Toward Modeling of Disease Transmission Networks

Stone D. Chen

Supervisor: Dr. David Welch

Bachelor of Science (Honours)
Department of Statistics
The University of Auckland
New Zealand

Acknowledgements

I would like to thank my supervisor, Dr. David Welch.

Contents

Abstract

This dissertation presents an extension of a Bayesian model for infectious disease surveillance data by Held et al. (2006). Their two-component model captures patterns in a time series of incidence data with a combination of a Poisson branching process for outbreaks and a parameter driven component for seasonality. We extend their model to multiple time series of incidence data that are spatially related by incorporating an Erdos-Renyi (ER) random graph. The ER random graph represents a transmission network between the different spatial regions and models dependencies between data in those regions. To obtain parameter estimates we use the Metropolis-Hastings algorithm and its variants. We demonstrate estimating the graph parameters on simulated data.

1 Introduction

On the 8th of May 2011, a 42-year-old female presented with hemolytic-uremic syndrome (HUS), a severe illness that that can lead to failure of the kidney and the heart. The cause was later identified as an aggressive strain of *Escherichia coli* known O104:H4. Over the next two months, O104:H4 caused a severe outbreak of illness throughout Germany, which at its peak caused 200 new cases per day; the same number typically seen in an entire year. By July 7th, O104:H4 had affected 3,842 people over 50 of whom later died (*EHEC O104* 2011 , pp. 2-3, 11). While there was initial confusion around the source of the *E. coli*, an intensive investigation and development of new *ad-hoc* methodology (Weiser et al. 2013) determined the source to be contaminated bean sprouts. This investigation involved determining the network of connections between the illness, restaurants, suppliers ,and distributors.

The aim of this dissertation is to develop a statistical method to model the shipping network of the contaminated sprouts using a graph. The dataset of interest contains count data from ~ 412 administrative districts in Germany over the time span of the outbreak. While this work presents a simple Erdos-Renyi random graph, the end goal is to estimate covariates that affect the probability of transmission between regions (e.g., the shipping network that transmitted the *E. coli* contaminated food).

Statistical methods to model infectious disease data typically are mechanistic models that use expert knowledge of the underyling process (Pawitan 2001, Ch. 1). One such model is the Susceptible-Exposed-Infectious-Recovery (SEIR) (Keeling and Rohani 2008 , pp. 41-43) model and its variants. In the SEIR model, an entire closed population of individuals is classified in one of the four eponymous states. The dynamics of the model works as follows: susceptible individuals can become exposed when in contact with an infectious invidual, exposed individuals will eventually become infectious, and infectious individuals will eventually be recovered and cannot be re-exposed or infect susceptibles (e.g. by immunity). If there is data on what times individuals become susceptible, exposed, infectious, and removed, parameters surrounding the epidemic of interest can be estimated. For example, in Groendyke et al. (2011), the time spent in each state is modeled with Gamma random variables and the parameters of the random variables can be estimated from the time data. Other parameters of interest such as the intial infected can also be estimated. They used an extended model to examine the canonical Haggeloch measles outbreak of 1861 (Groendyke et al. 2012). There they estimate a contact network between individuals, where covariates can affect the formation of the contact network and a corresponding transmission tree. Their network model is a type of exponential random graph model (ERGM) (Wasserman et al. 1994). The ERGMs were designed to be directly analogous to the generalized linear model (GLM). It allows for statistics surrounding global graph structure such as the number of triangles and k-stars to inform the probability of an edge in a graph. Similar methodology has been applied to other biological systems (e.g., parasite transmission (White et al. 2017)) as well as social networks. For example work by Lee et al. (2018) adapted this methodology to sharing of information on social networks. Here people are "infected" when they see a campaign and subsequently share it. Their paper also includes a different type of network model, the preferential attachment model (Albert and Barabasi 2002). In this model the probability of adding an edge to a vertex is directly proportional to the degree of the vertex and creates a "rich gets richer" dynamic and can better describe the degree distribution of social networks.

