# 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

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

. 1	•		. •	
ea	1	cai	5,1	on

To Mom and Dad 2

# <sup>1</sup> Acknowledgements

2 Soooooo many people

# Contents

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

Dexter Barrows Contents

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

Contents	Dexter	Barrow

		RStan SIRS Code	
F Spatial Epidemics		tial Epidemics 110	3
	F.1	Spatial SIR R Function Code	4
	F.2	RStan Spatial SIR Code	5
	F.3	IF2 Spatial SIR Code	6
	F.4	CUDA IF2 Spatial Fitting Code	7

# <sub>1</sub> List of Figures

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

List of Figures Dexter Barrows

4.9	As before, the solid grey lines show the true parameter values and	40	1
4.10	the dashed grey lines show the density means	40	2
4.10	the true states, the dots show the data, the dotted line shows the		3 4
	average system behaviour, the dashed line shows the bootstrap mean,		5
	and the grey ribbon shows the centre 95th quantile of the bootstrap		6
	trajectories.	41	7
4.11	IF2 point estimate densities are shown in black and HMCMC point		8
	estimate densities are shown in grey. The vertical black lines show		9
	the true parameter values.	42	10
4.12	Fitting times for IF2 and HMCMC, in seconds. The centre box in		11
	each plot shows the centre 50th quantile, with the bold centre line		12
	showing the median	42	13
5.1	Infection count data truncated at $T=30$ . The solid line shows the		14
	true underlying system states, and the dots show those states with		15
	added observation noise. Parameters used were $R_0 = 3.0$ , $r = 0.1$ ,		16
	$\eta = .05$ , $\sigma_{proc} = 0.5$ , and additive observation noise was drawn from	4.4	17
<b>r</b> 0	$\mathcal{N}(0,10)$	44	18
5.2	Infection count data truncated at $T=30$ from Figure [5.1]. The		19
	dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points	45	20 21
5.3	Forecast produced by the IF2 / parametric bootstrapping framework.	40	21
0.0	The dotted line shows the mean estimate of the forecasts, the dark		23
	grey ribbon shows the centre 95th quantile of the true state estimates,		24
	and the lighter grey ribbon shows the centre 95th quantile of the true		25
	state estimates with added observation noise drawn from $\mathcal{N}(0,\sigma)$ .	46	26
5.4	Forecast produced by the HMCMC / bootstrapping framework with		27
	M=200 trajectories. The dotted line shows the mean estimate of		28
	the forecasts, and the grey ribbon shows the centre 95th quantile	48	29
5.5	Error growth as a function of data truncation amount. Both meth-		30
	ods used 200 bootstrap trajectories. Note that the y-axis shows the	40	31
	natural log of the averaged SSE, not the total SSE	49	32
6.1	Five cycles generated by the SIRS function. The solid line the		33
	true number of cases, dots show case counts with added observation		34
	noise. The Parameter values were $R0 = 3.0$ , $\gamma = 0.1$ , $\eta = 1$ , $\sigma = 5$ ,		35
	and 10 initial cases.	54	36
6.2	S-map applied to the data from the previous figure. The solid line		37
	shows the infection counts with observation noise form the previous		38
	plot, and the dotted line is the S-map forecast. Parameters chosen were $F = 14$ and $\theta = 3$	5.4	39
	were $E = 14$ and $\theta = 3$	54	40

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

# Chapter 1

## Introduction

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

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

Securing precise, error-free data in the midst of an outbreak can be difficult if not impossible, so uncertainty in what we observe in building mathematical models of disease spread must be accounted for from the get-go. Further, models must differentiate between natural variation in the intensity of disease spread (process error) and error in data collection (observation error) in order to accurately determine the dynamics underlying a data set.

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

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

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

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

$$\frac{dS}{dt} = -\beta IS 
\frac{dI}{dt} = \beta IS - \gamma I 
\frac{dR}{dt} = \gamma I,$$
(1.1)

where S, I, and R are the number of individuals in each compartment,  $\beta$  is the "force" of infection acting on the susceptible population, and  $\gamma$  is a recovery rate. As an outbreak progresses, individuals transition from the susceptible compartment, through the infectious compartment, then finish in the removed compartment where they no longer impact the system dynamics. Many extensions of the SIR model exist are are commonly used, such as the SEIR model in which susceptible individuals pass through an exposed class where they have been infected but are not yet themselves infectious, and the SIRS model in which individuals become susceptible again after their immunity wanes.

Combining the phenomenological and mechanistic approaches are the semi-mechanistic techniques. These methods use a model to define the expected underlying dynamics of the system, but integrate data into the model in order to refine estimates of the model parameters and produce more accurate forecasts. Typically the first step in implementing such a technique is fitting the desired model to existing data. There are many ways to do this, most of which fall into two main categories: particle filter-based (PF) methods, and Markov chain Monte Carlo-based (MCMC) methods. From there data can either be integrated into the model by refitting the model to the new longer data set, or in an "on-line" fashion in which data points can be directly 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 methodology 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 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 spatiotemporal 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 phenomenological methods we have selected the sequential locally weighted global linear maps (S-map) [Sugihara ref].

# <sup>1</sup> Chapter 2

## <sub>2</sub> Hamiltonian MCMC

#### $\mathbf{2.1}$ Intro

- 4 Markov Chain Monte Carlo (MCMC) is part of a general class of methods designed
- 5 to sample from the posterior distribution of model parameters. It is an algorithm
- 6 used when we wish to fit a model M that depends on some parameter (or more
- 7 typically vector of parameters)  $\theta$  to observed data D. MCMC works by constructing
- 8 a Markov Chain whose stationary or equilibrium distribution is used to approximate
- 9 the desired posterior distribution.

#### 10 2.2 Markov Chains

- 11 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
- 13 in Figure [2.1].

15

14 The transition probabilities can be summarized as a matrix as

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

The probability vector  $\mu(x^{(1)})$  for a state  $x^{(1)}$  can be evolved using T by evaluating

4

5

6

7

8

9

13

14

15

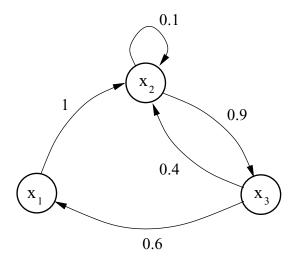


Figure 2.1: Finite state machine. (Andrieu et al., 2003)

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

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

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 10
- Aperiodic The chain must be able to explore the parameter space without 11 becoming trapped in a cycle 12

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

#### $_{\scriptscriptstyle 1}$ 2.3 Likelihood

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

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

8 In standard Maximum Likelihood approaches,  $\mathcal{L}(\theta)$  is searched to find a value of  $\theta$ 

that maximizes  $\mathcal{L}(\theta)$ , then this  $\theta$  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

11 it.

#### $_{12}$ 2.4 Prior distribution

Another significant component of MCMC is the user-specified prior distribution for

 $\theta$  or distributions for the individual components of  $\theta$  (Priors). Priors serve as a way

for us to tell the MCMC algorithm what we think consist of good values for the

6 parameters.

17 Note that if very little is known about the parameters, or we are worried about bi-

asing our estimate of the posterior, we can simply use a a wide uniform distribution.

19 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

21 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.

### 3 2.5 Proposal distribution

24 As part of the MCMC algorithm, when we find a state in the parameter space that

25 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

er 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

so it doesn't get "trapped" in a particular region of the parameter space.

2

3

4

5

6

7

8

9

14

22

23

25

26

27

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 out 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 the define the acceptance ratio, r, as

$$r = \frac{\mathcal{L}(\theta^*)p(\theta^*)q(\theta^*|\theta)}{\mathcal{L}(\theta)p(\theta)q(\theta|\theta^*)},$$
(2.4) 15

where  $\theta^*$  is the proposed sample to draw from the posterior, and  $\theta$  is the last accepted sample.

In the special case of the Metropolis Hastings variation of MCMC, the proposal 18 distribution is symmetric, meaning  $q(\theta^*|\theta) = q(\theta|\theta^*)$ , and so the acceptance ratio 19 simplifies to 20

$$r = \frac{\mathcal{L}(\theta^*)p(\theta^*)}{\mathcal{L}(\theta)p(\theta)}.$$
 (2.5) 21

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
                                                                                                                       */
        \theta^* \sim q(\cdot|\theta^{(i-1)})
\mathbf{2}
        u \sim \mathcal{U}(0,1)
        /* Evaluate acceptance ratio
                                                                                                                       */
                \underline{\mathcal{L}(\theta^*)p(\theta^*)}
        /* Step acceptance criterion
                                                                                                                       */
        if u < \min\{1, r\} then
\mathbf{5}
             \theta^{(i)} = \theta^*
6
        else
7
             \theta^{(i)} = \theta^{(i-1)}
8
   /* Samples from approximated posterior distribution
                                                                                                                       */
   Output: Chain of samples (\theta^{(1)}, \theta^{(2)}, ..., \theta^{(N)})
```

#### $_{\scriptscriptstyle 1}$ 2.7 Burn-in

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

#### $_{\scriptscriptstyle 10}$ 2.8 Thinning

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

space), but 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

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

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

$$K(r) = \frac{1}{2}r^{T}M^{-1}r,$$
(2.7) 17

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  $\theta$ .

The Hamiltonian of the system is defined as

$$H(\theta, r) = U(\theta) + K(r), \tag{2.8}$$

Where the Hamiltonian dynamics of the combined system can be simulated using the following system of ODEs. 23

$$\frac{d\theta}{dt} = M^{-1}r$$

$$\frac{dr}{dt} = -\nabla U(\theta)$$
(2.9) 25

1.

- It is tempting to try to integrate this system using the standard Euler evolution scheme, but in practice this leads to instability. Instead the "Leapfrog" scheme is used. This scheme is very similar to Euler scheme, except instead of using a fixed step size h for all evolutions, a step size of  $\varepsilon$  is used for most evolutions, with a half step size of  $\varepsilon/2$  for evolutions of  $\frac{dr}{dt}$  at the first step, and last step L. In this way the evolution steps "leapfrog" over each other while using future values from the other set of steps, leading to the scheme's name.
- The end product of the Leapfrog steps are the new proposed parameters  $(\theta^*, r^*)$ .
  These are either accepted or rejected using a mechanism similar to that of standard Metropolis-Hastings MCMC. Now, however, the acceptance ratio r is defined as

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

- where  $(\theta, r)$  are the last values in the chain.
- 15 Together, we have Algorithm [2].
- Note that the parameters  $\varepsilon$  and L have to be tuned in order to maintain stability 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 \*/ for $i = 1 : n \ do$ $\mathbf{2}$ $r(i) \leftarrow \mathcal{N}(0,1)$ 3 /\* Leapfrog initialization \*/ $\theta_0 \leftarrow \theta^{(i-1)}$ 4 $r_0 \leftarrow r - \nabla U(\theta_0) \cdot \varepsilon/2$ 5 /\* Leapfrog intermediate steps \*/ for j = 1 : L - 1 do 6 $\begin{bmatrix} \ddot{\theta_j} \leftarrow \theta_{j-1} + M^{-1}r_{j-1} \cdot \varepsilon \\ r_j \leftarrow r_{j-1} - \nabla U(\theta_j) \cdot \varepsilon \end{bmatrix}$ 7 8 /\* Leapfrog last steps \*/ $\theta^* \leftarrow \theta_{L-1} + M^{-1}r_{L-1} \cdot \varepsilon$ 9 $r^* \leftarrow \nabla U(\theta_L) \cdot \varepsilon/2 - r_{L-1}$ 10 /\* Evaluate acceptance ratio \*/ $r = \exp\left[H(\theta^{(i-1)}, r) - H(\theta^*, r^*)\right]$ 11 /\* Sample \*/ $u \sim \mathcal{U}(0,1)$ 12/\* Step acceptance criterion \*/ if $u < \min\{1, r\}$ then 13 $\theta^{(i)} = \theta^*$ 14 else 15 $\theta^{(i)} = \theta^{(i-1)}$ 16 /\* Samples from approximated posterior distribution \*/ Output: Chain of samples $(\theta^{(1)}, \theta^{(2)}, ..., \theta^{(N)})$

### $_{\scriptscriptstyle 1}$ 2.10 Fitting

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

8

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

$$\frac{dS}{dt} = -\beta IS$$

$$\frac{dI}{dt} = \beta IS - rI$$

$$\frac{dR}{dt} = rI$$
(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

```
\frac{16}{17}
            SIR ← function(Time, State, Pars) {
18
                 with(as.list(c(State, Pars)), {
19
20
                           \leftarrow R0*r/N
                                          # calculate Beta
21
22
                      BSI \leftarrow B*S*I
                                          # save product
                                          # save product
23
                      rI \ \leftarrow r {*} I
24
                                          # change in Susceptible people
25
                      dI = BSI - rI
                                          # change in Infected people
26
                      dR = rI
                                          # change in Removed (recovered people)
27
28
                      return(list(c(dS, dI, dR)))
29
30
                 })
31
32
            }
33
```

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

7

14

15

19

23

24

25

26

27

28

35

36

45

```
pars \leftarrow c(R0 \leftarrow 3.0, # new infected people per infected person 2 r \leftarrow 0.1, # recovery rate N \leftarrow 500) # population size \frac{1}{2}
```

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

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

```
y_i = y_i + c(S = 495, I = 5, R = 0) # initial conditions
```

The ode() function is called as

```
odeout \leftarrow ode(y_ini, times, SIR, pars) \begin{bmatrix} 20 \\ 21 \\ 21 \end{bmatrix}
```

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

```
set.seed(1001) # set RNG seed for reproducibility
sigma \leftarrow 5 # observation error standard deviation
infec_counts_raw \leftarrow odeout[,3] + rnorm(101, 0, sigma)
infec_counts \leftarrow 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

we obtain Figure [2.2].

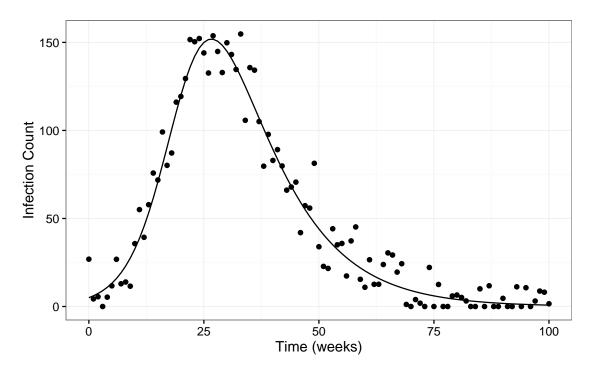


Figure 2.2: True SIR ODE solution infected counts, and with added observation noise

- The Hamiltonian MCMC model fitting was done using Stan (http://mc-stan.
- org/), a program written in C++ that does Baysian statistical inference using Hamil-
- 3 tonian MCMC. Stan's R interface (http://mc-stan.org/interfaces/rstan.html)
- 4 was used to ease implementation.
- In order to use an Explicit Euler-like stepping method in the later Stan model (both
- for speed and for integration method homogeneity with other methods against which
- 7 HMCMC was compared), the synthetic observation counts were treated as weekly
- 8 observations in which the counts on the other six days of the week were unobserved.
- For computational and organizational simplicity, these vales were set to -1 (all valid

observations are non-negative). This is done in R using

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

11

24

26

27

28

29

41

42

43

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

Now we call the Stan fitting function

```
12
stan_options \leftarrow list(
                                                # number of chains
                              chains = 4,
                                                                                     13
                                       = 2000, # iterations per chain
                              iter
                                                                                     14
                              warmup = 1000, # warmup interations
                                                                                     15
                                       = 2 )
                                                # thinning number
                                                                                     16
                           "d_sirode_euler.stan",
\texttt{fit} \leftarrow \texttt{stan(} \texttt{file}
                                                                                     17
               data
                         = sir_data,
                                                                                     18
                         = stan_options$chains,
                                                                                     19
               chains
                         = stan_options$iter,
                                                                                     20
               iter
                         = stan_options$warmup,
                                                                                     21
               warmup
               thin
                           stan_options$thin )
                                                                                     \frac{22}{23}
```

which fits the model in the file <code>d\_sirode\_euler.stan</code> to the data passed in through <code>sir\_data</code>. 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

```
30
data {
                                                                            31
                                                                           32
             <lower=1>
                                    // total integration steps
    int
                           Τ;
                                                                           33
    real
                           y[T];
                                    // observed number of cases
                                                                           34
    int
             <lower=1>
                           N;
                                    // population size
                                                                           35
    real
                           h;
                                    // step size
                                                                           36
                                                                           37
}
                                                                           38
```

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

```
parameters {
2
3 real <lower=0, upper=10> sigma; // observation error 47
```

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

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

```
15
            model {
16
17
18
                 real S[T];
19
                 real I[T];
                 real R[T];
20
21
                 S[1] \leftarrow y0[1];
22
                 I[1] \leftarrow y0[2];
23
                 R[1] \leftarrow y0[3];
24
25
                 y[1] ~ normal(y0[2], sigma);
26
27
                 for (t in 2:T) {
28
29
                      S[t] \leftarrow S[t-1] + h*( - S[t-1]*I[t-1]*R0*r/N );
30
                      I[t] \leftarrow I[t-1] + h*(S[t-1]*I[t-1]*R0*r/N - I[t-1]*r);
31
                      R[t] \leftarrow R[t-1] + h*(I[t-1]*r);
32
33
                      if (y[t] > 0) {
34
                          y[t] ~ normal( I[t], sigma );
35
                      }
36
37
                 }
38
39
                 y0[1] ~ normal(N - y[1], sigma);
40
                 y0[2] normal(y[1], sigma);
41
42
                                  lognormal(1,1);
                 theta[1]
43
                                ~ lognormal(1,1);
                 theta[2]
44
                                ~ lognormal(1,1);
                 sigma
45
46
            }
48
```

2

3

4

5

6

7

8

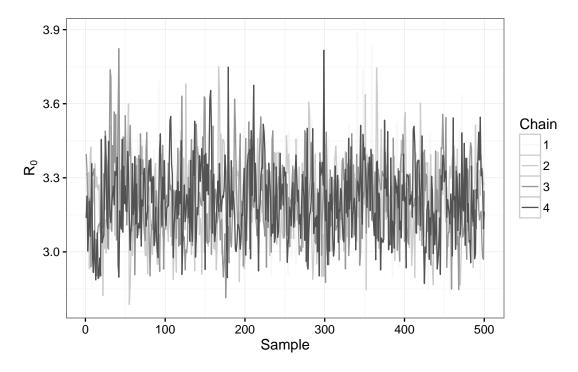


Figure 2.3: Traceplot of samples drawn for parameter  $R_0$ , excluding warmup

Examining the traceplot for 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].

