

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

13

To Mom and Dad

14

¹⁵ Acknowledgements

¹⁶ Sooooooooo many people

Contents

17

1	Introduction	1	18
2	Hamiltonian MCMC	4	19
2.1	Intro	4	20
2.2	Markov Chains	4	21
2.3	Likelihood	6	22
2.4	Prior distribution	6	23
2.5	Proposal distribution	6	24
2.6	Algorithm	7	25
2.7	Burn-in	8	26
2.8	Thinning	8	27
2.9	Hamiltonian Monte Carlo	9	28
2.10	Fitting	12	29
3	Iterated Filtering	20	30
3.1	Intro	20	31
3.2	Formulation	20	32
3.3	Algorithm	21	33
3.4	Particle Collapse	23	34
3.5	Iterated Filtering and Data Cloning	23	35
3.6	IF2	24	36
3.7	Fitting	26	37
4	Parameter Fitting	30	38
4.1	Fitting Setup	30	39
4.2	Calibrating Samples	33	40
4.3	IF2 Fitting	34	41
4.4	IF2 Convergence	36	42
4.5	IF2 Densities	38	43
4.6	HMCMC Fitting	39	44
4.7	HMCMC Densities	39	45
4.8	HMCMC and Bootstrapping	40	46

47	4.9 Multi-trajectory Parameter Estimation	41
48	5 Forecasting Frameworks	43
49	5.1 Data Setup	43
50	5.2 IF2	44
51	5.2.1 Parametric Bootstrapping	45
52	5.2.2 IF2 Forecasts	45
53	5.3 HMC MC	47
54	5.4 Truncation vs. Error	48
55	6 S-map and SIRS	50
56	6.1 S-maps	50
57	6.2 S-map Algorithm	51
58	6.3 SIRS Model	53
59	6.4 SIRS Model Forecasting	55
60	7 Spatial Epidemics	57
61	7.1 Spatial SIR	57
62	7.2 Dewdrop Regression	59
63	7.3 Spatial Model Forecasting	60
64	8 Discussion and Future Directions	62
65	8.1 Parallel and Distributed Computing	62
66	8.2 IF2, Bootstrapping, and Forecasting Methodology	65
67	A Hamiltonian MCMC	66
68	A.1 Full R code	66
69	A.2 Full Stan code	69
70	B Iterated Filtering	71
71	B.1 Full R code	71
72	B.2 Full C++ code	74
73	C Parameter Fitting	85
74	D Forecasting Frameworks	86
75	D.1 IF2 Parametric Bootstrapping Function	86
76	D.2 RStan Forward Simulator	88
77	E S-map and SIRS	90
78	E.1 SIRS R Function Code	90
79	E.2 SMAP Code	91
80	E.3 SMAP Parameter Optimization Code	93

E.4	RStan SIRS Code	95	81
E.5	IF2 SIRS Code	97	82
F	Spatial Epidemics	110	83
F.1	Spatial SIR R Function Code	110	84
F.2	RStan Spatial SIR Code	112	85
F.3	IF2 Spatial SIR Code	114	86
F.4	CUDA IF2 Spatial Fitting Code	125	87

List of Figures

89	2.1	Finite state machine. (<i>Andrieu et al., 2003</i>)	5
90	2.2	True SIR ODE solution infected counts, and with added observation	
91		noise	14
92	2.3	Traceplot of samples drawn for parameter R_0 , excluding warmup . .	17
93	2.4	Traceplot of samples drawn for parameter R_0 , including warmup. . .	18
94	2.5	Kernel density estimates produced by Stan	19
95	3.1	True SIR ODE solution infected counts, and with added observation	
96		noise.	28
97	3.2	Kernel estimates for four essential system parameters. True values	
98		are indicated by solid vertical lines, sample means by dashed lines. .	29
99	4.1	Simulated geometric autoregressive process show in Equation [4.2]. .	31
100	4.2	Density plot of values shown in Figure[4.1].	32
101	4.3	Stochastic SIR model simulated using an explicit Euler stepping scheme.	
102		The solid line is a single random trajectory, the dots show the data	
103		points obtained by adding in observation error defined as $\epsilon_{obs} =$	
104		$\mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile from 100 ran-	
105		dom trajectories.	32
106	4.5	True system trajectory (solid line), observed data (dots), and IF2	
107		estimated real state (dashed line).	35
108	4.4	Fitting errors.	35
109	4.6	The horizontal axis shows the IF2 pass number. The solid black lines	
110		show the evolution of the ML estimates, the solid grey lines show	
111		the true value, and the dashed grey lines show the mean parameter	
112		estimates from the particle swarm after the final pass.	37
113	4.7	The horizontal axis shows the IF2 pass number and the solid black	
114		lines show the evolution of the standard deviations of the particle	
115		swarm values.	37
116	4.8	As before, the solid grey lines show the true parameter values and	
117		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.	118 40 119
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.	120 121 122 123 41 124
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.	125 126 42 127
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.	128 129 42 130
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 $\mathcal{N}(0, 10)$	131 132 133 134 44 135
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.	136 137 45 138
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 $\mathcal{N}(0, \sigma)$	139 140 141 142 46 143
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.	144 145 48 146
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.	147 148 49 149
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.	150 151 152 54 153
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 $\theta = 3$	154 155 156 54 157

158	6.3	Error as a function of forecast length.	56
159	6.4	Runtimes for producing SIRS forecasts. The box shows the middle	
160		50th quantile, the bold line is the median, and the dots are outliers.	56
161	7.1	Evolution of a spatial epidemic in a ring topology. The outbreak	
162		was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$,	
163		$\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$	58
164	7.2	Evolution of a spatial epidemic as in Figure [7.1], with added obser-	
165		vation noise drawn from $\mathcal{N}(0, 10)$	59
166	7.3	Average SSE (log scale) across each location and all trials as a func-	
167		tion of the number of weeks ahead in the forecast.	61
168	7.4	Runtimes for producing spatial SIR forecasts. The box shows the	
169		middle 50th quantile, the bold line is the median, and the dots are	
170		outliers.	61

Chapter 1

171

Introduction

172

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.

173

174

175

176

177

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.

178

179

180

181

182

183

184

185

186

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.

187

188

189

190

191

192

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

193

194

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)

195

196

197

198 model. ARIMA assumes a linear underlying process and Gaussian error distri-
 199 butions. It uses three parameters representing the degree of autoregression (p),
 200 integration (trend removal) (d), and the moving average (q), where the orders of
 201 the autoregression and the moving average are determined through the use of an
 202 autocorrelation function (ACF) and partial autocorrelation function (PACF), re-
 203 spectively, applied to the the data *a priori*.

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

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

213 where S , I , and R are the number of individuals in each compartment, β is the
 214 “force” of infection acting on the susceptible population, and γ is a recovery rate.
 215 As an outbreak progresses, individuals transition from the susceptible compartment,
 216 through the infectious compartment, then finish in the removed compartment where
 217 they no longer impact the system dynamics. Many extensions of the SIR model exist
 218 are are commonly used, such as the SEIR model in which susceptible individuals pass
 219 through an exposed class where they have been infected but are not yet themselves
 220 infectious, and the SIRS model in which individuals become susceptible again after
 221 their immunity wanes.

222 Combining the phenomenological and mechanistic approaches are the semi-mechanistic
 223 techniques. These methods use a model to define the expected underlying dynamics
 224 of the system, but integrate data into the model in order to refine estimates of the
 225 model parameters and produce more accurate forecasts. Typically the first step in
 226 implementing such a technique is fitting the desired model to existing data. There
 227 are many ways to do this, most of which fall into two main categories: particle
 228 filter-based (PF) methods, and Markov chain Monte Carlo-based (MCMC) meth-
 229 ods. From there data can either be integrated into the model by refitting the model
 230 to the new longer data set, or in an “on-line” fashion in which data points can be di-
 231 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 phenomeno-
logical 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.

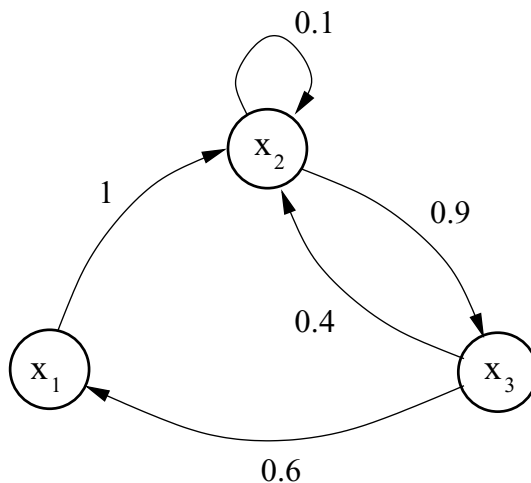
2.2 Markov Chains

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

```

349 **2.7 Burn-in**

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

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

358 **2.8 Thinning**

359 Some models will require very long chains to get a good approximation of the pos-
360 terior, which will consequently require a non-trivial amount of computer storage.
361 One way to reduce the burden of storing so many samples is by thinning. This
362 involves saving only every n^{th} step, which should still give a decent approximate of
363 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.

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

389 .

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

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

$$401 \quad r = \exp [H(\theta, r) - H(\theta^*, r^*)], \quad (2.10)$$

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

403 Together, we have Algorithm [2].

404 Note that the parameters ε and L have to be tuned in order to maintain stability
 405 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].

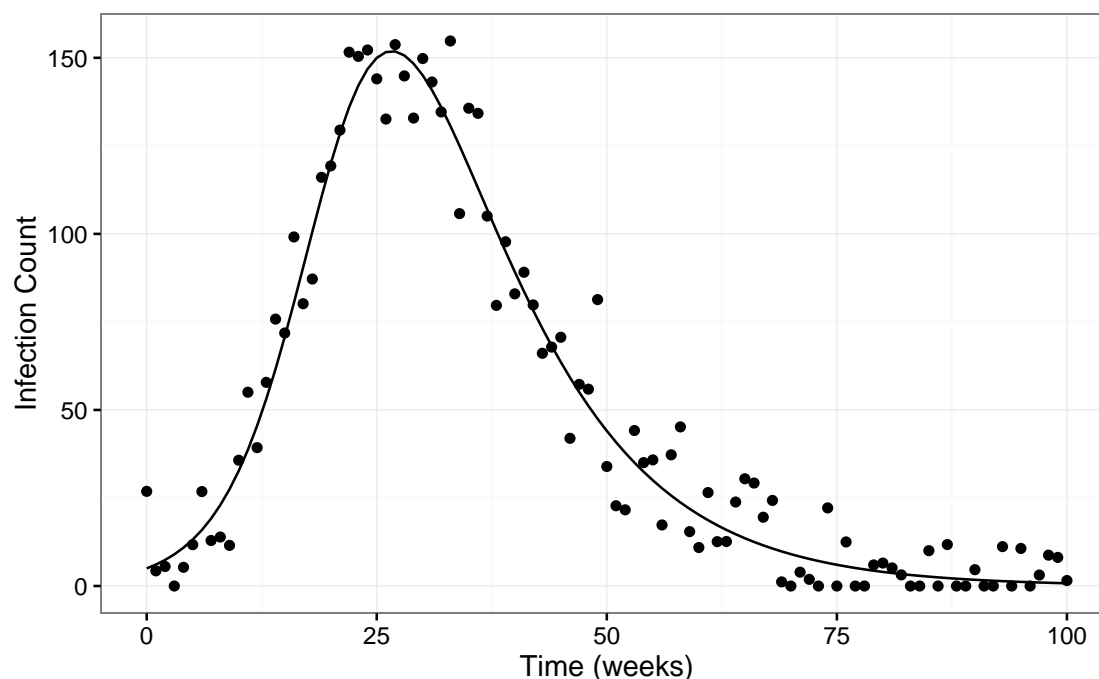


Figure 2.2: True SIR ODE solution infected counts, and with added observation noise

486 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>)
 488 was used to ease implementation.
 489

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

```
496 1 sPw <- 7 # steps per week
497 2 datlen <- (T-1)*7 + 1 # size of sparse data vector
498 3
499 4 data <- matrix(data = -1, nrow = T+1, ncol = sPw)
500 5 data[,1] <- infec_counts
501 6 standata <- as.vector(t(data))[1:datlen]
```

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

```
505 1 sir_data <- list( T = datlen, # simulation time
506
```



```

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

```

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)
2  options(mc.cores = parallel::detectCores())

```

Now we call the Stan fitting function

```

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

```

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

```

```

554 4      real <lower=0, upper=10>    R0;      // R0
555 5      real <lower=0, upper=10>    r;       // recovery rate
556 6      real <lower=0, upper=500>    y0[3];  // initial conditions
557 7
558 8      }
559

```

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

```

568 1      model {
569 2
570 3          real S[T];
571 4          real I[T];
572 5          real R[T];
573 6
574 7          S[1] <- y0[1];
575 8          I[1] <- y0[2];
576 9          R[1] <- y0[3];
577 10
578 11          y[1] ~ normal(y0[2], sigma);
579 12
580 13          for (t in 2:T) {
581 14
582 15              S[t] <- S[t-1] + h*( - S[t-1]*I[t-1]*R0*r/N );
583 16              I[t] <- I[t-1] + h*( S[t-1]*I[t-1]*R0*r/N - I[t-1]*r );
584 17              R[t] <- R[t-1] + h*( I[t-1]*r );
585 18
586 19              if (y[t] > 0) {
587 20                  y[t] ~ normal( I[t], sigma );
588 21              }
589 22
590 23          }
591 24
592 25          y0[1] ~ normal(N - y[1], sigma);
593 26          y0[2] ~ normal(y[1], sigma);
594 27
595 28          theta[1] ~ lognormal(1,1);
596 29          theta[2] ~ lognormal(1,1);
597 30          sigma ~ lognormal(1,1);
598 31
599 32      }
600

```

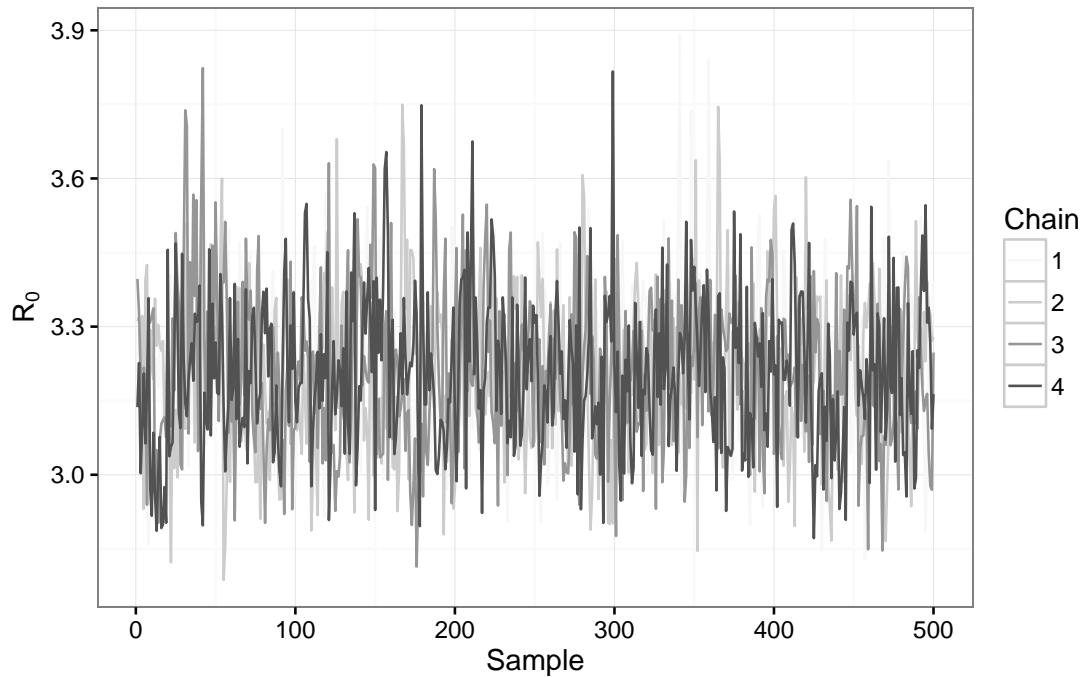


Figure 2.3: Traceplot of samples drawn for parameter R_0 , excluding warmup

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].

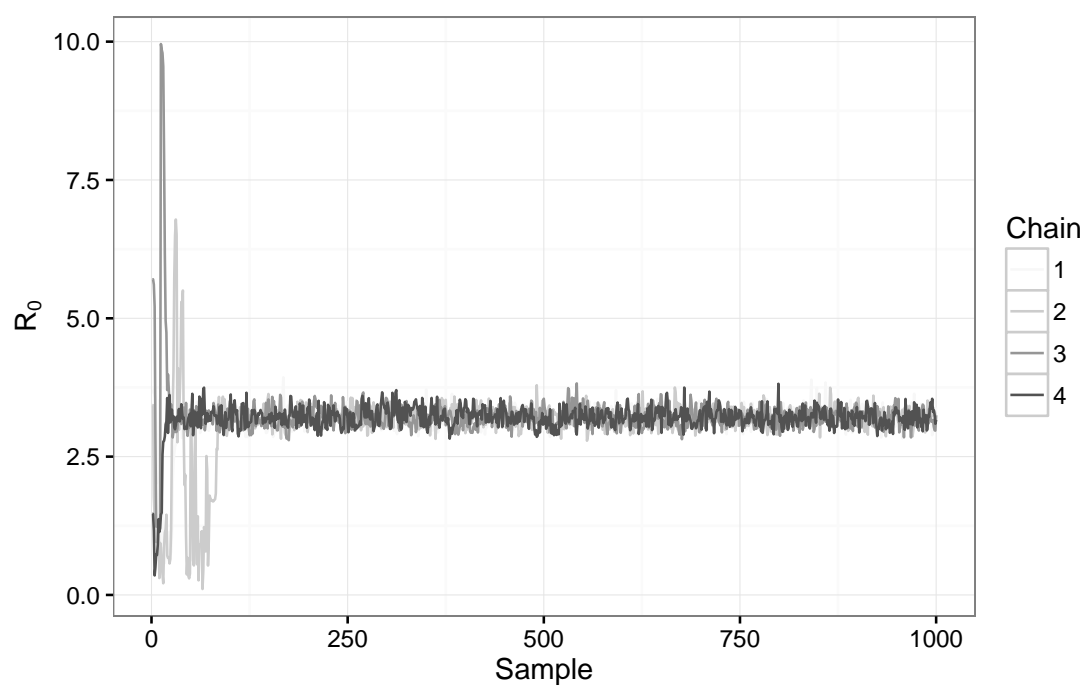


Figure 2.4: Traceplot of samples drawn for parameter R_0 , including warmup.

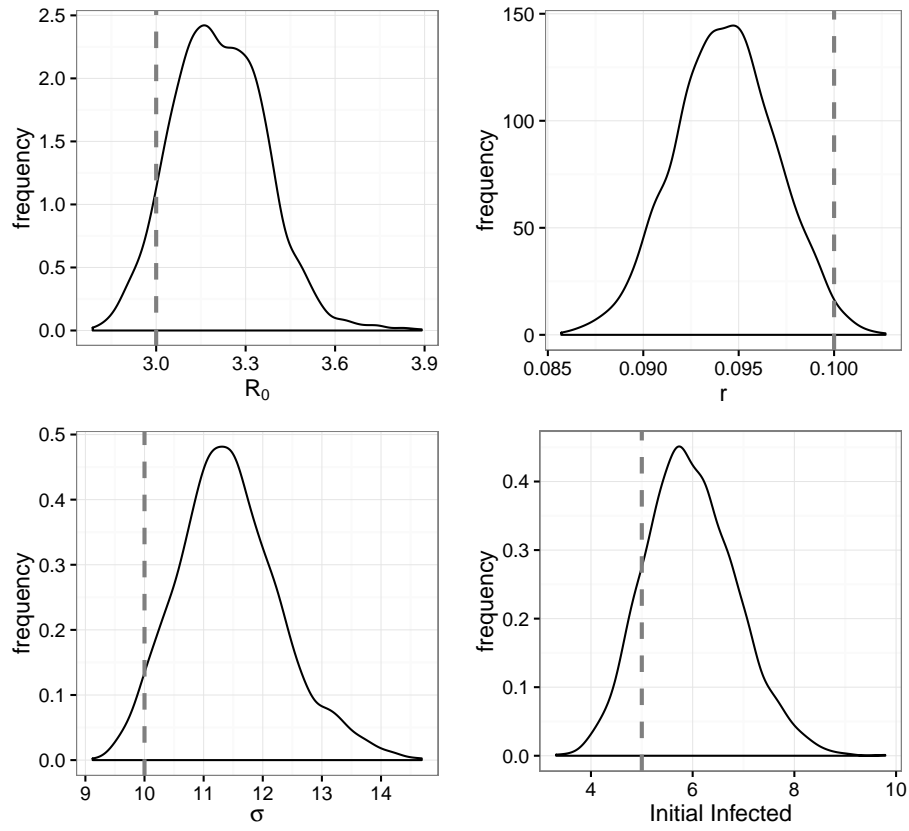


Figure 2.5: Kernel density estimates produced by Stan

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

$$X_{t+1} \sim f_1(X_t, \theta), \quad (3.1) \quad 632$$

And the observation process by 633

$$Y_t \sim f_2(X_t, \theta). \quad (3.2) \quad 634$$

Note that components of θ can contribute to both functions, but a typical formulation is to have some components contribute to $f_1(\cdot, \theta)$ and others to $f_2(\cdot, \theta)$. 635
636

The prediction part of the cycle utilises $f_1(\cdot, \theta)$ to update each particle's current state estimate to the next time step, while $f_2(\cdot, \theta)$ is used to evaluate a weighting w for each particle which will be used to determine how closely that particle is estimating the true underlying state of the system. Note that $f_2(\cdot, \theta)$ could be thought of as a probability of observing a piece of data y_t given the particle's current state estimate and parameter set, $P(y_t|X_t, \theta)$. Then, the new cohort of particles is drawn from the old cohort proportional to the weights. This process is repeated until the set of observations D is exhausted. 637
638
639
640
641
642
643
644

3.3 Algorithm 645

Now we will formalize the particle filter. 646

We will denote each particle $p^{(j)}$ as the j^{th} particle consisting of a state estimate at time t , $X_t^{(j)}$, a parameter set $\theta^{(j)}$, and a weight $w^{(j)}$. Note that the state estimates will evolve with the system as the cohort traverses the data. 647
648
649

The algorithm for a Sequential Importance Resampling particle is shown in Algorithm [3]. 650
651

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.

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

688 3.6 IF2

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

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

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

710 The algorithm for IF2 can be seen in Algorithm [4].
 711

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="/"))

```

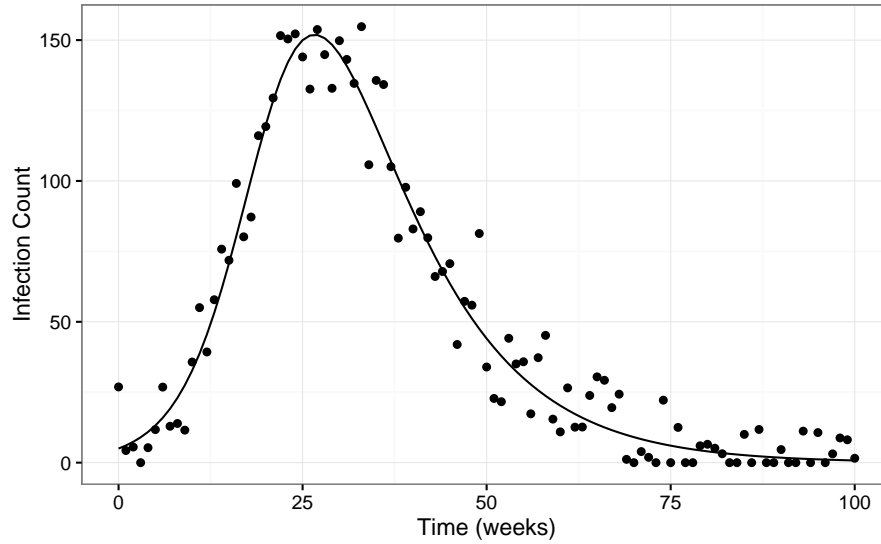


Figure 3.1: True SIR ODE solution infected counts, and with added observation noise.