While effective, this sort of mechanistic model requires data at a granularity not typical of surveillance data. For example the Hagelloch measles data contains detailed information on individuals such as the specific day when different symptoms appear as well as day of death, when applicable (Albert Pfeilsticker 1863). Surveillance data typically does not contain this information and also often contains other problems such as underreporting (Diggle et al. 2003 , pp. 233-266) which make it unsuitable to this type of model.

Instead, we begin with a model by Held et al. (2006) for infectious disease surveillance data. Held at al.'s method models infectious disease incidence (i.e., new cases) as a branching Poisson process with a cyclical endemic parameter. The cyclical endemic parameter models "seasonality" (e.g. seasonal flu patterns) while the Poisson branching process allows for "outbreaks" (e.g. swine flu, H1N1). Their method models a single time series of incidence data from a specific location or region. We extend the model to multiple time series of incidence data that are spatially related. We incorporate a graph that represents a transmission network between the different spatial regions (e.g., the administration district) and allow dependencies between count data in those regions.

2 The Two-Component Model

Held et al. (2006) presents a stochastic model for the statistical analysis of infectious disease counts that serves as the basis of the extended graph model.

The two components of the model are a simple Poisson branching process with autoregressive parameter λ and a seasonal component fit with a Fourier series. These components are described as the "epidemic" and "endemic" components respectively. Additionally, the two-component model allows for λ to change over time, which represents changing infectivity.

A branching process is a stochastic process that is used to model the evolution (Grimmett and Stirzaker 2004 , pp. 171-175, 243-255) of population over time. Suppose there's currently one parent alive (cf., one person infectious). Then in a branching process, this individual would give birth to a random number of offspring (cf., newly infected) and then immediately die. Then at the next step, each of those offspring become parents who individually give birth to a random number of offspring. If the number of offspring a parent gives birth to comes from a Poisson distribution with parameter λ, then the process is known as a Poisson branching process. It's in this sense that λ is an autoregressive parameter since it describes the relationship between the individuals in the previous time step to the current one. In this case we assume that the λ value is constant across individuals and over time (e.g. λ is some biological reproduction rate). In the case of the two-component model, λ is fixed for individuals at a given time, but is allowed to vary over time. This allows the model to capture the dynamics of disease outbreaks (Figure 1).

2.1 Two-Component Model Notation

Let $\vec{Z} = (Z_0, Z_1, ..., Z_n)$ be the count at each time step t and let each $Z_t = Y_t + X_t, t \in \{1, \dots, n\}$. The model is then specified through $Z_t|Z_{t-1}$.

2.2 Epidemic Component

The epidemic component is given by

$$Y_t|Z_{t-1} \sim \text{Pois}(\lambda_t Z_{t-1}),$$

where λ_t is the time varing infectivity parameter and Z_{t-1} is the the count in the previous time step.

We can think of λ_t as the infectivity of the disease at time t with an infected person causing new infections as $\text{Pois}(\lambda_t)$. Since each count at time Z_{t-1} generates new infected i.i.d $\text{Pois}(\lambda_t)$, then Z_t is the sum of those random variables $\sum_1^{Z_{n-1}} \text{Pois}(\lambda_t) = \text{Pois}(\lambda_t Z_{n-1})$.

In this model, the λ_t is allowed to vary over time. We restrict λ to be piecewise constant with K change points at locations $\theta_1 < \cdots < \theta_K$ with $\theta \in \{1, ..., n-1\}$. If $K = 0$ there is no change point and the λ parameter is constant throughout.

If we only consider the process $Z_t = Y_t$ (only the epidemic component), when $\lambda_t > 1$ an outbreak occurs (Held et al. 2006). When $\lambda_t < 1$ then the process "goes extinct" or reaches and remains at 0 with probability 1 (Grimmett and Stirzaker 2004 , pp. 245). Once the process reaches a point where $Z_t = 0$, it remains there as there are no more infected to create new infected at the next time step. Allowing λ_t to vary captures many scenarios, for example a particularly infectious strain of the flu could cause λ_t to increase above 1 and cause an outbreak. Later, new vaccines and quarantine procedures can cause the overall infectivity to decrease below 1.