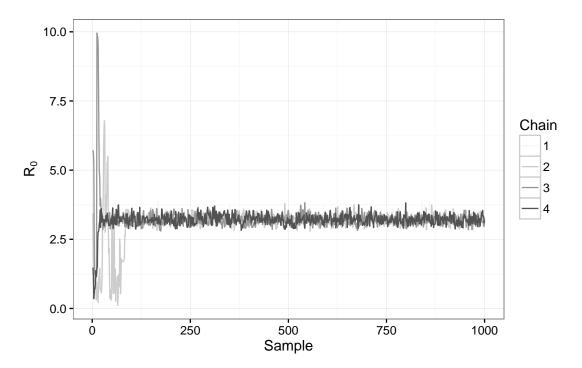


Figure 2.4: Traceplot of samples drawn for parameter  $R_0$ , including warmup.

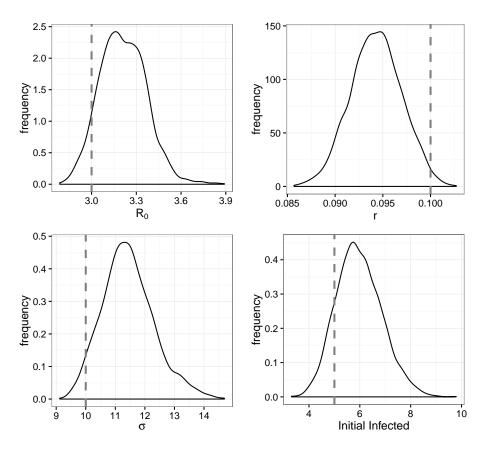


Figure 2.5: Kernel density estimates produced by Stan

# Chapter 3

# 2 Iterated Filtering

#### 3 3.1 Intro

- 4 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  $\theta$
- 6 given observed data D. Instead of constructing a Markov chain and approximating
- 7 its stationary distribution, a cohort of "particles" are used to move through the data
- s in an on-line (sequential) fashion with the cohort being culled of poorly-performing
- 9 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
- 11 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
- 13 Importance Resampling (SIR) particle filter.

#### $_{14}$ 3.2 Formulation

- Particle filters, also called Sequential Monte-Carlo (SMC) or bootstrap filters, fea-
- ture similar core functionality as the venerable Kalman Filter. As the algorithm
- 17 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
- 19 selections, track how closely these predictions approximate the new observed value,
- 20 and update the current cohort appropriately.
- 21 Two separate functions are used to simulate the evolution and observation processes.
- 22 The "true" state evolution is specified by

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

20

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

And the observation process by

$$Y_t \sim f_2(X_t, \theta). \tag{3.2}$$

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

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.

#### Algorithm 3.3

Now we will formalize the particle filter.

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.

The algorithm for a Sequential Importance Resampling particle is shown in Algo-19 rithm [3].

#### **Algorithm 3:** SIR particle filter

```
/* Select a starting point
  Input: Observations D = y_1, y_2, ..., y_T, initial particle distribution P_0 of size
               J
  /* Setup
                                                                                                   */
1 Initialize particle cohort by sampling (p^{(1)}, p^{(2)}, ..., p^{(J)}) from P_0
2 for t = 1 : T do
       /* Evolve
                                                                                                   */
       for j = 1:J do
3
       X_t^{(j)} \leftarrow f_1(X_{t-1}^{(j)}, \theta^{(j)})
       /* Weight
                                                                                                   */
       for j = 1:J do
5
       w^{(j)} \leftarrow P(y_t | X_t^{(j)}, \theta^{(j)}) = f_2(X_t^{(j)}, \theta^{(j)})
       /* Normalize
                                                                                                   */
      for j = 1:J do
7
       /* Resample
                                                                                                   */
      p^{(1:J)} \leftarrow \text{sample}(p^{(1:J)}, \text{prob} = w, \text{replace} = true)
  /* Samples from approximated posterior distribution
                                                                                                   */
  Output: Cohort of posterior samples (\theta^{(1)}, \theta^{(2)}, ..., \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

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

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.

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

#### 5 3.6 IF2

- 6 The successor to Iterated Filtering 1 (reference), Iterated Filtering 2 is simpler,
- 7 faster, and demonstrated better convergence toward maximum likelihood (refer-
- 8 ence). The core concept involves a two-pronged approach. First, Data cloning is
- 9 used to allow more time for the parameter stochastic process means to converge to
- maximum likelihood, and frequent cooled perturbation of the particle parameters
- allow better exploration of the parameter space while still allowing convergence to
- 12 good point estimates.
- 13 It is worth noting that IF2 is not designed to estimate the full posterior distribution,
- but in practice can be used to do so within reason. Further, IF2 thwarts the problem
- of particle collapse by keeping at least some perturbation in the system at all times.
- 16 It is important to note that while true particle collapse will not occur, there is still
- 17 risk of a pseudo-collapse in which all particles will be extremely close to one another
- 18 so as to be virtually indistinguishable. However this will only occur with the use of
- overly-aggressive cooling strategies or by specifying an excessive number of passes
- 20 through the data.
- 21 An important new quantity is the particle perturbation density denoted  $h(\theta|,\sigma)$ .
- Typically this is multi-normal with  $\sigma$  being a vector of variances proportional to the
- 23 expected values of  $\theta$ . In practice the proportionality can be derived from current
- 24 means or specified ahead of time. Further, these intensities must decrease over time.
- 25 This can be done via exponential or geometric cooling, a decreasing step function,
- 26 a combination of these, or though some other similar scheme.
- 27 The algorithm for IF2 can be seen in Algorithm [4].

28

#### Algorithm 4: IF2

```
/* Select a starting point
                                                                                                                */
    Input: Observations D = y_1, y_2, ..., y_T, initial particle distribution P_0 of size
                  J, decreasing sequence of perturbation intensity vectors
                  \sigma_1, \sigma_2, ..., \sigma_M
    /* Setup
                                                                                                                */
 1 Initialize particle cohort by sampling (p^{(1)}, p^{(2)}, ..., p^{(J)}) from P_0
    /* Particle seeding distribution
                                                                                                                */
 \mathbf{2} \ \Theta \leftarrow P_0
 3 for m = 1 : M do
         /* Pass perturbation
                                                                                                                */
         for j = 1:J do
 4
          p^{(j)} \sim h(\Theta^{(j)}, \sigma_m)
         for t = 1 : T do
 6
              for i = 1:J do
 7
                   /* Iteration perturbation
                                                                                                                */
                  p^{(j)} \sim h(p^{(j)}, \sigma_m)
 8
                  /* Evolve X_t^{(j)} \leftarrow f_1\big(X_{t-1}^{(j)}, \theta^{(j)}\big)
 9
                  /* Weight w^{(j)} \leftarrow P(y_t|X_t^{(j)},\theta^{(j)}) = f_2(X_t^{(j)},\theta^{(j)})
                                                                                                                */
10
              /* Normalize
                                                                                                                */
              for j = 1:J do
11
               w^{(j)} \leftarrow w^{(j)} / \sum_{1}^{J} w^{(j)}
12
              /* Resample
                                                                                                                */
             p^{(1:J)} \leftarrow \text{sample}(p^{(1:J)}, \text{prob} = w, \text{replace} = true)
13
         /* Collect particles for next pass
                                                                                                                */
         for j = 1 : J \text{ do}
14
             \Theta^{(j)} \leftarrow p^{(j)}
15
    /* Samples from approximated posterior distribution
                                                                                                                */
    Output: Cohort of posterior samples (\theta^{(1)}, \theta^{(2)}, ..., \theta^{(J)})
```

### $_{\scriptscriptstyle 1}$ 3.7 Fitting

7

- 2 Here we will examine a test case in which IF2 will be used to fit a Susceptible-
- 3 Infected-Removed (SIR) epidemic model to mock infectious count data.
- 4 The synthetic data was produced by taking the solution to a basic SIR ODE model,
- 5 sampling it at regular intervals, and perturbing those values by adding in observation
- 6 noise. The SIR model used was

$$\frac{dS}{dt} = -\beta IS 
\frac{dI}{dt} = \beta IS - rI 
\frac{dR}{dt} = rI$$
(3.4)

s where S is the number of individuals susceptible to infection, I is the number of

- 9 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

```
15
            SIR ← function(Time, State, Pars) {
16
17
                 with(as.list(c(State, Pars)), {
18
19

← R0*r/N

                                         # calculate Beta
20
                      BSI \leftarrow B*S*I
                                         # save product
21
                          \leftarrow \; r\!*\! \; I
                                         # save product
22
23
24
                      dS = -BSI
                                         # change in Susceptible people
                      dI = BSI - rI
                                         # change in Infected people
25
                      dR = rI
                                         # change in Removed (recovered people)
26
27
                      return(list(c(dS, dI, dR)))
28
29
                 })
30
31
            }
33
```

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

5

6

12

17

18

19

20

21

22

29

30

41

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

The ode() function is called as

```
odeout \leftarrow ode(y = true_init_cond, times = 0:(T-1), func = SIR, parms = true_pars) \frac{13}{14}
```

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

```
set.seed(1001) # set RNG seed for reproducibility
sigma \leftarrow 10 # observation error standard deviation
infec_counts_raw \leftarrow odeout[,3] + rnorm(101, 0, sigma)
infec_counts \leftarrow 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

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="/")) 44 45 46
```

Rinit")

5

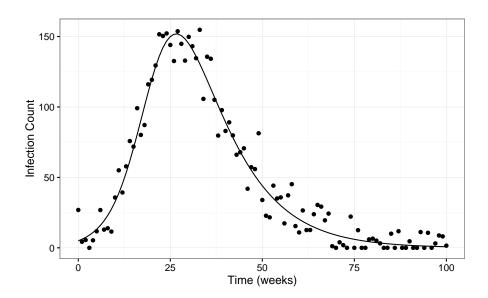


Figure 3.1: True SIR ODE solution infected counts, and with added observation noise.  $\,$ 