794 Then run and packed into a data frame using

```
795 1 paramdata <- data.frame(if2(infec_counts[1:Tlim], Tlim, N))
796 2 colnames(paramdata) <- c("R0", "r", "sigma", "Sinit", "Iinit", "
797   Rinit")
798
799
```

800 The final kernel estimates for four of the key parameters are shown in Figure
801 [3.2].

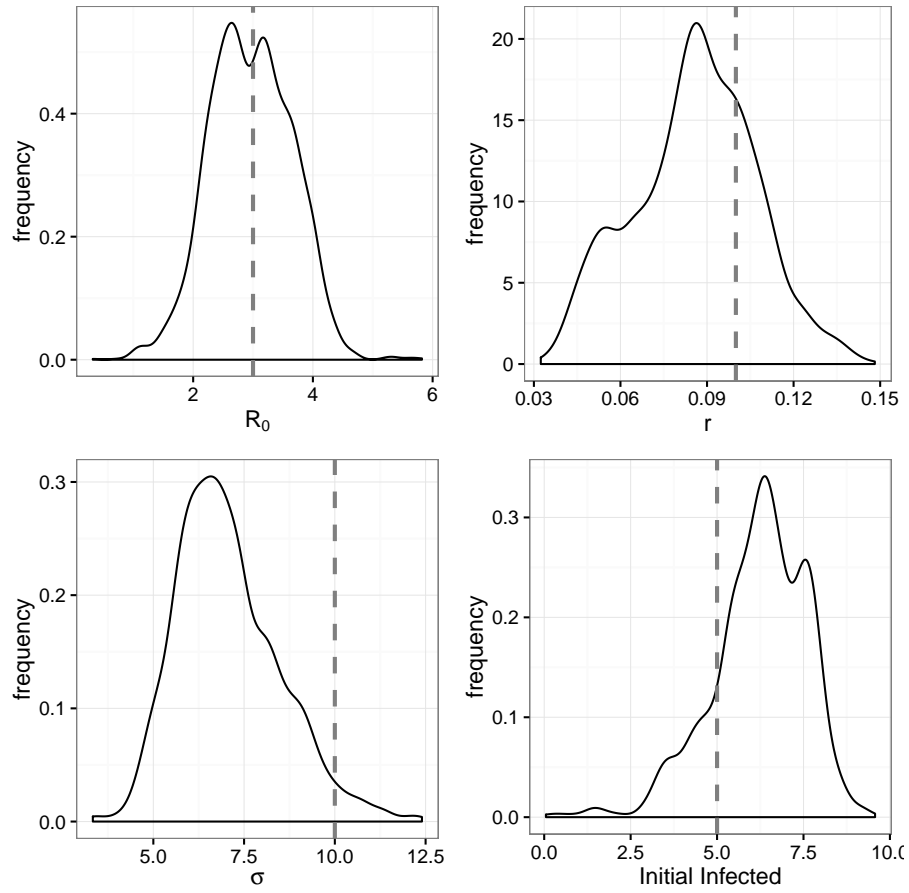


Figure 3.2: Kernel estimates for four essential system parameters. True values are indicated by solid vertical lines, sample means by dashed lines.

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).

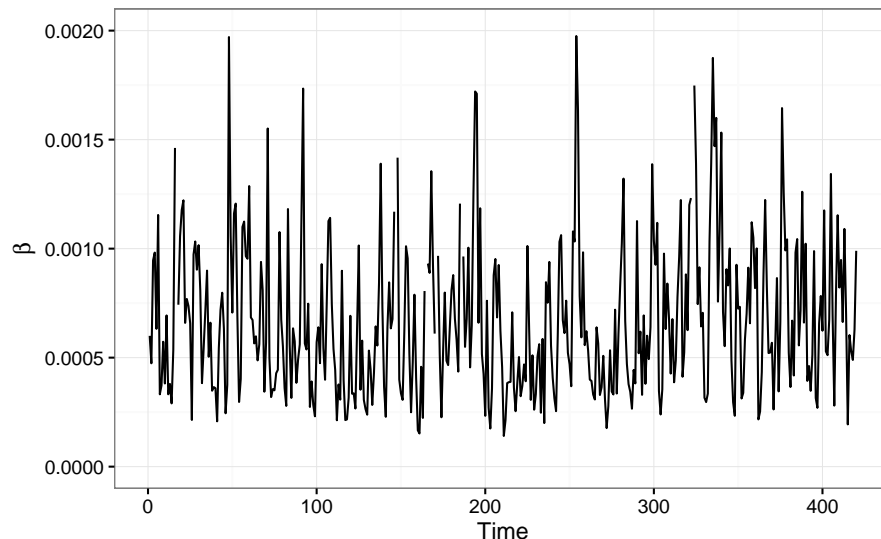


Figure 4.1: Simulated geometric autoregressive process show in Equation [4.2].

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].

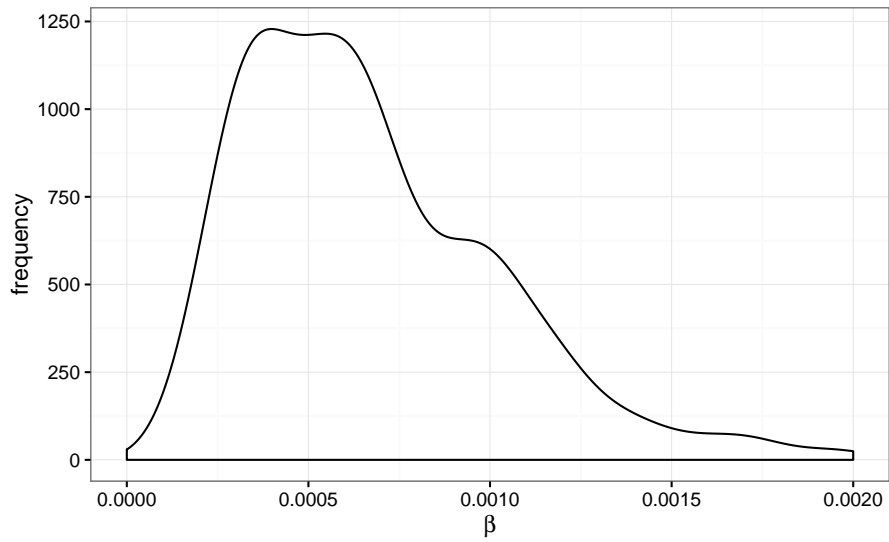


Figure 4.2: Density plot of values shown in Figure 4.1.

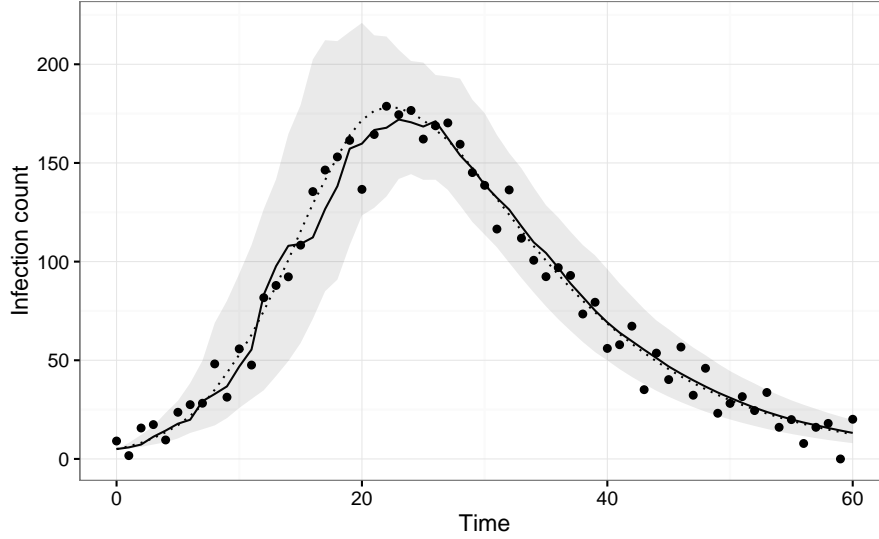


Figure 4.3: Stochastic SIR model simulated using an explicit Euler stepping scheme. The solid line is a single random trajectory, the dots show the data points obtained by adding in observation error defined as $\epsilon_{obs} = \mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile from 100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

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

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

4.3 IF2 Fitting

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

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

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

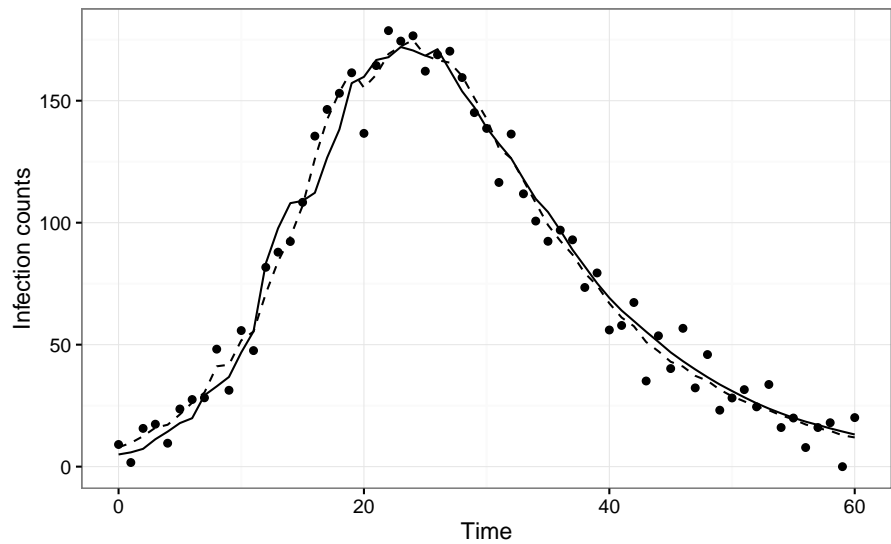


Figure 4.5: True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).

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].

886
887
888

889 4.4 IF2 Convergence

890 Since IF2 is an iterative algorithm where each pass through the data is expected to
891 push the parameter estimates towards the MLE, we can see the evolution of these
892 estimates as a function of the pass number. Plots showing evolution of the mean
893 estimates are shown in Figure [4.6] for the six most critical parameters.

894 Similarly, we can look at the evolution of the standard deviations of the parameter
895 estimates from the particle swarm as a function of the pass number, shown in Figure
896 [4.7].

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

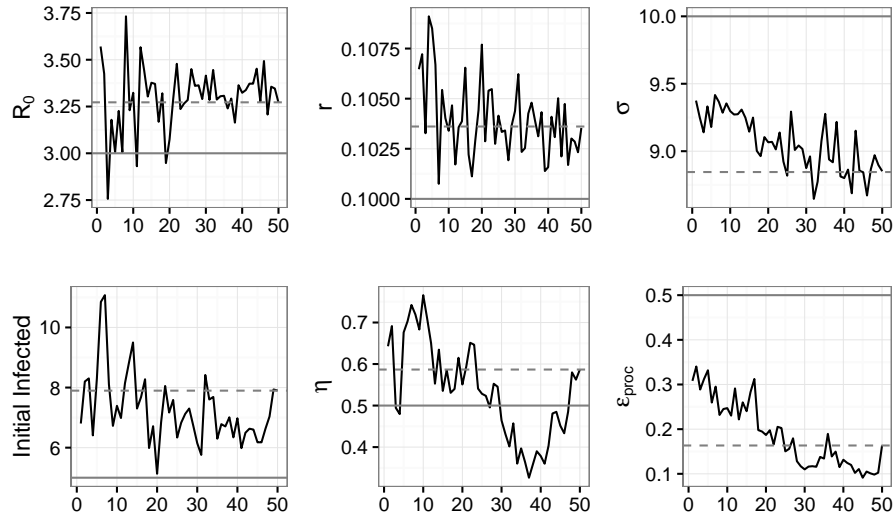


Figure 4.6: The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.

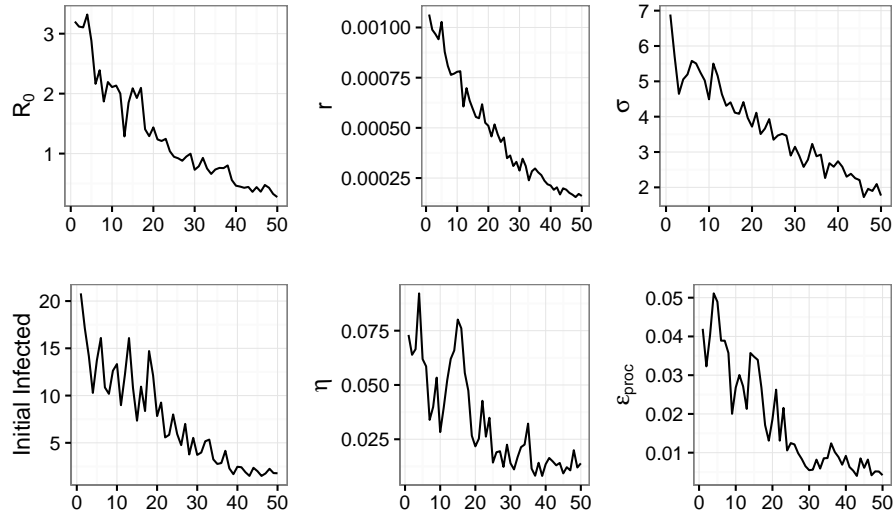


Figure 4.7: The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.

899 4.5 IF2 Densities

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

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

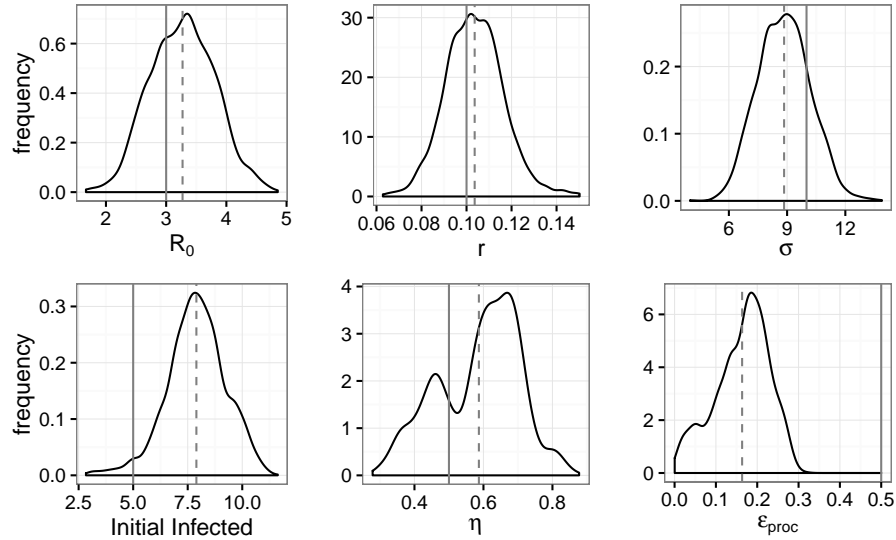


Figure 4.8: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

4.6 HMCMC Fitting

906

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.

907

908

909

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.

910

911

912

4.7 HMCMC Densities

913

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

914

915

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.

916

917

918

919

920

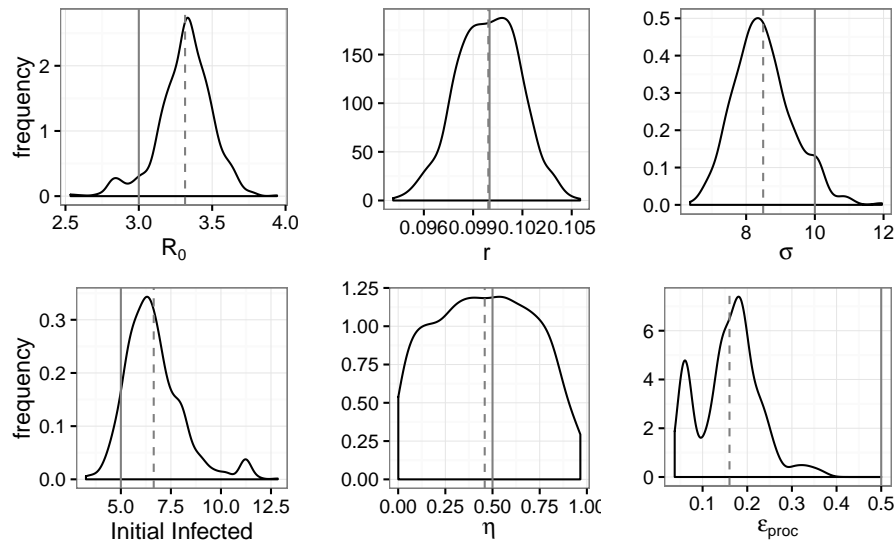


Figure 4.9: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

921 4.8 HMCMC and Bootstrapping

922 Unlike particle particle-filtering-based approaches, HMC does not produce state es-
 923 timates as a by-product of parameter fitting, but we can use information about
 924 the stochastic nodes related to the noise in the β geometric random walk to recon-
 925 struct state estimates. The results of 100 bootstrap trajectories is shown in Figure
 926 [4.10].

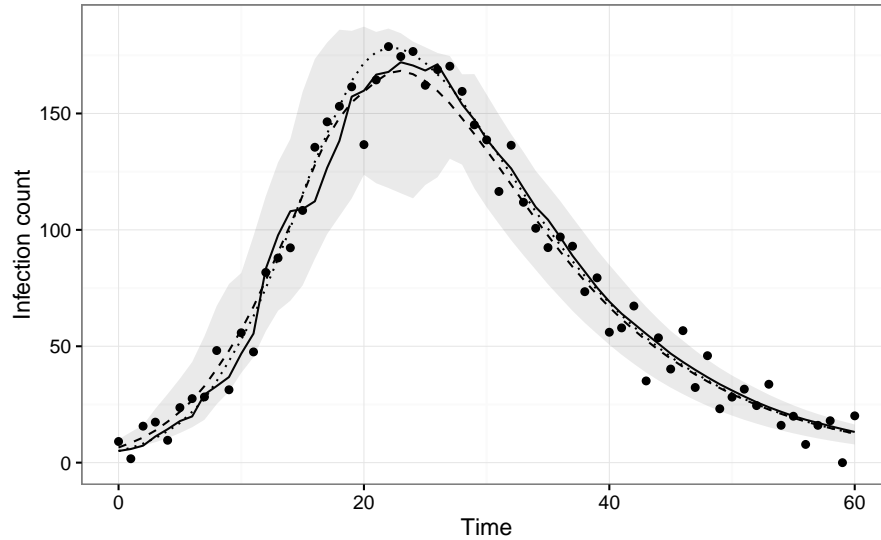


Figure 4.10: Result from 100 HMCBC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.

4.9 Multi-trajectory Parameter Estimation

927

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

928

929

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.

930

931

932

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

933

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.

934

935

936

937

938

939

940

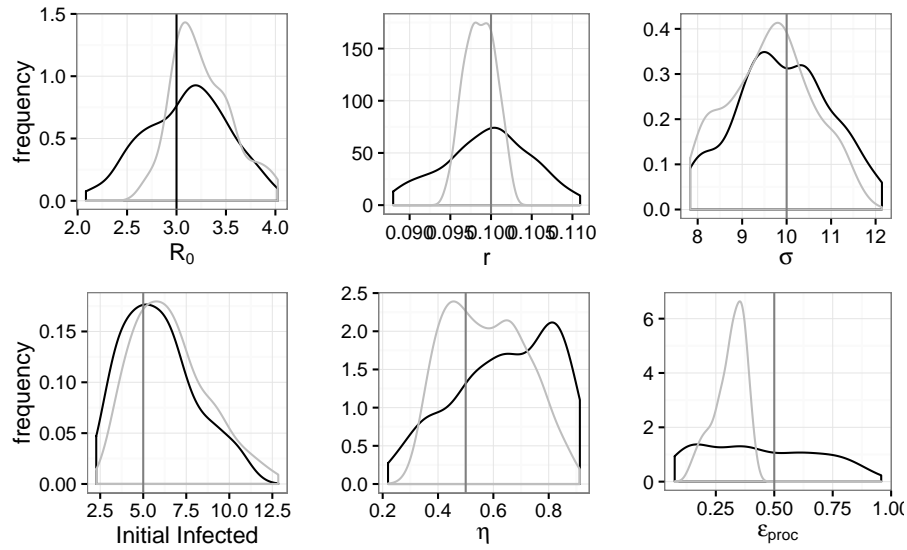


Figure 4.11: IF2 point estimate densities are shown in black and HMCMC point estimate densities are shown in grey. The vertical black lines show the true parameter values.

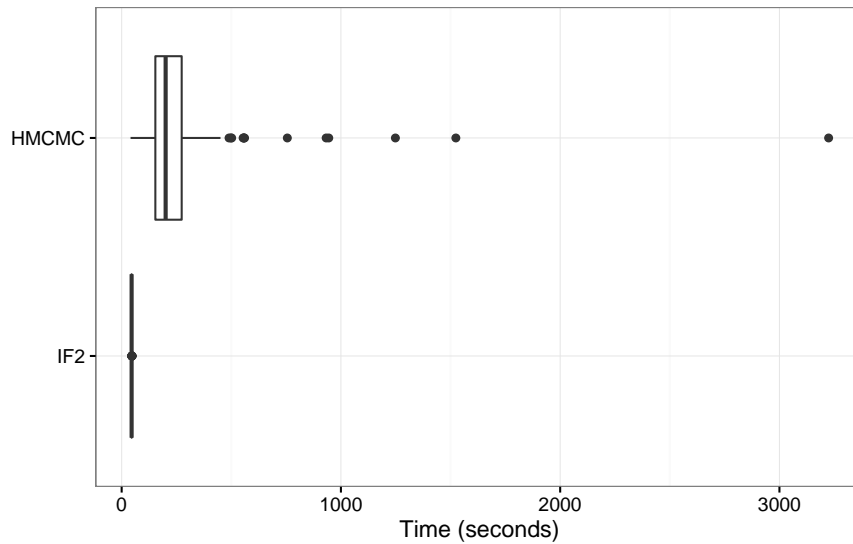


Figure 4.12: Fitting times for IF2 and HMCMC, in seconds. The centre box in each plot shows the centre 50th quantile, with the bold centre line showing the median.

Chapter 5

941

Forecasting Frameworks

942

5.1 Data Setup

943

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.

944

945

946

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

947

948

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.

949

950

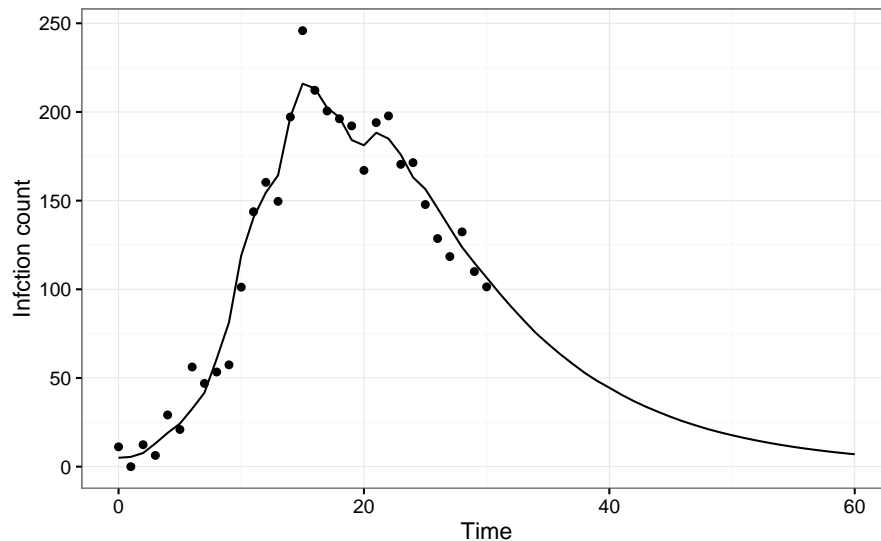


Figure 5.1: Infection count data truncated at $T = 30$. The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were $R_0 = 3.0$, $r = 0.1$, $\eta = .05$, $\sigma_{proc} = 0.5$, and additive observation noise was drawn from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

5.2 IF2

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

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

First, we can see the state estimates for each time point produced by the last IF2 pass in Figure [5.2].