2.3 Endemic Component

The endemic component in the model plays two roles: it allows the model to capture cyclical behaviors in disease (e.g. seasonal flu) and it also prevents the branching process from going extinct. The endemic count is modeled as

$$X_t \sim \text{Pois}(\nu_t)$$

$$\log \nu_t = \gamma_0 + \sum_{l=1}^{L} (\gamma_{2l-1} \sin(\rho l t) + \gamma_{2l} cos(\rho l t)).$$

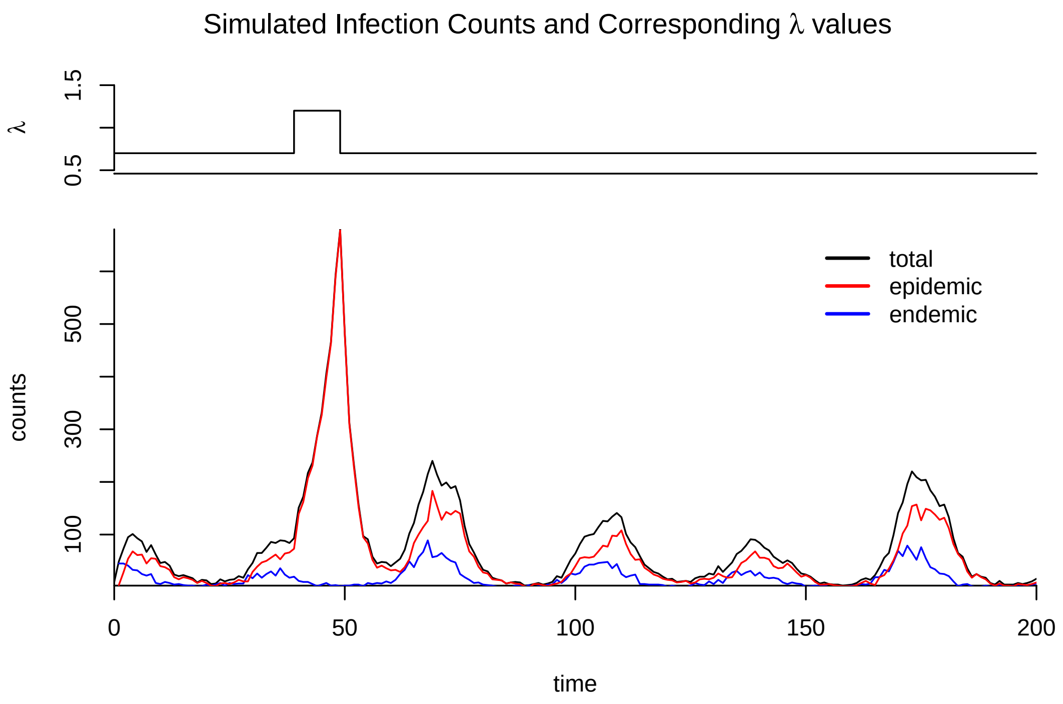

Figure 1: A simulated data set. **Top:** λ epidemic parameter starts at 0.7, increases to 1.2 and then returns to 0.7. **Bottom:** The simulated dataset. Red shows the count from the epidemic process, blue the endemic and black is the sum of the two.

That is $\log \nu_t$ is fit with a Fourier series. Following Held et al. (2006), the series is computed with $L = 1$ since it was determined higher order frequencies were insignificant. That is, the truncated series is still flexible enough to model cyclical patterns seen in disease. The γ's are estimated and inform the amplitude of the corresponding curve. The ρ term is the base frequency and is set to $2\pi/52$ for weekly data.