```
Then run and packed into a data frame using

paramdata ← data.frame(if2(infec_counts[1:Tlim], Tlim, N))
colnames(paramdata) ← c("R0", "r", "sigma", "Sinit", "Iinit", "
```

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

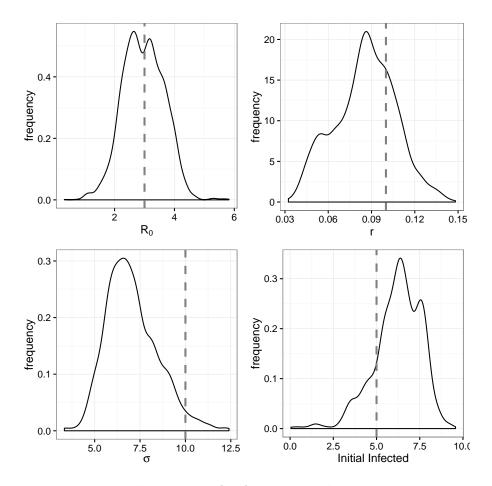


Figure 3.2: Kernel estimates for four essential system parameters. True values are indicated by solid vertical lines, sample means by dashed lines.

## Chapter 4

## <sup>2</sup> Parameter Fitting

### $_3$ 4.1 Fitting Setup

10

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

$$\frac{dS}{dt} = -\beta SI$$

$$\frac{dI}{dt} = \beta SI - \gamma I$$

$$\frac{dR}{dt} = \gamma I,$$
(4.1)

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

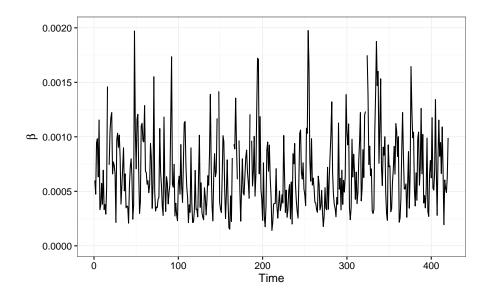


Figure 4.1: Simulated geometric autoregressive process show in Equation [4.2].

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

$$X_{t+1} = c + \rho X_t + \epsilon_t, \tag{4.3}$$

where  $\epsilon_t$  is normally distributed noise with mean 0 and standard deviation  $\sigma_E$ . This process has a theoretical expected mean of  $\mu = c/(1-\rho)$  and variance  $\sigma = \sigma_E^2/(1-\rho^2)$ . If we choose  $\eta = 0.5$ , the resulting log-normal distribution has a mean of  $6.80 \times 10^{-4}$  and standard deviation of  $4.46 \times 10^{-4}$ .

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  $6.92 \times 10^{-4}$  and standard the deviation to be  $3.99 \times 10^{-4}$ , 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].

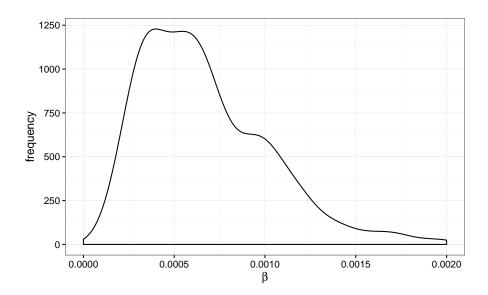


Figure 4.2: Density plot of values shown if Figure [4.1].

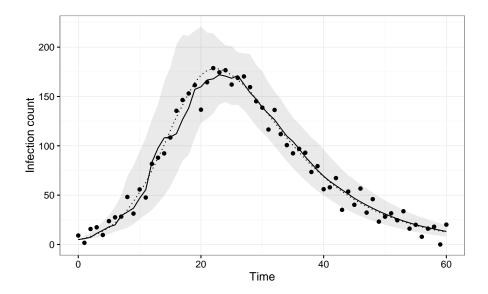


Figure 4.3: Stochastic SIR model simulated using an explicit Euler stepping scheme. The solid line is a single random trajectory, the dots show the data points obtained by adding in observation error defined as  $\epsilon_{obvs} = \mathcal{N}(0, 10)$ , and the grey ribbon is centre 95th quantile from 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  $\mu$ , where n is the number of samples. Then the Law of Large Numbers says that  $\hat{\mu}_n \to \mu$  as  $n \to \infty$ . Further, the Central Limit Theorem says that the error  $\hat{\mu}_n - \mu$  should shrink with number of samples such that  $\sqrt{n}(\hat{\mu}_n - \mu) \to \mathcal{N}(0, \sigma^2)$  as  $n \to \infty$ , where  $\sigma^2$  is the variance of the samples drawn.

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

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

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

$$\Theta^* V(\Theta^*)^T \tag{4.5}$$

where  $\Theta^*$  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

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

where P is the number of particles.

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

- 1 the expected number of IF2 particles needed to obtain the same value. This was
- 2 used as a starting value to "titrate" the IF2 iterations to the same point.
- 3 The resulting values were 1000 HMCMC warm-up iterations with 1000 samples
- 4 drawn post-warm-up, and 2500 IF2 particles sent through 50 passes, each method
- 5 giving an approximate MCSE of 0.0065.

### 6 4.3 IF2 Fitting

- 7 Now we will use an implementation of the IF2 algorithm to attempt to fit the
- 8 stochastic SIR model to the previous data. The goal here is just parameter in-
- 9 ference, but since IF2 works by applying a series on particle filters we essentially
- 10 get the average system state estimates for a very small additional computational
- 11 cost. Hence, we will will also look at that estimated behaviour in addition the the
- 12 parameter estimates.
- 13 The code used here is a mix of R and C++ implemented using RCpp. The fitting
- 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
- 16 starting with 0.
- 17 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.

2

3

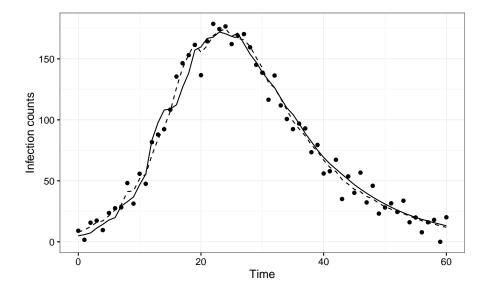


Figure 4.5: True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).

		IF2		HMCMC	
Name	True	Fit	Error	Fit	Error
$R_0$	3.0	3.27	$9.08 \times 10^{-2}$	3.12	$1.05 \times 10^{-1}$
r	$10^{-1}$	$1.04 \times 10^{-1}$	$3.61\times10^{-2}$	$9.99 \times 10^{-2}$	$-7.56 \times 10^{-4}$
Initial Infected	5	7.90	$5.80\times10^{-1}$	6.64	$3.28\times10^{-1}$
$\sigma$	10	8.84	$-1.15 \times 10^{-1}$	8.5	$-1.50 \times 10^{-1}$
$\eta$	$5 \times 10^{-1}$	$5.87\times10^{-1}$	$1.73\times10^{-1}$	$4.57\times10^{-1}$	$-8.27 \times 10^{-2}$
$arepsilon_{err}$	$5 \times 10^{-1}$	$1.63\times10^{-1}$	$-6.73 \times 10^{-1}$	$1.60 \times 10^{-1}$	$-6.80 \times 10^{-1}$

Figure 4.4: Fitting errors.

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

## 1 4.4 IF2 Convergence

- 2 Since IF2 is an iterative algorithm where each pass through he data is expected to
- 3 push the parameter estimates towards the MLE, we can see the evolution of these
- 4 estimates as a function of the pass number. Plots showing evolution of the mean
- 5 estimates are shown if Figure [4.6] for the six most critical parameters.
- 6 Similarly, we can look at the evolution of the standard deviations of the parameter
- 7 estimates from the particle swarm as a function of the pass number, shown in Figure
- 8 [4.7].
- 9 As expected there is a downward trend in all plots, with a very strong trend in all
- 10 but two of them.

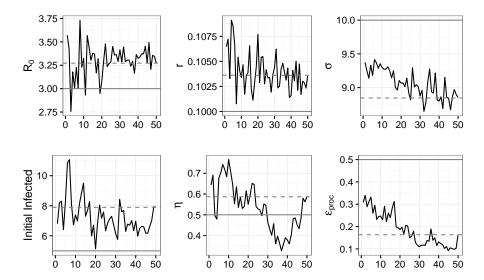


Figure 4.6: The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.

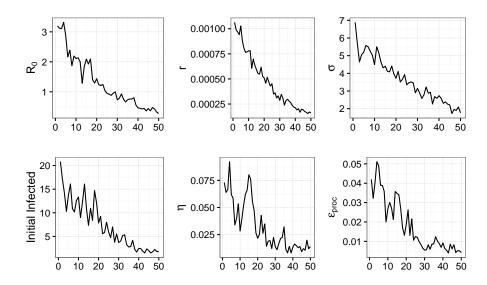


Figure 4.7: The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.

## 1 4.5 IF2 Densities

- 2 Of diagnostic importance are the densities of the parameter estimates given by the
- 3 final parameter swarm. These are shown if Figure [4.8].
- 4 It is worth noting that the IF2 parameters chosen were in part chosen so as to
- 5 not artificially narrow these densities; a more aggressive cooling schedule and/or
- an increased number of passes would have resulted in much narrower densities, and
- 7 indeed have the potential to collapse them to point estimates.

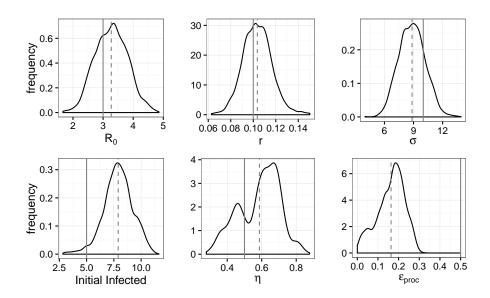


Figure 4.8: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

## 4.6 HMCMC Fitting

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

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

#### 4.7 HMCMC Densities

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

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

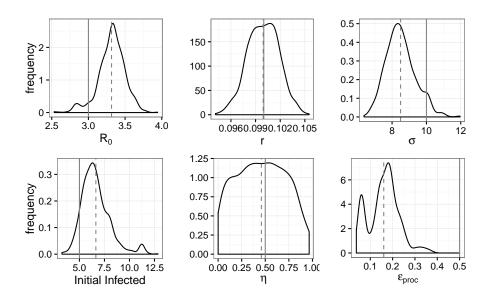


Figure 4.9: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

## 1 4.8 HMCMC and Bootstrapping

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

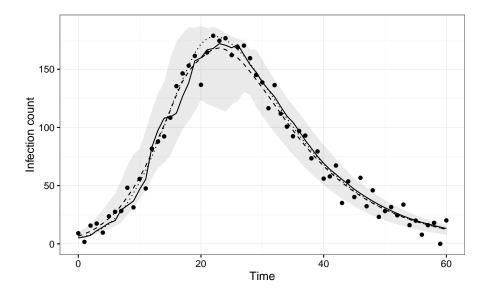


Figure 4.10: Result from 100 HMCMC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.

## 4.9 Multi-trajectory Parameter Estimation

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

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

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

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

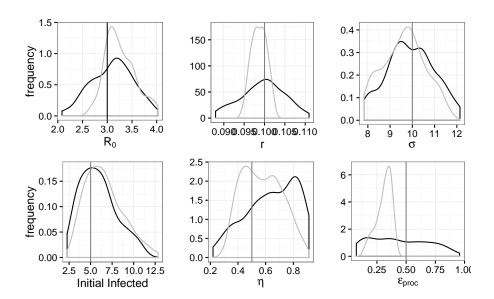


Figure 4.11: IF2 point estimate densities are shown in black and HMCMC point estimate densities are shown in grey. The vertical black lines show the true parameter values.

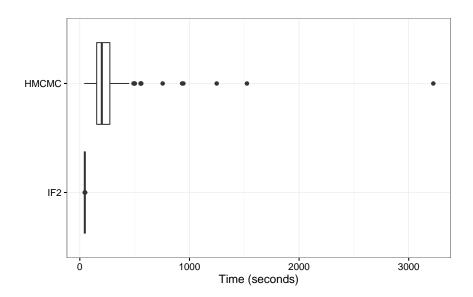


Figure 4.12: Fitting times for IF2 and HMCMC, in seconds. The centre box in each plot shows the centre 50th quantile, with the bold centre line showing the median.

Chapter 5				
Forecasting Frameworks	2			
5.1 Data Setup	3			
This section will focus on taking the stochastic SIR model from the previous section,				
truncating the synthetic data output from realizations of that model, and seeing how				
well IF2 and HMCMC can reconstruct out-of-sample forecasts.	6			
An example of a simulated system with truncated data can be seen in Figure				
[5.1].	8			
In essence we want to be able to give either IF2 of HMCMC only the data points				
and have it reconstruct the entirety of the true system states.				

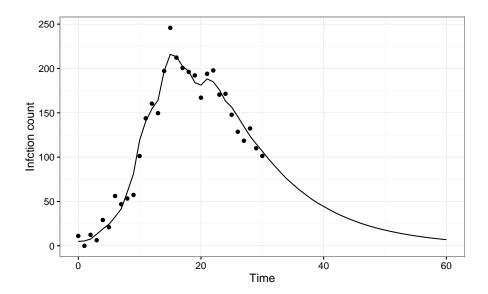


Figure 5.1: Infection count data truncated at T=30. The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were  $R_0 = 3.0$ , r = 0.1,  $\eta = .05$ ,  $\sigma_{proc} = 0.5$ , and additive observation noise was drawn from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

#### <sub>1</sub> 5.2 IF2

- 2 For IF2, we will take advantage of the fact that the particle filter will produce state
- 3 estimates for every datum in the time series given to it, as well as producing param-
- 4 eter maximum likelihood point estimates. Both of these sources of information will
- 5 be used to produce forecasts by parametric bootstrapping using the final parameter
- 6 estimates from the particle swarm after the last IF2 pass, then using the newly
- 7 generated parameter sets along with the system state point estimates from the first
- s fitting to simulate the systems forward into he future.
- 9 We will truncate the data at half the original time series length (to T=30), and fit
- the model as previously described.
- 11 First, we can see the state estimates for each time point produced by the last IF2
- 12 pass in Figure [5.2].
- 13 Recall that IF2 is not trying to generate parameter estimation densities, but rather
- 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
- 16 method, parametric bootstrapping.

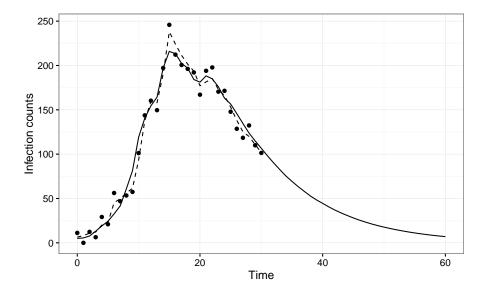


Figure 5.2: Infection count data truncated at T = 30 from Figure [5.1]. The dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points.

#### 5.2.1 Parametric Bootstrapping

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

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

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

#### 5.2.2 IF2 Forecasts

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

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

#### **Algorithm 5:** Parametric Bootstrap

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

/\* Generate artificial data sets \*/

 $\begin{array}{c|c} \mathbf{2} \ \mathbf{for} \ i = 1 : M \ \mathbf{do} \\ \mathbf{3} \ \mid \ D_i \leftarrow S(\theta^*) \end{array}$ 

. \_.

/\* Fit to new data sets \*/

4 for i = 1 : M do

 $\boldsymbol{5} \quad \theta_i \leftarrow IF2(D_i)$ 

Output: Distribution samples  $\theta_1, \theta_2, ..., \theta_M$ 

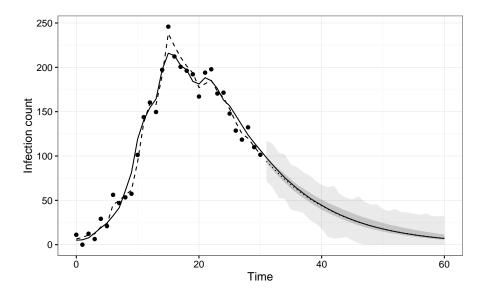


Figure 5.3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the centre 95th quantile of the true state estimates, and the lighter grey ribbon shows the centre 95th quantile of the true state estimates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

## **5.3** HMCMC

For HMCMC we can use a simpler bootstrapping approach. We do not get state estimates directly from the RStan fitting due to the way we implemented the model, but we can construct them using the process noise latent variables. Once we've done this we can forward simulate the system from the state estimate into the future.	2 3 4 5
As before we fit the stochastic SIR model to the partial data, but now perform bootstrapping as described above, and obtain the plot in Figure [5.4].	6 7
And as before we can evaluate the averaged SSE of the forecast for the data shown, giving $\overline{SSE} = 20.27$ .	8

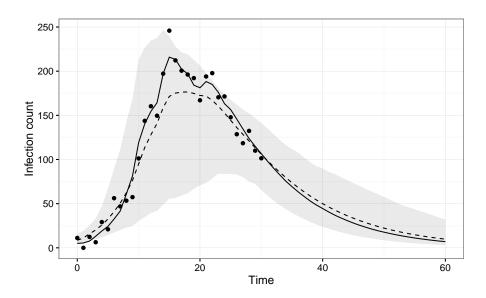


Figure 5.4: Forecast produced by the HMCMC / bootstrapping framework with M=200 trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the centre 95th quantile.

#### <sub>1</sub> 5.4 Truncation vs. Error

- 2 Of course the above mini-comparison only shows one truncation value for one trajec-
- 3 tory. Really, we need to know how each method performs on average given different
- 4 trajectories and truncation amounts. In effect we wish to "starve" each method
- 5 of data and see how poor the estimates become with each successive data point
- 6 loss.
- 7 Using each method, we can fit the stochastic SIR model to successively smaller time
- 8 series to see the effect of truncation on forecast averaged SSE. This was performed
- 9 with 10 new trajectories drawn for each of the desired lengths. The results are
- shown in Figure [5.5].
- 11 IF2 and HMCMC perform very closely, with IF2 maintaining a small advantage up
- to a truncation of about 25-30 data points.
- 13 Since the parametric bootstrapping approach used by IF2 requires a significant
- 14 number of additional fits, its computational cost is significantly higher than the
- is simpler bootstrapping approach used by the HMCMC framework, about 35.5x as
- 16 expensive. However the now much longer running time can somewhat alleviated
- by parallelizing the parametric bootstrapping process; as each of the parametric
- bootstrap fittings in entirely independent, this can be done without a great deal of

2

3

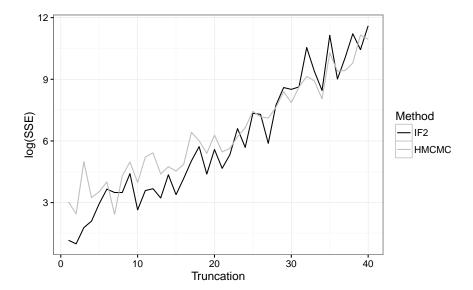


Figure 5.5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

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

## Chapter 6

## <sub>2</sub> S-map and SIRS

### 3 **6.1** S-maps

- 4 A family of forecasting methods that shy away from the mechanistic model-based
- approaches outlined in the previous sections have been developed by Sugihara (refer-
- 6 ences) over the last several decades. As these methods do not include a mechanistic
- 7 model in their forecasting process, they also do not attempt to perform parameter
- 8 inference. Instead they attempt to reconstruct the underlying dynamical process as
- 9 a weighted linear model from a time series.
- 10 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
- 13 operating is the product of linear system dynamics, and in fact was developed to
- 14 accommodate non-linear dynamics.
- The S-map works by first constructing a time series embedding of length E, known
- 16 as the library and denoted  $\{x_i\}$ . Consider a time series of length T denoted
- $x_1, x_2, ..., x_T$ . Each element in the time series with indices in the range E, E+1, ..., T
- will have a corresponding entry in the library such that a given element  $x_t$  will cor-
- respond to a library vector of the form  $\mathbf{x_i} = (x_t, x_{t-1}, ..., x_{t-E+1})$ . 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
- 23 itself), is generated using an exponentially weighted function of the library  $\{x_i\}$ ,
- predictions  $\{y_i\}$ , and predictor vector  $\mathbf{x_t}$ .
- This function is defined as follows:

3

4

5

6

8

10

11

12

16

17

18

19

20

First construct a matrix A and vector b defined as

$$A(i,j) = w(||\mathbf{x_i} - \mathbf{x_t}||)\mathbf{x_i}(j)$$

$$b(i) = w(||\mathbf{x_i} - \mathbf{x_t}||)y_i$$
(6.1) 2

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

The weighting function w is defined as

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

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

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

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

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

## 6.2 S-map Algorithm

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

```
Algorithm 6: S-map
  /* Select a starting point
                                                                                         */
  Input: Time series x_1, x_2, ..., x_T, embedding dimension E, distance
             penalization \theta, forecast length L, predictor vector \mathbf{x_t}
  /* Construct library \{x_i\}
                                                                                         */
1 for i = E : T do
\mathbf{z} \mid \mathbf{x_i} = (x_i, x_{i-1}, ..., x_{i-E-1})
  \slash Construct mapping from library vectors to predictions
                                                                                         */
3 for i = 1 : (T_E + 1) do
   for j = 1 : E do
       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
                                                                                         */
s 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}
```

3

4

5

6

7

8

9

11

18

19

#### SIRS Model 6.3

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

With that in mind, the only fair way to compare the efficacy of s-mapping to IF2 or Hamiltonian MCMC is to generate data from a SIRS model with a seasonal component, and have all methods operate on the resulting time series.

The basic skeleton of the SIRS model is similar to the stochastic SIR model described 10 previously. The deterministic ODE component of the model is as follows.

$$\frac{dS}{dt} = -\Gamma(t)\beta SI + \eta R$$

$$\frac{dI}{dt} = \Gamma(t)\beta SI - \gamma I$$

$$\frac{dR}{dt} = \gamma I - \eta R,$$
(6.4) 12

There are two new features here. We have a re-susceptibility rate  $\eta$  through which 13 people become able to be reinfected, and a seasonality factor  $\Gamma$  defined as 14

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

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

As before,  $\beta$  is allowed to walk restricted by a geometric mean, described by 20

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

When simulated for the equivalent of 5 years (260 weeks), and adding noise drawn 22 from  $\mathcal{N}(0, \sigma)$  we obtain Figure [6.1]. 23

We can see how the S-map can reconstruct the next cycle in the time series in Figure 24 [6.2].25

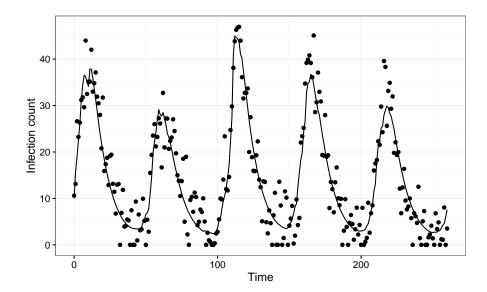


Figure 6.1: Five cycles generated by the SIRS function. The solid line the true number of cases, dots show case counts with added observation noise. The Parameter values were  $R0=3.0,\,\gamma=0.1,\,\eta=1,\,\sigma=5,$  and 10 initial cases.

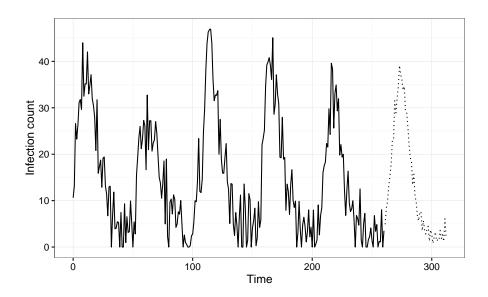


Figure 6.2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise form the previous plot, and the dotted line is the S-map forecast. Parameters chosen were E=14 and  $\theta=3$ .

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  $1.49 \times 10^{-1}$  seconds, for IF2 it is about  $4.70 \times 10^{4}$ , and for HMCMC it is about  $9.20 \times 10^{3}$ . This is a speed-up of over 316,000x compared to IF2 and over 61,800x compared to HMCMC.

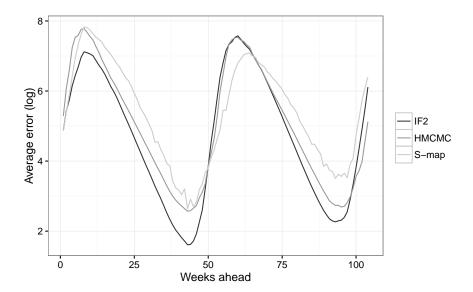


Figure 6.3: Error as a function of forecast length.

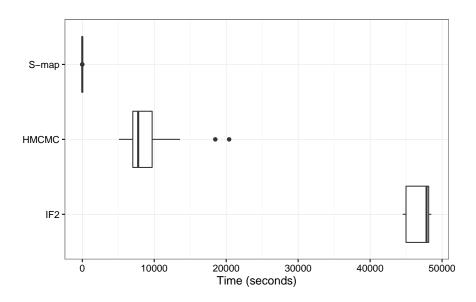


Figure 6.4: Runtimes for producing SIRS forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

## Chapter 7

## Spatial Epidemics

### 7.1 Spatial SIR

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

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

$$\frac{dS_i}{dt} = -\left(1 - \phi \frac{N_i}{N_i + 1}\right) \beta_i S_i I_i - \left(\frac{\phi}{N_i + 1}\right) S_i \sum_{j=0}^{N_i} \beta_j I_j$$

$$\frac{dI_i}{dt} = \left(1 - \phi \frac{N_i}{N_i + 1}\right) \beta_i S_i I_i + \left(\frac{\phi}{N_i + 1}\right) S_i \sum_{j=0}^{N_i} \beta_j I_j - \gamma I$$

$$\frac{dR_i}{dt} = \gamma I,$$
(7.1) 1:

Neighbours for a particular location are numbered  $j = 1, ..., N_i$ . 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.

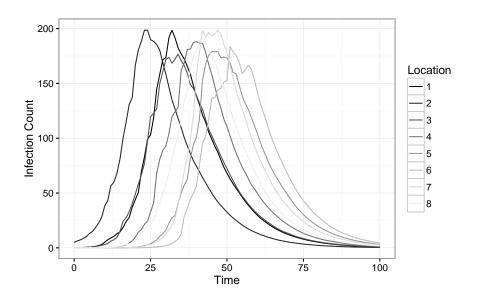


Figure 7.1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were  $R_0 = 3.0$ ,  $\gamma = 0.1$ ,  $\eta = 0.5$ ,  $\sigma_{err} = 0.5$ , and  $\phi = 0.5$ .

1 As before we let  $\beta$  embark on a geometric random walk defined as

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

- Note that as  $\beta$  is a state variable, each location has its own stochastic process driving
- the evolution of its  $\beta$  state.
- 5 If we imagine a circular topology in which each of 10 locations is connected to
- exactly two neighbours (i.e. location 1 is connected to locations N and 2, location 2
- 7 is connected to locations 1 and 3, etc.), and we start each location with completely
- s susceptible populations except for a handful of infected individuals in one of the
- 9 locations, we obtain a plot of the outbreak progression in Figure [7.1].
- 10 If we add noise to the data from Figure [7.1], we obtain Figure [7.2].

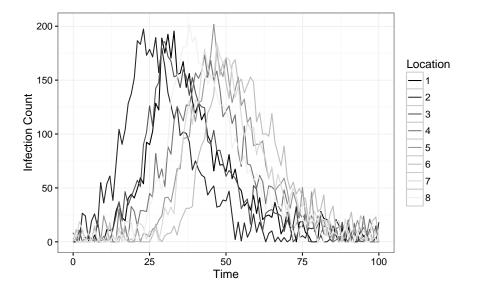


Figure 7.2: Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from  $\mathcal{N}(0, 10)$ .

## 7.2 Dewdrop Regression

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

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

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

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

#### 5 7.3 Spatial Model Forecasting

- 6 In order to compare the forecasting efficacy of Dewdrop Regression with S-mapping
- 7 against IF2 and HMCMC, we generated 20 independent spatial data sets up to time
- 8 T=50 weeks in each of L=10 locations and forecasted 10 weeks into the future.
- 9 Forecasts were compared to that of the true model evolution, and the average SSE
- 10 for each week ahead in the forecast were computed. The number of bootstrapping
- $_{11}$  trajectories used by IF2 and HMCMC was reduced from 200 to 50 to curtail running
- 12 times.
- 13 The results are shown in Figure [7.3].
- 14 The results show a clear delineation in forecast fidelity between methods. IF2 main-
- tains an advantage regardless of how long the forecast produced. Interestingly, Dew-
- drop Regression with S-mapping performs almost as well as IF2, and outperforms
- 17 HMCMC. HMCMC lags behind both methods by a healthy margin.
- 18 If we examine the runtimes for each forecast framework, we obtain the data in Figure [7.4].
- 20 As before, the S-map with Dewdrop Regression runs faster than the other two
- 21 methods with a huge margin. It is again hard to see exactly how large the margin
- 22 is from the figure due to the scale, but we can examine the average values: the
- 23 average running time for S-mapping with Dewdrop Regression was about 249 sec-
- onds, whereas the average times for IF2 and HMCMC were about  $2.90 \times 10^4$  and
- $3.88 \times 10^4$ , respectively. This is a speed-up of just over 116x over IF2 and 156x over
- 26 HMCMC.
- 27 Considering how well S-mapping performed with regards to forecast error, it shows a
- 28 significant advantage over HMCMC in particular it outperforms it in both forecast
- 29 error and running times.

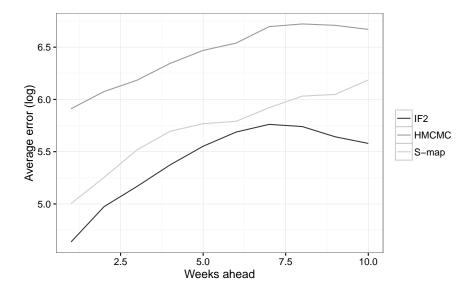


Figure 7.3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.

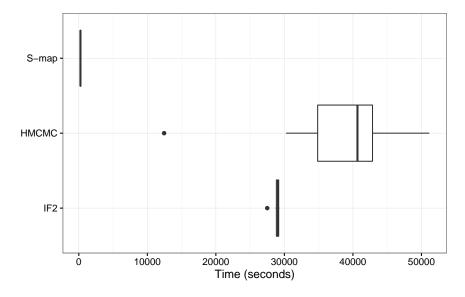


Figure 7.4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

## Chapter 8

# Discussion and Future Directions

### 4 8.1 Parallel and Distributed Computing

- Whenever running times are discussed, we must consider the current computing
- 6 landscape and hardware boundaries. In 1965, Intel co-founder Gordon E. Moore
- 7 published a paper in which he observed that the number of transistors per unit area
- 8 in integrated circuits double roughly every year. The consequence of this growth
- 9 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.).
- 13 Recently, transistor growth has begin to falter. This is due to several factors. The
- 14 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
- 17 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.
- 20 To compensate for these limitations, chip manufacturers have instead redesigned
- 21 the internal chip structures to consists to smaller "cores" within a single CPU die.
- 22 The resulting processing power per processor then stays on track with Moore's Law,
- 23 but keeps the clock speeds of each individual core, and consequently the thermal
- dissipation requirement, under control.
- 25 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 musts 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 HM-CMC 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

9

- 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. And 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.

### Appendix A

### <sub>2</sub> Hamiltonian MCMC

#### 3 A.1 Full R code

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

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

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

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

32