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

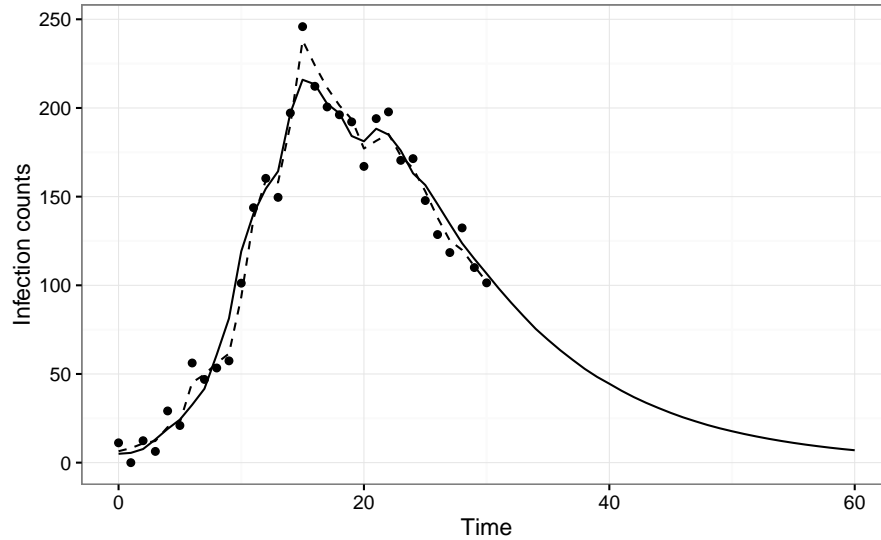


Figure 5.2: Infection count data truncated at $T = 30$ from Figure [5.1]. The dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points.

5.2.1 Parametric Bootstrapping

967

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.

968

969

970

971

[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.]

972

973

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

974

975

5.2.2 IF2 Forecasts

976

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].

977

978

979

980

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 $\overline{SSE} = 1.67$.

981

982

983

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

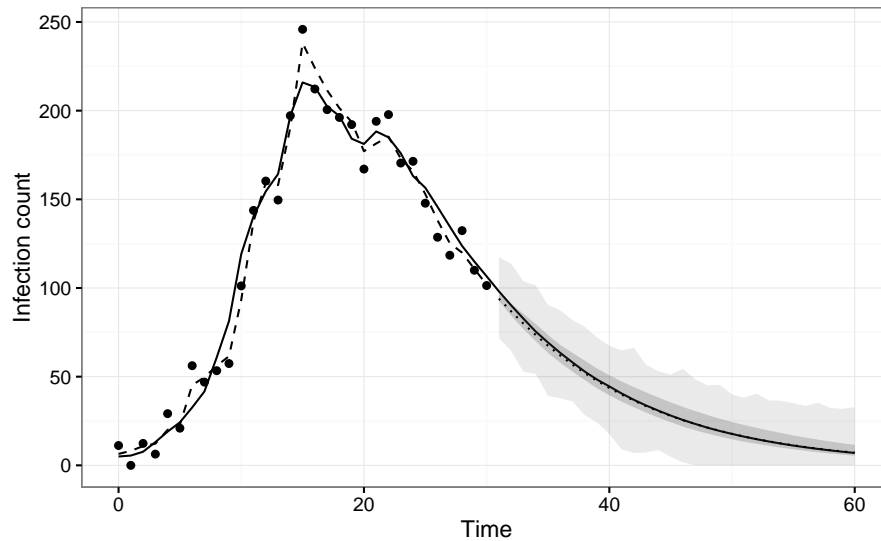


Figure 5.3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the centre 95th quantile of the true state estimates, and the lighter grey ribbon shows the centre 95th quantile of the true state estimates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

5.3 HMCMC

984

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.

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].

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

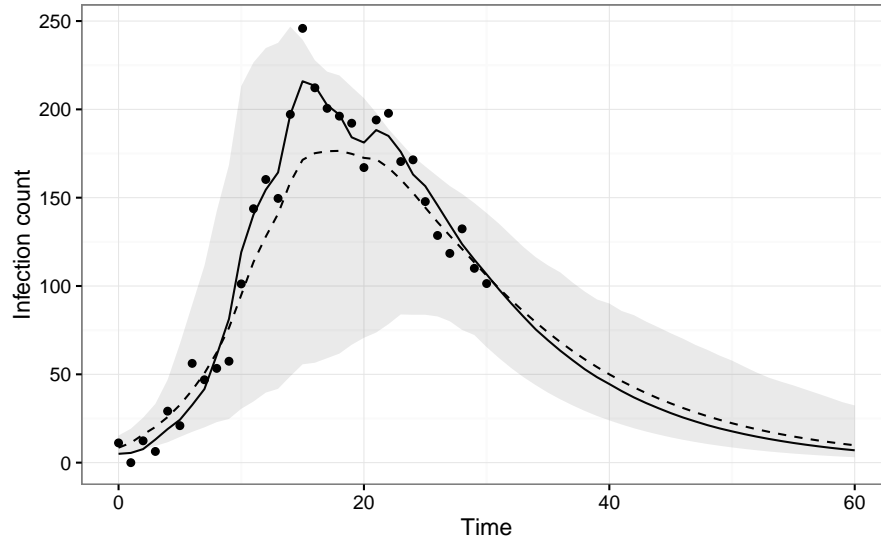


Figure 5.4: Forecast produced by the HMC MC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the centre 95th quantile.

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

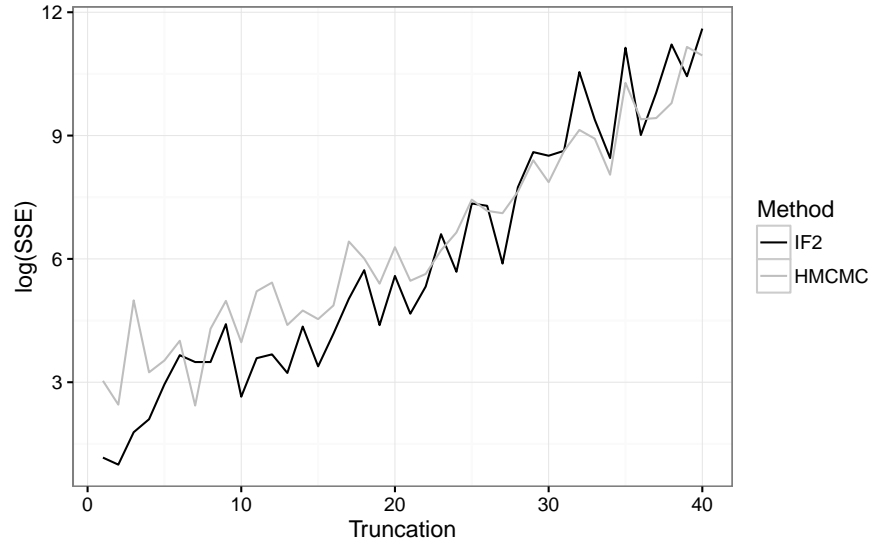


Figure 5.5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

additional effort. The code used here has this capability, but it was not utilised in 1011
the comparison so as to accurately represent total computational cost, not potential 1012
running time. 1013

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

1039

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

1040

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.

1041

1042

1043

The weighting function w is defined as

1044

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

1045

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.

1046

1047

1048

1049

1050

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

1051

1052

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

1053

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.

1054

1055

1056

6.2 S-map Algorithm

1057

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

1058

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

1059

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.

1064

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.

1067

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.

1068

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

1070

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

1072

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

1073

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.

1077

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

1078

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

1079

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

1080

1081

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

1082

1083

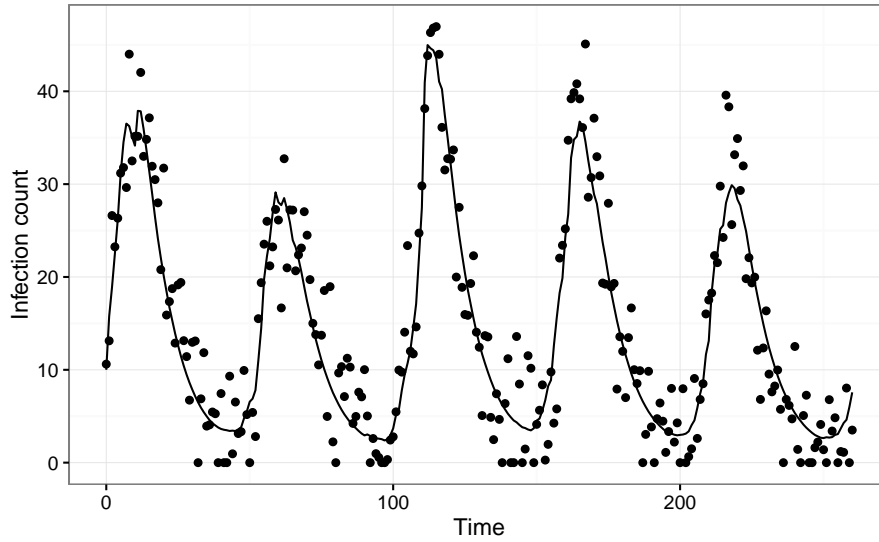


Figure 6.1: Five cycles generated by the SIRS function. The solid line the the true number of cases, dots show case counts with added observation noise. The Parameter values were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 1$, $\sigma = 5$, and 10 initial cases.

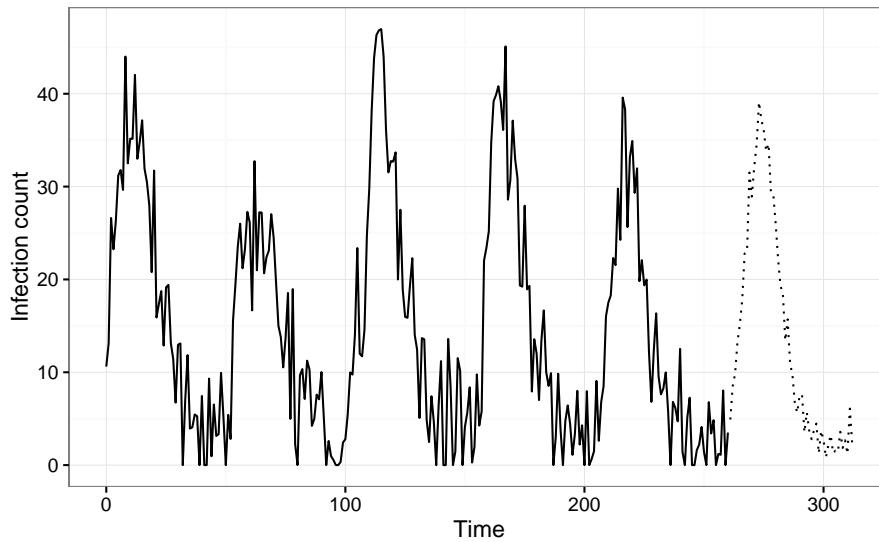


Figure 6.2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise from the previous plot, and the dotted line is the S-map forecast. Parameters chosen were $E = 14$ and $\theta = 3$.

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.

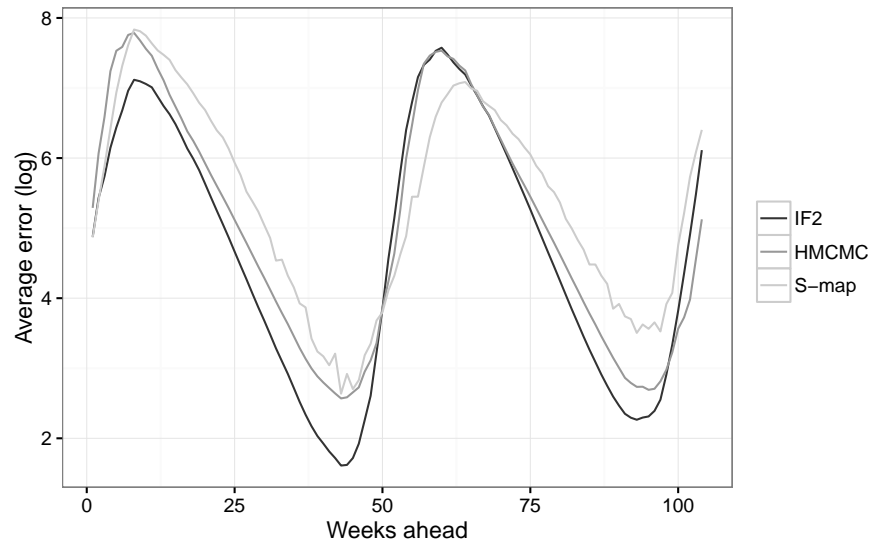


Figure 6.3: Error as a function of forecast length.

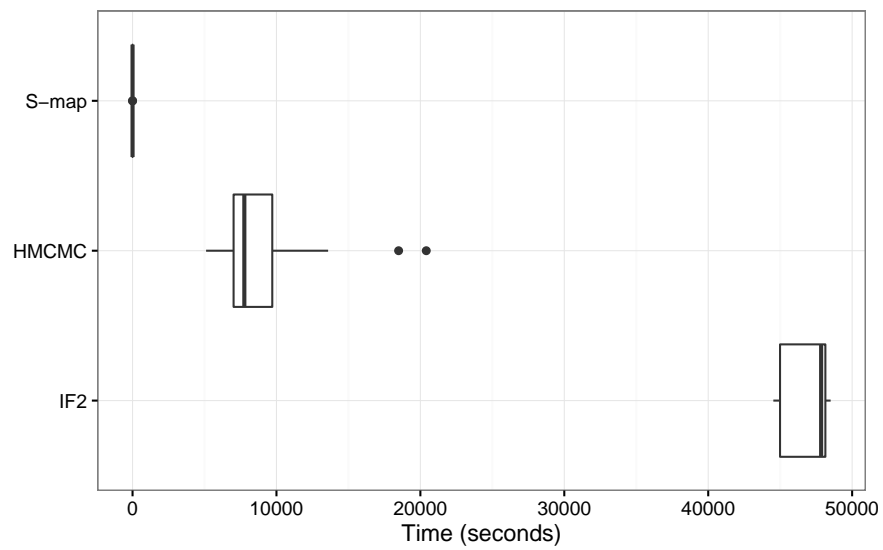


Figure 6.4: Runtimes for producing SIRS forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

Chapter 7

1109

Spatial Epidemics

1110

7.1 Spatial SIR

1111

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

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

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

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

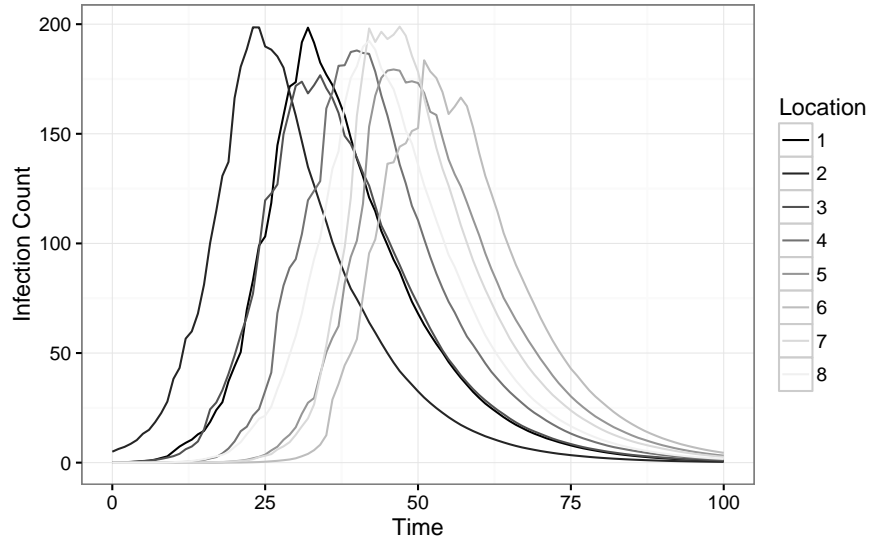


Figure 7.1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$.

1127 As before we let β embark on a geometric random walk defined as

$$1128 \quad \beta_{i,t+1} = \exp \left(\log(\beta_{i,t}) + \eta(\log(\bar{\beta}) - \log(\beta_{i,t})) + \epsilon_t \right). \quad (7.2)$$

1129 Note that as β is a state variable, each location has its own stochastic process driving
1130 the evolution of its β state.

1131 If we imagine a circular topology in which each of 10 locations is connected to
1132 exactly two neighbours (i.e. location 1 is connected to locations N and 2, location 2
1133 is connected to locations 1 and 3, etc.), and we start each location with completely
1134 susceptible populations except for a handful of infected individuals in one of the
1135 locations, we obtain a plot of the outbreak progression in Figure [7.1].

1136 If we add noise to the data from Figure [7.1], we obtain Figure [7.2].

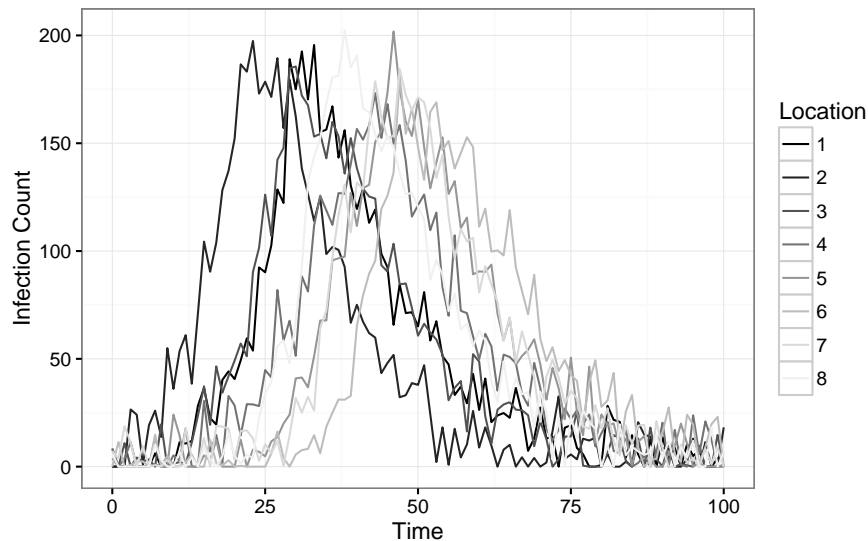


Figure 7.2: Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from $\mathcal{N}(0, 10)$.

7.2 Dewdrop Regression

1137

Dewdrop regression (references) aims to overcome the primary disadvantage suffered 1138
by methods such as the S-map or its cousin Simplex Projection: the requirement 1139
of long time series from which to build a library. Suggested by Sugihara’s group in 1140
2008, Dewdrop Regression works by stitching together shorter, related, time series, 1141
in order to give the S-map or similar methods enough data to operate on. The 1142
underlying idea is that as long as the underlying dynamics of the time series display 1143
similar behaviour (such as potentially collapsing to the same attractor), they can 1144
be treated as part of the same overarching system. 1145

It is not enough to simply concatenate the shorter time series together – several 1146
procedures must be carried out and a few caveats observed. First, as the individual 1147
time series can be or drastically differing scales and breadths, they all must be 1148
rescaled to unit mean and variance. Then the library is constructed as before with 1149
an embedding dimension E , but any library vectors that span any of the seams 1150
joining the time series are discarded. Further, and predictions stemming from a 1151
library vector must stay within the time series from which they originated. In this 1152
way we are allowing the “shadow” of of the underlying dynamics of the separate 1153
time series to infer the forecasts for segments of other time series. Once the library 1154
has been constructed, S-mapping can be carried out as previously specified. 1155

This procedure is especially well-suited to a the spatial model we are using. While 1156
the dynamics are stochastic, they still display very similar means and variances. 1157

1158 This means the rescaling process in Dewdrop Regression is not necessary and can
1159 be skipped. Further, the overall variation between the epidemic curves in each
1160 location is on the smaller side, meaning the S-map will have a high-quality library
1161 from which to build forecasts.

1162 7.3 Spatial Model Forecasting

1163 In order to compare the forecasting efficacy of Dewdrop Regression with S-mapping
1164 against IF2 and HMCMC, we generated 20 independent spatial data sets up to time
1165 $T = 50$ weeks in each of $L = 10$ locations and forecasted 10 weeks into the future.
1166 Forecasts were compared to that of the true model evolution, and the average *SSE*
1167 for each week ahead in the forecast were computed. The number of bootstrapping
1168 trajectories used by IF2 and HMCMC was reduced from 200 to 50 to curtail running
1169 times.

1170 The results are shown in Figure [7.3].

1171 The results show a clear delineation in forecast fidelity between methods. IF2 main-
1172 tains an advantage regardless of how long the forecast produced. Interestingly, Dew-
1173 drop Regression with S-mapping performs almost as well as IF2, and outperforms
1174 HMCMC. HMCMC lags behind both methods by a healthy margin.

1175 If we examine the runtimes for each forecast framework, we obtain the data in Figure
1176 [7.4].

1177 As before, the S-map with Dewdrop Regression runs faster than the other two
1178 methods with a huge margin. It is again hard to see exactly how large the margin
1179 is from the figure due to the scale, but we can examine the average values: the
1180 average running time for S-mapping with Dewdrop Regression was about 249 sec-
1181 onds, whereas the average times for IF2 and HMCMC were about 2.90×10^4 and
1182 3.88×10^4 , respectively. This is a speed-up of just over 116x over IF2 and 156x over
1183 HMCMC.

1184 Considering how well S-mapping performed with regards to forecast error, it shows a
1185 significant advantage over HMCMC in particular – it outperforms it in both forecast
1186 error and running times.

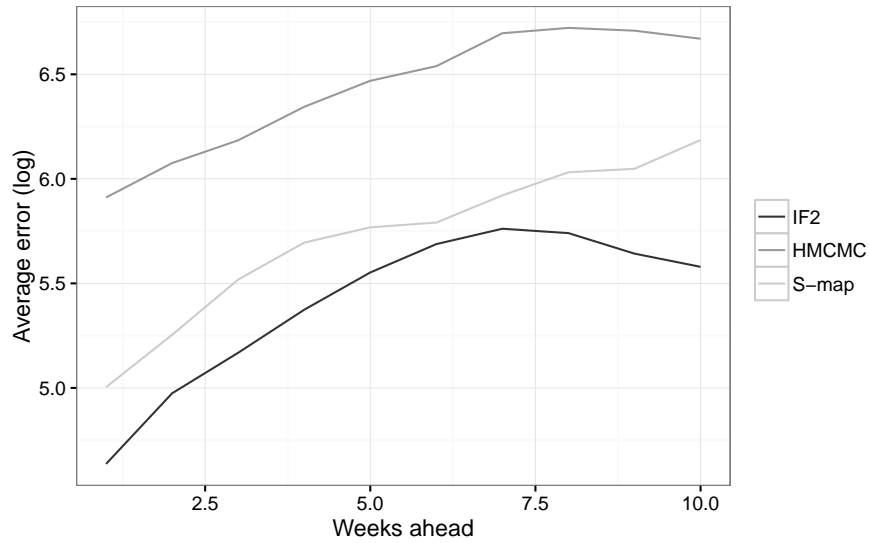


Figure 7.3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.