2.4 Likelihood

The probability of the full time series $\vec{Z}$ given the initial starting count Z_0 can be factored as a product of the probabilities of each Z_t given the previous counts Z_{t-1},

$$P(\vec{Z}|Z_0, \theta, K, \lambda^{(1)}, \ldots, \lambda^{(K+1)}, \gamma_0, \gamma_1, \gamma_2) = \prod_{t=1}^{b} P(Z_t|Z_t-1, \theta, K, \lambda^{(1)}, \ldots, \lambda^{(K+1)}, \gamma_0, \gamma_1, \gamma_2),$$

where Z_t is distributed as sum of the Poisson endemic and epidemic components,

$$Z_t|Z_{t-1}, \theta, K, \lambda^{(1)}, \ldots, \lambda^{(K+1)}, \gamma_0, \gamma_1, \gamma_2 \quad \sim \quad \text{Pois}(\nu_t + \lambda_t Z_{t-1}),$$

where the endemic component ν_t is described as,

$$\log \nu_t = \gamma_0 + \gamma_1 \sin(\rho t) + \gamma_2 \cos(\rho t),$$

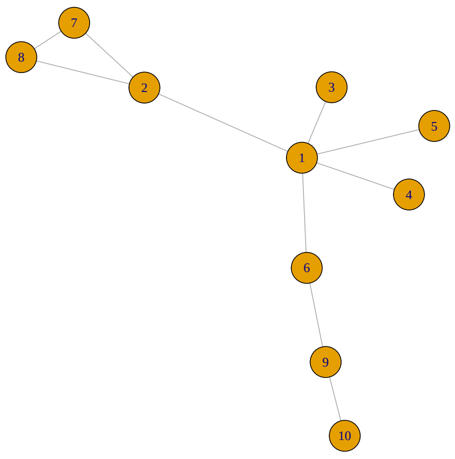

Figure 2: A graph configuration. The vertices $i \in \{1, \ldots, 10\}$ represent cities each with their own disease counts $Z_{i,t}$. The edges between the graph represent whether the counts between the cities can affect each other. In this example city 1's disease counts at time t are influenced by both its own counts and city 2, 4, 5 and 6's (i.e., every city connected to it) disease counts. City 10's disease counts are only influenced by its own and city 9's.

with $\rho = 2\pi/52$. Lastly, λ_t is piecewise function described by

$$\lambda_t = \begin{cases} \lambda^{(1)}, & t < \theta_0 \\ \lambda^{(k)}, & \theta_k \leq t < \theta_{(k-1)} \\ \lambda^{(K+1)}, & t \geq \theta_K \end{cases}.$$

3 Multiple locations connected by a graph

We would like to extend our data from a univariate time series of counts Z_t to multiple time series of counts $Z_{i,t}$ where i now indexes separate time series. In this case we will say the i indexes individual "cities" so Z_i represents the disease counts in city i. Now, a city i's epidemic disease count at time t is modeled as a function of both its own disease count $Z_{i,t-1}$ and possibly other cites' counts as well.

This dependence between cities is represented by a graph G where N_v, the number of vertices in graph is fixed and equal to the number of cities. An (undirected) edge $\{i, j\}$ connects vertices i and j if the count in city i influences the count in city j and vice versa. We call $V = \{1, \ldots, N_v\}$ the vertex set of the graph where and $E(G)$ is the set of unordered pairs of vertices $\{i, j\}$ (where $i, j \in V$ and $i \neq j$) that describes the edges present in graph G. We represent the edge $\{i, j\}$ as e_{ij} and the indicator $1[e_{ij}] = 1$ if $\{i, j\} \in E(G)$ and 0 otherwise.

We then model the disease count of city i at time t as a function of $Z_{i,t-1}$ (as before, its own count at time $t-1$) as well as all the count of the cities j it is connected to, $Z_{j,t-1}$. Then the counts at city i at time t are modeled as

$$Z_t | Z_{t-1}, G = X_{i,t} + Y_{i,t} | G$$

where

$$X_{i,t} : \text{count in city } i \text{ at time step } t, \text{ due to endemic factors,}$$

$$Y_{i,t}|G : \text{count in city } i \text{ at time step } t, \text{ due to epidemic factors,}$$

$$Z_{i,t}|G = X_{i,t} + Y_{i,t}|G : \text{count in city } i \text{ at time step } t.$$