```
color = factor(chains, labels = 1:stan_
                                     options$chains))) +
                                                                                          2
                     labs(x = "Sample", y = expression(R[0]), color = "
                                                                                          3
                         Chain") +
                                                                                          4
                     scale_color_brewer(palette="Greys") +
                                                                                          5
                     theme_bw()
                                                                                          6
                                                                                          7
122 print(tracefitR0)
                                                                                          8
123 ggsave(tracefitR0, filename="traceplotR0.pdf", height=4, width=6.5)
                                                                                          9
                                                                                         10
|125| exfit \leftarrow extract(fit, permuted = FALSE, inc_warmup = TRUE)
                                                                                         11
|126| plotdata \leftarrow melt(exfit[,,"R0"])
                                                                                         12
127 tracefitR0 \leftarrow ggplot() +
                                                                                         13
                     geom_line(data = plotdata,
                                                                                         14
                                 aes(x = iterations,
                                                                                         15
                                 y = value,
                                                                                         16
                                 color = factor(chains, labels = 1:stan_
                                                                                         17
                                     options$chains))) +
                                                                                         18
                     labs(x = "Sample", y = expression(R[0]), color = "
                                                                                         19
                         Chain") +
                                                                                         20
                     scale_color_brewer(palette="Greys") +
                                                                                         21
                     theme_bw()
                                                                                         22
                                                                                         23
136 print(tracefitR0)
                                                                                         24
   ggsave(tracefitR0, filename="traceplotR0_inc.pdf", height=4, width
                                                                                         25
        =6.5)
                                                                                         26
                                                                                         27
|139| \operatorname{sso} \leftarrow \operatorname{as.shinystan}(\operatorname{fit})
                                                                                         28
|140| \operatorname{sso} \leftarrow \operatorname{launch\_shinystan(sso)}|
                                                                                         38
```

### A.2 Full Stan code

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

```
## Dexter Barrows
                                                                                34
  ## McMaster University
                                                                                35
3 ## 2016
                                                                                36
                                                                                37
  data {
                                                                                38
                                                                                39
       int
                <lower=1>
                                      // total integration steps
                                                                                40
                             Τ;
                                      // observed number of cases
       real
                             y[T];
                                                                                41
       int
                <lower=1>
                             N;
                                      // population size
                                                                                42
                                      // step size
       real
                             h;
                                                                                43
                                                                                44
  }
                                                                                45
13
                                                                                46
```

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

### Appendix B

### Iterated Filtering

#### B.1 Full R code

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

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

3

4

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

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

### <sup>1</sup> B.2 Full C++ code

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

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

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

```
particles[n].R0 = R0can;
1
2
               do {
3
4
                    rcan = rtrue + rtrue*randn();
5
               } while (rcan < 0);</pre>
   96
               particles[n].r = rcan;
6
7
               do {
8
                    sigmacan = merr + merr*randn();
9
10
               } while (sigmacan < 0);</pre>
               particles[n].sigma = sigmacan;
11
12
               do {
13
                    Iinitcan = i_infec + i_infec*randn();
14
               } while (Iinitcan < 0 || N < Iinitcan);</pre>
15
16
               particles[n].Sinit = N - Iinitcan;
               particles[n].Iinit = Iinitcan;
17
               particles[n].Rinit = 0.0;
19
20
           }
21 111
           // START PASSES THROUGH DATA
22
23 113
           printf("Starting filter\n");
24
           printf("----\n");
25 115
26
           printf("Pass\n");
27
28
           for (int pass = 0; pass < nPasses; pass++) {</pre>
29
30 120
               printf("...%d / %d\n", pass, nPasses);
31
32 122
33 123
               perturbParticles(particles, N, NP, pass, coolrate);
34 124
               // initialize particle system states
35
               for (int n = 0; n < NP; n++) {
36
37 127
                    particles[n].S = particles[n].Sinit;
38
                    particles[n].I = particles[n].Iinit;
39
                    particles[n].R = particles[n].Rinit;
40
41 131
               }
42
43
44
               // between-pass perturbations
45
               for (int t = 1; t < T; t++) {
46
47
                   // between-iteration perturbations
48
                   perturbParticles(particles, N, NP, pass, coolrate);
49
50
```

```
// generate individual predictions and weight
                                                                           1
               for (int n = 0; n < NP; n++) {
                                                                           2
                                                                           3
                   exp_euler_SIR(1.0/10.0, 0.0, 1.0, N, &particles[n]);
                                                                           4
                                                                           5
                   double merr_par = particles[n].sigma;
                                                                           6
                   double y_diff = data[t] - particles[n].I;
                                                                           7
                                                                           8
                   w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff
                                                                           9
                       *y_diff / (2.0*merr_par*merr_par) );
                                                                          10
                                                                          11
               }
                                                                          12
                                                                          13
               // cumulative sum
                                                                          14
               for (int n = 1; n < NP; n++) {
                                                                          15
                   w[n] += w[n-1];
                                                                          16
               }
                                                                          17
                                                                          18
               // save particle states to resample from
                                                                          19
               for (int n = 0; n < NP; n++){
                                                                          20
                   copyParticle(&particles_old[n], &particles[n]);
                                                                          21
               }
                                                                          22
                                                                          23
               // resampling
                                                                          24
               for (int n = 0; n < NP; n++) {
                                                                          25
                                                                          26
                   double w_r = randu() * w[NP-1];
                                                                          27
                   int i = 0;
                                                                          28
                   while (w_r > w[i]) {
                                                                          29
                       i++;
                                                                          30
                                                                          31
                                                                          32
                   // i is now the index to copy state from
                                                                          33
                   copyParticle(&particles[n], &particles_old[i]);
                                                                          34
                                                                          35
               }
                                                                          36
                                                                          37
           }
                                                                          38
                                                                          39
       }
                                                                          40
                                                                          41
       ParticleInfo pInfo;
                                                                          42
       particleDiagnostics(&pInfo, particles, NP);
                                                                          43
                                                                          44
       printf("Parameter results (mean | sd)\n");
                                                                          45
       printf("----\n");
185
                                                                          46
                       %f %f\n", pInfo.R0mean, pInfo.R0sd);
       printf("R0
                                                                          47
       printf("r
                        %f %f\n", pInfo.rmean, pInfo.rsd);
                                                                          48
       printf("sigma %f %f\n", pInfo.sigmamean, pInfo.sigmasd);
                                                                          49
       printf("S_init %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
                                                                          50
```

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

```
double dI = B*S*I - r*I;
                                                                                1
            double dR = r*I;
                                                                                2
            // step forward by h
                                                                                3
            S += h*dS;
                                                                                4
            I += h*dI;
                                                                                5
            R += h*dR;
                                                                                6
       }
                                                                                7
                                                                                8
        particle -> S = S;
                                                                                9
        particle ->I = I;
                                                                               10
        particle ->R = R;
                                                                               11
                                                                               12
249 }
                                                                               13
                                                                               14
                                                                               15
252 /* Particle pertubation function to be run between iterations and
                                                                               16
       passes
                                                                               17
                                                                               18
                                                                               19
255 void perturbParticles(Particle * particles, int N, int NP, int
                                                                               20
       passnum, double coolrate) {
                                                                               21
                                                                               22
        double coolcoef = pow(coolrate, passnum);
                                                                               23
                                                                               24
        double spreadR0
                             = coolcoef * R0true / 10.0;
                                                                               25
        double spreadr
                             = coolcoef * rtrue / 10.0;
                                                                               26
        double spreadsigma = coolcoef * merr
                                                   / 10.0;
                                                                               27
        double spreadIinit = coolcoef * I0
                                                   / 10.0;
                                                                               28
                                                                               29
        double R0can, rcan, sigmacan, Iinitcan;
                                                                               30
                                                                               31
        for (int n = 0; n < NP; n++) {
                                                                               32
                                                                               33
            do {
                                                                               34
                R0can = particles[n].R0 + spreadR0*randn();
                                                                               35
270
            } while (R0can < 0);
                                                                               36
            particles[n].R0 = R0can;
                                                                               37
                                                                               38
273
            do {
                                                                               39
                rcan = particles[n].r + spreadr*randn();
                                                                               40
            } while (rcan < 0);</pre>
                                                                               41
            particles[n].r = rcan;
                                                                               42
                                                                               43
            do {
                                                                               44
                sigmacan = particles[n].sigma + spreadsigma*randn();
                                                                               45
            } while (sigmacan < 0);</pre>
                                                                               46
            particles[n].sigma = sigmacan;
                                                                               47
                                                                               48
            do {
                                                                               49
                Iinitcan = particles[n].Iinit + spreadIinit*randn();
                                                                               50
```

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

```
Rinitmean += particles[n].Rinit;
                                                                               1
                                                                               2
       }
                                                                               3
                                                                               4
        R0mean
                    /= NP;
                                                                               5
        rmean
                    /= NP;
                                                                               6
                    /= NP;
                                                                               7
        sigmamean
                    /= NP;
        Sinitmean
                                                                               8
        Iinitmean
                    /= NP;
                                                                               9
       Rinitmean
                    /= NP;
                                                                              10
                                                                              11
        double R0sd = 0, rsd = 0, sigmasd = 0, Sinitsd = 0, Iinitsd =
                                                                              12
           0, Rinitsd = 0;
                                                                              13
                                                                              14
        for (int n = 0; n < NP; n++) {
                                                                              15
                                                                              16
            R0sd
                    += ( particles[n].R0 - R0mean ) * ( particles[n].R0
                                                                              17
               - R0mean );
                                                                              18
                    += ( particles[n].r - rmean ) * ( particles[n].r -
                                                                              19
            rsd
               rmean );
                                                                              20
            sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[
                                                                              21
               n].sigma - sigmamean );
                                                                              22
            Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[
                                                                              23
               n].Sinit - Sinitmean );
                                                                              24
            Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[
                                                                              25
               n]. Iinit - Iinitmean );
                                                                              26
            Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[
                                                                              27
               n].Rinit - Rinitmean );
                                                                              28
                                                                              29
       }
                                                                              30
                                                                              31
       R0sd
                    /= NP:
                                                                              32
       rsd
                    /= NP;
                                                                              33
        sigmasd
                    /= NP;
                                                                              34
                    /= NP;
        Sinitsd
                                                                              35
                    /= NP;
        Iinitsd
                                                                              36
       Rinitsd
                    /= NP;
                                                                              37
                                                                              38
        if (R0sd + rsd + sigmasd) < 1e-5)
                                                                              39
            retVal = true;
                                                                              40
        else
                                                                              41
            retVal = false;
                                                                              42
                                                                              43
        return retVal;
                                                                              44
                                                                              45
370 }
                                                                              46
                                                                              47
372 void particleDiagnostics(ParticleInfo * partInfo, Particle *
                                                                              48
       particles, int NP) {
                                                                              49
                                                                              50
```

```
1 374
           double
                                 = 0.0,
                    R0mean
2
                    rmean
                                 = 0.0,
3
                    sigmamean
                                 = 0.0,
                                 = 0.0,
4 377
                    Sinitmean
5
                    Iinitmean
                                 = 0.0,
  379
                    Rinitmean
                                 = 0.0;
6
7
           // means
8
9
10
           for (int n = 0; n < NP; n++) {
11
               R0mean
                            += particles[n].R0;
12
               rmean
                            += particles[n].r;
13
                            += particles[n].sigma;
14
               sigmamean
15
               Sinitmean
                            += particles[n].Sinit;
16
               Iinitmean
                            += particles[n]. Iinit;
               Rinitmean
                            += particles[n].Rinit;
17
           }
19
20
21
           R0mean
                        /= NP;
22
           rmean
                        /= NP;
                        /= NP;
23
           sigmamean
           Sinitmean
                        /= NP;
24
           Iinitmean
                        /= NP;
25
26
           Rinitmean
                        /= NP;
27
           // standard deviations
28
29
           double
30
                   R0sd
                            = 0.0,
31
                    rsd
                            = 0.0.
32
                    sigmasd = 0.0,
33
                    Sinitsd = 0.0,
                    Iinitsd = 0.0,
34
                    Rinitsd = 0.0;
35
36
           for (int n = 0; n < NP; n++) {
37
38
                        += ( particles[n].R0 - R0mean ) * ( particles[n].R0
39
                   - R0mean );
40
                        += ( particles[n].r - rmean ) * ( particles[n].r -
41
                   rmean );
42
               sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[
43
44
                   n].sigma - sigmamean );
               Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[
45
                   n].Sinit - Sinitmean );
46
               Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[
47
48
                   n]. Iinit - Iinitmean );
               Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[
49
50
                   n].Rinit - Rinitmean );
```

```
418
                                                                               1
       }
                                                                               2
                                                                               3
       R0sd
                    /= NP;
                                                                               4
       rsd
                    /= NP;
                                                                               5
       sigmasd
                    /= NP;
                                                                               6
       Sinitsd
                    /= NP;
                                                                               7
       Iinitsd
                    /= NP;
                                                                               8
       Rinitsd
                    /= NP;
                                                                               9
                                                                              10
       partInfo->R0mean
                             = R0mean;
                                                                              11
       partInfo->R0sd
                             = R0sd;
                                                                              12
       partInfo->sigmamean = sigmamean;
                                                                              13
       partInfo->sigmasd = sigmasd;
                                                                              14
       partInfo->rmean
                             = rmean;
                                                                              15
       partInfo->rsd
                             = rsd;
                                                                              16
       partInfo->Sinitmean = Sinitmean;
                                                                              17
       partInfo->Sinitsd
                            = Sinitsd;
                                                                              18
       partInfo->Iinitmean = Iinitmean;
                                                                              19
       partInfo->Iinitsd
                             = Iinitsd;
                                                                              20
       partInfo->Rinitmean = Rinitmean;
                                                                              21
       partInfo->Rinitsd = Rinitsd;
                                                                              22
                                                                              23
   }
                                                                              24
                                                                              25
443 double randu() {
                                                                              26
                                                                              27
       return (double) rand() / (double) RAND_MAX;
                                                                              28
                                                                              29
447 }
                                                                              30
                                                                              31
                                                                              32
450 /*
       Return a normally distributed random number with mean 0 and
                                                                              33
       standard deviation 1
                                                                              34
       Uses the polar form of the Box-Muller transformation
                                                                              35
       From http://www.design.caltech.edu/erik/Misc/Gaussian.html
                                                                              36
       */
                                                                              37
454 double randn() {
                                                                              38
                                                                              39
       double x1, x2, w, y1;
                                                                              40
                                                                              41
       do {
                                                                              42
            x1 = 2.0 * randu() - 1.0;
                                                                              43
           x2 = 2.0 * randu() - 1.0;
                                                                              44
           w = x1 * x1 + x2 * x2;
                                                                              45
       } while ( w >= 1.0 );
                                                                              46
                                                                              47
       w = sqrt((-2.0 * log(w)) / w);
                                                                              48
       y1 = x1 * w;
                                                                              49
                                                                              50
```

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

Appendix C	1
Parameter Fitting	2

### Appendix D

### <sub>2</sub> Forecasting Frameworks

### 3 D.1 IF2 Parametric Bootstrapping Function

4 The parametric bootstrapping machinery used to produce forecasts.

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