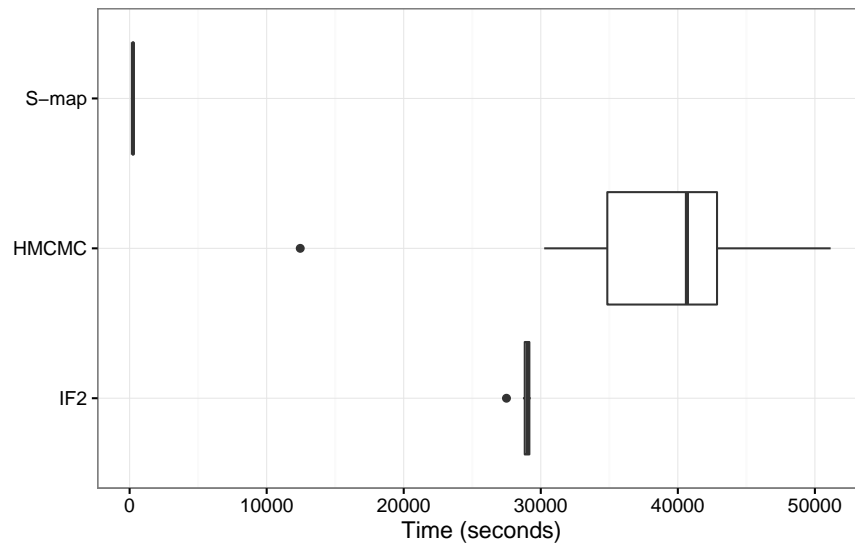


Figure 7.4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

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.

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

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

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

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

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

S-mapping, like the other two methods, is parallelizable to a degree. However, the S-map is already a great deal faster than the other two methods, and in the worst case (paired with Dewdrop Regression and applied to a spatiotemporal data set) still only takes a few minutes to run. Setting this observation aside, if one were investing in developing a faster S-map implementation, this is certainly possible. By far the most computationally expensive component of the algorithm is the SVD

decomposition, and algorithms exist to accelerate it via parallelization. Further, each point in the forecast can be computed separately; in the cases similar to the one here with application to spatiotemporal prediction, there can be a significant number of these points.

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

8.2 IF2, Bootstrapping, and Forecasting Methodology

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

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

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

1316 Appendix A

1317 Hamiltonian MCMC

1318 A.1 Full R code

1319 This code will run all the indicated analysis and produce all plots.

```
1320 1 ## Dexter Barrows
1321 2 ## McMaster University
1322 3 ## 2016
1323 4
1324 5 library(deSolve)
1325 6 library(rstan)
1326 7 library(shinystan)
1327 8 library(ggplot2)
1328 9 library(RColorBrewer)
1329 10 library(reshape2)
1330 11
1331 12 SIR ← function(Time, State, Pars) {
1332 13
1333 14     with(as.list(c(State, Pars)), {
1334 15
1335 16         B ← R0*r/N
1336 17         BSI ← B*S*I
1337 18         rI ← r*I
1338 19
1339 20         dS = -BSI
1340 21         dI = BSI - rI
1341 22         dR = rI
1342 23
1343 24         return(list(c(dS, dI, dR)))
1344 25
1345 26     })
1346 27
1347 28 }
```

```

29 | 1349
30 | pars ← c(R0 ← 3.0,      # average number of new infected individuals
           per infectious person
31 |         r ← 0.1,      # recovery rate
32 |         N ← 500)      # population size
33 | 1351
34 | T ← 100
35 | y_ini ← c(S = 495, I = 5, R = 0)
36 | times ← seq(0, T, by = 1)
37 | 1355
38 | odeout ← ode(y_ini, times, SIR, pars)
39 | 1356
40 | set.seed(1001)
41 | sigma ← 10
42 | infec_counts_raw ← odeout[,3] + rnorm(T+1, 0, sigma)
43 | infec_counts      ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
44 | 1363
45 | g ← qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)",
           ylab = "Infection Count") +
46 |   geom_point(aes(y = infec_counts)) +
47 |   theme_bw()
48 | 1366
49 | print(g)
50 | ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
51 | 1367
52 | sPw ← 7
53 | datlen ← (T-1)*7 + 1
54 | 1368
55 | data ← matrix(data = -1, nrow = T+1, ncol = sPw)
56 | data[,1] ← infec_counts
57 | standata ← as.vector(t(data))[1:datlen]
58 | 1369
59 | sir_data ← list( T = datlen,      # simulation time
60 |                 y = standata,    # infection count data
61 |                 N = 500,         # population size
62 |                 h = 1/sPw )      # step size per day
63 | 1370
64 | rstan_options(auto_write = TRUE)
65 | options(mc.cores = parallel::detectCores())
66 | stan_options ← list( chains = 4,   # number of chains
67 |                     iter  = 2000, # iterations per chain
68 |                     warmup = 1000, # warmup iterations
69 |                     thin   = 2)    # thinning number
70 | fit ← stan(file      = "d_sirode_euler.stan",
71 |            data      = sir_data,
72 |            chains     = stan_options$chains,
73 |            iter       = stan_options$iter,
74 |            warmup     = stan_options$warmup,
75 |            thin       = stan_options$thin )
76 | 1371

```

```

1399 77 exfit ← extract(fit, permuted = TRUE, inc_warmup = FALSE)
1400 78
1401 79 R0points ← exfit$R0
1402 80 R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0])
1403      , ylab = "frequency") +
1404 81     geom_vline(aes(xintercept=R0), linetype="dashed", size=1,
1405      color="grey50") +
1406 82     theme_bw()
1407 83
1408 84 print(R0kernel)
1409 85 ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)
1410 86
1411 87 rpoints ← exfit$r
1412 88 rkernell ← qplot(rpoints, geom = "density", xlab = "r", ylab = "
1413     frequency") +
1414 89     geom_vline(aes(xintercept=r), linetype="dashed", size=1,
1415     color="grey50") +
1416 90     theme_bw()
1417 91
1418 92 print(rkernell)
1419 93 ggsave(rkernell, filename="kernelr.pdf", height=3, width=3.25)
1420 94
1421 95 sigmapoints ← exfit$sigma
1422 96 sigmakernell ← qplot(sigmapoints, geom = "density", xlab = expression
1423     (sigma), ylab = "frequency") +
1424 97     geom_vline(aes(xintercept=sigma), linetype="dashed", size=1,
1425     color="grey50") +
1426 98     theme_bw()
1427 99
1428 100 print(sigmakernell)
1429 101 ggsave(sigmakernell, filename="kernelsigma.pdf", height=3, width
1430     =3.25)
1431 102
1432 103 infecpoints ← exfit$y0[,2]
1433 104 infeckernell ← qplot(infecpoints, geom = "density", xlab = "Initial
1434     Infected", ylab = "frequency") +
1435 105     geom_vline(aes(xintercept=y_ini[['I']]), linetype="dashed",
1436     size=1, color="grey50") +
1437 106     theme_bw()
1438 107
1439 108 print(infeckernell)
1440 109 ggsave(infeckernell, filename="kernelinfec.pdf", height=3, width
1441     =3.25)
1442 110
1443 111 exfit ← extract(fit, permuted = FALSE, inc_warmup = FALSE)
1444 112 plotdata ← melt(exfit[,,"R0"])
1445 113 tracefitR0 ← ggplot() +
1446 114     geom_line(data = plotdata,
1447 115     aes(x = iterations,
1448 116     y = value,

```

```

117         color = factor(chains, labels = 1:stan_      1449
                        options$chains))) +          1450
118     labs(x = "Sample", y = expression(R[0]), color = "  1451
           Chain") +                                1452
119     scale_color_brewer(palette="Greys") +          1453
120     theme_bw()                                     1454
121                                                    1455
122 print(tracefitR0)                                  1456
123 ggsave(tracefitR0, filename="traceplotR0.pdf", height=4, width=6.5) 1457
124                                                    1458
125 exfit ← extract(fit, permuted = FALSE, inc_warmup = TRUE) 1459
126 plotdata ← melt(exfit[,,"R0"])                    1460
127 tracefitR0 ← ggplot() +                            1461
128     geom_line(data = plotdata,                      1462
129               aes(x = iterations,                   1463
130                   y = value,                        1464
131                   color = factor(chains, labels = 1:stan_ 1465
                                   options$chains))) + 1466
132     labs(x = "Sample", y = expression(R[0]), color = "  1467
           Chain") +                                1468
133     scale_color_brewer(palette="Greys") +          1469
134     theme_bw()                                     1470
135                                                    1471
136 print(tracefitR0)                                  1472
137 ggsave(tracefitR0, filename="traceplotR0_inc.pdf", height=4, width  1473
           =6.5)                                     1474
138                                                    1475
139 sso ← as.shinystan(fit)                            1476
140 sso ← launch_shinystan(sso)                        1477

```

A.2 Full Stan code

1479

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

1480

```

1  ## Dexter Barrows      1481
2  ## McMaster University 1482
3  ## 2016                1483
4                          1484
5  data {                 1485
6                          1486
7      int      <lower=1>  T;    // total integration steps 1487
8      real     y[T];       // observed number of cases      1488
9      int      <lower=1>  N;    // population size           1489
10     real     h;          // step size                      1490
11                                     1491
12 }                             1492
13                               1493
                               1494

```

```

1495 14 parameters {
1496 15
1497 16     real <lower=0, upper=10>    R0;      // R0
1498 17     real <lower=0, upper=10>    r;       // recovery rate
1499 18     real <lower=0, upper=20>    sigma;   // observation error
1500 19     real <lower=0, upper=500>    y0[3];  // initial conditions
1501 20
1502 21 }
1503 22
1504 23 model {
1505 24
1506 25     real S[T];
1507 26     real I[T];
1508 27     real R[T];
1509 28
1510 29     S[1] <- y0[1];
1511 30     I[1] <- y0[2];
1512 31     R[1] <- y0[3];
1513 32
1514 33     y[1] ~ normal(y0[2], sigma);
1515 34
1516 35     for (t in 2:T) {
1517 36
1518 37         S[t] <- S[t-1] + h*( - S[t-1]*I[t-1]*R0*r/N );
1519 38         I[t] <- I[t-1] + h*( S[t-1]*I[t-1]*R0*r/N - I[t-1]*r );
1520 39         R[t] <- R[t-1] + h*( I[t-1]*r );
1521 40
1522 41         if (y[t] > 0) {
1523 42             y[t] ~ normal( I[t], sigma );
1524 43         }
1525 44
1526 45     }
1527 46
1528 47     y0[1] ~ normal(N - y[1], sigma);
1529 48     y0[2] ~ normal(y[1], sigma);
1530 49
1531 50     R0 ~ lognormal(1,1);
1532 51     r ~ lognormal(1,1);
1533 52     sigma ~ lognormal(1,1);
1534 53
1535 54 }

```


Appendix B

1537

Iterated Filtering

1538

B.1 Full R code

1539

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

1540

```
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
```

1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569

```

1570 29 N      ← 500
1571 30 sigma  ← 10
1572 31 i_infec ← 5
1573 32
1574 33 ## Generate true trajecory and synthetic data
1575 34 ##
1576 35
1577 36 true_init_cond ← c(S = N - i_infec,
1578 37                  I = i_infec,
1579 38                  R = 0)
1580 39
1581 40 true_pars ← c(R0 = 3.0,
1582 41              r = 0.1,
1583 42              N = 500.0)
1584 43
1585 44 odeout ← ode(true_init_cond, 0:T, SIR, true_pars)
1586 45 trueTraj ← odeout[,3]
1587 46
1588 47 set.seed(1001)
1589 48
1590 49 infec_counts_raw ← odeout[,3] + rnorm(T+1, 0, sigma)
1591 50 infec_counts     ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
1592 51
1593 52 g ← qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)",
1594 53          ylab = "Infection Count") +
1595 54          geom_point(aes(y = infec_counts)) +
1596 55          theme_bw()
1597 56
1598 56 print(g)
1599 57 ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
1600 58
1601 59 ## Rcpp stuff
1602 60 ##
1603 61
1604 62 sourceCpp(paste(getwd(),"d_if2.cpp",sep="/"))
1605 63
1606 64 paramdata ← data.frame(if2(infec_counts, T+1, N))
1607 65 colnames(paramdata) ← c("R0", "r", "sigma", "Sinit", "Iinit", "Rinit",
1608 66                        ")")
1609 67
1610 67 ## Parameter density kernels
1611 68 ##
1612 69
1613 70 R0points ← paramdata$R0
1614 71 R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0])
1615 72              , ylab = "frequency") +
1616 73              geom_vline(aes(xintercept=true_pars[["R0"]]), linetype="
1617 74              dashed", size=1, color="grey50") +
1618 75              theme_bw()
1619 76

```

```

75 print(R0kernel) 1620
76 ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25) 1621
77 1622
78 rpoints ← paramdata$r 1623
79 rkernell ← qplot(rpoints, geom = "density", xlab = "r", ylab = " 1624
    frequency") + 1625
80     geom_vline(aes(xintercept=true_pars[["r"]]), linetype=" 1626
        dashed", size=1, color="grey50") + 1627
81     theme_bw() 1628
82 1629
83 print(rkernell) 1630
84 ggsave(rkernell, filename="kernelr.pdf", height=3, width=3.25) 1631
85 1632
86 sigmapoints ← paramdata$sigma 1633
87 sigmakernell ← qplot(sigmapoints, geom = "density", xlab = expression 1634
    (sigma), ylab = "frequency") + 1635
88     geom_vline(aes(xintercept=sigma), linetype="dashed", size=1, 1636
        color="grey50") + 1637
89     theme_bw() 1638
90 1639
91 print(sigmakernell) 1640
92 ggsave(sigmakernell, filename="kernelsigma.pdf", height=3, width 1641
    =3.25) 1642
93 1643
94 infecpoints ← paramdata$Iinit 1644
95 infeckernell ← qplot(infecpoints, geom = "density", xlab = "Initial 1645
    Infected", ylab = "frequency") + 1646
96     geom_vline(aes(xintercept=true_init_cond[["I"]]), linetype=" 1647
        dashed", size=1, color="grey50") + 1648
97     theme_bw() 1649
98 1650
99 print(infeckernell) 1651
100 ggsave(infeckernell, filename="kernelinfec.pdf", height=3, width 1652
    =3.25) 1653
101 1654
102 # show grid 1655
103 grid.arrange(R0kernel, rkernell, sigmakernell, infeckernell, ncol = 2, 1656
    nrow = 2) 1657
104 1658
105 pdf("if2kernels.pdf", height = 6.5, width = 6.5) 1659
106 grid.arrange(R0kernel, rkernell, sigmakernell, infeckernell, ncol = 2, 1660
    nrow = 2) 1661
107 dev.off() 1662
108 #ggsave(filename="if2kernels.pdf", g2, height=6.5, width=6.5) 1663

```

B.2 Full C++ code

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

```

1667 1 /* Author: Dexter Barrows
1668 2   Github: dbarrows.github.io
1669 3
1670 4   */
1671 5
1672 6 #include <stdio.h>
1673 7 #include <math.h>
1674 8 #include <sys/time.h>
1675 9 #include <time.h>
1676 10 #include <stdlib.h>
1677 11 #include <string>
1678 12 #include <cmath>
1679 13 #include <cstdlib>
1680 14 #include <fstream>
1681 15
1682 16 // #include "rand.h"
1683 17 // #include "timer.h"
1684 18
1685 19 #define Treal    100           // time to simulate over
1686 20 #define R0true   3.0           // infectiousness
1687 21 #define rtrue    0.1           // recovery rate
1688 22 #define Nreal    500.0        // population size
1689 23 #define merr     10.0          // expected measurement error
1690 24 #define I0       5.0           // Initial infected individuals
1691 25
1692 26 #include <Rcpp.h>
1693 27 using namespace Rcpp;
1694 28
1695 29
1696 30 struct Particle {
1697 31     double R0;
1698 32     double r;
1699 33     double sigma;
1700 34     double S;
1701 35     double I;
1702 36     double R;
1703 37     double Sinit;
1704 38     double Iinit;
1705 39     double Rinit;
1706 40 };
1707 41
1708 42 struct ParticleInfo {
1709 43     double R0mean;    double R0sd;
1710 44     double rmean;     double rsd;
1711 45     double sigmamean; double sigmasd;
1712 46     double Sinitmean; double Sinitsd;

```

```

47     double Iinitmean;    double Iinitstd;      1714
48     double Rinitmean;    double Rinitstd;      1715
49 };                                             1716
50                                             1717
51                                             1718
52 int timeval_subtract (double *result, struct timeval *x, struct  1719
    timeval *y);                                1720
53 int check_double(double x,double y);          1721
54 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *  1722
    particle);                                  1723
55 void copyParticle(Particle * dst, Particle * src);          1724
56 void perturbParticles(Particle * particles, int N, int NP, int  1725
    passnum, double coolrate);                  1726
57 bool isCollapsed(Particle * particles, int NP);             1727
58 void particleDiagnostics(ParticleInfo * partInfo, Particle *  1728
    particles, int NP);                           1729
59 NumericMatrix if2(NumericVector * data, int T, int N);      1730
60 double randu();                                             1731
61 double randn();                                             1732
62                                             1733
63 // [[Rcpp::export]]                                       1734
64 NumericMatrix if2(NumericVector data, int T, int N) {      1735
65                                             1736
66     int      NP          = 2500;                      1737
67     int      nPasses     = 50;                        1738
68     double   coolrate    = 0.975;                    1739
69                                             1740
70     int      i_infec     = I0;                        1741
71                                             1742
72     NumericMatrix paramdata(NP, 6);                   1743
73                                             1744
74     srand(time(NULL)); // Seed PRNG with system time      1745
75                                             1746
76     double w[NP]; // particle weights                    1747
77                                             1748
78     Particle particles[NP]; // particle estimates for current  1749
        step                                             1750
79     Particle particles_old[NP]; // intermediate particle states for  1751
        resampling                                       1752
80                                             1753
81     printf("Initializing particle states\n");           1754
82                                             1755
83     // initialize particle parameter states (seeding)      1756
84     for (int n = 0; n < NP; n++) {                    1757
85                                             1758
86         double R0can, rcan, sigmaican, Iinitcan;        1759
87                                             1760
88         do {                                             1761
89             R0can = R0true + R0true*randn();             1762
90         } while (R0can < 0);                             1763

```

```

1764 91         particles[n].R0 = R0can;
1765 92
1766 93     do {
1767 94         rcan = rtrue + rtrue*randn();
1768 95     } while (rcan < 0);
1769 96     particles[n].r = rcan;
1770 97
1771 98     do {
1772 99         sigmacan = merr + merr*randn();
1773 100     } while (sigmacan < 0);
1774 101     particles[n].sigma = sigmacan;
1775 102
1776 103     do {
1777 104         Iinitcan = i_infec + i_infec*randn();
1778 105     } while (Iinitcan < 0 || N < Iinitcan);
1779 106     particles[n].Sinit = N - Iinitcan;
1780 107     particles[n].Iinit = Iinitcan;
1781 108     particles[n].Rinit = 0.0;
1782 109
1783 110 }
1784 111
1785 112 // START PASSES THROUGH DATA
1786 113
1787 114 printf("Starting filter\n");
1788 115 printf("-----\n");
1789 116 printf("Pass\n");
1790 117
1791 118
1792 119 for (int pass = 0; pass < nPasses; pass++) {
1793 120
1794 121     printf("...%d / %d\n", pass, nPasses);
1795 122
1796 123     perturbParticles(particles, N, NP, pass, coolrate);
1797 124
1798 125     // initialize particle system states
1799 126     for (int n = 0; n < NP; n++) {
1800 127
1801 128         particles[n].S = particles[n].Sinit;
1802 129         particles[n].I = particles[n].Iinit;
1803 130         particles[n].R = particles[n].Rinit;
1804 131
1805 132     }
1806 133
1807 134     // between-pass perturbations
1808 135
1809 136     for (int t = 1; t < T; t++) {
1810 137
1811 138         // between-iteration perturbations
1812 139         perturbParticles(particles, N, NP, pass, coolrate);
1813 140

```

```

141 // generate individual predictions and weight
142 for (int n = 0; n < NP; n++) {
143
144     exp_euler_SIR(1.0/10.0, 0.0, 1.0, N, &particles[n]);
145
146     double merr_par = particles[n].sigma;
147     double y_diff    = data[t] - particles[n].I;
148
149     w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff
150         *y_diff / (2.0*merr_par*merr_par) );
151
152 }
153
154 // cumulative sum
155 for (int n = 1; n < NP; n++) {
156     w[n] += w[n-1];
157 }
158
159 // save particle states to resample from
160 for (int n = 0; n < NP; n++){
161     copyParticle(&particles_old[n], &particles[n]);
162 }
163
164 // resampling
165 for (int n = 0; n < NP; n++) {
166
167     double w_r = randu() * w[NP-1];
168     int i = 0;
169     while (w_r > w[i]) {
170         i++;
171     }
172
173     // i is now the index to copy state from
174     copyParticle(&particles[n], &particles_old[i]);
175
176 }
177
178 }
179
180 ParticleInfo pInfo;
181 particleDiagnostics(&pInfo, particles, NP);
182
183 printf("Parameter results (mean | sd)\n");
184 printf("-----\n");
185 printf("R0      %f %f\n", pInfo.R0mean, pInfo.R0sd);
186 printf("r       %f %f\n", pInfo.rmean, pInfo.rsd);
187 printf("sigma    %f %f\n", pInfo.sigamean, pInfo.sigmasd);
188 printf("S_init   %f %f\n", pInfo.Sinitmean, pInfo.Sinitd);
189

```

```

1864 190     printf("I_init      %f %f\n", pInfo.Iinitmean, pInfo.Iinitstd);
1865 191     printf("R_init      %f %f\n", pInfo.Rinitmean, pInfo.Rinitstd);
1866 192
1867 193     printf("\n");
1868 194
1869 195
1870 196
1871 197     // Get particle results to pass back to R
1872 198
1873 199     for (int n = 0; n < NP; n++) {
1874 200
1875 201         paramdata(n, 0) = particles[n].R0;
1876 202         paramdata(n, 1) = particles[n].r;
1877 203         paramdata(n, 2) = particles[n].sigma;
1878 204         paramdata(n, 3) = particles[n].Sinit;
1879 205         paramdata(n, 4) = particles[n].Iinit;
1880 206         paramdata(n, 5) = particles[n].Rinit;
1881 207
1882 208     }
1883 209
1884 210     return paramdata;
1885 211
1886 212 }
1887 213
1888 214
1889 215 /* Use the Explicit Euler integration scheme to integrate SIR model
1890     forward in time
1891 216     double h      - time step size
1892 217     double t0     - start time
1893 218     double tn     - stop time
1894 219     double * y    - current system state; a three-component vector
1895                     representing [S I R], susceptible-infected-recovered
1896 220
1897 221     */
1898 222 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
1899     particle) {
1900 223
1901 224     int num_steps = floor( (tn-t0) / h );
1902 225
1903 226     double S = particle->S;
1904 227     double I = particle->I;
1905 228     double R = particle->R;
1906 229
1907 230     double R0    = particle->R0;
1908 231     double r     = particle->r;
1909 232     double B     = R0 * r / N;
1910 233
1911 234     for(int i = 0; i < num_steps; i++) {
1912 235         // get derivatives
1913 236         double dS = - B*S*I;