As before we have the epidemic component as $X_{i,t} \sim \text{Pois}(\nu_t)$. For the epidemic component we now include the additional counts from connected cities as

$$Y_{i,t}|G \sim \text{ Pois}(\lambda_t \sum_{j \neq i}^{N_v} Z_{j,t-1} 1[e_{ij} = 1] + \lambda_t Z_{i,t-1})|G,$$

where $\lambda_t \sum_{j \neq i}^{N_v} Z_{j,t} 1[e_{ij} = 1]$ are the counts from all cities connected to city i. Then the epidemic component for city i is the sum of all counts in every city j connected to i in addition to the counts in city i.

3.1 Migration

One issue with this model is that connecting two isolated cities essentially doubles the infectivity parameter, since we include the counts from both cities. In this formulation a connection between two cities is equivalent to treating them as a single city. To make the model more realistic, a migration parameter $m \in (0, 1)$ is introduced so

$$Y_{i,t}|G \sim \text{ Pois}(m\lambda_t \sum_{j \neq i}^{N_v} Z_{j,t-1} 1[e_{ij} = 1] + \lambda_t Z_{i,t-1})|G.$$

The migration parameter m is then the fraction of counts in city j that can cause infections in city i, where $m = 1$ is equivalent to the previous model where all incidences in city j are counted, whereas when $m = 0$, no incidences are counted and the model is equivalent to all cities i, j not being connected in graph G (i.e, $\{i, j\} \notin E(G)$)

4 Bayesian Inference

In Bayesian analysis, we aim to estimate the posterior distribution of the parameters which we can calculate via Bayes' theorem:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}.$$

$P(\theta|D)$ is the posterior distribution of the parameters, θ, given the data D. $P(D|\theta)$ is the probability of the data D given the parameter value θ. $P(\theta)$ is the prior distribution of θ and represents the belief that the parameters will take certain values before seeing the data. If we have little prior information about a parameter, we can choose an uninformative prior to reflect that ignorance. $P(D)$ is the probability of the data marginalized over the parameter space. One issue with Bayesian estimation is finding the $P(\text{D})$ term, which requires computation of $\int P(\text{D}|\theta)P(\theta)$ over the entire parameter space. With many parameters, this integral is typically analytically intractable. To handle this Markov chain Monte-Carlo methods are used which we discuss in section Methods/Implementation.

We now specify the priors of the two-component and graph portions of the model.

4.1 Priors for λ, γ, K and $\vec{\theta}$

The prior distribution for the γ parameters is Normal with variance $\sigma^2 = 3^2$ to describe an uninformative prior:

$$\gamma_i \sim N(0, 3), i \in \{0, 1, 2\}.$$

Since the λ values parameterize a Poisson distribution we set the prior to be

$$\lambda^{(k)} \sim \text{Gamma}(1, 1),\ \ k \in \{1, \ldots, K+1\}.$$

The $\text{Gamma}(1, 1)$, is selected since it's the conjugate prior of the Poisson. If Gamma is interpreted as the sum of exponentials, then the shape $= 1$, rate $= 1$ parameterization represents seeing a single occurence in 1 unit of time and also represents an uninformative prior. An uninformative prior is important when we later incorporate a graph since the magnitude of the λ_i's can vary with the connectivity of the graph.

The number of change points K takes values in $\{1, \ldots, N\}$ where N is the total number of time points of counts collected. In the Held et al. (2006) paper the number of change points is uniformly distributed $P(K = k) = 1/N$ representing uncertainty in the number of change points. This is changed to

$$K \sim \text{Pois}(2)$$

representing that idea that the disease count data is already of interest due to a potential change in the infectivity of the disease. $K \sim \text{Pois}(2)$ places the highest mass on $K = 2$ change points which can capture a spike in disease counts (as seen in the simulated data) before returning to a baseline endemic rate. It also places mass on $K = 1$ (e.g. capturing a long term decrease in infectivity due to intervention) and $K = 3$ change points. It also acts as a regularizer to help reduce overfitting the count time series where each time point is given a unique λ_t value.