```
statemeans_parent ← colMeans(statedata_parent)
                                                                                1
27
     names(statemeans_parent) ← c("S","I","R","B")
                                                                                2
                                                                                3
                                                                                4
     ## use parametric bootstrapping to generate forcasts
                                                                                5
                                                                                6
     trajectories \leftarrow foreach(i = 1:nTrials, .combine = rbind, .packages
                                                                                7
         = "Rcpp") %dopar% {
                                                                                8
                                                                                9
       source(stoc_sir_file)
                                                                                10
                                                                                11
       ## draw new data
                                                                                12
                                                                                13
                                                                                14
       pars ← with( as.list(parmeans_parent),
                                                                                15
                      c(R0 = R0,
                                                                                16
                      r = r,
                                                                                17
                      N = N,
                                                                                18
                      eta = eta,
                                                                                19
                      berr = berr) )
                                                                                20
                                                                                21
       init_cond ← with( as.list(parmeans_parent),
                                                                                22
                            c(S = Sinit,
                                                                                23
                              I = Iinit,
                                                                                24
                              R = Rinit))
                                                                                25
                                                                                26
       # generate trajectory
                                                                                27
       sdeout \leftarrow StocSIR(init\_cond, pars, Tlim + 1, steps)
                                                                                28
       colnames(sdeout) ← c('S','I','R','B')
                                                                                29
                                                                                30
       # add noise
                                                                                31
       counts_raw ← sdeout[,'I'] + rnorm(dim(sdeout)[1], 0, parmeans_
                                                                                32
          parent[['sigma']])
                                                                                33
                    ← ifelse(counts_raw < 0, 0, counts_raw)</p>
                                                                                34
         counts
                                                                                35
         ## refit using new data
                                                                                36
                                                                                37
                                                                                38
         rm(if2) # because stupid things get done in packages
                                                                                39
         sourceCpp(if2file)
                                                                                40
         if2time \leftarrow system.time( if2data \leftarrow if2(counts, Tlim+1, N, NP,
                                                                                41
             nPasses, coolrate) )
                                                                                42
                                                                                43
         paramdata \leftarrow data.frame( if2data\$paramdata )
                                                                                44
       names(paramdata) \leftarrow c("R0", "r", "sigma", "eta", "berr", "Sinit",
                                                                                45
            "Iinit", "Rinit")
                                                                                46
       parmeans ← colMeans(paramdata)
                                                                                47
       names(parmeans) ← c("R0", "r", "sigma", "eta", "berr", "Sinit",
                                                                                48
          "Iinit", "Rinit")
                                                                                49
                                                                                50
```

```
## generate the rest of the trajectory
1
2
3
4
           # pack new parameter estimates
5
           pars ← with( as.list(parmeans),
                           c(R0 = R0,
6
                           r = r,
7
                           N = N,
8
9
                           eta = eta,
10
    80
                           berr = berr) )
    81
           init\_cond \leftarrow c(S = statemeans\_parent[['S']],
11
                              I = statemeans_parent[['I']],
    82
12
                              R = statemeans_parent[['R']])
    83
13
14
    84
           # generate remaining trajectory part
15
16
           sdeout_future ← StocSIR(init_cond, pars, T-Tlim, steps)
    87
           colnames(sdeout_future) ← c('S','I','R','B')
17
    89
           return ( c( counts = unname(sdeout_future[,'I']),
19
20
                         parmeans,
21
                         time = if2time[['user.self']]) )
22
23
         }
24
25
26
    96
         return(trajectories)
27
    98 }
<del>2</del>8
```

#### D.2 RStan Forward Simulator

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

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

```
14
                                                                                                                   1
      B0 \leftarrow R0 * r / N
                                                                                                                   2
      B \leftarrow B0
                                                                                                                   3
                                                                                                                   4
18
      out[1,] \leftarrow c(S,I,R,B)
                                                                                                                   5
                                                                                                                   6
      h \leftarrow 1 \text{ / steps}
                                                                                                                   7
                                                                                                                   8
      for ( i in 1:(T*steps) ) {
                                                                                                                   9
                                                                                                                  10
             if (i <= bveclim) {</pre>
                                                                                                                  11
                B \leftarrow \exp(\log(B) + eta*(\log(B0) - \log(B)) + berrvec[i])
                                                                                                                  12
                                                                                                                  13
             } else {
                   B \leftarrow \exp(\log(B) + eta*(\log(B0) - \log(B)) + rnorm(1, 0,
                                                                                                                  14
                        berr))
                                                                                                                  15
             }
                                                                                                                  16
                                                                                                                  17
          \texttt{BSI} \leftarrow \texttt{B} {\star} \texttt{S} {\star} \texttt{I}
                                                                                                                  18
          rI \leftarrow r*I
                                                                                                                  19
                                                                                                                  20
          \mathsf{dS} \leftarrow \mathsf{-BSI}
                                                                                                                  21
          dI \leftarrow BSI - rI
                                                                                                                  22
          \text{dR} \leftarrow \text{rI}
                                                                                                                  23
                                                                                                                  24
          S \leftarrow S + h*dS #newInf
                                                                                                                  25
          I \leftarrow I + h*dI #newInf - h*dR
                                                                                                                  26
          R \leftarrow R + h*dR #h*dR
                                                                                                                  27
                                                                                                                  28
41
          if (i %% steps == 0)
                                                                                                                  29
             out[i/steps+1,] \leftarrow c(S,I,R,B)
                                                                                                                  30
                                                                                                                  31
      }
                                                                                                                  32
45
                                                                                                                  33
       return(out)
                                                                                                                  34
47
                                                                                                                  35
48 }
                                                                                                                  <del>36</del>
```

## <sub>1</sub> Appendix E

### <sub>2</sub> S-map and SIRS

#### 3 E.1 SIRS R Function Code

R code to simulate the outlines SIRS function.

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

40

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

#### E.2 SMAP Code

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

```
\begin{array}{c} 1 \\ 1 \\ 2 \\ 3 \\ 3 \\ 5 \\ \end{array} \begin{array}{c} \text{Hibrary(pracma)} \\ 42 \\ 43 \\ 44 \\ 45 \\ \end{array} \begin{array}{c} 41 \\ 43 \\ 44 \\ 45 \\ 45 \\ \end{array}
```

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

10

### E.3 SMAP Parameter Optimization Code

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

```
11
                                                                                      12
  library(deSolve)
                                                                                      13
2 library(ggplot2)
                                                                                      14
3 library (RColorBrewer)
                                                                                      15
  library(pracma)
                                                                                      16
                                                                                      17
  set.seed(1010)
                                                                                      18
                                                                                      19
8 ## external files
                                                                                      20
                                                                                      21
10 stoc_sirs_file \leftarrow paste(getwd(), "../sir-functions", "StocSIRS.r",
                                                                                      22
      sep = "/")
                                                                                      23
11 smap_file
                  ← paste(getwd(), "smap.r", sep = "/")
                                                                                      24
12 source(stoc_sirs_file)
                                                                                      25
13 source(smap_file)
                                                                                      26
                                                                                      27
                                                                                      28
                                                                                      29
17 ## parameters
                                                                                      30
18 ##
                                                                                      31
19 T
         \leftarrow 6 * 52
                                                                                      32
20 Tlim \leftarrow T - 52
                                                                                      33
  |i_i| infec \leftarrow 10
                                                                                      34
22 steps \leftarrow 7
                                                                                      35
23 N
         \leftarrow 500
                                                                                      36
24 \mid \text{sigma} \quad \leftarrow 5
                                                                                      37
                                                                                      38
26 true_pars \leftarrow c( R0 = 3.0, # new infected people per infected
                                                                                      39
                                                                                      40
      person
                   r = 0.1, # recovery rate
                                                                                      41
              N = 500, # population size
                                                                                      42
               eta = 0.5, # geometric random walk
                                                                                      43
               berr = 0.5, # Beta geometric walk noise
                                                                                      44
                   re = 1) # resuceptibility rate
                                                                                      45
                                                                                      46
```

```
33 true_init_cond \leftarrow c(S = N - i_infec,
1
2
                             I = i_infec,
3
                             R = 0)
4
5
   37 ## trial parameter values to check.options
   38 ##
6
7
   39 Elist \leftarrow 1:20
   40 thetalist \leftarrow 10 * \exp(-(seq(0, 9.5, 0.5)))
8
   41 | \text{nTrials} \leftarrow 100
9
10
11
   43 ssemat \leftarrow matrix(NA, 20, 20)
12
   45 for (i in 1:length(Elist)) {
13
        for (j in 1:length(thetalist)) {
14
15
16
           ssemean \leftarrow 0
17
           for (k in 1:nTrials) {
18
19
             E \leftarrow Elist[i]
20
21
             theta ← thetalist[j]
22
             ## get true trajectory
23
24
             sdeout ← StocSIRS(true_init_cond, true_pars, T, steps)
25
26
             ## perturb to get data
27
             ##
28
             infec_counts_raw \( \text{sdeout[1:(Tlim+1),'I'] + rnorm(Tlim+1,0,} \)
   61
29
30
             31
32
                 raw)
33 63
             predictions ← smap(infec_counts, E, theta, 52)
34
35
             err ← sdeout[(Tlim+2):dim(sdeout)[1],'I'] - predictions
36
37
             sse \leftarrow sum(err^2)
38
             ssemean ← ssemean + (sse / nTrials)
39
40
           }
41
42
           ssemat[i,j] \leftarrow ssemean
43
44
45
         }
46
47
      }
48
   79 quartz()
49
   80 image(-ssemat)
```

22

```
81 quartz()
  filled.contour(-ssemat)
                                                                                           2
                                                                                           3
84 #print(ssemat)
                                                                                           4
85 \text{ #cms} \leftarrow \text{colMeans(ssemat)}
                                                                                           5
86 \text{ #rms} \leftarrow \text{rowMeans(ssemat)}
                                                                                           6
87
                                                                                           7
88 #Emin ← Elist[which.min(rms)]
                                                                                           8
89 |#thetamin \leftarrow thetalist[which.min(cms)]
                                                                                           9
90 #print(Emin)
                                                                                          10
91 #print(thetamin)
                                                                                          11
                                                                                          12
  mininds ← which(ssemat==min(ssemat),arr.ind=TRUE)
                                                                                          13
                                                                                          14
95 Emin ← Elist[mininds[,'row']]
                                                                                          15
96 thetamin \leftarrow thetalist[mininds[,'col']]
                                                                                          16
                                                                                          17
98 print(Emin)
                                                                                          18
99 print(thetamin)
                                                                                          28
```

#### E.4 RStan SIRS Code

This code implements a periodic SIRS model in Rstan.

```
23
 data {
                                                                             24
                                                                             25
                                    // total integration steps
      int
               <lower=1>
                                                                             26
                                    // observed number of cases
      real
                            y[T];
                                                                             27
               <lower=1>
      int
                           N;
                                    // population size
                                                                             28
      real
                           h;
                                    // step size
                                                                             29
7
                                                                             30
8
 }
                                                                             31
                                                                             32
 parameters {
                                                                             33
                                                                             34
      real <lower=0, upper=10>
                                        R0;
                                                 // R0
                                                                             35
      real <lower=0, upper=10>
                                                 // recovery rate
                                                                             36
                                        r;
      real <lower=0, upper=10>
                                                 // resusceptibility rate
                                                                             37
                                        re;
      real <lower=0, upper=20>
                                                 // observation error
                                        sigma;
                                                                             38
      real <lower=0, upper=30>
                                        Iinit;
                                                   // initial infected
                                                                             39
      real <lower=0, upper=1>
                                        eta;
                                                 // geometric walk
                                                                             40
         attraction strength
                                                                             41
      real <lower=0, upper=1>
                                        berr;
                                                 // beta walk noise
                                                                             42
      real <lower=-1.5, upper=1.5>
                                        Bnoise[T]; // Beta vector
                                                                             43
                                                                             44
 }
                                                                             45
                                                                             46
```

```
23 //transformed parameters {
               real B0 \leftarrow R0 \star r / N;
2
    24 //
    25 //}
3
4
5
    27 model {
    28
6
7
            real S[T];
            real I[T];
8
9
            real R[T];
            real B[T];
10
11
            real B0;
12
13
            real pi;
            real Bfac;
14
15
16
            pi \leftarrow 3.1415926535;
17
            B0 \leftarrow R0 * r / N;
18
19
            B[1] \leftarrow B0;
20
21
22
    44
            S[1] \leftarrow N - Iinit;
            I[1] \leftarrow Iinit;
23
24
            R[1] \leftarrow 0.0;
25
26
            for (t in 2:T) {
27
                 Bnoise[t] ~ normal(0,berr);
28
                 Bfac \leftarrow \exp(2*\cos((2*pi/365)*t) - 2);
29
                 B[t] \leftarrow \exp(\log(B0) + eta * (\log(B[t-1]) - \log(B0)) +
30
31
                     Bnoise[t] );
32
                 S[t] \leftarrow S[t-1] + h*( - Bfac*B[t]*S[t-1]*I[t-1] + re*R[t-1] );
33
34
                 I[t] \leftarrow I[t-1] + h*( Bfac*B[t]*S[t-1]*I[t-1] - I[t-1]*r );
                 R[t] \leftarrow R[t-1] + h*( I[t-1]*r - re*R[t-1] );
35
36
37
                 if (y[t] > 0) {
                      y[t] ^ normal( I[t], sigma );
38
39
40
            }
41
    63
42
            R0
                      ~ lognormal(1,1);
43
                      ^{\sim} lognormal(1,1);
44
            r
                      ~ lognormal(1,1);
45
            sigma
                      ~ lognormal(1,1);
46
            re
            Iinit
                     ~ normal(y[1], sigma);
47
48
    70 }
\frac{49}{50}
```

#### IF2 SIRS Code E.5

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

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

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

```
srand(time(NULL));
                                // Seed PRNG with system time
                                                                              1
                                                                              2
                         // particle weights
       double w[NP];
                                                                              3
                                                                              4
       Particle particles[NP];  // particle estimates for current
                                                                              5
                                                                              6
       Particle particles_old[NP]; // intermediate particle states for
                                                                              7
           resampling
                                                                              8
                                                                              9
       printf("Initializing particle states\n");
                                                                              10
                                                                              11
       // initialize particle parameter states (seeding)
                                                                              12
       for (int n = 0; n < NP; n++) {
                                                                              13
                                                                              14
            double R0can, rcan, recan, sigmacan, Iinitcan, etacan,
                                                                              15
               berrcan;
                                                                              16
                                                                              17
            do {
                                                                              18
106
                R0can = R0true + R0true*randn();
                                                                              19
            } while (R0can < 0);
                                                                              20
            particles[n].R0 = R0can;
                                                                              21
                                                                              22
            do {
                                                                              23
                rcan = rtrue + rtrue*randn();
                                                                              24
            } while (rcan < 0);</pre>
                                                                              25
            particles[n].r = rcan;
                                                                              26
                                                                              27
            do {
                                                                              28
                recan = retrue + retrue*randn();
                                                                              29
            } while (recan < 0);</pre>
                                                                              30
            particles[n].re = recan;
                                                                              31
                                                                              32
            particles[n].B = (double) R0can * rcan / N;
                                                                              33
                                                                              34
            do {
                                                                              35
                sigmacan = merr + merr*randn();
                                                                              36
            } while (sigmacan < 0);</pre>
                                                                              37
            particles[n].sigma = sigmacan;
                                                                              38
                                                                              39
            do {
                                                                              40
                etacan = etatrue + PSC*etatrue*randn();
                                                                              41
            } while (etacan < 0 || etacan > 1);
                                                                              42
            particles[n].eta = etacan;
                                                                              43
                                                                              44
            do {
                                                                              45
                berrcan = berrtrue + PSC*berrtrue*randn();
                                                                              46
            } while (berrcan < 0);</pre>
                                                                              47
            particles[n].berr = berrcan;
                                                                              48
                                                                              49
            do {
                                                                              50
```

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

```
*y_diff / (2.0*merr_par*merr_par) );
                                                                               1
                                                                               2
                }
                                                                               3
                                                                               4
                // cumulative sum
                                                                               5
                for (int n = 1; n < NP; n++) {
                                                                               6
                    w[n] += w[n-1];
                                                                               7
                }
                                                                               8
                                                                               9
                // save particle states to resample from
                                                                              10
                for (int n = 0; n < NP; n++){
                                                                              11
                     copyParticle(&particles_old[n], &particles[n]);
                                                                              12
                }
                                                                              13
                                                                              14
                // resampling
                                                                              15
200
                for (int n = 0; n < NP; n++) {
                                                                              16
                                                                              17
                     double w_r = randu() * w[NP-1];
                                                                              18
                    int i = 0;
                                                                              19
                    while (w_r > w[i]) {
                                                                              20
                         i++;
                                                                              21
                    }
                                                                              22
                                                                              23
                    // i is now the index to copy state from
                                                                              24
                    copyParticle(&particles[n], &particles_old[i]);
                                                                              25
                                                                              26
                }
                                                                              27
                                                                              28
                // between-iteration perturbations, not after last time
                                                                              29
                                                                              30
                if (t < (T-1))
                                                                              31
                    perturbParticles(particles, N, NP, pass, coolrate);
                                                                              32
                                                                              33
                if (pass == (nPasses-1)) {
                                                                              34
                    State sMeans;
                                                                              35
                     getStateMeans(&sMeans, particles, NP);
                                                                              36
                    statemeans(t,0) = sMeans.S;
                                                                              37
                     statemeans(t,1) = sMeans.I;
                                                                              38
                     statemeans(t,2) = sMeans.R;
                                                                              39
                }
                                                                              40
                                                                              41
            }
                                                                              42
                                                                              43
            ParticleInfo pInfo;
                                                                              44
            particleDiagnostics(&pInfo, particles, NP);
                                                                              45
                                                                              46
            means(pass, 0) = pInfo.R0mean;
                                                                              47
            means(pass, 1) = pInfo.rmean;
                                                                              48
            means(pass, 2) = pInfo.remean;
                                                                              49
            means(pass, 3) = pInfo.sigmamean;
                                                                              50
```

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

```
paramdata(n, 5) = particles[n].berr;
                                                                              1
            paramdata(n, 6) = particles[n].Sinit;
                                                                              2
            paramdata(n, 7) = particles[n].Iinit;
                                                                              3
            paramdata(n, 8) = particles[n].Rinit;
                                                                              4
                                                                              5
       }
                                                                              6
                                                                              7
       for (int n = 0; n < NP; n++) {
                                                                              8
                                                                              9
            statedata(n, 0) = particles[n].S;
                                                                             10
            statedata(n, 1) = particles[n].I;
                                                                             11
            statedata(n, 2) = particles[n].R;
                                                                             12
            statedata(n, 3) = particles[n].B;
                                                                             13
                                                                             14
       }
                                                                             15
                                                                             16
                                                                             17
                                                                             18
       return Rcpp::List::create( Rcpp::Named("paramdata") = paramdata
                                                                             19
                                                                             20
                                     Rcpp::Named("means") = means,
                                                                             21
                                     Rcpp::Named("statemeans") =
                                                                             22
                                        statemeans,
                                                                             23
                                     Rcpp::Named("statedata") = statedata
                                                                             24
                                                                             25
                                     Rcpp::Named("sds") = sds);
                                                                             26
                                                                             27
308 }
                                                                             28
                                                                             29
                                                                             30
311 /*
       Use the Explicit Euler integration scheme to integrate SIR model
                                                                             31
        forward in time
                                                                             32
       double h - time step size
                                                                             33
       double t0 - start time
                                                                             34
       double tn - stop time
                                                                             35
       double * y - current system state; a three-component vector
                                                                             36
          representing [S I R], susceptible-infected-recovered
                                                                             37
                                                                             38
                                                                             39
318 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle
                                                                             40
       * particle) {
                                                                             41
                                                                             42
       int num_steps = floor( (tn-t0) / h );
                                                                             43
                                                                             44
       double S = particle->S;
                                                                             45
       double I = particle->I;
                                                                             46
       double R = particle->R;
                                                                             47
                                                                             48
       double R0
                    = particle->R0;
                                                                             49
       double r
                    = particle->r;
                                                                             50
```

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