```



```

237     double dI = B*S*I - r*I;
238     double dR = r*I;
239     // step forward by h
240     S += h*dS;
241     I += h*dI;
242     R += h*dR;
243 }
244
245 particle->S = S;
246 particle->I = I;
247 particle->R = R;
248
249 }
250
251
252 /* Particle pertubation function to be run between iterations and
253    passes
254    */
255 void perturbParticles(Particle * particles, int N, int NP, int
256    passnum, double coolrate) {
257     double coolcoef = pow(coolrate, passnum);
258
259     double spreadR0      = coolcoef * R0true / 10.0;
260     double spreadr       = coolcoef * rtrue  / 10.0;
261     double spreadsigma   = coolcoef * merr   / 10.0;
262     double spreadIinit   = coolcoef * I0     / 10.0;
263
264     double R0can, rcan, sigmacan, Iinitcan;
265
266     for (int n = 0; n < NP; n++) {
267         do {
268             R0can = particles[n].R0 + spreadR0*randn();
269         } while (R0can < 0);
270         particles[n].R0 = R0can;
271
272         do {
273             rcan = particles[n].r + spreadr*randn();
274         } while (rcan < 0);
275         particles[n].r = rcan;
276
277         do {
278             sigmacan = particles[n].sigma + spreadsigma*randn();
279         } while (sigmacan < 0);
280         particles[n].sigma = sigmacan;
281
282         do {
283             Iinitcan = particles[n].Iinit + spreadIinit*randn();
284

```

```

1964 285         } while (Iinitcan < 0 || Iinitcan > 500);
1965 286         particles[n].Iinit = Iinitcan;
1966 287         particles[n].Sinit = N - Iinitcan;
1967 288
1968 289     }
1969 290
1970 291 }
1971 292
1972 293
1973 294 /* Convenience function for particle resampling process
1974 295
1975 296     */
1976 297 void copyParticle(Particle * dst, Particle * src) {
1977 298
1978 299     dst->R0      = src->R0;
1979 300     dst->r        = src->r;
1980 301     dst->sigma    = src->sigma;
1981 302     dst->S        = src->S;
1982 303     dst->I        = src->I;
1983 304     dst->R        = src->R;
1984 305     dst->Sinit    = src->Sinit;
1985 306     dst->Iinit    = src->Iinit;
1986 307     dst->Rinit    = src->Rinit;
1987 308
1988 309 }
1989 310
1990 311
1991 312 /* Checks to see if particles are collapsed
1992 313     This is done by checking if the standard deviations between the
1993     particles' parameter
1994 314     values are significantly close to one another. Spread threshold
1995     may need to be tuned.
1996 315
1997 316     */
1998 317 bool isCollapsed(Particle * particles, int NP) {
1999 318
2000 319     bool retVal;
2001 320
2002 321     double R0mean = 0, rmean = 0, sigmamean = 0, Sinitmean = 0,
2003         Iinitmean = 0, Rinitmean = 0;
2004 322
2005 323     // means
2006 324
2007 325     for (int n = 0; n < NP; n++) {
2008 326
2009 327         R0mean      += particles[n].R0;
2010 328         rmean       += particles[n].r;
2011 329         sigmamean   += particles[n].sigma;
2012 330         Sinitmean   += particles[n].Sinit;
2013 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, Sinitd = 0, Iinitd =
344         0, Rinitd = 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         Sinitd += ( particles[n].Sinit - Sinitmean ) * ( particles[
354             n].Sinit - Sinitmean );
355         Iinitd += ( particles[n].Iinit - Iinitmean ) * ( particles[
356             n].Iinit - Iinitmean );
357         Rinitd += ( particles[n].Rinit - Rinitmean ) * ( particles[
358             n].Rinit - Rinitmean );
359
360     }
361
362     R0sd    /= NP;
363     rsd     /= NP;
364     sigmasd /= NP;
365     Sinitd  /= NP;
366     Iinitd  /= NP;
367     Rinitd  /= 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

```

```

2064 374     double  R0mean      = 0.0,
2065 375           rmean      = 0.0,
2066 376           sigmamean  = 0.0,
2067 377           Sinitmean   = 0.0,
2068 378           Iinitmean   = 0.0,
2069 379           Rinitmean   = 0.0;
2070 380
2071 381     // means
2072 382
2073 383     for (int n = 0; n < NP; n++) {
2074 384
2075 385         R0mean      += particles[n].R0;
2076 386         rmean      += particles[n].r;
2077 387         sigmamean  += particles[n].sigma;
2078 388         Sinitmean   += particles[n].Sinit;
2079 389         Iinitmean   += particles[n].Iinit;
2080 390         Rinitmean   += particles[n].Rinit;
2081 391
2082 392     }
2083 393
2084 394     R0mean      /= NP;
2085 395     rmean      /= NP;
2086 396     sigmamean  /= NP;
2087 397     Sinitmean   /= NP;
2088 398     Iinitmean   /= NP;
2089 399     Rinitmean   /= NP;
2090 400
2091 401     // standard deviations
2092 402
2093 403     double  R0sd      = 0.0,
2094 404           rsd        = 0.0,
2095 405           sigmasd    = 0.0,
2096 406           Sinitsd    = 0.0,
2097 407           Iinitsd    = 0.0,
2098 408           Rinitsd    = 0.0;
2099 409
2100 410     for (int n = 0; n < NP; n++) {
2101 411
2102 412         R0sd      += ( particles[n].R0 - R0mean ) * ( particles[n].R0
2103 413                 - R0mean );
2104 414         rsd      += ( particles[n].r - rmean ) * ( particles[n].r -
2105 415                 rmean );
2106 416         sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[
2107 417                 n].sigma - sigmamean );
2108 418         Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[
2109 419                 n].Sinit - Sinitmean );
2110 420         Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[
2111 421                 n].Iinit - Iinitmean );
2112 422         Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[
2113 423                 n].Rinit - Rinitmean );

```

418		2114
419	}	2115
420		2116
421	R0sd /= NP;	2117
422	rsd /= NP;	2118
423	sigmasd /= NP;	2119
424	Sinit_sd /= NP;	2120
425	Iinit_sd /= NP;	2121
426	Rinit_sd /= NP;	2122
427		2123
428	partInfo->R0mean = R0mean;	2124
429	partInfo->R0sd = R0sd;	2125
430	partInfo->sigmamean = sigmamean;	2126
431	partInfo->sigmasd = sigmasd;	2127
432	partInfo->rmean = rmean;	2128
433	partInfo->rsd = rsd;	2129
434	partInfo->Sinitmean = Sinitmean;	2130
435	partInfo->Sinit_sd = Sinit_sd;	2131
436	partInfo->Iinitmean = Iinitmean;	2132
437	partInfo->Iinit_sd = Iinit_sd;	2133
438	partInfo->Rinitmean = Rinitmean;	2134
439	partInfo->Rinit_sd = Rinit_sd;	2135
440		2136
441	}	2137
442		2138
443	double randu() {	2139
444		2140
445	return (double) rand() / (double) RAND_MAX;	2141
446		2142
447	}	2143
448		2144
449		2145
450	/* Return a normally distributed random number with mean 0 and	2146
	standard deviation 1	2147
451	Uses the polar form of the Box-Muller transformation	2148
452	From http://www.design.caltech.edu/erik/Misc/Gaussian.html	2149
453	*/	2150
454	double randn() {	2151
455		2152
456	double x1, x2, w, y1;	2153
457		2154
458	do {	2155
459	x1 = 2.0 * randu() - 1.0;	2156
460	x2 = 2.0 * randu() - 1.0;	2157
461	w = x1 * x1 + x2 * x2;	2158
462	} while (w >= 1.0);	2159
463		2160
464	w = sqrt((-2.0 * log(w)) / w);	2161
465	y1 = x1 * w;	2162
466		2163

```
2164 467 |     return y1;  
2165 468 |  
2166 469 | }
```

Appendix C

2168

Parameter Fitting

2169

Appendix D

Forecasting Frameworks

D.1 IF2 Parametric Bootstrapping Function

The parametric bootstrapping machinery used to produce forecasts.

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



```

26 statemeans_parent ← colMeans(statedata_parent) 2203
27 names(statemeans_parent) ← c("S", "I", "R", "B") 2204
28 2205
29 2206
30 ## use parametric bootstrapping to generate forecasts 2207
31 ## 2208
32 trajectories ← foreach( i = 1:nTrials, .combine = rbind, .packages 2209
    = "Rcpp") %dopar% { 2210
33 2211
34   source(stoc_sir_file) 2212
35 2213
36   ## draw new data 2214
37   ## 2215
38 2216
39   pars ← with( as.list(parmmeans_parent), 2217
40               c(R0 = R0, 2218
41                 r = r, 2219
42                 N = N, 2220
43                 eta = eta, 2221
44                 berr = berr) ) 2222
45 2223
46   init_cond ← with( as.list(parmmeans_parent), 2224
47                    c(S = Sinit, 2225
48                      I = Iinit, 2226
49                      R = Rinit) ) 2227
50 2228
51   # generate trajectory 2229
52   sdeout ← StocSIR(init_cond, pars, Tlim + 1, steps) 2230
53   colnames(sdeout) ← c('S', 'I', 'R', 'B') 2231
54 2232
55   # add noise 2233
56   counts_raw ← sdeout[, 'I'] + rnorm(dim(sdeout)[1], 0, parmeans_ 2234
    parent[['sigma']]) 2235
57   counts ← ifelse(counts_raw < 0, 0, counts_raw) 2236
58 2237
59   ## refit using new data 2238
60   ## 2239
61 2240
62   rm(if2) # because stupid things get done in packages 2241
63   sourceCpp(if2file) 2242
64   if2time ← system.time( if2data ← if2(counts, Tlim+1, N, NP, 2243
    nPasses, coolrate) ) 2244
65 2245
66   paramdata ← data.frame( if2data$paramdata ) 2246
67   names(paramdata) ← c("R0", "r", "sigma", "eta", "berr", "Sinit", 2247
    "Iinit", "Rinit") 2248
68   parmeans ← colMeans(paramdata) 2249
69   names(parmeans) ← c("R0", "r", "sigma", "eta", "berr", "Sinit", 2250
    "Iinit", "Rinit") 2251
70 2252

```

```

2253 71  ## generate the rest of the trajectory
2254 72  ##
2255 73
2256 74  # pack new parameter estimates
2257 75  pars ← with( as.list(parmmeans),
2258 76              c(R0 = R0,
2259 77                r = r,
2260 78                N = N,
2261 79                eta = eta,
2262 80                berr = berr) )
2263 81  init_cond ← c(S = statemeans_parent[['S']],
2264 82              I = statemeans_parent[['I']],
2265 83              R = statemeans_parent[['R']])
2266 84
2267 85  # generate remaining trajectory part
2268 86  sdeout_future ← StocSIR(init_cond, pars, T-Tlim, steps)
2269 87  colnames(sdeout_future) ← c('S','I','R','B')
2270 88
2271 89  return ( c( counts = unname(sdeout_future[['I']]),
2272 90            parmmeans,
2273 91            time = if2time[['user.self']] ) )
2274 92
2275 93
2276 94  }
2277 95
2278 96  return(trajectories)
2279 97
2280 98  }

```

2282 D.2 RStan Forward Simulator

2283 The code used to reconstruct the state estimates, then project the trajectory forward
 2284 past data.

```

2285 1 StocSIRstan ← function(y, pars, T, steps, berrvec, bveclim) {
2286 2
2287 3   out ← matrix(NA, nrow = (T+1), ncol = 4)
2288 4
2289 5   R0 ← pars[['R0']]
2290 6   r ← pars[['r']]
2291 7   N ← pars[['N']]
2292 8   eta ← pars[['eta']]
2293 9   berr ← pars[['berr']]
2294 10
2295 11  S ← y[['S']]
2296 12  I ← y[['I']]
2297 13  R ← y[['R']]

```

14		2299
15	$B_0 \leftarrow R_0 * r / N$	2300
16	$B \leftarrow B_0$	2301
17		2302
18	$out[1,] \leftarrow c(S, I, R, B)$	2303
19		2304
20	$h \leftarrow 1 / steps$	2305
21		2306
22	for (i in 1:(T*steps)) {	2307
23		2308
24	if (i <= bveclim) {	2309
25	$B \leftarrow \exp(\log(B) + \eta * (\log(B_0) - \log(B)) + berrvec[i])$	2310
26	} else {	2311
27	$B \leftarrow \exp(\log(B) + \eta * (\log(B_0) - \log(B)) + rnorm(1, 0,$	2312
	berr))	2313
28	}	2314
29		2315
30	$BSI \leftarrow B * S * I$	2316
31	$rI \leftarrow r * I$	2317
32		2318
33	$dS \leftarrow -BSI$	2319
34	$dI \leftarrow BSI - rI$	2320
35	$dR \leftarrow rI$	2321
36		2322
37	$S \leftarrow S + h * dS$ #newInf	2323
38	$I \leftarrow I + h * dI$ #newInf - h*dR	2324
39	$R \leftarrow R + h * dR$ #h*dR	2325
40		2326
41	if (i %% steps == 0)	2327
42	$out[i/steps+1,] \leftarrow c(S, I, R, B)$	2328
43		2329
44	}	2330
45		2331
46	return(out)	2332
47		2333
48	}	2334
		2335

2336 Appendix E

2337 S-map and SIRS

2338 E.1 SIRS R Function Code

2339 R code to simulate the outlines SIRS function.

```
2340 1 StocSIRS ← function(y, pars, T, steps) {
2341 2
2342 3   out ← matrix(NA, nrow = (T+1), ncol = 4)
2343 4
2344 5   R0 ← pars[['R0']]
2345 6   r ← pars[['r']]
2346 7   N ← pars[['N']]
2347 8   eta ← pars[['eta']]
2348 9   berr ← pars[['berr']]
2349 10  re ← pars[['re']]
2350 11
2351 12  S ← y[['S']]
2352 13  I ← y[['I']]
2353 14  R ← y[['R']]
2354 15
2355 16  B0 ← R0 * r / N
2356 17  B ← B0
2357 18
2358 19  out[1,] ← c(S,I,R,B)
2359 20
2360 21  h ← 1 / steps
2361 22
2362 23  for ( i in 1:(T*steps) ) {
2363 24
2364 25      #Bfac ← 1/2 - cos((2*pi/365)*i)/2
2365 26      Bfac ← exp(2*cos((2*pi/365)*i) - 2)
2366 27
2367 28      B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr) )
```

```

29 |                                     2369
30 |     BSI ← Bfac*B*S*I               2370
31 |     rI ← r*I                       2371
32 |     reR ← re*R                     2372
33 |                                     2373
34 |     dS ← -BSI + reR                2374
35 |     dI ← BSI - rI                  2375
36 |     dR ← rI - reR                  2376
37 |                                     2377
38 |     S ← S + h*dS #newInf           2378
39 |     I ← I + h*dI #newInf - h*dR    2379
40 |     R ← R + h*dR #h*dR             2380
41 |                                     2381
42 |     if (i %% steps == 0)           2382
43 |         out[i/steps+1,] ← c(S,I,R,B) 2383
44 |                                     2384
45 | }                                   2385
46 |                                     2386
47 | colnames(out) ← c("S","I","R","B") 2387
48 | return(out)                        2388
49 |                                     2389
50 | }                                   2390
51 |                                     2391
52 | ### suggested parameters           2392
53 | #                                   2393
54 | # T      ← 200                     2394
55 | # i_infec ← 10                     2395
56 | # steps   ← 7                     2396
57 | # N       ← 500                    2397
58 | # sigma   ← 5                     2398
59 | #                                   2399
60 | # pars ← c(R0 = 3.0, # new infected people per infected person 2400
61 | #         r = 0.1, # recovery rate                               2401
62 | #         N = 500, # population size                             2402
63 | #         eta = 0.5, # geometric random walk                    2403
64 | #         berr = 0.5, # Beta geometric walk noise               2404
65 | #         re = 1) # resuceptibility rate                         2405

```

E.2 SMAP Code

2407

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

2408

```

1 | library(pracma)                    2409
2 |                                     2410
3 | smap ← function(data, E, theta, stepsAhead) { 2411
4 |                                     2412
5 |     # construct library             2413

```

2414

```

2415 6   tseries ← as.vector(data)
2416 7   liblen ← length(tseries) - E + 1 - stepsAhead
2417 8   lib     ← matrix(NA, liblen, E)
2418 9
2419 10  for (i in 1:E) {
2420 11    lib[,i] ← tseries[(E-i+1):(liblen+E-i)]
2421 12  }
2422 13
2423 14  # predict from the last index
2424 15  tslen ← length(tseries)
2425 16  predictee ← rev(t(as.matrix(tseries[(tslen-E+1):tslen])))
2426 17  predictions ← numeric(stepsAhead)
2427 18
2428 19  #allPredictees ← matrix(NA, stepsAhead, E)
2429 20
2430 21  # for each prediction index (number of steps ahead)
2431 22  for(i in 1:stepsAhead) {
2432 23
2433 24    # set up weight calculation
2434 25    predmat ← repmat(predictee, liblen, 1)
2435 26    distances ← sqrt( rowSums( abs(lib - predmat)^2 ) )
2436 27    meanDist ← mean(distances)
2437 28
2438 29    # calculate weights
2439 30    weights ← exp( - (theta * distances) / meanDist )
2440 31
2441 32    # construct A, B
2442 33
2443 34    preds ← tseries[(E+i):(liblen+E+i-1)]
2444 35
2445 36    A ← cbind( rep(1.0, liblen), lib ) * repmat(as.matrix(
2446 37      weights), 1, E+1)
2447 38    B ← as.matrix(preds * weights)
2448 39
2449 40    # solve system for C
2450 41
2451 42    Asvd ← svd(A)
2452 43    C ← Asvd$v %*% diag(1/Asvd$d) %*% t(Asvd$u) %*% B
2453 44
2454 45    # get prediction
2455 46
2456 47    predsum ← sum(C * c(1,predictee))
2457 48
2458 49    # save
2459 50
2460 51    predictions[i] ← predsum
2461 52
2462 53    # next predictee
2463 54
2464 55    #predictee ← c( predsum, predictee[-E] )

```

```

55         #allPredicttees[i,] ← predictee
56     }
57 }
58
59     return(predictions)
60
61 }

```

E.3 SMAP Parameter Optimization Code

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

```

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

```

```

2511 33 true_init_cond ← c(S = N - i_infec,
2512 34                   I = i_infec,
2513 35                   R = 0)
2514 36
2515 37 ## trial parameter values to check.options
2516 38 ##
2517 39 Elist ← 1:20
2518 40 thetalist ← 10*exp(-(seq(0,9.5,0.5)))
2519 41 nTrials ← 100
2520 42
2521 43 ssemat ← matrix(NA, 20, 20)
2522 44
2523 45 for (i in 1:length(Elist)) {
2524 46   for (j in 1:length(thetalist)) {
2525 47
2526 48     ssemean ← 0
2527 49
2528 50     for (k in 1:nTrials) {
2529 51
2530 52       E ← Elist[i]
2531 53       theta ← thetalist[j]
2532 54
2533 55       ## get true trajectory
2534 56       ##
2535 57       sdeout ← StocSIRS(true_init_cond, true_pars, T, steps)
2536 58
2537 59       ## perturb to get data
2538 60       ##
2539 61       infec_counts_raw ← sdeout[1:(Tlim+1), 'I'] + rnorm(Tlim+1, 0,
2540 62                   sigma)
2541 62       infec_counts ← ifelse(infec_counts_raw < 0, 0, infec_counts_
2542 63       raw)
2543 63
2544 64       predictions ← smap(infec_counts, E, theta, 52)
2545 65
2546 66       err ← sdeout[(Tlim+2):dim(sdeout)[1], 'I'] - predictions
2547 67       sse ← sum(err^2)
2548 68
2549 69       ssemean ← ssemean + (sse / nTrials)
2550 70
2551 71     }
2552 72
2553 73     ssemat[i,j] ← ssemean
2554 74
2555 75   }
2556 76 }
2557 77 }
2558 78
2559 79 quartz()
2560 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

```

```

2607 23 //transformed parameters {
2608 24 //      real B0 ← R0 * r / N;
2609 25 //}
2610 26
2611 27 model {
2612 28
2613 29     real S[T];
2614 30     real I[T];
2615 31     real R[T];
2616 32     real B[T];
2617 33     real B0;
2618 34
2619 35     real pi;
2620 36     real Bfac;
2621 37
2622 38     pi ← 3.1415926535;
2623 39
2624 40     B0 ← R0 * r / N;
2625 41
2626 42     B[1] ← B0;
2627 43
2628 44     S[1] ← N - Iinit;
2629 45     I[1] ← Iinit;
2630 46     R[1] ← 0.0;
2631 47
2632 48     for (t in 2:T) {
2633 49
2634 50         Bnoise[t] ~ normal(0,berr);
2635 51         Bfac ← exp(2*cos((2*pi/365)*t) - 2);
2636 52         B[t] ← exp( log(B0) + eta * ( log(B[t-1]) - log(B0) ) +
2637         Bnoise[t] );
2638 53
2639 54         S[t] ← S[t-1] + h*( - Bfac*B[t]*S[t-1]*I[t-1] + re*R[t-1] );
2640 55         I[t] ← I[t-1] + h*( Bfac*B[t]*S[t-1]*I[t-1] - I[t-1]*r );
2641 56         R[t] ← R[t-1] + h*( I[t-1]*r - re*R[t-1] );
2642 57
2643 58         if (y[t] > 0) {
2644 59             y[t] ~ normal( I[t], sigma );
2645 60         }
2646 61
2647 62     }
2648 63
2649 64     R0      ~ lognormal(1,1);
2650 65     r       ~ lognormal(1,1);
2651 66     sigma   ~ lognormal(1,1);
2652 67     re      ~ lognormal(1,1);
2653 68     Iinit   ~ normal(y[1], sigma);
2654 69
2655 70 }

```

E.5 IF2 SIRS Code

2657

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

2658

```

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;

```

2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705

```

2706 46     double berr;
2707 47     double B;
2708 48     double S;
2709 49     double I;
2710 50     double R;
2711 51     double Sinit;
2712 52     double Iinit;
2713 53     double Rinit;
2714 54 };
2715 55
2716 56 struct ParticleInfo {
2717 57     double R0mean;      double R0sd;
2718 58     double rmean;       double rsd;
2719 59     double remean;       double resd;
2720 60     double sigmamean;    double sigmasd;
2721 61     double etamean;      double etasd;
2722 62     double berrmean;     double berrsd;
2723 63     double Sinitmean;    double Sinitsd;
2724 64     double Iinitmean;    double Iinitsd;
2725 65     double Rinitmean;    double Rinitsd;
2726 66 };
2727 67
2728 68
2729 69 int timeval_subtract (double *result, struct timeval *x, struct
2730     timeval *y);
2731 70 int check_double(double x, double y);
2732 71 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle
2733     * particle);
2734 72 void copyParticle(Particle * dst, Particle * src);
2735 73 void perturbParticles(Particle * particles, int N, int NP, int
2736     passnum, double coolrate);
2737 74 void particleDiagnostics(ParticleInfo * partInfo, Particle *
2738     particles, int NP);
2739 75 void getStateMeans(State * state, Particle* particles, int NP);
2740 76 NumericMatrix if2(NumericVector * data, int T, int N);
2741 77 double randu();
2742 78 double randn();
2743 79
2744 80 // [[Rcpp::export]]
2745 81 Rcpp::List if2_sirs(NumericVector data, int T, int N, int NP, int
2746     nPasses, double coolrate) {
2747 82
2748 83     int npar = 9;
2749 84
2750 85     NumericMatrix paramdata(NP, npar);
2751 86     NumericMatrix means(nPasses, npar);
2752 87     NumericMatrix sds(nPasses, npar);
2753 88     NumericMatrix statemeans(T, 3);
2754 89     NumericMatrix statedata(NP, 4);
2755 90