The probability of a specific location for a change point given the number of change points is uniformly distributed among all the possible change points,

$$P(\theta | K = k) = \binom{N}{k}^{-1}.$$

Then the unnormalized posterior is then the product of the likelihood and the priors,

$$P(\theta, K, \lambda^{(1)}, \ldots, \lambda^{(K+1)}, \gamma_0, \gamma_1, \gamma_2 | Z) \propto$$

$$\prod_{t=1}^{N} P(Z_t | Z_{t-1}, \theta, K, \lambda^{(1)}, \ldots, \lambda^{(K+1)}, \gamma_0, \gamma_1, \gamma_2) P(\theta | K) P(K) \prod_{k=1}^{K+1} P(\lambda^{(k)}) \prod_{i=0}^{2} P(\gamma_i).$$

4.2 Graph Prior

We model the graph G modeled as an Erdos-Renyi (ER) random graph. An Erdos-Renyi random graph has a fixed vertex set $V(G) = \{1, \ldots, N_v\}$ and is parameterized by $p \in [0, 1]$. Then an edge e_{ij} is in the edge set $E(G)$ with probability p independent of every other edge in the graph. That is an ER graph is parameterized as $G \sim ER(p)$ and the likelihood of a graph G given probability p on an edge is

$$\begin{aligned} G|p &= \prod_{i,j \in V(G), i \neq j} p^{1[e_{ij}]}(1-p)^{1-1[e_{ij}]} \\ &= p^{N_e}(1-p)^{\binom{N_v}{2} - N_e} \end{aligned}$$

where N_e is the number of edges.

We then place a prior on p as $p \sim \text{Unif}(0, 1)$ representing a uncertainty about the sparseness of the graph. Now computing the marginal probability of a graph we find

$$\begin{aligned} P(G) &= \int_0^1 P(G, p)dp = \int_0^1 P(G|p)P(p)dp \\ &= \int_0^1 p^{N_e}(1-p)^{\binom{N_v}{2} - N_e} * 1 * dp = \frac{1}{(\binom{N_v}{2} + 1)\binom{\binom{N_v}{2}}{N_e}}, \end{aligned}$$

where $\binom{\binom{N_v}{2}}{N_e}$ is $\binom{N_v}{2}$ choose N_e and this calculation is used to induce the prior on the graph. It may be desirable to place a *Beta* prior in p with higher mass on lower probabilities.

4.3 Posterior

To summarize, the posterior distribution is proportional to the product of the likelihood and the priors of the graph and non-graph parameters

$$\begin{aligned} &P(\vec{\theta}, K, \lambda^{(1)}, \ldots, \lambda^{(K+1)}, \gamma_0, \gamma_1, \gamma_2, G, p|Z) \propto \\ &\prod_{i=1}^{N_v}\prod_{t=1}^{N} P(Z_{i,t}|Z_{t-1}, \theta, K, \lambda^{(1)}, \ldots, \lambda^{(K+1)}, \gamma_0, \gamma_1, \gamma_2, G, p) * \\ &\qquad P(\theta|K)P(K)\prod_{k=1}^{K+1} P(\lambda^{(k)}) \prod_{i=0}^{2} P(\gamma_i)P(G|p)P(p), \end{aligned}$$

where $\vec{\theta}$ is a K dimensional vector of locations of change points (and takes values in $\{1, \ldots, N\}$ where n is end of the time series); $\lambda^{(1)}, \ldots, \lambda^{(K+1)}$ parameterize the epidemic component at the corresponding change points; $\gamma_0, \gamma_1, \gamma_2$ parameterize the Fourier series that drives the endemic component; G is Erdos-Renyi (ER) random graph that represents the dependencies between cities and $p \in (0, 1)$ parameterizes the ER graphs and is the probability of an edge being in the graph.