```
double spreadr
                             = coolcoef * rtrue / 10.0;
                                                                               1
        double spreadre
                             = coolcoef * retrue / 10.0;
                                                                               2
        double spreadsigma = coolcoef * merr / 10.0;
                                                                               3
        double spreadIinit = coolcoef * I0 / 10.0;
                                                                               4
        double spreadeta = coolcoef * etatrue / 10.0;
                                                                               5
        double spreadberr
                             = coolcoef * berrtrue / 10.0;
                                                                               6
                                                                               7
                                                                               8
       double R0can, rcan, recan, sigmacan, Iinitcan, etacan, berrcan;
                                                                               9
                                                                               10
        for (int n = 0; n < NP; n++) {
                                                                               11
                                                                               12
            do {
                                                                               13
                R0can = particles[n].R0 + spreadR0*randn();
                                                                               14
            } while (R0can < 0);</pre>
                                                                               15
390
            particles[n].R0 = R0can;
                                                                               16
                                                                               17
            do {
                                                                               18
                rcan = particles[n].r + spreadr*randn();
                                                                               19
            } while (rcan < 0);
                                                                               20
            particles[n].r = rcan;
                                                                               21
                                                                               22
            do {
                                                                               23
                recan = particles[n].re + spreadre*randn();
                                                                               24
            } while (recan < 0);</pre>
                                                                               25
            particles[n].re = recan;
                                                                               26
                                                                               27
            do {
                                                                               28
                sigmacan = particles[n].sigma + spreadsigma*randn();
                                                                               29
            } while (sigmacan < 0);</pre>
                                                                               30
            particles[n].sigma = sigmacan;
                                                                               31
                                                                               32
            do {
                                                                               33
                etacan = particles[n].eta + PSC*spreadeta*randn();
                                                                               34
            } while (etacan < 0 \mid \mid etacan > 1);
                                                                               35
            particles[n].eta = etacan;
                                                                               36
                                                                               37
            do {
                                                                               38
                berrcan = particles[n].berr + PSC*spreadberr*randn();
                                                                               39
            } while (berrcan < 0);</pre>
                                                                               40
            particles[n].berr = berrcan;
                                                                               41
                                                                               42
                                                                               43
                Iinitcan = particles[n].Iinit + spreadIinit*randn();
                                                                               44
            } while (Iinitcan < 0 || Iinitcan > 500);
                                                                               45
            particles[n].Iinit = Iinitcan;
                                                                               46
            particles[n].Sinit = N - Iinitcan;
                                                                               47
                                                                               48
       }
                                                                               49
                                                                               50
```

```
1 425 }
2 426
3
4 428
           Convinience function for particle resampling process
5
6
       void copyParticle(Particle * dst, Particle * src) {
7
8
9
           dst->R0
                        = src -> R0;
10
           dst->r
                        = src -> r;
           dst->re
                        = src->re;
11
12
           dst->sigma = src->sigma;
           dst->eta
                        = src->eta;
13
14
           dst->berr
                        = src->berr;
15
           dst->B
                        = src -> B;
16
           dst->S
                        = src -> S;
           dst->I
                        = src->I;
17
           dst->R
                        = src -> R;
           dst->Sinit = src->Sinit;
19
20
           dst->Iinit
                        = src->Iinit;
           dst->Rinit = src->Rinit;
21
22
   447 }
23
24
  449 void particleDiagnostics(ParticleInfo * partInfo, Particle *
25
26
          particles, int NP) {
27
           double
28
                    R0mean
                                 = 0.0.
                                 = 0.0,
29
                    rmean
30
                                 = 0.0,
                    remean
31
                    sigmamean
                                 = 0.0.
32
                    etamean
                                 = 0.0.
33
                    berrmean
                                 = 0.0,
34 457
                    Sinitmean
                                 = 0.0,
                                 = 0.0,
35
                    Iinitmean
                    Rinitmean
                                 = 0.0;
36
37
           // means
38
39
           for (int n = 0; n < NP; n++) {
40
41 464
               R0mean
                            += particles[n].R0;
42
                            += particles[n].r;
43
               rmean
44
               remean
                            += particles[n].re;
45
               etamean
                            += particles[n].eta,
                            += particles[n].berr,
46
               berrmean
47
               sigmamean
                            += particles[n].sigma;
                            += particles[n].Sinit;
48
               Sinitmean
49
               Iinitmean
                            += particles[n]. Iinit;
  473
               Rinitmean
                            += particles[n].Rinit;
```

```
1
       }
                                                                               2
                                                                               3
                    /= NP;
        R0mean
                                                                               4
        rmean
                     /= NP;
                                                                               5
                     /= NP;
        remean
                                                                               6
                    /= NP;
                                                                               7
        sigmamean
                    /= NP;
        etamean
                                                                               8
                    /= NP;
                                                                               9
        berrmean
        Sinitmean
                    /= NP;
                                                                              10
        Iinitmean
                    /= NP;
                                                                              11
        Rinitmean
                    /= NP;
                                                                              12
                                                                              13
        // standard deviations
                                                                              14
                                                                              15
        double
                R0sd
                         = 0.0,
                                                                              16
                rsd
                         = 0.0,
                                                                              17
                resd
                         = 0.0,
                                                                              18
                sigmasd = 0.0,
                                                                              19
                etasd
                         = 0.0,
                                                                              20
                berrsd = 0.0,
                                                                              21
                Sinitsd = 0.0,
                                                                              22
                Iinitsd = 0.0,
                                                                              23
                Rinitsd = 0.0;
                                                                              24
                                                                              25
        for (int n = 0; n < NP; n++) {
                                                                              26
500
                                                                              27
                    += ( particles[n].R0 - R0mean ) * ( particles[n].R0
            R0sd
                                                                              28
                                                                              29
               - R0mean );
                    += ( particles[n].r - rmean ) * ( particles[n].r -
                                                                              30
               rmean );
                                                                              31
                   += ( particles[n].re - rmean ) * ( particles[n].re -
                                                                              32
            resd
                rmean );
                                                                              33
            sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[
                                                                              34
               n].sigma - sigmamean );
                                                                              35
            etasd += ( particles[n].eta - etamean ) * ( particles[n].
                                                                              36
               eta - etamean );
                                                                              37
            berrsd += ( particles[n].berr - berrmean ) * ( particles[n
                                                                              38
               ].berr - berrmean );
                                                                              39
            Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[
                                                                              40
               n].Sinit - Sinitmean );
                                                                              41
            Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[
                                                                              42
               n].Iinit - Iinitmean );
                                                                              43
            Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[
                                                                              44
               n].Rinit - Rinitmean );
                                                                              45
                                                                              46
       }
                                                                              47
                                                                              48
        R0sd
                    /= NP:
                                                                              49
                    /= NP;
        rsd
                                                                              50
```

```
1 515
                        /= NP;
           resd
                        /= NP;
2
           sigmasd
3 517
           etasd
                        /= NP;
                       /= NP;
4 518
           berrsd
5
           Sinitsd
                        /= NP;
           Iinitsd
                        /= NP;
6
7 521
           Rinitsd
                       /= NP;
8
           partInfo->R0mean
                                = R0mean;
9
10 524
           partInfo->R0sd
                                = R0sd;
11 525
           partInfo->rmean
                                = rmean;
           partInfo->rsd
                                = rsd;
12
                                = remean;
           partInfo->remean
13
           partInfo->resd
14
                                = resd;
15
           partInfo->sigmamean = sigmamean;
16
           partInfo->sigmasd
                                = sigmasd;
17 531
           partInfo->etamean
                                = etamean;
18 532
           partInfo->etasd
                                = etasd;
19 533
           partInfo->berrmean = berrmean;
20
           partInfo->berrsd
                                = berrsd;
21 535
           partInfo->Sinitmean = Sinitmean;
           partInfo->Sinitsd
                                = Sinitsd;
22
23 537
           partInfo->Iinitmean = Iinitmean;
           partInfo->Iinitsd
                                = Iinitsd;
24
25 539
           partInfo->Rinitmean = Rinitmean;
26
           partInfo->Rinitsd
                                = Rinitsd;
27
  542 }
28
29
  544 double randu() {
30
31
32
           return (double) rand() / (double) RAND_MAX;
33
34 548 }
35
  550 void getStateMeans(State * state, Particle* particles, int NP) {
36
37
           double Smean = 0, Imean = 0, Rmean = 0;
38
39
           for (int n = 0; n < NP; n++) {
40
41 555
               Smean += particles[n].S;
               Imean += particles[n].I;
42
               Rmean += particles[n].R;
43
44
          }
45
           state->S = (double) Smean / NP;
46
           state->I = (double) Imean / NP;
47
           state->R = (double) Rmean / NP;
48
49
50
  564 }
```

```
1
                                                                               2
567 /* Return a normally distributed random number with mean 0 and
                                                                               3
       standard deviation 1
                                                                               4
       Uses the polar form of the Box-Muller transformation
                                                                               5
       From http://www.design.caltech.edu/erik/Misc/Gaussian.html
                                                                               6
       */
                                                                               7
571 double randn() {
                                                                               8
                                                                               9
       double x1, x2, w, y1;
                                                                              10
                                                                              11
       do {
                                                                              12
           x1 = 2.0 * randu() - 1.0;
                                                                              13
           x2 = 2.0 * randu() - 1.0;
                                                                              14
           w = x1 * x1 + x2 * x2;
                                                                              15
       } while ( w >= 1.0 );
                                                                              16
                                                                              17
       w = sqrt((-2.0 * log(w)) / w);
581
                                                                              18
       y1 = x1 * w;
                                                                              19
                                                                              20
       return y1;
                                                                              21
                                                                              22
586 }
                                                                              \frac{23}{24}
```

# Appendix F

## <sub>2</sub> Spatial Epidemics

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

4 R code to simulate the outlined Spatial SIR function.

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

```
N \leftarrow pars[['N']]
                                                                                                        1
27
         eta ← pars[['eta']]
                                                                                                        2
         berr ← pars[['berr']]
                                                                                                        3
         phi ← pars[['phi']]
                                                                                                        4
                                                                                                        5
         B0 \leftarrow rep(R0*r/N, nloc)
                                                                                                        6
                                                                                                        7
         ## state vectors
                                                                                                        8
         S \leftarrow ymat[,'S']
                                                                                                        9
         I \leftarrow ymat[,'I']
                                                                                                       10
         R \leftarrow ymat[,'R']
                                                                                                       11
         B \leftarrow B0
                                                                                                       12
                                                                                                       13
         ## assign starting to output matrix
                                                                                                       14
         out[,,1] \leftarrow cbind(ymat, B0)
                                                                                                       15
                                                                                                       16
         h \leftarrow 1 \text{ / steps}
                                                                                                       17
                                                                                                       18
         for ( i in 1:(T*steps) ) {
                                                                                                       19
                                                                                                       20
               B \leftarrow \exp(\log(B) + eta*(\log(B0) - \log(B)) + rnorm(nloc, 0,
                                                                                                       21
                   berr) )
                                                                                                       22
                                                                                                       23
               for (loc in 1:nloc) {
                                                                                                       24
                 n \leftarrow neinum[loc]
                                                                                                       25
                 sphi \leftarrow 1 - phi*(n/(n+1))
                                                                                                       26
                 ophi \leftarrow phi/(n+1)
                                                                                                       27
                 nBIsum ← B[neibmat[loc,1:n]] %*% I[neibmat[loc,1:n]]
                                                                                                       28
                 BSI[loc] \leftarrow S[loc]*( sphi*B[loc]*I[loc] + ophi*nBIsum )
                                                                                                       29
               }
                                                                                                       30
                                                                                                       31
              #if(i == 1)
                                                                                                       32
               # print(BSI)
                                                                                                       33
                                                                                                       34
               \texttt{rI} \leftarrow \texttt{r*I}
                                                                                                       35
                                                                                                       36
               \mathsf{dS} \leftarrow \mathsf{-BSI}
                                                                                                       37
               \texttt{dI} \leftarrow \texttt{BSI} - \texttt{rI}
                                                                                                       38
               dR \leftarrow rI
                                                                                                       39
                                                                                                       40
               S \leftarrow S + h*dS
                                                                                                       41
               I \leftarrow I + h*dI
                                                                                                       42
              R \leftarrow R + h*dR
                                                                                                       43
                                                                                                       44
              if (i %% steps == 0) {
                                                                                                       45
                    out[,,i/steps+1] \leftarrow cbind(S,I,R,B)
                                                                                                       46
               }
                                                                                                       47
                                                                                                       48
         }
                                                                                                       49
                                                                                                       50
```

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

## <sub>21</sub> F.2 RStan Spatial SIR Code

This code implements a Spatial SIR model in Rstan.

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

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

```
1
                     S[loc, t] \leftarrow S[loc, t-1] + h*( - BSI[loc, t] );
2
                      I[loc, t] \leftarrow I[loc, t-1] + h*(BSI[loc, t] - rI[loc, t])
4
5
                     R[loc, t] \leftarrow R[loc, t-1] + h*(rI[loc, t]);
6
                      if (y[loc, t] > 0) {
7
                          y[loc, t] ~ normal( I[loc, t], sigma );
8
9
10
                }
11
12
13
                        lognormal(1,1);
            R0
14
                      ^{\sim} lognormal(1,1);
15
                     ~ lognormal(1,1);
16
            sigma
    81
            for (loc in 1:nloc) {
17
                 Iinit[loc] ~ normal(y[loc, 1], sigma);
    83
            }
19
20
    84
    85 }
\frac{21}{22}
```

### F.3 IF2 Spatial SIR Code

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

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

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

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

```
1
                                                                      2
for (int pass = 0; pass < nPasses; pass++) {
                                                                      3
                                                                      4
    printf("...%d / %d\n", pass, nPasses);
                                                                      5
                                                                      6
    // reset particle system evolution states
                                                                      7
    for (int n = 0; n < NP; n++) {
                                                                      8
        for (int loc = 0; loc < nloc; loc++) {
                                                                      9
            particles[n].S[loc] = N - particles[n].Iinit[loc];
                                                                     10
            particles[n].I[loc] = particles[n].Iinit[loc];
                                                                     11
            particles[n].R[loc] = 0.0;
                                                                     12
            particles[n].B[loc] = (double) particles[n].R0 *
                                                                     13
                particles[n].r / N;
                                                                     14
        }
                                                                     15
    }
                                                                     16
                                                                     17
    if (pass == (nPasses-1)) {
                                                                     18
        double means[nloc];
                                                                     19
        for (int loc = 0; loc < nloc; loc++) {
                                                                     20
            means[loc] = 0.0;
                                                                     21
            for (int n = 0; n < NP; n++) {
                                                                     22
                means[loc] += particles[n].I[loc] / NP;
                                                                     23
                                                                     24
            infecmeans(loc, 0) = means[loc];
                                                                     25
        }
                                                                     26
    }
                                                                     27
                                                                     28
    for (int t = 1; t < T; t++) {
                                                                     29
                                                                     30
        // generate individual predictions and weight
                                                                     31
        for (int n = 0; n < NP; n++) {
                                                                     32
                                                                     33
            exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[n],
                                                                     34
                neinum, neibmat, nloc);
                                                                     35
                                                                     36
            double merr_par = particles[n].sigma;
                                                                     37
                                                                     38
            w[n] = 1.0;
                                                                     39
            for (int loc = 0; loc < nloc; loc++) {
                                                                     40
                double y_diff = data(loc, t) - particles[n].I[
                                                                     41
                    loc];
                                                                     42
                w[n] *= 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( -
                                                                     43
                    y_diff*y_diff / (2.0*merr_par*merr_par) );
                                                                     44
            }
                                                                     45
                                                                     46
        }
                                                                     47
                                                                     48
        // cumulative sum
                                                                     49
        for (int n = 1; n < NP; n++) {
                                                                     50
```

```
1 151
                       w[n] += w[n-1];
2 152
                   }
                   // save particle states to resample from
4 154
5
                   for (int n = 0; n < NP; n++){
                        copyParticle(&particles_old[n], &particles[n], nloc)
6
7
                   }
8
9
10
                   // resampling
                   for (int n = 0; n < NP; n++) {
11
12
                        double w_r = randu() * w[NP-1];
13
14
                        int i = 0;
15
                       while (w_r > w[i]) {
16
                            i++;
                        }
17
                       // i is now the index to copy state from
19
                        copyParticle(&particles[n], &particles_old[i], nloc)
20
21
22
                   }
23
24
                   // between-iteration perturbations, not after last time
25
26
27 174
                   if (t < (T-1))
                        perturbParticles(particles, N, NP, nloc, pass,
28
                           coolrate);
29
30 176
31
                   if (pass == (nPasses-1)) {
                        double means[nloc];
32 178
33
                        for (int loc = 0; loc < nloc; loc++) {
34 180
                            means[loc] = 0.0;
                            for (int n = 0; n < NP; n++) {
35
                                means[loc] += particles[n].I[loc] / NP;
36
37
                            infecmeans(loc, t) = means[loc];
38
                        }
39
                   }
40
41
               }
42
43
               // between-pass perturbations, not after last pass
44
               if (pass < (nPasses + 1))
45
                   perturbParticles(particles, N, NP, nloc, pass, coolrate)
46
47
                       ;
48
49 194
           }
50
```