```

```

91      srand(time(NULL));          // Seed PRNG with system time      2756
92
93      double w[NP];              // particle weights                2757
94
95      Particle particles[NP];     // particle estimates for current  2758
96      step                       2759
97      Particle particles_old[NP]; // intermediate particle states for  2760
98      resampling                 2761
99
100     printf("Initializing particle states\n"); 2762
101
102     // initialize particle parameter states (seeding) 2763
103     for (int n = 0; n < NP; n++) { 2764
104
105         double R0can, rcan, recan, sigmacan, Iinitcan, etacan, 2765
106         berrcan; 2766
107
108         do { 2767
109             R0can = R0true + R0true*randn(); 2768
110             } while (R0can < 0); 2769
111         particles[n].R0 = R0can; 2770
112
113         do { 2771
114             rcan = rtrue + rtrue*randn(); 2772
115             } while (rcan < 0); 2773
116         particles[n].r = rcan; 2774
117
118         do { 2775
119             recan = retrue + retrue*randn(); 2776
120             } while (recan < 0); 2777
121         particles[n].re = recan; 2778
122
123         particles[n].B = (double) R0can * rcan / N; 2779
124
125         do { 2780
126             sigmacan = merr + merr*randn(); 2781
127             } while (sigmacan < 0); 2782
128         particles[n].sigma = sigmacan; 2783
129
130         do { 2784
131             etacan = etatrue + PSC*etatrue*randn(); 2785
132             } while (etacan < 0 || etacan > 1); 2786
133         particles[n].eta = etacan; 2787
134
135         do { 2788
136             berrcan = berrtrue + PSC*berrtrue*randn(); 2789
137             } while (berrcan < 0); 2790
138         particles[n].berr = berrcan; 2791
139
140         do { 2792

```

```

2806 138         Iinitcan = I0 + I0*randn();
2807 139     } while (Iinitcan < 0 || N < Iinitcan);
2808 140     particles[n].Sinit = N - Iinitcan;
2809 141     particles[n].Iinit = Iinitcan;
2810 142     particles[n].Rinit = 0.0;
2811 143
2812 144 }
2813 145
2814 146 // START PASSES THROUGH DATA
2815 147
2816 148 printf("Starting filter\n");
2817 149 printf("-----\n");
2818 150 printf("Pass\n");
2819 151
2820 152
2821 153 for (int pass = 0; pass < nPasses; pass++) {
2822 154
2823 155     printf("...%d / %d\n", pass, nPasses);
2824 156
2825 157     // reset particle system evolution states
2826 158     for (int n = 0; n < NP; n++) {
2827 159
2828 160         particles[n].S = particles[n].Sinit;
2829 161         particles[n].I = particles[n].Iinit;
2830 162         particles[n].R = particles[n].Rinit;
2831 163         particles[n].B = (double) particles[n].R0 * particles[n
2832 164             ].r / N;
2833 164
2834 165     }
2835 166
2836 167     if (pass == (nPasses-1)) {
2837 168         State sMeans;
2838 169         getStateMeans(&sMeans, particles, NP);
2839 170         statemeans(0,0) = sMeans.S;
2840 171         statemeans(0,1) = sMeans.I;
2841 172         statemeans(0,2) = sMeans.R;
2842 173     }
2843 174
2844 175     for (int t = 1; t < T; t++) {
2845 176
2846 177         // generate individual predictions and weight
2847 178         for (int n = 0; n < NP; n++) {
2848 179
2849 180             exp_euler_SIRS(1.0/7.0, (double) t-1, (double) t, N,
2850 181                 &particles[n]);
2851 181
2852 182             double merr_par = particles[n].sigma;
2853 183             double y_diff = data[t] - particles[n].I;
2854 184
2855 185             w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff

```

```

186         *y_diff / (2.0*merr_par*merr_par) );
187     }
188
189     // cumulative sum
190     for (int n = 1; n < NP; n++) {
191         w[n] += w[n-1];
192     }
193
194     // save particle states to resample from
195     for (int n = 0; n < NP; n++){
196         copyParticle(&particles_old[n], &particles[n]);
197     }
198
199     // resampling
200     for (int n = 0; n < NP; n++) {
201
202         double w_r = randu() * w[NP-1];
203         int i = 0;
204         while (w_r > w[i]) {
205             i++;
206         }
207
208         // i is now the index to copy state from
209         copyParticle(&particles[n], &particles_old[i]);
210
211     }
212
213     // between-iteration perturbations, not after last time
214     // step
215     if (t < (T-1))
216         perturbParticles(particles, N, NP, pass, coolrate);
217
218     if (pass == (nPasses-1)) {
219         State sMeans;
220         getStateMeans(&sMeans, particles, NP);
221         statemeans(t,0) = sMeans.S;
222         statemeans(t,1) = sMeans.I;
223         statemeans(t,2) = sMeans.R;
224     }
225 }
226
227 ParticleInfo pInfo;
228 particleDiagnostics(&pInfo, particles, NP);
229
230 means(pass, 0) = pInfo.R0mean;
231 means(pass, 1) = pInfo.rmean;
232 means(pass, 2) = pInfo.remean;
233 means(pass, 3) = pInfo.sigamean;

```

```

2906 234     means(pass, 4) = pInfo.etamean;
2907 235     means(pass, 5) = pInfo.berrmean;
2908 236     means(pass, 6) = pInfo.Sinitmean;
2909 237     means(pass, 7) = pInfo.Iinitmean;
2910 238     means(pass, 8) = pInfo.Rinitmean;
2911 239
2912 240     sds(pass, 0) = pInfo.R0sd;
2913 241     sds(pass, 1) = pInfo.rsd;
2914 242     sds(pass, 2) = pInfo.resd;
2915 243     sds(pass, 3) = pInfo.sigmasd;
2916 244     sds(pass, 4) = pInfo.etasd;
2917 245     sds(pass, 5) = pInfo.berrsd;
2918 246     sds(pass, 6) = pInfo.Sinitsd;
2919 247     sds(pass, 7) = pInfo.Iinitsd;
2920 248     sds(pass, 8) = pInfo.Rinitsd;
2921 249
2922 250     // between-pass perturbations, not after last pass
2923 251     if (pass < (nPasses + 1))
2924 252         perturbParticles(particles, N, NP, pass, coolrate);
2925 253
2926 254 }
2927 255
2928 256 ParticleInfo pInfo;
2929 257 particleDiagnostics(&pInfo, particles, NP);
2930 258
2931 259 printf("Parameter results (mean | sd)\n");
2932 260 printf("-----\n");
2933 261 printf("R0      %f %f\n", pInfo.R0mean, pInfo.R0sd);
2934 262 printf("r      %f %f\n", pInfo.rmean, pInfo.rsd);
2935 263 printf("re     %f %f\n", pInfo.remean, pInfo.resd);
2936 264 printf("sigma   %f %f\n", pInfo.sigamean, pInfo.sigmasd);
2937 265 printf("eta     %f %f\n", pInfo.etamean, pInfo.etasd);
2938 266 printf("berr    %f %f\n", pInfo.berrmean, pInfo.berrsd);
2939 267 printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
2940 268 printf("I_init   %f %f\n", pInfo.Iinitmean, pInfo.Iinitsd);
2941 269 printf("R_init   %f %f\n", pInfo.Rinitmean, pInfo.Rinitsd);
2942 270
2943 271 printf("\n");
2944 272
2945 273
2946 274
2947 275 // Get particle results to pass back to R
2948 276
2949 277 for (int n = 0; n < NP; n++) {
2950 278
2951 279     paramdata(n, 0) = particles[n].R0;
2952 280     paramdata(n, 1) = particles[n].r;
2953 281     paramdata(n, 2) = particles[n].re;
2954 282     paramdata(n, 3) = particles[n].sigma;
2955 283     paramdata(n, 4) = particles[n].eta;

```



```

284     paramdata(n, 5) = particles[n].berr;           2956
285     paramdata(n, 6) = particles[n].Sinit;         2957
286     paramdata(n, 7) = particles[n].Iinit;         2958
287     paramdata(n, 8) = particles[n].Rinit;         2959
288                                                     2960
289 }                                                  2961
290                                                     2962
291 for (int n = 0; n < NP; n++) {                    2963
292                                                     2964
293     statedata(n, 0) = particles[n].S;             2965
294     statedata(n, 1) = particles[n].I;             2966
295     statedata(n, 2) = particles[n].R;             2967
296     statedata(n, 3) = particles[n].B;             2968
297                                                     2969
298 }                                                  2970
299                                                     2971
300                                                     2972
301                                                     2973
302     return Rcpp::List::create( Rcpp::Named("paramdata") = paramdata 2974
303                                ,                                     2975
304                                Rcpp::Named("means") = means,         2976
305                                Rcpp::Named("statemeans") =           2977
306                                    statemeans,                       2978
307                                Rcpp::Named("statedata") = statedata   2979
308                                ,                                     2980
309                                Rcpp::Named("sds") = sds);             2981
308 }                                                  2982
309                                                     2983
310                                                     2984
311 /* Use the Explicit Euler integration scheme to integrate SIR model 2985
312    forward in time
313    double h      - time step size
314    double t0     - start time
315    double tn     - stop time
316    double * y    - current system state; a three-component vector
317                   representing [S I R], susceptible-infected-recovered 2992
318 */                                                  2993
317 */                                                  2994
318 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle 2995
319 * particle) {                                     2996
319                                                     2997
320     int num_steps = floor( (tn-t0) / h );          2998
321                                                     2999
322     double S = particle->S;                         3000
323     double I = particle->I;                         3001
324     double R = particle->R;                         3002
325                                                     3003
326     double R0   = particle->R0;                     3004
327     double r    = particle->r;                     3005

```

```

3006 328     double re    = particle->re;
3007 329     double B0    = R0 * r / N;
3008 330     double eta    = particle->eta;
3009 331     double berr   = particle->berr;
3010 332
3011 333     double B = particle->B;
3012 334
3013 335     for(int i = 0; i < num_steps; i++) {
3014 336
3015 337         //double Bfac = 0.5 - 0.95*cos( (2.0*M_PI/365)*(t0*num_steps
3016         +i) )/2.0;
3017 338         double Bfac = exp(2*cos((2*M_PI/365)*(t0*num_steps+i)) - 2);
3018 339         B = exp( log(B) + eta*(log(B0) - log(B)) + berr*randn() );
3019 340
3020 341         double BSI = Bfac*B*S*I;
3021 342         double rI  = r*I;
3022 343         double reR = re*R;
3023 344
3024 345         // get derivatives
3025 346         double dS = - BSI + reR;
3026 347         double dI = BSI - rI;
3027 348         double dR = rI - reR;
3028 349
3029 350         // step forward by h
3030 351         S += h*dS;
3031 352         I += h*dI;
3032 353         R += h*dR;
3033 354
3034 355     }
3035 356
3036 357     particle->S = S;
3037 358     particle->I = I;
3038 359     particle->R = R;
3039 360     particle->B = B;
3040 361
3041 362 }
3042 363
3043 364
3044 365 /* Particle pertubation function to be run between iterations and
3045     passes
3046 366
3047 367     */
3048 368 void perturbParticles(Particle * particles, int N, int NP, int
3049     passnum, double coolrate) {
3050 369
3051 370     //double coolcoef = exp( - (double) passnum / coolrate );
3052 371     double coolcoef = pow(coolrate, passnum);
3053 372
3054 373
3055 374     double spreadR0      = coolcoef * R0true / 10.0;

```

```

375 double spreadr      = coolcoef * rtrue / 10.0;      3056
376 double spreadre     = coolcoef * retrue / 10.0;     3057
377 double spreadsigma  = coolcoef * merr / 10.0;       3058
378 double spreadIinit  = coolcoef * I0 / 10.0;         3059
379 double spreadeta    = coolcoef * etatrue / 10.0;    3060
380 double spreadberr   = coolcoef * berrtrue / 10.0;   3061
381                                     3062
382                                     3063
383 double R0can, rcan, recan, sigmacan, Iinitcan, etacan, berrcan; 3064
384                                     3065
385 for (int n = 0; n < NP; n++) { 3066
386                                     3067
387     do { 3068
388         R0can = particles[n].R0 + spreadR0*randn(); 3069
389     } while (R0can < 0); 3070
390     particles[n].R0 = R0can; 3071
391                                     3072
392     do { 3073
393         rcan = particles[n].r + spreadr*randn(); 3074
394     } while (rcan < 0); 3075
395     particles[n].r = rcan; 3076
396                                     3077
397     do { 3078
398         recan = particles[n].re + spreadre*randn(); 3079
399     } while (recan < 0); 3080
400     particles[n].re = recan; 3081
401                                     3082
402     do { 3083
403         sigmacan = particles[n].sigma + spreadsigma*randn(); 3084
404     } while (sigmacan < 0); 3085
405     particles[n].sigma = sigmacan; 3086
406                                     3087
407     do { 3088
408         etacan = particles[n].eta + PSC*spreadeta*randn(); 3089
409     } while (etacan < 0 || etacan > 1); 3090
410     particles[n].eta = etacan; 3091
411                                     3092
412     do { 3093
413         berrcan = particles[n].berr + PSC*spreadberr*randn(); 3094
414     } while (berrcan < 0); 3095
415     particles[n].berr = berrcan; 3096
416                                     3097
417     do { 3098
418         Iinitcan = particles[n].Iinit + spreadIinit*randn(); 3099
419     } while (Iinitcan < 0 || Iinitcan > 500); 3100
420     particles[n].Iinit = Iinitcan; 3101
421     particles[n].Sinit = N - Iinitcan; 3102
422                                     3103
423 } 3104
424                                     3105

```

```

3106 425 }
3107 426
3108 427
3109 428 /* Convenience function for particle resampling process
3110 429
3111 430 */
3112 431 void copyParticle(Particle * dst, Particle * src) {
3113 432
3114 433     dst->R0      = src->R0;
3115 434     dst->r        = src->r;
3116 435     dst->re       = src->re;
3117 436     dst->sigma    = src->sigma;
3118 437     dst->eta      = src->eta;
3119 438     dst->berr     = src->berr;
3120 439     dst->B        = src->B;
3121 440     dst->S        = src->S;
3122 441     dst->I        = src->I;
3123 442     dst->R        = src->R;
3124 443     dst->Sinit    = src->Sinit;
3125 444     dst->Iinit    = src->Iinit;
3126 445     dst->Rinit    = src->Rinit;
3127 446
3128 447 }
3129 448
3130 449 void particleDiagnostics(ParticleInfo * partInfo, Particle *
3131 450 particles, int NP) {
3132 451
3133 451     double   R0mean      = 0.0,
3134 452             rmean       = 0.0,
3135 453             remean      = 0.0,
3136 454             sigmamean   = 0.0,
3137 455             etamean     = 0.0,
3138 456             berrmean    = 0.0,
3139 457             Sinitmean   = 0.0,
3140 458             Iinitmean   = 0.0,
3141 459             Rinitmean   = 0.0;
3142 460
3143 461     // means
3144 462
3145 463     for (int n = 0; n < NP; n++) {
3146 464
3147 465         R0mean      += particles[n].R0;
3148 466         rmean       += particles[n].r;
3149 467         remean      += particles[n].re;
3150 468         etamean     += particles[n].eta;
3151 469         berrmean    += particles[n].berr;
3152 470         sigmamean   += particles[n].sigma;
3153 471         Sinitmean   += particles[n].Sinit;
3154 472         Iinitmean   += particles[n].Iinit;
3155 473         Rinitmean   += particles[n].Rinit;

```

```

474 |
475 | }
476 |
477 | R0mean      /= NP;
478 | rmean       /= NP;
479 | remean      /= NP;
480 | sigmamean   /= NP;
481 | etamean     /= NP;
482 | berrmean    /= NP;
483 | Sinitmean   /= NP;
484 | Iinitmean   /= NP;
485 | Rinitmean   /= NP;
486 |
487 | // standard deviations
488 |
489 | double R0sd   = 0.0,
490 |        rsd    = 0.0,
491 |        resd   = 0.0,
492 |        sigmasd = 0.0,
493 |        etasd  = 0.0,
494 |        berrsd = 0.0,
495 |        Sinitsd = 0.0,
496 |        Iinitsd = 0.0,
497 |        Rinitsd = 0.0;
498 |
499 | for (int n = 0; n < NP; n++) {
500 |
501 |     R0sd += ( particles[n].R0 - R0mean ) * ( particles[n].R0
502 |           - R0mean );
503 |     rsd  += ( particles[n].r - rmean ) * ( particles[n].r -
504 |           rmean );
505 |     resd += ( particles[n].re - rmean ) * ( particles[n].re -
506 |           rmean );
507 |     sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[
508 |           n].sigma - sigmamean );
509 |     etasd  += ( particles[n].eta - etamean ) * ( particles[n].
510 |           eta - etamean );
511 |     berrsd += ( particles[n].berr - berrmean ) * ( particles[n
512 |           ].berr - berrmean );
513 |     Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[
514 |           n].Sinit - Sinitmean );
515 |     Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[
516 |           n].Iinit - Iinitmean );
517 |     Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[
518 |           n].Rinit - Rinitmean );
519 |
520 | }
521 |
522 | R0sd      /= NP;
523 | rsd       /= NP;

```

```

3206 515     resd          /= NP;
3207 516     sigmasd       /= NP;
3208 517     etasd         /= NP;
3209 518     berrsd        /= NP;
3210 519     Sinitsd       /= NP;
3211 520     Iinitsd       /= NP;
3212 521     Rinitsd       /= NP;
3213 522
3214 523     partInfo->R0mean    = R0mean;
3215 524     partInfo->R0sd      = R0sd;
3216 525     partInfo->rmean     = rmean;
3217 526     partInfo->rsd       = rsd;
3218 527     partInfo->remean    = remean;
3219 528     partInfo->resd      = resd;
3220 529     partInfo->sigmamean = sigmamean;
3221 530     partInfo->sigmasd   = sigmasd;
3222 531     partInfo->etamean   = etamean;
3223 532     partInfo->etasd     = etasd;
3224 533     partInfo->berrmean   = berrmean;
3225 534     partInfo->berrsd    = berrsd;
3226 535     partInfo->Sinitmean  = Sinitmean;
3227 536     partInfo->Sinitsd   = Sinitsd;
3228 537     partInfo->Iinitmean  = Iinitmean;
3229 538     partInfo->Iinitsd   = Iinitsd;
3230 539     partInfo->Rinitmean  = Rinitmean;
3231 540     partInfo->Rinitsd   = Rinitsd;
3232 541
3233 542 }
3234 543
3235 544 double randu() {
3236 545
3237 546     return (double) rand() / (double) RAND_MAX;
3238 547
3239 548 }
3240 549
3241 550 void getStateMeans(State * state, Particle* particles, int NP) {
3242 551
3243 552     double Smean = 0, Imean = 0, Rmean = 0;
3244 553
3245 554     for (int n = 0; n < NP; n++) {
3246 555         Smean += particles[n].S;
3247 556         Imean += particles[n].I;
3248 557         Rmean += particles[n].R;
3249 558     }
3250 559
3251 560     state->S = (double) Smean / NP;
3252 561     state->I = (double) Imean / NP;
3253 562     state->R = (double) Rmean / NP;
3254 563
3255 564 }

```

565		3256
566		3257
567	/* Return a normally distributed random number with mean 0 and	3258
	standard deviation 1	3259
568	Uses the polar form of the Box-Muller transformation	3260
569	From http://www.design.caltech.edu/erik/Misc/Gaussian.html	3261
570	*/	3262
571	double randn() {	3263
572		3264
573	double x1, x2, w, y1;	3265
574		3266
575	do {	3267
576	x1 = 2.0 * randu() - 1.0;	3268
577	x2 = 2.0 * randu() - 1.0;	3269
578	w = x1 * x1 + x2 * x2;	3270
579	} while (w >= 1.0);	3271
580		3272
581	w = sqrt((-2.0 * log(w)) / w);	3273
582	y1 = x1 * w;	3274
583		3275
584	return y1;	3276
585		3277
586	}	3278
		3279

3280 Appendix F

3281 Spatial Epidemics

3282 F.1 Spatial SIR R Function Code

3283 R code to simulate the outlined Spatial SIR function.

```
3284 1 ## ymat:  Contains the initial conditions where:
3285 2 #        - rows are locations
3286 3 #        - columns are S, I, R
3287 4 ## pars:  Contains the parameters: global values for R0, r, N, eta,
3288          berr
3289 5 ## T:      The stop time. Since 0 is included, there should be T+1
3290          time steps in the simulation
3291 6 ## neinum: Number of neighbors for each location, in order
3292 7 ## neibmat: Contains lists of neighbors for each location
3293 8 #        - rows are parent locations (nodes)
3294 9 #        - columns are locations each parent is attached to (edges)
3295 10 StocSSIR ← function(ymat, pars, T, steps, neinum, neibmat) {
3296 11
3297 12     ## number of locations
3298 13     nloc ← dim(ymat)[1]
3299 14
3300 15     ## storage
3301 16     ## dims are locations, (S,I,R,B), times
3302 17     # output array
3303 18     out ← array(NA, c(nloc, 4, T+1), dimnames = list(NULL, c("S", "I",
3304          "R", "B"), NULL))
3305 19     # temp storage
3306 20     BSI ← numeric(nloc)
3307 21     rI ← numeric(nloc)
3308 22
3309 23     ## extract parameters
3310 24     R0 ← pars[['R0']]
3311 25     r ← pars[['r']]
```



```

26 N ← pars[['N']] 3313
27 eta ← pars[['eta']] 3314
28 berr ← pars[['berr']] 3315
29 phi ← pars[['phi']] 3316
30 3317
31 B0 ← rep(R0*r/N, nloc) 3318
32 3319
33 ## state vectors 3320
34 S ← ymat[, 'S'] 3321
35 I ← ymat[, 'I'] 3322
36 R ← ymat[, 'R'] 3323
37 B ← B0 3324
38 3325
39 ## assign starting to output matrix 3326
40 out[, , 1] ← cbind(ymat, B0) 3327
41 3328
42 h ← 1 / steps 3329
43 3330
44 for ( i in 1:(T*steps) ) { 3331
45 3332
46 B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(nloc, 0, 3333
47 berr) ) 3334
48 3335
49 for (loc in 1:nloc) { 3336
50 n ← neinum[loc] 3337
51 sphi ← 1 - phi*(n/(n+1)) 3338
52 ophi ← phi/(n+1) 3339
53 nBIsu ← B[neibmat[loc, 1:n]] %*% I[neibmat[loc, 1:n]] 3340
54 BSI[loc] ← S[loc]*( sphi*B[loc]*I[loc] + ophi*nBIsu ) 3341
55 } 3342
56 3343
57 #if(i == 1) 3344
58 # print(BSI) 3345
59 3346
60 rI ← r*I 3347
61 3348
62 dS ← -BSI 3349
63 dI ← BSI - rI 3350
64 dR ← rI 3351
65 3352
66 S ← S + h*dS 3353
67 I ← I + h*dI 3354
68 R ← R + h*dR 3355
69 3356
70 if (i %% steps == 0) { 3357
71 out[, , i/steps+1] ← cbind(S, I, R, B) 3358
72 } 3359
73 } 3360
74 3361

```

```

3363 75     #out[, ,2] ← cbind(S,I,R,B)
3364 76
3365 77     return(out)
3366 78
3367 79 }
3368 80
3369 81 ### Suggested parameters
3370 82 #
3371 83 # T          ← 60
3372 84 # i_infec ← 5
3373 85 # steps    ← 7
3374 86 # N        ← 500
3375 87 # sigma    ← 10
3376 88 #
3377 89 # pars ← c(R0 = 3.0,      # new infected people per infected person
3378 90 #         r = 0.1,      # recovery rate
3379 91 #         N = 500,      # population size
3380 92 #         eta = 0.5,    # geometric random walk
3381 93 #         berr = 0.5)   # Beta geometric walk noise
3382