4.4 Metropolis-Hastings Algorithm

To approximate the posterior distribution, we draw samples from it using the Metropolis-Hastings algorithm (and its variant, the Metropolis-Hastings-Green algorithm). The algorithm creates a Markov Chain whose states are the parameter space and whose stationary distribution is the posterior distribution. That is the Markov chain will eventually enter states in proportion to the posterior distribution.

The algorithm is as follows:

1. Initialize the Markov chain at some state θ_0
2. From the current state θ at time n propose a new state j according to q. The probability of proposing a transition $\theta \to \theta^*$ is $q(\theta^*|\theta)$
3. Compute the acceptance probability
$$\alpha(\theta^*|\theta) = \min\{1, \frac{P(D|\theta^*)P(\theta^*)}{P(D|\theta)P(\theta)}\frac{q(\theta|\theta^*)}{q(\theta^*|\theta)}\}$$
4. Generate $U \sim \text{Unif}(0,1)$
5. If $U < \alpha(\theta|\theta^*)$ then accept the move and the parameter value at time $n+1$ is $\theta_{n+1} = \theta^*$. If $U \geq \alpha(\theta|\theta^*)$ reject the move. Then the chain remains in the same state at time $n+1$ and $\theta_{n+1} = \theta$.
6. Repeat steps 1-5 for a large number of iterations.

The fraction $\frac{q(\theta^*|\theta)}{q(\theta|\theta^*)}$ is known as the Hastings ratio and is a function of the proposal distribution. It allows the base Metropolis algorithm (which requires symmetric proposals) to work for arbitrary q. Then the transition probability is ratio of the posterior distributions whose denominators, $P(D)$, cancels leaving $\frac{P(D|\theta^*)P(\theta^*)}{P(D|\theta)P(\theta)}$. The proposal ratio becomes important in asymmetric proposal distributions which occurs in some of the graph proposals below. For example if two parameter θ, θ^* values are equally likely, $P(\theta^*|D) = P(\theta|D)$ then the posterior distribution should have approximately equal frequency of these two states. However if the proposal distribution proposes entering θ^* twice as frequently as θ^* then without the Hastings ratio there would be twice the frequency of θ^*, since they're accepted at the same rate but one is proposed twice is frequently. A similar issue is also seen when jumping between parameter spaces. To handle this the Metropolis-Hastings-Green algorithm is introduced which adds an additional Jacobian term $|J|$ to handle the change in dimension. This is described further below.

4.5 Two-Component Model

4.6 Model Checking

4.6.1 Prior Checks

By setting the likelihood to always equal 1, the posterior is only a function of the priors

$$\begin{aligned} P(\theta|D) &\propto P(D|\theta)P(\theta) \\ &\propto 1 * P(\theta) \end{aligned}$$

that is the samples returned by the MCMC should be drawn from the prior distribution. This helps shows that the prior distributions are properly implemented in the model as well as show potentially unintended assumptions about the priors. Furthermore, it can help show any issues with the Hastings computation. Other prior analysis include prior sensitivity analysis where the effect of different priors on the posterior distribution are examined. This is less important when the datasets are large sense the likelihood portion generally dominates the posterior calculation.

4.6.2 Effective Sample Size

The effective sample size is a downward adjustment of the number of samples drawn from the posterior distribution. Since the samples are not independent, their overall variance is lower than would be expected from truly independent samples. To adjust for this the following is used to estimate the effective sample size

$$\widehat{n_{eff}} = \frac{mn}{1 + 2\sum_{t=1}^{T} \widehat{\rho}_t},$$

where m is the number of chains, n is the samples per chain and $\widehat{\rho}_t$ is an estimate of the autocorrelation as seen in Gelman et al. (2013), pp. 286. The effective sample size can be used to calculate estimates of the variance of the means of the posterior distributions.

4.6.3 Well-Calibrated

A goal of frequentist statistical inference is to obtain confidence intervals (CI), where a 95% CI for a parameter would capture the true parameter in 95% of replications.