```
// pack parameter data (minus initial conditions)
                                                                              1
       for (int n = 0; n < NP; n++) {
                                                                              2
            paramdata(n, 0) = particles[n].R0;
                                                                              3
            paramdata(n, 1) = particles[n].r;
                                                                              4
            paramdata(n, 2) = particles[n].sigma;
                                                                              5
            paramdata(n, 3) = particles[n].eta;
                                                                              6
            paramdata(n, 4) = particles[n].berr;
                                                                              7
            paramdata(n, 5) = particles[n].phi;
                                                                              8
       }
                                                                              9
                                                                             10
       // Pack initial condition data
                                                                             11
       for (int n = 0; n < NP; n++) {
                                                                             12
            for (int loc = 0; loc < nloc; loc++) {
                                                                             13
                initInfec(loc, n) = particles[n].Iinit[loc];
                                                                             14
            }
                                                                             15
       }
                                                                             16
                                                                             17
       // Pack final state means data
                                                                             18
       double Smeans[nloc], Imeans[nloc], Rmeans[nloc], Bmeans[nloc];
                                                                             19
       for (int loc = 0; loc < nloc; loc++) {
                                                                             20
            Smeans[loc] = 0.0;
                                                                             21
            Imeans[loc] = 0.0;
                                                                             22
218
            Rmeans[loc] = 0.0;
                                                                             23
            Bmeans[loc] = 0.0;
                                                                             24
            for (int n = 0; n < NP; n++) {
                                                                             25
                Smeans[loc] += particles[n].S[loc] / NP;
                                                                             26
                Imeans[loc] += particles[n].I[loc] / NP;
                                                                             27
                Rmeans[loc] += particles[n].R[loc] / NP;
                                                                             28
                Bmeans[loc] += particles[n].B[loc] / NP;
                                                                             29
            }
                                                                             30
            finalstate(loc, 0) = Smeans[loc];
                                                                             31
            finalstate(loc, 1) = Imeans[loc];
                                                                             32
            finalstate(loc, 2) = Rmeans[loc];
                                                                             33
            finalstate(loc, 3) = Bmeans[loc];
                                                                             34
       }
                                                                             35
                                                                             36
                                                                             37
       return Rcpp::List::create( Rcpp::Named("paramdata") = paramdata
                                                                             38
                                                                             39
                                     Rcpp::Named("initInfec") = initInfec
                                                                             40
                                                                             41
                                     Rcpp::Named("infecmeans") =
                                                                             42
                                         infecmeans,
                                                                             43
                                     Rcpp::Named("finalstate") =
                                                                             44
                                         finalstate);
                                                                             45
                                                                             46
                                                                             47
                                                                             48
240 }
                                                                             49
                                                                             50
```

```
1 242
          Use the Explicit Euler integration scheme to integrate SIR model
2 243 /*
           forward in time
           double h - time step size
4 244
                       - start time
5 245
           double t0
6 246
           double tn - stop time
7 247
           double * y - current system state; a three-component vector
              representing [S I R], susceptible-infected-recovered
8
9 248
10 249
11 250 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle
          * particle,
12
                            NumericVector neinum, NumericMatrix neibmat, int
13 251
14
                                nloc) {
15 252
16 253
           int num_steps = floor( (tn-t0) / h );
17 254
18 255
           double * S = particle->S;
19 256
           double * I = particle->I;
20 257
           double * R = particle->R;
21 258
           double * B = particle->B;
22 259
23 260
           // create last state vectors
           double S_last[nloc];
24 261
25 262
           double I_last[nloc];
26 263
           double R_last[nloc];
27 264
           double B_last[nloc];
28 265
29 266
           double R0
                       = particle->R0;
30 267
           double r
                       = particle->r;
                       = R0 * r / N;
31 268
           double B0
32 269
           double eta = particle->eta;
           double berr = particle->berr;
33 270
34 271
           double phi = particle->phi;
35 272
36 273
           //printf("sphi \t\ ophi \t\ BSI \t\ rI \t\ dS \t\ dI \t\
37
              t \mid dR \mid t \mid S \mid t \mid I \mid t \mid R \mid n");
38 274
39 275
           for(int t = 0; t < num_steps; t++) {
40 276
41 277
               for (int loc = 0; loc < nloc; loc++) {
                   S_last[loc] = S[loc];
42
                   I_last[loc] = I[loc];
43 279
                   R_last[loc] = R[loc];
44 280
45 281
                   B_{ast[loc]} = B[loc];
               }
46
47 283
               for (int loc = 0; loc < nloc; loc++) {
49 285
                   B[loc] = exp(log(B_last[loc]) + eta*(log(B0) - log(
50 286
```

```
B_last[loc])) + berr*randn() );
                                                                           1
                                                                           2
               int n = neinum[loc];
                                                                           3
               double sphi = 1.0 - phi*((double) n/(n+1.0));
                                                                           4
               double ophi = phi/(n+1.0);
                                                                           5
                                                                           6
               double nBIsum = 0.0;
                                                                           7
               for (int j = 0; j < n; j++)
                                                                           8
                   nBIsum += B_last[(int) neibmat(loc, j) - 1] * I_last
                                                                           9
                       [(int) neibmat(loc, j) - 1];
                                                                          10
                                                                          11
               double BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc]
                                                                          12
                   + ophi*nBIsum );
                                                                          13
               double rI = r*I_last[loc];
                                                                          14
                                                                          15
299
               // get derivatives
                                                                          16
               double dS = -BSI;
                                                                          17
               double dI = BSI - rI;
                                                                          18
               double dR = rI;
                                                                          19
                                                                          20
               // step forward by h
                                                                          21
               S[loc] += h*dS;
                                                                          22
               I[loc] += h*dI;
                                                                          23
               R[loc] += h*dR;
                                                                          24
                                                                          25
               //if (loc == 1)
                                                                          26
               27
                   |%f\t|\n", sphi, ophi, BSI, rI, dS, dI, dR, S[1], I
                                                                          28
                  [1], R[1]);
                                                                          29
                                                                          30
           }
                                                                          31
                                                                          32
       }
                                                                          33
                                                                          34
       /*particle->S = S;
                                                                          35
       particle ->I = I;
                                                                          36
       particle -> R = R;
                                                                          37
       particle -> B = B; */
                                                                          38
                                                                          39
321 }
                                                                          40
                                                                          41
323 /*
       Initializes particles
                                                                          42
                                                                          43
325 void initializeParticles(Particle ** particles, int NP, int nloc,
                                                                          44
      int N) {
                                                                          45
                                                                          46
       // allocate space for doubles
                                                                          47
       *particles = (Particle*) malloc (NP*sizeof(Particle));
                                                                          48
                                                                          49
       // allocate space for arays inside particles
330
                                                                          50
```

```
1 331
           for (int n = 0; n < NP; n++) {
               (*particles)[n].S = (double*) malloc(nloc*sizeof(double));
2
               (*particles)[n].I = (double*) malloc(nloc*sizeof(double));
3
               (*particles)[n].R = (double*) malloc(nloc*sizeof(double));
4 334
5
               (*particles)[n].B = (double*) malloc(nloc*sizeof(double));
               (*particles)[n]. Iinit = (double*) malloc(nloc*sizeof(double)
6
7
                   );
           }
8
9
10
           // initialize all all parameters
           for (int n = 0; n < NP; n++) {
11
12
               double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
13
14
                   phican;
15
16
               do {
                    R0can = R0true + R0true*randn();
17
               } while (R0can < 0);</pre>
               (*particles)[n].R0 = R0can;
19
20
21
               do {
                    rcan = rtrue + rtrue*randn();
22
               } while (rcan < 0);
23
               (*particles)[n].r = rcan;
24
25
26
               for (int loc = 0; loc < nloc; loc++)
                    (*particles)[n].B[loc] = (double) R0can * rcan / N;
27
28
29
               do {
                    sigmacan = merr + merr*randn();
30
31
               } while (sigmacan < 0);</pre>
32
               (*particles)[n].sigma = sigmacan;
33
               do {
34
                    etacan = etatrue + PSC*etatrue*randn();
35
               } while (etacan < 0 || etacan > 1);
36
               (*particles)[n].eta = etacan;
37
38
               do {
39
                    berrcan = berrtrue + PSC*berrtrue*randn();
40
               } while (berrcan < 0);</pre>
41
               (*particles)[n].berr = berrcan;
42
43
44
               do {
                    phican = phitrue + PSC*phitrue*randn();
45
               } while (phican <= 0 \mid \mid phican >= 1);
46
47
               (*particles)[n].phi = phican;
48
               for (int loc = 0; loc < nloc; loc++) {
49
50
                    do {
```

```
Iinitcan = I0 + I0*randn();
                                                                               1
                } while (Iinitcan < 0 || N < Iinitcan);</pre>
                                                                               2
                (*particles)[n]. Iinit[loc] = Iinitcan;
                                                                               3
            }
                                                                               4
                                                                               5
       }
                                                                               6
                                                                               7
386 }
                                                                               8
                                                                               9
388 /* Particle pertubation function to be run between iterations and
                                                                              10
       passes
                                                                              11
                                                                              12
       */
                                                                              13
391 void perturbParticles(Particle * particles, int N, int NP, int nloc,
                                                                              14
        int passnum, double coolrate) {
                                                                              15
                                                                              16
       //double coolcoef = exp( - (double) passnum / coolrate );
                                                                              17
       double coolcoef = pow(coolrate, passnum);
                                                                              18
                                                                              19
       double spreadR0
                             = coolcoef * R0true / 10.0;
                                                                              20
       double spreadr
                            = coolcoef * rtrue / 10.0;
                                                                              21
       double spreadsigma = coolcoef * merr / 10.0;
                                                                              22
       double spreadIinit = coolcoef * I0 / 10.0;
                                                                              23
       double spreadeta
                            = coolcoef * etatrue / 10.0;
                                                                              24
       double spreadberr = coolcoef * berrtrue / 10.0;
                                                                              25
       double spreadphi
                           = coolcoef * phitrue / 10.0;
                                                                              26
                                                                              27
       double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
                                                                              28
                                                                              29
       for (int n = 0; n < NP; n++) {
                                                                              30
                                                                              31
            do {
                                                                              32
                R0can = particles[n].R0 + spreadR0*randn();
                                                                              33
            } while (R0can < 0);</pre>
                                                                              34
            particles[n].R0 = R0can;
                                                                              35
                                                                              36
            do {
                                                                              37
                rcan = particles[n].r + spreadr*randn();
                                                                              38
            } while (rcan < 0);</pre>
                                                                              39
            particles[n].r = rcan;
                                                                              40
                                                                              41
            do {
                                                                              42
                sigmacan = particles[n].sigma + spreadsigma*randn();
                                                                              43
            } while (sigmacan < 0);</pre>
                                                                              44
            particles[n].sigma = sigmacan;
                                                                              45
                                                                              46
            do {
                                                                              47
                etacan = particles[n].eta + PSC*spreadeta*randn();
                                                                              48
            } while (etacan < 0 || etacan > 1);
                                                                              49
            particles[n].eta = etacan;
                                                                              50
```

```
1 427
               do {
2 428
                    berrcan = particles[n].berr + PSC*spreadberr*randn();
3 429
4 430
               } while (berrcan < 0);</pre>
5
               particles[n].berr = berrcan;
  432
6
7 433
               do {
                    phican = particles[n].phi + PSC*spreadphi*randn();
8
               } while (phican \leq 0 \mid \mid phican \geq 1);
9
10
               particles[n].phi = phican;
11
               for (int loc = 0; loc < nloc; loc++) {
12
                    do {
13
                        Iinitcan = particles[n].Iinit[loc] + spreadIinit*
14
15
                            randn();
16
                    } while (Iinitcan < 0 || Iinitcan > 500);
                    particles[n].Iinit[loc] = Iinitcan;
17
               }
           }
19
20
21 446 }
22
           Convinience function for particle resampling process
  448 /*
23
24
  450 void copyParticle(Particle * dst, Particle * src, int nloc) {
25
26
27
           dst->R0
                        = src -> R0;
28
           dst->r
                        = src -> r;
29
           dst->sigma = src->sigma;
           dst->eta
                        = src->eta;
30
           dst->berr
31
                        = src->berr;
32
           dst->phi
                       = src->phi;
33
34 459
           for (int n = 0; n < nloc; n++) {
               dst->S[n]
35
                                 = src->S[n];
               dst->I[n]
                                 = src->I[n];
36
               dst->R[n]
                                 = src -> R[n];
               dst->B[n]
                                 = src->B[n];
38
               dst->Iinit[n]
                               = src->Iinit[n];
39
           }
40
41
   467 }
42
43
44
45
      double randu() {
46
47
           return (double) rand() / (double) RAND_MAX;
49
50 475 }
```

42

43

```
1
                                                                              2
   void getStateMeans(State * state, Particle* particles, int NP) {
                                                                              3
                                                                              4
       double Smean = 0, Imean = 0;
                                                                              5
                                                                              6
       for (int n = 0; n < NP; n++) {
                                                                              7
           Smean += particles[n].S;
                                                                              8
            Imean += particles[n].I;
                                                                              9
           Rmean += particles[n].R;
                                                                             10
                                                                             11
                                                                             12
       state->S = (double) Smean / NP;
                                                                             13
       state->I = (double) Imean / NP;
                                                                             14
       state->R = (double) Rmean / NP;
                                                                             15
                                                                             16
492 }
                                                                             17
493 */
                                                                             18
                                                                             19
495 /* Return a normally distributed random number with mean 0 and
                                                                             20
       standard deviation 1
                                                                             21
       Uses the polar form of the Box-Muller transformation
                                                                             22
       From http://www.design.caltech.edu/erik/Misc/Gaussian.html
                                                                             23
                                                                             24
   double randn() {
                                                                             25
                                                                             26
       double x1, x2, w, y1;
                                                                             27
                                                                             28
       do {
                                                                             29
           x1 = 2.0 * randu() - 1.0;
                                                                             30
           x2 = 2.0 * randu() - 1.0;
                                                                             31
           w = x1 * x1 + x2 * x2;
                                                                             32
       } while ( w >= 1.0 );
                                                                             33
                                                                             34
       w = sqrt((-2.0 * log(w)) / w);
                                                                             35
       y1 = x1 * w;
                                                                             36
                                                                             37
       return y1;
                                                                             38
                                                                             39
514 }
                                                                             49
```

### 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
44
45
46
```

```
2
    5 /*
          Runs a particle filter on synthetic noisy data and attempts to
    6
          reconstruct underlying true state at each time step. Note that
4
5
          this program uses gnuplot to plot the data, so an x11
    8
          environment must be present. Also the multiplier of 1024 in the
6
7
          definition of NP below should be set to a multiple of the number
          of multiprocessors of your GPU for optimal results.
   10
8
9
10
          Also, the accompanying "pf.plg" file contains the instructions
          gnuplot will use. It must be present in the same directory as
11
          the executable generated by compiling this file.
12
13
          Compile with:
14
15
16
          nvcc -arch=sm_20 -02 pf_cuda.cu timer.cpp rand.cpp -o pf_cuda.x
17
          */
18
19
   22 #include <cuda.h>
20
21
   23 #include <iostream>
22 24 #include <fstream>
   25 #include <curand.h>
23
   26 #include <curand_kernel.h>
24
   27 #include <string>
25
   28 #include <sstream>
   29 #include <cmath>
27
28
   31 #include "timer.h"
29
30 32 #include "rand.h"
   33 #include "readdata.h"
31
32
33 35 #define NP
                                       // number of particles
                           (2*2500)
34 36 #define N
                           500.0
                                       // population size
                                       // infectiousness
   37 #define R0true
                           3.0
35
   38 #define rtrue
                           0.1
                                       // recovery rate
36
37
   39 #define etatrue
                           0.5
                                       // real drift attraction strength
   40 #define berrtrue
                           0.5
                                       // real beta drift noise
38
   41 #define phitrue
                           0.5
                                       // real connectivity strength
39
   42 #define merr
                           10.0
                                       // expected measurement error
40
   43 #define I0
                           5.0
                                       // Initial infected individuals
41
   44 #define PSC
                           0.5
                                       // sensitive parameter perturbation
42
         scaling
43
   45 #define NLOC
                           10
44
45
                       3.141592654f
   47 #define PI
46
47
   49 // Wrapper for CUDA calls, from CUDA API
48
   50 // Modified to also print the error code and string
49
   51 # define CUDA_CALL(x) do { if ((x) != cudaSuccess ) {
```

```
std::cout << " Error at " << __FILE__ << ":" << __LINE__ << std
                                                                              2
                                                                              3
      std::cout << " Error was " << x << " " << cudaGetErrorString(x)</pre>
                                                                              4
          << std::endl;
                                                                              5
       return EXIT_FAILURE ;}} while (0)
                                                                              6
                                                                              7
                                                                              8
  typedef struct {
                                                                              9
       float R0;
                                                                             10
       float r;
                                                                             11
       float sigma;
                                                                             12
       float eta;
                                                                             13
       float berr;
                                                                             14
       float phi;
                                                                             15
       /*
                                                                             16
      float * S;
                                                                             17
       float * I;
                                                                             18
      float * R;
                                                                             19
       float * B;
                                                                             20
       float * Iinit;
                                                                             21
                                                                             22
      float S[NLOC];
                                                                             23
      float I[NLOC];
                                                                             24
      float R[NLOC];
                                                                             25
      float B[NLOC];
                                                                             26
      float Iinit[NLOC];
                                                                             27
       curandState randState; // PRNG state
                                                                             28
76 } Particle;
                                                                             29
                                                                             30
  __host__ std::string getHRmemsize (size_t memsize);
                                                                             31
  __host__ std::string getHRtime (float runtime);
                                                                             32
                                                                             33
  __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
                                                                             34
       * particle, int * neinum, int * neibmat, int nloc);
                                                                             35
  __device__ void copyParticle(Particle * dst, Particle * src, int
                                                                             36
      nloc);
                                                                             37
                                                                             38
84
                                                                             39
85 /* Initialize all PRNG states, get starting state vector using
                                                                             40
      initial distribution
                                                                             41
      */
                                                                             42
  __global__ void initializeParticles (Particle * particles, int nloc)
                                                                             43
                                                                             44
                                                                             45
       int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread |
89
                                                                             46
           ΙD
                                                                             47
                                                                             48
      if (id < NP) {
                                                                             49
                                                                             50
```

```
// initialize PRNG state
1
               curandState state;
2
               curand_init(id, 0, 0, &state);
3
4
5
               // allocate space for arays inside particle
               //particles[id].S = (float*) malloc(nloc*sizeof(float));
6
               //particles[id].I = (float*) malloc(nloc*sizeof(float));
7
               //particles[id].R = (float*) malloc(nloc*sizeof(float));
8
               //particles[id].B = (float*) malloc(nloc*sizeof(float));
9
10
               //particles[id].Iinit = (float*) malloc(nloc*sizeof(float));
11
               // initialize all parameters
12
13
14
               float R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
15
                   phican;
16
               do {
17
                   R0can = R0true + R0true*curand_normal(&state);
               } while (R0can < 0);</pre>
19
20
               particles[id].R0 = R0can;
21 112
22 113
               do {
23 114
                   rcan = rtrue + rtrue*curand_normal(&state);
               } while (rcan < 0);
24
25 116
               particles[id].r = rcan;
26 117
               for (int loc = 0; loc < nloc; loc++)
27
                   particles[id].B[loc] = (float) R0can * rcan / N;
28
29 120
30 121
               do {
31 122
                   sigmacan = merr + merr*curand_normal(&state);
32 123
               } while (sigmacan < 0);</pre>
33 124
               particles[id].sigma = sigmacan;
34 125
               do {
35
                   etacan = etatrue + PSC*etatrue*curand_normal(&state);
36 127
37 128
               \} while (etacan < 0 || etacan > 1);
38 129
               particles[id].eta = etacan;
39
40 131
               do {
41 132
                   berrcan = berrtrue + PSC*berrtrue*curand_normal(&state);
               } while (berrcan < 0);</pre>
42
               particles[id].berr = berrcan;
43
44 135
45 136
               do {
                   phican = phitrue + PSC*phitrue*curand_normal(&state);
46
47
               } while (phican \leq 0 || phican \geq 1);
48
               particles[id].phi = phican;
49
50
               for (int loc = 0; loc < nloc; loc++) {
```

```
do {
                                                                               1
                    Iinitcan = I0 + I0*curand_normal(&state);
                                                                               2
                } while (Iinitcan < 0 || N < Iinitcan);</pre>
                                                                               3
                particles[id].Iinit[loc] = Iinitcan;
                                                                               4
            }
                                                                               5
                                                                               6
            particles[id].randState = state;
                                                                               7
                                                                               8
       }
                                                                               9
                                                                              10
152 }
                                                                              11
                                                                              12
154|_{-global_{-}} void resetStates (Particle * particles, int nloc) {
                                                                              13
                                                                              14
       int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
                                                                              15
                                                                              16
                                                                              17
       for (int loc = 0; loc < nloc; loc++) {
                                                                              18
            particles[id].S[loc] = N - particles[id].Iinit[loc];
                                                                              19
            particles[id].I[loc] = particles[id].Iinit[loc];
                                                                              20
            particles[id].R[loc] = 0.0;
                                                                              21
                                                                              22
                                                                              23
164 }
                                                                              24
                                                                              25
166 __global__ void clobberParams (Particle * particles, int nloc) {
                                                                              26
                                                                              27
       int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
                                                                              28
                                                                              29
                                                                              30
       particles[id].R0 = R0true;
                                                                              31
       particles[id].r = rtrue;
                                                                              32
       particles[id].sigma = merr;
                                                                              33
       particles[id].eta = etatrue;
                                                                              34
       particles[id].berr = berrtrue;
                                                                              35
       particles[id].phi = phitrue;
                                                                              36
                                                                              37
       for (int loc = 0; loc < nloc; loc++) {
                                                                              38
            particles[id].Iinit[loc] = I0;
                                                                              39
       }
                                                                              40
                                                                              41
181
                                                                              42
182 }
                                                                              43
                                                                              44
                                                                              45
185 /* Project particles forward, perturb, and save weight based on
                                                                              46
      data
                                                                              47
       int t - time step number (1, ..., T)
                                                                              48
       */
                                                                              49
188 __global__ void project (Particle * particles, int * neinum, int *
                                                                              50
```