```

F.2 RStan Spatial SIR Code

This code implements a Spatial SIR model in Rstan.

```

3384
3385 1 data {
3386 2
3387 3     int      <lower=1>    T;      // total integration steps
3388 4     int      <lower=1>    nloc;   // number of locations
3389 5     real      y[nloc, T]; // observed number of cases
3390 6     int      <lower=1>    N;      // population size
3391 7     real      h;         // step size
3392 8     int      <lower=0>    neinum[nloc]; // number of neighbors
3393 9     each location has
3394 10    int      neibmat[nloc, nloc]; // neighbor list for
3395 11    each location
3396 12
3397 13 }
3398 14
3399 15 parameters {
3400 16
3401 17     real <lower=0, upper=10>    R0;      // R0
3402 18     real <lower=0, upper=10>    r;      // recovery rate
3403 19     real <lower=0, upper=20>    sigma;  // observation error
3404 20     real <lower=0, upper=30>    Iinit[nloc]; // initial
3405 21     infected for each location
3406 22     real <lower=0, upper=1>    eta;    // geometric walk
3407 23     attraction strength
3408

```

```

20     real <lower=0, upper=1>          berr;    // beta walk noise      3409
21     real <lower=-1.5, upper=1.5>    Bnoise[nloc,T]; // Beta vector  3410
22     real <lower=0, upper=1>          phi;     // interconnectivity    3411
        strength                      3412
23                                     3413
24 }                                   3414
25                                   3415
26 model {                             3416
27                                   3417
28     real S[nloc, T];                3418
29     real I[nloc, T];                3419
30     real R[nloc, T];                3420
31     real B[nloc, T];                3421
32     real B0;                        3422
33                                   3423
34     real BSI[nloc, T];               3424
35     real rI[nloc, T];               3425
36     int n;                          3426
37     real sphl;                      3427
38     real ophi;                      3428
39     real nBIsum;                    3429
40                                   3430
41     B0 ← R0 * r / N;                3431
42                                   3432
43     for (loc in 1:nloc) {           3433
44         S[loc, 1] ← N - Iinit[loc]; 3434
45         I[loc, 1] ← Iinit[loc];      3435
46         R[loc, 1] ← 0.0;             3436
47         B[loc, 1] ← B0;              3437
48     }                               3438
49                                   3439
50     for (t in 2:T) {                3440
51         for (loc in 1:nloc) {       3441
52                                   3442
53             Bnoise[loc, t] ~ normal(0,berr); 3443
54             B[loc, t] ← exp( log(B[loc, t-1]) + eta * ( log(B0) - 3444
                    log(B[loc, t-1]) ) + Bnoise[loc, t] ); 3445
55                                   3446
56             n ← neinum[loc];          3447
57             sphl ← 1.0 - phi*( n/(n+1.0) ); 3448
58             ophi ← phi/(n+1.0);       3449
59                                   3450
60             nBIsum ← 0.0;             3451
61             for (j in 1:n)            3452
62                 nBIsum ← nBIsum + B[neibmat[loc, j], t-1] * I[ 3453
                    neibmat[loc, j], t-1]; 3454
63                                   3455
64             BSI[loc, t] ← S[loc, t-1]*( sphl*B[loc, t-1]*I[loc, t-1] 3456
                    + ophi*nBIsum ); 3457
65             rI[loc, t] ← r*I[loc, t-1]; 3458

```

```

3459 66
3460 67     S[loc, t] ← S[loc, t-1] + h*( - BSI[loc, t] );
3461 68     I[loc, t] ← I[loc, t-1] + h*( BSI[loc, t] - rI[loc, t] );
3462    ;
3463 69     R[loc, t] ← R[loc, t-1] + h*( rI[loc, t] );
3464 70
3465 71     if (y[loc, t] > 0) {
3466 72         y[loc, t] ~ normal( I[loc, t], sigma );
3467 73     }
3468 74
3469 75     }
3470 76 }
3471 77
3472 78 R0      ~ lognormal(1,1);
3473 79 r       ~ lognormal(1,1);
3474 80 sigma   ~ lognormal(1,1);
3475 81 for (loc in 1:nloc) {
3476 82     Iinit[loc] ~ normal(y[loc, 1], sigma);
3477 83 }
3478 84
3479 85 }

```

3481 F.3 IF2 Spatial SIR Code

3482 This code implements a Spatial SIR model using IF2 in C++.

```

3483 1 /* Author: Dexter Barrows
3484 2   Github: dbarrows.github.io
3485 3
3486 4   */
3487 5
3488 6 #include <stdio.h>
3489 7 #include <math.h>
3490 8 #include <sys/time.h>
3491 9 #include <time.h>
3492 10 #include <stdlib.h>
3493 11 #include <string>
3494 12 #include <cmath>
3495 13 #include <cstdlib>
3496 14 #include <fstream>
3497 15
3498 16 // #include "rand.h"
3499 17 // #include "timer.h"
3500 18
3501 19 #define Treal      100          // time to simulate over
3502 20 #define R0true     3.0          // infectiousness
3503 21 #define rtrue      0.1          // recovery rate

```

```

22 #define Nreal      500.0      // population size      3505
23 #define etatrue    0.5        // real drift attraction strength  3506
24 #define berrtrue   0.5        // real beta drift noise          3507
25 #define phitrue    0.5        // real connectivity strength     3508
26 #define merr       10.0       // expected measurement error     3509
27 #define I0         5.0        // Initial infected individuals    3510
28                                     3511
29 #define PSC        0.5        // perturbation scale factor for  3512
    more sensitive parameters      3513
30                                     3514
31 #include <Rcpp.h>                3515
32 using namespace Rcpp;           3516
33                                 3517
34 struct Particle {                3518
35     double R0;                   3519
36     double r;                    3520
37     double sigma;                3521
38     double eta;                  3522
39     double berr;                 3523
40     double phi;                  3524
41     double * S;                  3525
42     double * I;                  3526
43     double * R;                  3527
44     double * B;                  3528
45     double * Iinit;              3529
46 };                               3530
47                                 3531
48                                 3532
49 int timeval_subtract (double *result, struct timeval *x, struct  3533
    timeval *y);                  3534
50 int check_double(double x,double y); 3535
51 void initializeParticles(Particle ** particles, int NP, int nloc, 3536
    int N);                        3537
52 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle 3538
    * particle,                    3539
53     NumericVector neinum, NumericMatrix neibmat, int 3540
    nloc) ;                       3541
54 void copyParticle(Particle * dst, Particle * src, int nloc);      3542
55 void perturbParticles(Particle * particles, int N, int NP, int nloc, 3543
    int passnum, double coolrate); 3544
56 double randu();                3545
57 double randn();                 3546
58                                 3547
59 // [[Rcpp::export]]             3548
60 Rcpp::List if2_spa(NumericMatrix data, int T, int N, int NP, int 3549
    nPasses, double coolrate, NumericVector neinum, NumericMatrix 3550
    neibmat, int nloc) {          3551
61                                     3552
62     NumericMatrix paramdata(NP, 6); // for R0, r, sigma, eta, 3553
    berr, phi                      3554

```

```

3555 63 NumericMatrix initInfec(nloc, NP); // for Iinit
3556 64 NumericMatrix infecmeans(nloc, T); // mean infection counts for
3557      each location
3558 65 NumericMatrix finalstate(nloc, 4); // SIRB means for each
3559      location
3560 66
3561 67 srand(time(NULL)); // Seed PRNG with system time
3562 68
3563 69 double w[NP]; // particle weights
3564 70
3565 71 // initialize particles
3566 72 printf("Initializing particle states\n");
3567 73 Particle * particles = NULL; // particle estimates for
3568      current step
3569 74 Particle * particles_old = NULL; // intermediate particle
3570      states for resampling
3571 75 initializeParticles(&particles, NP, nloc, N);
3572 76 initializeParticles(&particles_old, NP, nloc, N);
3573 77
3574 78 /*
3575 79 // copy particle test
3576 80 copyParticle(&particles[0], &particles_old[0], nloc);
3577 81
3578 82 // perturb particle test
3579 83 perturbParticles(particles, N, NP, nloc, 1, coolrate);
3580 84
3581 85 // evolution test
3582 86 // reset particle system evolution states
3583 87 for (int n = 0; n < NP; n++) {
3584 88     for (int loc = 0; loc < nloc; loc++) {
3585 89         particles[n].S[loc] = N - particles[n].Iinit[loc];
3586 90         particles[n].I[loc] = particles[n].Iinit[loc];
3587 91         particles[n].R[loc] = 0.0;
3588 92         particles[n].B[loc] = (double) particles[n].R0 *
3589             particles[n].r / N;
3590 93     }
3591 94 }
3592 95 printf("Before S:%f | I:%f | R:%f\n", particles[0].S[0],
3593     particles[0].I[0], particles[0].R[0]);
3594 96 exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[0], neinum,
3595     neibmat, nloc);
3596 97 printf("After S:%f | I:%f | R:%f\n", particles[0].S[0],
3597     particles[0].I[0], particles[0].R[0]);
3598 98 */
3599 99
3600 100 // START PASSES THROUGH DATA
3601 101
3602 102 printf("Starting filter\n");
3603 103 printf("-----\n");
3604 104 printf("Pass\n");

```

```

105 |
106 |
107 |     for (int pass = 0; pass < nPasses; pass++) {
108 |
109 |         printf("...%d / %d\n", pass, nPasses);
110 |
111 |         // reset particle system evolution states
112 |         for (int n = 0; n < NP; n++) {
113 |             for (int loc = 0; loc < nloc; loc++) {
114 |                 particles[n].S[loc] = N - particles[n].Iinit[loc];
115 |                 particles[n].I[loc] = particles[n].Iinit[loc];
116 |                 particles[n].R[loc] = 0.0;
117 |                 particles[n].B[loc] = (double) particles[n].R0 *
118 |                     particles[n].r / N;
119 |             }
120 |         }
121 |
122 |         if (pass == (nPasses-1)) {
123 |             double means[nloc];
124 |             for (int loc = 0; loc < nloc; loc++) {
125 |                 means[loc] = 0.0;
126 |                 for (int n = 0; n < NP; n++) {
127 |                     means[loc] += particles[n].I[loc] / NP;
128 |                 }
129 |                 infecmeans(loc, 0) = means[loc];
130 |             }
131 |         }
132 |
133 |         for (int t = 1; t < T; t++) {
134 |
135 |             // generate individual predictions and weight
136 |             for (int n = 0; n < NP; n++) {
137 |
138 |                 exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[n],
139 |                     neinum, neibmat, nloc);
140 |
141 |                 double merr_par = particles[n].sigma;
142 |
143 |                 w[n] = 1.0;
144 |                 for (int loc = 0; loc < nloc; loc++) {
145 |                     double y_diff = data(loc, t) - particles[n].I[
146 |                         loc];
147 |                     w[n] *= 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( -
148 |                         y_diff*y_diff / (2.0*merr_par*merr_par) );
149 |                 }
150 |             }
151 |
152 |             // cumulative sum
153 |             for (int n = 1; n < NP; n++) {

```

```

3655 151         w[n] += w[n-1];
3656 152     }
3657 153
3658 154     // save particle states to resample from
3659 155     for (int n = 0; n < NP; n++){
3660 156         copyParticle(&particles_old[n], &particles[n], nloc)
3661 157         ;
3662 158     }
3663 159
3664 159     // resampling
3665 160     for (int n = 0; n < NP; n++) {
3666 161
3667 162         double w_r = randu() * w[NP-1];
3668 163         int i = 0;
3669 164         while (w_r > w[i]) {
3670 165             i++;
3671 166         }
3672 167
3673 168         // i is now the index to copy state from
3674 169         copyParticle(&particles[n], &particles_old[i], nloc)
3675 170         ;
3676 171     }
3677 172
3678 172
3679 173     // between-iteration perturbations, not after last time
3680 174     step
3681 174     if (t < (T-1))
3682 175         perturbParticles(particles, N, NP, nloc, pass,
3683 176         coolrate);
3684 176
3685 177     if (pass == (nPasses-1)) {
3686 178         double means[nloc];
3687 179         for (int loc = 0; loc < nloc; loc++) {
3688 180             means[loc] = 0.0;
3689 181             for (int n = 0; n < NP; n++) {
3690 182                 means[loc] += particles[n].I[loc] / NP;
3691 183             }
3692 184             infecmeans(loc, t) = means[loc];
3693 185         }
3694 186     }
3695 187
3696 188 }
3697 189
3698 190 // between-pass perturbations, not after last pass
3699 191 if (pass < (nPasses + 1))
3700 192     perturbParticles(particles, N, NP, nloc, pass, coolrate)
3701 193     ;
3702 193
3703 194 }
3704 195

```



```

196 // pack parameter data (minus initial conditions) 3705
197 for (int n = 0; n < NP; n++) { 3706
198     paramdata(n, 0) = particles[n].R0; 3707
199     paramdata(n, 1) = particles[n].r; 3708
200     paramdata(n, 2) = particles[n].sigma; 3709
201     paramdata(n, 3) = particles[n].eta; 3710
202     paramdata(n, 4) = particles[n].berr; 3711
203     paramdata(n, 5) = particles[n].phi; 3712
204 } 3713
205 3714
206 // Pack initial condition data 3715
207 for (int n = 0; n < NP; n++) { 3716
208     for (int loc = 0; loc < nloc; loc++) { 3717
209         initInfec(loc, n) = particles[n].Iinit[loc]; 3718
210     } 3719
211 } 3720
212 3721
213 // Pack final state means data 3722
214 double Smeans[nloc], Imeans[nloc], Rmeans[nloc], Bmeans[nloc]; 3723
215 for (int loc = 0; loc < nloc; loc++) { 3724
216     Smeans[loc] = 0.0; 3725
217     Imeans[loc] = 0.0; 3726
218     Rmeans[loc] = 0.0; 3727
219     Bmeans[loc] = 0.0; 3728
220     for (int n = 0; n < NP; n++) { 3729
221         Smeans[loc] += particles[n].S[loc] / NP; 3730
222         Imeans[loc] += particles[n].I[loc] / NP; 3731
223         Rmeans[loc] += particles[n].R[loc] / NP; 3732
224         Bmeans[loc] += particles[n].B[loc] / NP; 3733
225     } 3734
226     finalstate(loc, 0) = Smeans[loc]; 3735
227     finalstate(loc, 1) = Imeans[loc]; 3736
228     finalstate(loc, 2) = Rmeans[loc]; 3737
229     finalstate(loc, 3) = Bmeans[loc]; 3738
230 } 3739
231 3740
232 3741
233 return Rcpp::List::create( Rcpp::Named("paramdata") = paramdata 3742
234     , 3743
235     Rcpp::Named("initInfec") = initInfec 3744
236     , 3745
237     Rcpp::Named("infecmeans") = 3746
238     infecmeans, 3747
239     Rcpp::Named("finalstate") = 3748
240     finalstate); 3749
241 } 3750
3751
3752
3753
3754

```

```

3755 242
3756 243 /* Use the Explicit Euler integration scheme to integrate SIR model
3757         forward in time
3758 244     double h      - time step size
3759 245     double t0     - start time
3760 246     double tn     - stop time
3761 247     double * y    - current system state; a three-component vector
3762                     representing [S I R], susceptible-infected-recovered
3763 248
3764 249     */
3765 250 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle
3766     * particle,
3767 251                     NumericVector neinum, NumericMatrix neibmat, int
3768                     nloc) {
3769 252
3770 253     int num_steps = floor( (tn-t0) / h );
3771 254
3772 255     double * S = particle->S;
3773 256     double * I = particle->I;
3774 257     double * R = particle->R;
3775 258     double * B = particle->B;
3776 259
3777 260     // create last state vectors
3778 261     double S_last[nloc];
3779 262     double I_last[nloc];
3780 263     double R_last[nloc];
3781 264     double B_last[nloc];
3782 265
3783 266     double R0    = particle->R0;
3784 267     double r     = particle->r;
3785 268     double B0    = R0 * r / N;
3786 269     double eta   = particle->eta;
3787 270     double berr  = particle->berr;
3788 271     double phi   = particle->phi;
3789 272
3790 273     //printf("sphi \t\t| ophi \t\t| BSI \t\t| rI \t\t| dS \t\t| dI \
3791         \t\t| dR \t\t| S\t\t| I \t\t| R |\n");
3792 274
3793 275     for(int t = 0; t < num_steps; t++) {
3794 276
3795 277         for (int loc = 0; loc < nloc; loc++) {
3796 278             S_last[loc] = S[loc];
3797 279             I_last[loc] = I[loc];
3798 280             R_last[loc] = R[loc];
3799 281             B_last[loc] = B[loc];
3800 282         }
3801 283
3802 284         for (int loc = 0; loc < nloc; loc++) {
3803 285
3804 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

```

```

3855 331     for (int n = 0; n < NP; n++) {
3856 332         (*particles)[n].S = (double*) malloc(nloc*sizeof(double));
3857 333         (*particles)[n].I = (double*) malloc(nloc*sizeof(double));
3858 334         (*particles)[n].R = (double*) malloc(nloc*sizeof(double));
3859 335         (*particles)[n].B = (double*) malloc(nloc*sizeof(double));
3860 336         (*particles)[n].Iinit = (double*) malloc(nloc*sizeof(double)
3861                                     );
3862 337     }
3863 338
3864 339     // initialize all all parameters
3865 340     for (int n = 0; n < NP; n++) {
3866 341
3867 342         double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
3868             phican;
3869 343
3870 344         do {
3871 345             R0can = R0true + R0true*randn();
3872 346         } while (R0can < 0);
3873 347         (*particles)[n].R0 = R0can;
3874 348
3875 349         do {
3876 350             rcan = rtrue + rtrue*randn();
3877 351         } while (rcan < 0);
3878 352         (*particles)[n].r = rcan;
3879 353
3880 354         for (int loc = 0; loc < nloc; loc++)
3881 355             (*particles)[n].B[loc] = (double) R0can * rcan / N;
3882 356
3883 357         do {
3884 358             sigmacan = merr + merr*randn();
3885 359         } while (sigmacan < 0);
3886 360         (*particles)[n].sigma = sigmacan;
3887 361
3888 362         do {
3889 363             etacan = etatrue + PSC*etatrue*randn();
3890 364         } while (etacan < 0 || etacan > 1);
3891 365         (*particles)[n].eta = etacan;
3892 366
3893 367         do {
3894 368             berrcan = berrtrue + PSC*berrtrue*randn();
3895 369         } while (berrcan < 0);
3896 370         (*particles)[n].berr = berrcan;
3897 371
3898 372         do {
3899 373             phican = phitrue + PSC*phitrue*randn();
3900 374         } while (phican <= 0 || phican >= 1);
3901 375         (*particles)[n].phi = phican;
3902 376
3903 377         for (int loc = 0; loc < nloc; loc++) {
3904 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;

```

```

3955 427
3956 428     do {
3957 429         berrcan = particles[n].berr + PSC*spreadberr*randn();
3958 430     } while (berrcan < 0);
3959 431     particles[n].berr = berrcan;
3960 432
3961 433     do {
3962 434         phican = particles[n].phi + PSC*spreadphi*randn();
3963 435     } while (phican <= 0 || phican >= 1);
3964 436     particles[n].phi = phican;
3965 437
3966 438     for (int loc = 0; loc < nloc; loc++) {
3967 439         do {
3968 440             Iinitcan = particles[n].Iinit[loc] + spreadIinit*
3969             randn();
3970 441         } while (Iinitcan < 0 || Iinitcan > 500);
3971 442         particles[n].Iinit[loc] = Iinitcan;
3972 443     }
3973 444 }
3974 445
3975 446 }
3976 447
3977 448 /* Convenience function for particle resampling process
3978 449 */
3979 450 void copyParticle(Particle * dst, Particle * src, int nloc) {
3980 451
3981 452     dst->R0      = src->R0;
3982 453     dst->r        = src->r;
3983 454     dst->sigma    = src->sigma;
3984 455     dst->eta      = src->eta;
3985 456     dst->berr     = src->berr;
3986 457     dst->phi      = src->phi;
3987 458
3988 459     for (int n = 0; n < nloc; n++) {
3989 460         dst->S[n]      = src->S[n];
3990 461         dst->I[n]      = src->I[n];
3991 462         dst->R[n]      = src->R[n];
3992 463         dst->B[n]      = src->B[n];
3993 464         dst->Iinit[n]  = src->Iinit[n];
3994 465     }
3995 466
3996 467 }
3997 468
3998 469
3999 470
4000 471 double randu() {
4001 472
4002 473     return (double) rand() / (double) RAND_MAX;
4003 474
4004 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

```

```

4051 3      */
4052 4
4053 5  /*  Runs a particle filter on synthetic noisy data and attempts to
4054 6      reconstruct underlying true state at each time step. Note that
4055 7      this program uses gnuplot to plot the data, so an x11
4056 8      environment must be present. Also the multiplier of 1024 in the
4057 9      definition of NP below should be set to a multiple of the number
4058 10     of multiprocessors of your GPU for optimal results.
4059 11
4060 12     Also, the accompanying "pf.plg" file contains the instructions
4061 13     gnuplot will use. It must be present in the same directory as
4062 14     the executable generated by compiling this file.
4063 15
4064 16     Compile with:
4065 17
4066 18     nvcc -arch=sm_20 -O2 pf_cuda.cu timer.cpp rand.cpp -o pf_cuda.x
4067 19
4068 20     */
4069 21
4070 22 #include <cuda.h>
4071 23 #include <iostream>
4072 24 #include <fstream>
4073 25 #include <curand.h>
4074 26 #include <curand_kernel.h>
4075 27 #include <string>
4076 28 #include <sstream>
4077 29 #include <cmath>
4078 30
4079 31 #include "timer.h"
4080 32 #include "rand.h"
4081 33 #include "readdata.h"
4082 34
4083 35 #define NP          (2*2500)      // number of particles
4084 36 #define N           500.0        // population size
4085 37 #define R0true      3.0           // infectiousness
4086 38 #define rtrue       0.1           // recovery rate
4087 39 #define etatrue     0.5           // real drift attraction strength
4088 40 #define berrtrue    0.5           // real beta drift noise
4089 41 #define phitrue     0.5           // real connectivity strength
4090 42 #define merr        10.0          // expected measurement error
4091 43 #define I0          5.0           // Initial infected individuals
4092 44 #define PSC         0.5           // sensitive parameter perturbation
4093 45     scaling
4094 46 #define NLOC        10
4095 47
4096 47 #define PI          3.141592654f
4097 48
4098 49 // Wrapper for CUDA calls, from CUDA API
4099 50 // Modified to also print the error code and string
4100 51 # define CUDA_CALL(x) do { if ((x) != cudaSuccess ) {

```



```

52         \
std::cout << " Error at " << __FILE__ << ":" << __LINE__ << std
::endl; \
53 std::cout << " Error was " << x << " " << cudaGetErrorString(x)
<< std::endl; \
54 return EXIT_FAILURE ;}} while (0)
\
55
56 typedef struct {
57     float R0;
58     float r;
59     float sigma;
60     float eta;
61     float berr;
62     float phi;
63     /*
64     float * S;
65     float * I;
66     float * R;
67     float * B;
68     float * Iinit;
69     */
70     float S[NLOC];
71     float I[NLOC];
72     float R[NLOC];
73     float B[NLOC];
74     float Iinit[NLOC];
75     curandState randState; // PRNG state
76 } Particle;
77
78 __host__ std::string getHRmemsize (size_t memsize);
79 __host__ std::string getHRtime (float runtime);
80
81 __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
* particle, int * neinum, int * neibmat, int nloc);
82 __device__ void copyParticle(Particle * dst, Particle * src, int
nloc);
83
84
85 /* Initialize all PRNG states, get starting state vector using
initial distribution
86 */
87 __global__ void initializeParticles (Particle * particles, int nloc)
{
88
89     int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
ID
90
91     if (id < NP) {
92