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

```
234
                                                                               1
   __global__ void cumsumWeights (double * w) {
                                                                               2
                                                                               3
        int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
                                                                               4
            ΙD
                                                                               5
                                                                               6
        // compute cumulative weights
                                                                               7
       if (id == 0) {
                                                                               8
            for (int i = 1; i < NP; i++)
                                                                               9
                w[i] += w[i-1];
                                                                              10
       }
                                                                              11
                                                                              12
245 }
                                                                              13
                                                                              14
                                                                              15
248 /*
       Resample from all particle states within cell
                                                                              16
249
                                                                              17
250 __global__ void resample (Particle * particles, Particle *
                                                                              18
       particles_old, double * w, int nloc) {
                                                                              19
                                                                              20
       int id = blockIdx.x*blockDim.x + threadIdx.x;
                                                                              21
                                                                              22
       if (id < NP) {
                                                                              23
                                                                              24
            // resampling proportional to weights
                                                                              25
            double w_r = curand_uniform(&particles[id].randState) * w[NP
                                                                              26
                                                                              27
               -1];
            int i = 0;
                                                                              28
            while (w_r > w[i]) {
                                                                              29
                i++;
                                                                              30
                                                                              31
                                                                              32
            // i is now the index of the particle to copy from
                                                                              33
            copyParticle(&particles[id], &particles_old[i], nloc);
                                                                              34
                                                                              35
       }
                                                                              36
                                                                              37
268 }
                                                                              38
                                                                              39
270 // launch this with probably just nloc threads... block structure/
                                                                              40
       size probably not important
                                                                              41
271 __global__ void reduceStates (Particle * particles, float *
                                                                              42
       countmeans, int t, int T, int nloc) {
                                                                              43
                                                                              44
       int id = blockIdx.x*blockDim.x + threadIdx.x;
                                                                              45
                                                                              46
       if (id < nloc) {</pre>
                                                                              47
                                                                              48
            int loc = id;
                                                                              49
278
                                                                              50
```

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

```
etacan = particles[id].eta + PSC*spreadeta*curand_normal
                   (&particles[id].randState);
                                                                             2
           \} while (etacan < 0 || etacan > 1);
                                                                             3
           particles[id].eta = etacan;
                                                                             4
                                                                             5
           do {
                                                                             6
                berrcan = particles[id].berr + PSC*spreadberr*
                                                                             7
                   curand_normal(&particles[id].randState);
                                                                             8
           } while (berrcan < 0);</pre>
                                                                             9
           particles[id].berr = berrcan;
                                                                            10
                                                                            11
           do {
                                                                            12
                phican = particles[id].phi + PSC*spreadphi*curand_normal
                                                                            13
                   (&particles[id].randState);
                                                                            14
           } while (phican <= 0 || phican >= 1);
                                                                            15
           particles[id].phi = phican;
                                                                            16
                                                                            17
           for (int loc = 0; loc < nloc; loc++) {
                                                                            18
                do {
                                                                            19
                    Iinitcan = particles[id].Iinit[loc] + spreadIinit*
                                                                            20
                       curand_normal(&particles[id].randState);
                                                                            21
                } while (Iinitcan < 0 || Iinitcan > 500);
                                                                            22
                particles[id].Iinit[loc] = Iinitcan;
                                                                            23
           }
                                                                            24
                                                                            25
       }
                                                                            26
                                                                            27
348 }
                                                                            28
                                                                            29
                                                                            30
351 int main (int argc, char *argv[]) {
                                                                            31
                                                                            32
                                                                            33
       int T, nloc;
                                                                            34
                                                                            35
       double restime;
                                                                            36
       struct timeval tdr0, tdr1, tdrMaster;
                                                                            37
                                                                            38
       gettimeofday (&tdr0, NULL);
                                                                            39
                                                                            40
                                                                            41
       // Parse arguments
                                                                            42
           ***********
                                                                            43
                                                                            44
       if (argc < 4) {
                                                                            45
           std::cout << "Not enough arguments" << std::endl;</pre>
                                                                            46
           return 0;
                                                                            47
       }
                                                                            48
                                                                            49
       std::string arg1(argv[1]); // infection counts
                                                                            50
```

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

```
409
     std::cout << "\t" << getHRtime(restime) << std::endl;</pre>
                                                         1
                                                         2
        5
                                                         6
     // CUDA data
                                                         7
        **************
                                                         8
                                                         9
     std::cout << "Allocating device storage" << std::endl;</pre>
                                                        10
                                                        11
     gettimeofday (&tdr0, NULL);
                                                        12
                                                        13
              * d_data;
                             // device copy of data
     float
                                                        14
                             // particles
     Particle
              * particles;
                                                        15
     Particle * particles_old;
                              // intermediate particle states
                                                        16
     double
              * W;
                              // weights
                                                        17
              * d_neinum;
                              // device copy of adjacency
     int
                                                        18
                                                        19
        matrix
              * d_neibmat; // device copy of neighbour
                                                        20
        counts matrix
                                                        21
             * countmeans; // host copy of reduced
                                                        22
        infection count means from last pass
                                                        23
     float     * d_countmeans;     // device copy of reduced
                                                        24
       infection count means from last pass
                                                        25
                                                         26
     27
              );
                                                         28
     29
        Particle)) );
                                                        30
     CUDA_CALL( cudaMalloc( (void**) &particles_old , NP*sizeof(
                                                        31
        Particle)) );
                                                        32
                                                        33
     CUDA_CALL( cudaMalloc( (void**) &w
                                          , NP*sizeof(
        double))
                                                        34
               );
     35
                                                        36
     37
                                                        38
     CUDA_CALL( cudaMalloc( (void**) &d_countmeans , nloc*T*sizeof(
                                                        39
        float)) );
                                                         40
                                                        41
                                                        42
     gettimeofday (&tdr1, NULL);
                                                        43
     timeval_subtract (&restime, &tdr1, &tdr0);
                                                        44
                                                        45
     std::cout << "\t" << getHRtime(restime) << std::endl;</pre>
                                                        46
                                                        47
     size_t avail, total;
                                                        48
     cudaMemGetInfo( &avail, &total );
443
                                                        49
     size_t used = total - avail;
444
                                                        50
```

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

```
cudaMemGetInfo( &avail, &total );
                                                                                                                                                                                  1
used = total - avail;
                                                                                                                                                                                  2
std::cout << "\t[" << getHRmemsize(used) << "] used of [" <= getHrmemsize(used) << "] used of [" <= getHrmemsize(used) << "] used of [" <= getHrmemsize(used) <= getHrmemsize
                                                                                                                                                                                  3
        getHRmemsize(total) << "]" <<std::endl;</pre>
                                                                                                                                                                                  4
                                                                                                                                                                                  5
                                                                                                                                                                                  6
         *******************
                                                                                                                                                                                  7
                                                                                                                                                                                  8
                                                                                                                                                                                  9
// Starting filtering
                                                                                                                                                                                10
        **********
                                                                                                                                                                                11
                                                                                                                                                                                12
for (int pass = 0; pass < 50; pass++) {
                                                                                                                                                                                13
                                                                                                                                                                                14
          std::cout << "pass = " << pass << std::endl;
                                                                                                                                                                                15
                                                                                                                                                                                16
          // ** TEMP **
                                                                                                                                                                                17
          //clobberParams <<< nBlocks, nThreads >>> (particles, nloc);
                                                                                                                                                                                18
          // ** TEMP **
                                                                                                                                                                                19
                                                                                                                                                                                20
          nThreads
                                         = 32;
                                                                                                                                                                                21
                                          = ceil( (float) NP / nThreads);
                                                                                                                                                                                22
                                                                                                                                                                                23
          resetStates <<< nBlocks, nThreads >>> (particles, nloc);
                                                                                                                                                                                24
          CUDA_CALL( cudaGetLastError() );
                                                                                                                                                                                25
          CUDA_CALL( cudaDeviceSynchronize() );
                                                                                                                                                                                26
                                                                                                                                                                                27
          std::cout << "Filtering over [1," << Tlim << "]"<< std::endl</pre>
                                                                                                                                                                                28
                                                                                                                                                                                29
                   ;
                                                                                                                                                                                30
          gettimeofday (&tdrMaster, NULL);
                                                                                                                                                                                31
                                                                                                                                                                                32
          gettimeofday (&tdr0, NULL);
                                                                                                                                                                                33
                                                                                                                                                                                34
          nThreads = 1;
                                                                                                                                                                                35
          nBlocks = 10;
                                                                                                                                                                                36
                                                                                                                                                                                37
          if (pass == 49) {
                                                                                                                                                                                38
                     reduceStates <<< nBlocks, nThreads >>> (particles,
                                                                                                                                                                                39
                              d_countmeans, 0, T, nloc);
                                                                                                                                                                                40
                     CUDA_CALL( cudaGetLastError() );
                                                                                                                                                                                41
                     CUDA_CALL( cudaDeviceSynchronize() );
                                                                                                                                                                                42
          }
                                                                                                                                                                                43
                                                                                                                                                                                44
          gettimeofday (&tdr1, NULL);
                                                                                                                                                                                45
          timeval_subtract (&restime, &tdr1, &tdr0);
                                                                                                                                                                                46
          std::cout << "Reduction " << getHRtime(restime) <<</pre>
                                                                                                                                                                                47
                   std::endl;
                                                                                                                                                                                48
                                                                                                                                                                                49
          int Tlim = T;
                                                                                                                                                                                50
```

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

```
timeval_subtract (&restime, &tdr1, &tdr0);
                                                         1
   std::cout << "Cumulative sum " << getHRtime(</pre>
                                                         2
       restime) << std::endl;
                                                         3
}
                                                         4
                                                         5
// Save particles for resampling from
                                                         6
   *******
                                                         7
                                                         8
nThreads
           = 32;
                                                         9
nBlocks
          = ceil( (float) NP / nThreads);
                                                        10
                                                        11
stashParticles <<< nBlocks, nThreads >>> (particles,
                                                        12
   particles_old, nloc);
                                                        13
CUDA_CALL( cudaGetLastError() );
                                                        14
CUDA_CALL( cudaDeviceSynchronize() );
                                                        15
                                                        16
                                                        17
// Resampling
                                                        18
   ***********
                                                        19
                                                        20
nThreads
           = 32;
                                                        21
nBlocks
          = ceil( (float) NP/ nThreads);
                                                        22
                                                        23
if (t == 1)
                                                        24
   gettimeofday (&tdr0, NULL);
                                                        25
                                                        26
resample <<< nBlocks, nThreads >>> (particles,
                                                        27
   particles_old, w, nloc);
                                                        28
CUDA_CALL( cudaGetLastError() );
                                                        29
CUDA_CALL( cudaDeviceSynchronize() );
                                                        30
                                                        31
if (t == 1) {
                                                        32
   gettimeofday (&tdr1, NULL);
                                                        33
    timeval_subtract (&restime, &tdr1, &tdr0);
                                                        34
    std::cout << "\tResampling " << getHRtime(restime)</pre>
                                                        35
       << std::endl;
                                                        36
}
                                                        37
                                                        38
// Reduction
                                                        39
   **********
                                                        40
                                                        41
//if (t == (Tlim-1)) {
                                                        42
                                                        43
if (pass == 49) {
                                                        44
                                                        45
   if (t == 1)
                                                        46
       gettimeofday (&tdr0, NULL);
                                                        47
                                                        48
   nThreads = 1:
                                                        49
   nBlocks = 10;
                                                        50
```

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

```
1
        std::cout.precision(10);
                                                                                  2
                                                                                  3
        countmeans = (float*) malloc (nloc*T*sizeof(float));
                                                                                  4
        cudaMemcpy(countmeans, d_countmeans, nloc*T*sizeof(float),
                                                                                  5
           cudaMemcpyDeviceToHost);
                                                                                  6
                                                                                  7
        std::string filename = "cuIF2states.dat";
                                                                                  8
                                                                                  9
        std::cout << "Writing results to file '" << filename << "' ..."</pre>
                                                                                 10
           << std::endl;
                                                                                 11
                                                                                 12
        std::ofstream outfile;
                                                                                 13
        outfile.open(filename.c_str());
                                                                                 14
                                                                                 15
        for(int loc = 0; loc < nloc; loc++) {</pre>
                                                                                 16
            for (int t = 0; t < T; t++) {
                                                                                 17
                outfile << countmeans[loc*T + t] << " ";</pre>
                                                                                 18
                                                                                 19
            outfile << std::endl;</pre>
                                                                                 20
        }
                                                                                 21
                                                                                 22
                                                                                 23
        double * h_w = (double*) malloc (NP*sizeof(double));
                                                                                 24
        cudaMemcpy(h_w, w, NP*sizeof(double), cudaMemcpyDeviceToHost);
                                                                                 25
                                                                                 26
        for (int n = 0; n < NP; n++) {
                                                                                 27
            std::cout << h_w[n] << " ";
                                                                                 28
                                                                                 29
        */
                                                                                 30
                                                                                 31
                                                                                 32
        for (int i = 0; i < nCells; i++) {
                                                                                 33
            outfile << trueCounts[t*nCells + i];</pre>
                                                                                 34
            if (i % dim == 0)
                                                                                 35
                outfile << std::endl;</pre>
                                                                                 36
                                                                                 37
                outfile << " ";
                                                                                 38
                                                                                 39
        */
                                                                                 40
                                                                                 41
700
        outfile.close();
                                                                                 42
                                                                                 43
        gettimeofday (&tdr1, NULL);
                                                                                 44
        timeval_subtract (&restime, &tdr1, &tdrMaster);
                                                                                 45
        std::cout << "Total PF time (excluding setup) " << getHRtime(</pre>
                                                                                 46
           restime) << std::endl;</pre>
                                                                                 47
                                                                                 48
706
        cudaFree(d_data);
                                                                                 49
        cudaFree(particles);
                                                                                 50
```

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

```
1
            for (int loc = 0; loc < nloc; loc++) {
                                                                               2
                                                                               3
                B[loc] = exp(log(B_last[loc]) + eta*(log(B0) - log(B0))
                                                                               4
                    B_last[loc])) + berr*curand_normal(&(particle->
                                                                               5
                    randState)) );
                                                                               6
                                                                               7
                int n = neinum[loc];
                                                                               8
                float sphi = 1.0 - phi*((float) n/(n+1.0));
                                                                               9
                float ophi = phi/(n+1.0);
                                                                              10
                                                                              11
                float nBIsum = 0.0;
                                                                              12
                for (int j = 0; j < n; j++)
                                                                              13
                    nBIsum += B_last[neibmat[nloc*loc + j]-1] * I_last[
                                                                              14
                        neibmat[nloc*loc + j]-1];
                                                                              15
                                                                              16
                float BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] +
                                                                              17
                    ophi*nBIsum );
                                                                              18
                float rI = r*I_last[loc];
                                                                              19
                                                                              20
                // get derivatives
                                                                              21
                float dS = -BSI;
                                                                              22
                float dI = BSI - rI;
                                                                              23
                float dR = rI;
                                                                              24
                                                                              25
                // step forward by h
                                                                              26
                S[loc] += h*dS;
                                                                              27
                I[loc] += h*dI;
                                                                              28
                R[loc] += h*dR;
                                                                              29
                                                                              30
            }
                                                                              31
                                                                              32
       }
                                                                              33
                                                                              34
       free(S_last);
                                                                              35
       free(I_last);
                                                                              36
       free(R_last);
                                                                              37
       free(B_last);
                                                                              38
                                                                              39
790 }
                                                                              40
                                                                              41
792 /*
       Convinience function for particle resampling process
                                                                              42
                                                                              43
   __device__ void copyParticle(Particle * dst, Particle * src, int
                                                                              44
       nloc) {
                                                                              45
                                                                              46
       dst->R0
                    = src -> R0;
                                                                              47
       dst->r
                    = src ->r;
                                                                              48
       dst->sigma = src->sigma;
                                                                              49
799
       dst->eta
                    = src->eta;
                                                                              50
```

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

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

The parameter estimation means as compared to IF2 and HMCMC are shown in  $\,$   $_{17}$  Figure [].  $\,$ 

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