```

```

4151 93 // initialize PRNG state
4152 94 curandState state;
4153 95 curand_init(id, 0, 0, &state);
4154 96
4155 97 // allocate space for arrays inside particle
4156 98 //particles[id].S = (float*) malloc(nloc*sizeof(float));
4157 99 //particles[id].I = (float*) malloc(nloc*sizeof(float));
4158 100 //particles[id].R = (float*) malloc(nloc*sizeof(float));
4159 101 //particles[id].B = (float*) malloc(nloc*sizeof(float));
4160 102 //particles[id].Iinit = (float*) malloc(nloc*sizeof(float));
4161 103
4162 104 // initialize all parameters
4163 105
4164 106 float R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
4165 107 phican;
4166 107
4167 108 do {
4168 109     R0can = R0true + R0true*curand_normal(&state);
4169 110 } while (R0can < 0);
4170 111 particles[id].R0 = R0can;
4171 112
4172 113 do {
4173 114     rcan = rtrue + rtrue*curand_normal(&state);
4174 115 } while (rcan < 0);
4175 116 particles[id].r = rcan;
4176 117
4177 118 for (int loc = 0; loc < nloc; loc++)
4178 119     particles[id].B[loc] = (float) R0can * rcan / N;
4179 120
4180 121 do {
4181 122     sigmacan = merr + merr*curand_normal(&state);
4182 123 } while (sigmacan < 0);
4183 124 particles[id].sigma = sigmacan;
4184 125
4185 126 do {
4186 127     etacan = etatrue + PSC*etatrue*curand_normal(&state);
4187 128 } while (etacan < 0 || etacan > 1);
4188 129 particles[id].eta = etacan;
4189 130
4190 131 do {
4191 132     berrcan = berrtrue + PSC*berrtrue*curand_normal(&state);
4192 133 } while (berrcan < 0);
4193 134 particles[id].berr = berrcan;
4194 135
4195 136 do {
4196 137     phican = phitrue + PSC*phitrue*curand_normal(&state);
4197 138 } while (phican <= 0 || phican >= 1);
4198 139 particles[id].phi = phican;
4199 140
4200 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 *

```

```

4251     neibmat, int nloc) {
4252 189
4253 190     int id = blockIdx.x*blockDim.x + threadIdx.x;    // global id
4254 191
4255 192     if (id < NP) {
4256 193         // project forward
4257 194         exp_euler_SSIR(1.0/7.0, 0.0, 1.0, &particles[id], neinum,
4258         neibmat, nloc);
4259 195     }
4260 196
4261 197 }
4262 198
4263 199 __global__ void weight(float * data, Particle * particles, double *
4264     w, int t, int T, int nloc) {
4265 200
4266 201     int id = blockIdx.x*blockDim.x + threadIdx.x;    // global id
4267 202
4268 203     if (id < NP) {
4269 204
4270 205         float merr_par = particles[id].sigma;
4271 206
4272 207         // Get weight and save
4273 208         double w_local = 1.0;
4274 209         for (int loc = 0; loc < nloc; loc++) {
4275 210             float y_diff = data[loc*T + t] - particles[id].I[loc];
4276 211             w_local *= 1.0/(merr_par*sqrt(2.0*PI)) * exp( - y_diff*
4277             y_diff / (2.0*merr_par*merr_par) );
4278 212
4279 213         }
4280 214         w[id] = w_local;
4281 215
4282 216     }
4283 217
4284 218 }
4285 219
4286 220 __global__ void stashParticles (Particle * particles, Particle *
4287     particles_old, int nloc) {
4288 221
4289 222     int id = blockIdx.x*blockDim.x + threadIdx.x;    // global id
4290 223
4291 224     if (id < NP) {
4292 225         // COPY PARTICLE
4293 226         copyParticle(&particles_old[id], &particles[id], nloc);
4294 227     }
4295 228
4296 229 }
4297 230
4298 231
4299 232 /* The 0th thread will perform cumulative sum on the weights.
4300 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;

```

```

4351 279     double countmean_local = 0.0;
4352 280     for (int n = 0; n < NP; n++) {
4353 281         countmean_local += particles[n].I[loc] / NP;
4354 282     }
4355 283
4356 284     countmeans[loc*T + t] = (float) countmean_local;
4357 285
4358 286 }
4359 287
4360 288 }
4361 289
4362 290 __global__ void perturbParticles(Particle * particles, int nloc, int
4363     passnum, double coolrate) {
4364 291
4365 292     //double coolcoef = exp( - (double) passnum / coolrate );
4366 293     double coolcoef = pow(coolrate, passnum);
4367 294
4368 295     double spreadR0      = coolcoef * R0true / 10.0;
4369 296     double spreadr       = coolcoef * rtrue / 10.0;
4370 297     double spreadsigma   = coolcoef * merr / 10.0;
4371 298     double spreadIinit   = coolcoef * I0 / 10.0;
4372 299     double spreadeta     = coolcoef * etatrue / 10.0;
4373 300     double spreadberr    = coolcoef * berrtrue / 10.0;
4374 301     double spreadphi     = coolcoef * phitrue / 10.0;
4375 302
4376 303     double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
4377 304
4378 305     int id = blockIdx.x*blockDim.x + threadIdx.x;
4379 306
4380 307     if (id < NP) {
4381 308
4382 309         do {
4383 310             R0can = particles[id].R0 + spreadR0*curand_normal(&
4384             particles[id].randState);
4385 311         } while (R0can < 0);
4386 312         particles[id].R0 = R0can;
4387 313
4388 314         do {
4389 315             rcan = particles[id].r + spreadr*curand_normal(&
4390             particles[id].randState);
4391 316         } while (rcan < 0);
4392 317         particles[id].r = rcan;
4393 318
4394 319         do {
4395 320             sigmacan = particles[id].sigma + spreadsigma*
4396             curand_normal(&particles[id].randState);
4397 321         } while (sigmacan < 0);
4398 322         particles[id].sigma = sigmacan;
4399 323
4400 324         do {

```

```

325         etacan = particles[id].eta + PSC*spreadeta*curand_normal 4401
           (&particles[id].randState);                               4402
326     } while (etacan < 0 || etacan > 1);                           4403
327     particles[id].eta = etacan;                                    4404
328                                                                     4405
329     do {                                                           4406
330         berrcan = particles[id].berr + PSC*spreadberr*           4407
           curand_normal(&particles[id].randState);                 4408
331     } while (berrcan < 0);                                         4409
332     particles[id].berr = berrcan;                                  4410
333                                                                     4411
334     do {                                                           4412
335         phican = particles[id].phi + PSC*spreadphi*curand_normal 4413
           (&particles[id].randState);                               4414
336     } while (phican <= 0 || phican >= 1);                         4415
337     particles[id].phi = phican;                                    4416
338                                                                     4417
339     for (int loc = 0; loc < nloc; loc++) {                         4418
340         do {                                                       4419
341             Iinitcan = particles[id].Iinit[loc] + spreadIinit*   4420
               curand_normal(&particles[id].randState);             4421
342         } while (Iinitcan < 0 || Iinitcan > 500);                 4422
343         particles[id].Iinit[loc] = Iinitcan;                     4423
344     }                                                             4424
345                                                                     4425
346 }                                                                 4426
347                                                                     4427
348 }                                                                 4428
349                                                                     4429
350                                                                     4430
351 int main (int argc, char *argv[]) {                               4431
352                                                                     4432
353                                                                     4433
354     int T, nloc;                                                  4434
355                                                                     4435
356     double restime;                                               4436
357     struct timeval tdr0, tdr1, tdrMaster;                        4437
358                                                                     4438
359     gettimeofday (&tdr0, NULL);                                   4439
360                                                                     4440
361                                                                     4441
362     // Parse arguments                                           4442
           *****                                                  4443
363                                                                     4444
364     if (argc < 4) {                                               4445
365         std::cout << "Not enough arguments" << std::endl;     4446
366         return 0;                                                 4447
367     }                                                            4448
368                                                                     4449
369     std::string arg1(argv[1]); // infection counts              4450

```

```

4451 370     std::string arg2(argv[2]); // neighbour counts
4452 371     std::string arg3(argv[3]); // neighbour indices
4453 372
4454 373     std::cout << "Arguments:" << std::endl;
4455 374     std::cout << "Infection data:      " << arg1 << std::endl;
4456 375     std::cout << "Neighbour counts:    " << arg2 << std::endl;
4457 376     std::cout << "Neighbour indices:  " << arg3 << std::endl;
4458 377
4459 378     //
4460
4461     *****
4462 379
4463 380
4464 381     // Read count data
4465     *****
4466 382
4467 383     std::cout << "Getting count data" << std::endl;
4468 384     float * data = getDataFloat(arg1, &T, &nloc);
4469 385     size_t datasize = nloc*T*sizeof(float);
4470 386
4471 387     //
4472     *****
4473
4474 388
4475 389     // Read neinum matrix data
4476     *****
4477 390
4478 391     std::cout << "Getting neighbour count data" << std::endl;
4479 392     int * neinum = getDataInt(arg2, NULL, NULL);
4480 393     size_t neinumsize = nloc * sizeof(int);
4481 394
4482 395     //
4483     *****
4484
4485 396
4486 397     // Read neibmat matrix data
4487     *****
4488 398
4489 399     std::cout << "Getting neighbour count data" << std::endl;
4490 400     int * neibmat = getDataInt(arg3, NULL, NULL);
4491 401     size_t neibmatsize = nloc * nloc * sizeof(int);
4492 402
4493 403     //
4494     *****
4495
4496 404
4497 405
4498 406     gettimeofday (&tdr1, NULL);
4499 407     timeval_subtract (&restime, &tdr1, &tdr0);
4500 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
455

```

```

4551 445
4552 446     std::cout << "\t[" << getHRmemsize(used) << "]" used of [" <<
4553     getHRmemsize(total) << "]" <<std::endl;
4554 447
4555 448     std::cout << "Copying data to device" << std::endl;
4556 449
4557 450     gettimeofday (&tdr0, NULL);
4558 451
4559 452     CUDA_CALL( cudaMemcpy(d_data      , data      , datasize      ,
4560     cudaMemcpyHostToDevice) );
4561 453     CUDA_CALL( cudaMemcpy(d_neinum    , neinum    , neinumsize    ,
4562     cudaMemcpyHostToDevice) );
4563 454     CUDA_CALL( cudaMemcpy(d_neibmat   , neibmat   , neibmatsize   ,
4564     cudaMemcpyHostToDevice) );
4565 455
4566 456     gettimeofday (&tdr1, NULL);
4567 457     timeval_subtract (&restime, &tdr1, &tdr0);
4568 458
4569 459     std::cout << "\t" << getHRtime(restime) << std::endl;
4570 460
4571 461     //
4572     *****
4573
4574 462
4575 463
4576 464
4577 465     // Initialize particles
4578     *****
4579 466
4580 467     std::cout << "Initializing particles" << std::endl;
4581 468
4582 469     gettimeofday (&tdr0, NULL);
4583 470
4584 471     int nThreads      = 32;
4585 472     int nBlocks       = ceil( (float) NP / nThreads);
4586 473
4587 474     initializeParticles <<< nBlocks, nThreads >>> (particles, nloc);
4588 475     CUDA_CALL( cudaGetLastError() );
4589 476     CUDA_CALL( cudaDeviceSynchronize() );
4590 477
4591 478     initializeParticles <<< nBlocks, nThreads >>> (particles_old,
4592     nloc);
4593 479     CUDA_CALL( cudaGetLastError() );
4594 480     CUDA_CALL( cudaDeviceSynchronize() );
4595 481
4596 482     gettimeofday (&tdr1, NULL);
4597 483     timeval_subtract (&restime, &tdr1, &tdr0);
4598 484
4599 485     std::cout << "\t" << getHRtime(restime) << std::endl;
4600 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;

```

```

4651 530
4652 531     for (int t = 1; t < Tlim; t++) {
4653 532
4654 533         // Projection
4655         *****
4656 534
4657 535         nThreads      = 32;
4658 536         nBlocks       = ceil( (float) NP / nThreads);
4659 537
4660 538         //if (t == 1)
4661 539         //  gettimeofday (&tdr0, NULL);
4662 540
4663 541         project <<< nBlocks, nThreads >>> (particles, d_neinum,
4664         d_neibmat, nloc);
4665 542         CUDA_CALL( cudaGetLastError() );
4666 543         CUDA_CALL( cudaDeviceSynchronize() );
4667 544
4668 545         //if (t == 1) {
4669 546         //  gettimeofday (&tdr1, NULL);
4670 547         //  timeval_subtract (&restime, &tdr1, &tdr0);
4671 548         //  std::cout << "\tProjection " << getHRtime(restime)
4672         << std::endl;
4673 549         //}
4674 550
4675 551         // Weighting
4676         *****
4677 552
4678 553         nThreads      = 32;
4679 554         nBlocks       = ceil( (float) NP / nThreads);
4680 555
4681 556         weight <<< nBlocks, nThreads >>>(d_data, particles, w, t
4682         , T, nloc);
4683 557         CUDA_CALL( cudaGetLastError() );
4684 558         CUDA_CALL( cudaDeviceSynchronize() );
4685 559
4686 560         // Cumulative sum
4687         *****
4688 561
4689 562         nThreads      = 1;
4690 563         nBlocks       = 1;
4691 564
4692 565         if (t == 1)
4693 566             gettimeofday (&tdr0, NULL);
4694 567
4695 568         cumsumWeights <<< nBlocks, nThreads >>> (w);
4696 569         CUDA_CALL( cudaGetLastError() );
4697 570         CUDA_CALL( cudaDeviceSynchronize() );
4698 571
4699 572         if (t == 1) {
4700 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;

```

```

4751 617
4752 618         reduceStates <<< nBlocks, nThreads >>> (particles,
4753         d_countmeans, t, T, nloc);
4754 619     CUDA_CALL( cudaGetLastError() );
4755 620     CUDA_CALL( cudaDeviceSynchronize() );
4756 621
4757 622     if (t == 1) {
4758 623         gettimeofday (&tdr1, NULL);
4759 624         timeval_subtract (&restime, &tdr1, &tdr0);
4760 625         std::cout << "Reduction          " << getHRtime(
4761         restime) << std::endl;
4762 626     }
4763 627
4764 628 }
4765 629
4766 630 // Perturb particles
4767         *****
4768 631
4769 632     nThreads      = 32;
4770 633     nBlocks       = ceil( (float) NP/ nThreads);
4771 634
4772 635     perturbParticles <<< nBlocks, nThreads >>> (particles,
4773     nloc, pass, 0.975);
4774 636     CUDA_CALL( cudaGetLastError() );
4775 637     CUDA_CALL( cudaDeviceSynchronize() );
4776 638
4777 639     //}
4778 640     /*
4779 641     nThreads      = RB_DIM;
4780 642     nBlocks       = nCells;
4781 643
4782 644
4783 645
4784 646     reduce <<< nBlocks, nThreads >>> (d_E, t, particles,
4785     Beta_last, nCells);
4786 647     CUDA_CALL( cudaGetLastError() );
4787 648     CUDA_CALL( cudaDeviceSynchronize() );
4788 649
4789 650     if (t == 1) {
4790 651         gettimeofday (&tdr1, NULL);
4791 652         timeval_subtract (&restime, &tdr1, &tdr0);
4792 653         std::cout << "Reduction          " << getHRtime(
4793         restime) << std::endl;
4794 654     }
4795 655     */
4796 656
4797 657
4798 658 } // end time
4799 659
4800 660 } // end pass

```

```

661 |                                     | 4801
662 | std::cout.precision(10);           | 4802
663 |                                     | 4803
664 | countmeans = (float*) malloc (nloc*T*sizeof(float)); | 4804
665 | cudaMemcpy(countmeans, d_countmeans, nloc*T*sizeof(float), | 4805
        |     cudaMemcpyDeviceToHost);       | 4806
666 |                                     | 4807
667 | std::string filename = "cuIF2states.dat"; | 4808
668 |                                     | 4809
669 | std::cout << "Writing results to file '" << filename << "' ... " | 4810
        |     << std::endl;                 | 4811
670 |                                     | 4812
671 | std::ofstream outfile;             | 4813
672 | outfile.open(filename.c_str());     | 4814
673 |                                     | 4815
674 | for(int loc = 0; loc < nloc; loc++) { | 4816
675 |     for (int t = 0; t < T; t++) {   | 4817
676 |         outfile << countmeans[loc*T + t] << " "; | 4818
677 |     }                               | 4819
678 |     outfile << std::endl;           | 4820
679 | }                                   | 4821
680 |                                     | 4822
681 | /*                                 | 4823
682 | double * h_w = (double*) malloc (NP*sizeof(double)); | 4824
683 | cudaMemcpy(h_w, w, NP*sizeof(double), cudaMemcpyDeviceToHost); | 4825
684 |                                     | 4826
685 | for (int n = 0; n < NP; n++) {     | 4827
686 |     std::cout << h_w[n] << " ";   | 4828
687 | }                                   | 4829
688 | */                                 | 4830
689 |                                     | 4831
690 | /*                                 | 4832
691 | for (int i = 0; i < nCells; i++) { | 4833
692 |     outfile << trueCounts[t*nCells + i]; | 4834
693 |     if (i % dim == 0)              | 4835
694 |         outfile << std::endl;       | 4836
695 |     else                           | 4837
696 |         outfile << " ";             | 4838
697 | }                                   | 4839
698 | */                                 | 4840
699 |                                     | 4841
700 | outfile.close();                   | 4842
701 |                                     | 4843
702 | gettimeofday (&tdr1, NULL);       | 4844
703 | timeval_subtract (&restime, &tdr1, &tdrMaster); | 4845
704 | std::cout << "Total PF time (excluding setup) " << getHRtime( | 4846
        |     restime) << std::endl;       | 4847
705 |                                     | 4848
706 | cudaFree(d_data);                 | 4849
707 | cudaFree(particles);              | 4850

```

```

4851 708     cudaFree(particles_old);
4852 709     cudaFree(w);
4853 710     cudaFree(d_neinum);
4854 711     cudaFree(d_neibmat);
4855 712     cudaFree(d_countmeans);
4856 713
4857 714     exit (EXIT_SUCCESS);
4858 715
4859 716 }
4860 717
4861 718
4862 719 /* Use the Explicit Euler integration scheme to integrate SIR model
4863     forward in time
4864 720     float h      - time step size
4865 721     float t0     - start time
4866 722     float tn     - stop time
4867 723     float * y    - current system state; a three-component vector
4868                   representing [S I R], susceptible-infected-recovered
4869 724     */
4870 725 __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
4871     * particle, int * neinum, int * neibmat, int nloc) {
4872 726
4873 727     int num_steps = floor( (tn-t0) / h );
4874 728
4875 729     float * S = particle->S;
4876 730     float * I = particle->I;
4877 731     float * R = particle->R;
4878 732     float * B = particle->B;
4879 733
4880 734     // create last state vectors
4881 735     float * S_last = (float*) malloc (nloc*sizeof(float));
4882 736     float * I_last = (float*) malloc (nloc*sizeof(float));
4883 737     float * R_last = (float*) malloc (nloc*sizeof(float));
4884 738     float * B_last = (float*) malloc (nloc*sizeof(float));
4885 739
4886 740     float R0      = particle->R0;
4887 741     float r       = particle->r;
4888 742     float B0      = R0 * r / N;
4889 743     float eta     = particle->eta;
4890 744     float berr    = particle->berr;
4891 745     float phi     = particle->phi;
4892 746
4893 747     for(int t = 0; t < num_steps; t++) {
4894 748
4895 749         for (int loc = 0; loc < nloc; loc++) {
4896 750             S_last[loc] = S[loc];
4897 751             I_last[loc] = I[loc];
4898 752             R_last[loc] = R[loc];
4899 753             B_last[loc] = B[loc];
4900 754         }

```



```

755 |                                     | 4901
756 |     for (int loc = 0; loc < nloc; loc++) { | 4902
757 |                                     | 4903
758 |         B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log( | 4904
759 |             B_last[loc])) + berr*curand_normal(&(particle-> | 4905
760 |                 randState)) ); | 4906
761 |                                     | 4907
762 |         int n = neinum[loc]; | 4908
763 |         float sphl = 1.0 - phi*( (float) n/(n+1.0) ); | 4909
764 |         float ophi = phi/(n+1.0); | 4910
765 |                                     | 4911
766 |         float nBIsun = 0.0; | 4912
767 |         for (int j = 0; j < n; j++) | 4913
768 |             nBIsun += B_last[neibmat[nloc*loc + j]-1] * I_last[ | 4914
769 |                 neibmat[nloc*loc + j]-1]; | 4915
770 |                                     | 4916
771 |         float BSI = S_last[loc]*( sphl*B_last[loc]*I_last[loc] + | 4917
772 |             ophi*nBIsun ); | 4918
773 |         float rI = r*I_last[loc]; | 4919
774 |                                     | 4920
775 |         // get derivatives | 4921
776 |         float dS = - BSI; | 4922
777 |         float dI = BSI - rI; | 4923
778 |         float dR = rI; | 4924
779 |                                     | 4925
780 |         // step forward by h | 4926
781 |         S[loc] += h*dS; | 4927
782 |         I[loc] += h*dI; | 4928
783 |         R[loc] += h*dR; | 4929
784 |                                     | 4930
785 |     } | 4931
786 | } | 4932
787 | | 4933
788 | free(S_last); | 4934
789 | free(I_last); | 4935
790 | free(R_last); | 4936
791 | free(B_last); | 4937
792 | } | 4938
793 | | 4939
794 | | 4940
795 | | 4941
796 | /* Convenience function for particle resampling process | 4942
797 | */ | 4943
798 | __device__ void copyParticle(Particle * dst, Particle * src, int | 4944
799 |     nloc) { | 4945
800 | | 4946
801 |     dst->R0 = src->R0; | 4947
802 |     dst->r = src->r; | 4948
803 |     dst->sigma = src->sigma; | 4949
804 |     dst->eta = src->eta; | 4950

```

```

4951 800     dst->berr    = src->berr;
4952 801     dst->phi     = src->phi;
4953 802
4954 803     for (int n = 0; n < nloc; n++) {
4955 804         dst->S[n]      = src->S[n];
4956 805         dst->I[n]      = src->I[n];
4957 806         dst->R[n]      = src->R[n];
4958 807         dst->B[n]      = src->B[n];
4959 808         dst->Iinit[n]   = src->Iinit[n];
4960 809     }
4961 810
4962 811 }
4963 812
4964 813 /* Convert memory size in bytes to human-readable format
4965 814 */
4966 815 std::string getHRmemsize (size_t memsize) {
4967 816
4968 817     std::stringstream ss;
4969 818     std::string valstring;
4970 819
4971 820     int kb = 1024;
4972 821     int mb = kb*1024;
4973 822     int gb = mb*1024;
4974 823
4975 824     if (memsize <= kb)
4976 825         ss << memsize << " B";
4977 826     else if (memsize > kb && memsize <= mb)
4978 827         ss << (float) memsize/ kb << " KB";
4979 828     else if (memsize > mb && memsize <= gb)
4980 829         ss << (float) memsize/ mb << " MB";
4981 830     else
4982 831         ss << (float) memsize/ gb << " GB";
4983 832
4984 833     valstring = ss.str();
4985 834
4986 835     return valstring;
4987 836
4988 837 }
4989 838
4990 839
4991 840 /* Convert time in seconds to human readable format
4992 841 */
4993 842 std::string getHRtime (float runtime) {
4994 843
4995 844     std::stringstream ss;
4996 845     std::string valstring;
4997 846
4998 847     int mt = 60;
4999 848     int ht = mt*60;
5000 849     int dt = ht*24;

```

850		5001
851	if (runtime <= mt)	5002
852	ss << runtime << " s";	5003
853	else if (runtime > mt && runtime <= ht)	5004
854	ss << runtime/mt << " m";	5005
855	else if (runtime > ht && runtime <= dt)	5006
856	ss << runtime/dt << " h";	5007
857	else	5008
858	ss << runtime/ht << " d";	5009
859		5010
860	valstring = ss.str();	5011
861		5012
862	return valstring;	5013
863		5014
864	}	5015
		5016

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

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