A credible interval plays a similar role in Bayesian inference. We would like a 95% credible interval for a parameter to capture the "true" parameter value 95% of the time. If this is the case, then the model is considered "well-calibrated". To do this for Bayesian models, we draw samples of the estimated parameters from their prior distributions $\theta_{sample_1} \sim P(\theta)$ and then using the sampled parameters we simulate data according to the likelihood function $D_1 \sim P(D|\theta_{sample_1})$. The model is then used to compute 95% credible intervals as if it were real data. This process is repeated for $(\theta_{sample_2}, D_2), \ldots, (\theta_{sample_N}, D_N)$ and their corresponding credible intervals are collected. We can then compare to see if the 95% credible intervals from the N parameter samples, covers the true sampled parameter 95% of the time (Carpenter 2017; Cook et al. 2006).

4.6.4 Posterior Predictive Checking

If the model is a good fit for the data, then simulating data by sampling according the posterior distribution, should result in simulated data that looks similar to the true data. If this is not the case, the model is potentially inadequate for capturing important features of the data. This can be done in a qualitative manner where obviously poor models can be investigated (Gelman et al. 2013,pp. 141–159; Kruschke 2014,pp. 130–131)

5 Methods/Implementation

Here we describe the Metropolis-Hastings algorithms used to draw samples from the posterior distributions of the parameters of interest. The operators generate and evaluate proposals and are implemented as an R function. Each operator accepts a list of parameters that represents the current state of the Markov chain, then proposes an new state for a given parameter (or set of parameters). The function then calculates the log acceptance ratio and determines whether to accept its proposal or reject it. It then returns the proposed parameters or returns the original parameters (if it rejects the proposal). The algorithms are implemented in base R (R Core Team 2019) with some elements of the likelihood computation implemented in Rcpp (Eddelbuettel and François 2011). Plots are generated in base R and the bayesplot (Gabry and Mahr 2019). Parallel chains are implemented using doParallel (Corporation and Weston 2019) and diagostics done with coda (Plummer et al. 2006).

5.1 Likelihood Computation

Each operator receives the log-likelihood of its proposal by passing the proposal to either `compute_log_like()` function or the `compute_log_like_diff()` function. The `compute_log_like()` function returns the unnormalized log-likelihood of the proposed parameters. This is computationally more efficient since computing the normalized likelihood of a Poisson distribution requires the evalution of $Z_t!$ for each data point. The naive method of computing the entire log-likelihood was used as it was computationally fast enough on the simulated dataset (without the graph).

This differs from Held et al. (2006) and Green (1995) who perform separate likelihood computations for each of their three operators: birth (adding a change point), death (removing a change point) and changing a λ value. Each of these operators changes only a portion of the total likelihood computation and as such, it is computationally more efficient to only compute the ratio of the changes. For example, let $\boldsymbol{\theta}^*$ be the proposed change point vector where $K^* = K + 1$ (i.e. a new change point is added). Let m be the index of the proposed change point and the rest of the change points remain the same. Then the only part of the likelihood computation that changes is in the interval between timepoints $[\theta_{m-1}, \theta_{m+1})$ and since its the ratio between the likelihoods of the proposed vs the current parameter set is important, the parts that remain identical have a likelihood of 1 (or log-likelihood of 0).

Similar reductions in comptutation can be done following changes in the graph and are implemented in `compute_log_like_diff()`. If an edge e^*_{ik} is proposed then only $Y_{i,t}$ and $Y_{k,t}$ and the corresponding Z's are affected. As such only those two cities need to be updated.

$$Y^*_{i,t}|G \sim \text{Pois}(m\lambda_t Z_{k,t-1} + m\lambda_t \sum_{j=1}^{N_v} Z_{j,t-1} 1[e_{ij} \in E(G)] + \lambda_t Z_{i,t-1})$$

$$Y^*_{k,t}|G \sim \text{Pois}(m\lambda_t Z_{i,t-1} + m\lambda_t \sum_{j=1}^{N_v} Z_{j,t-1} 1[e_{ij} \in E(G)] + \lambda_t Z_{i,t-1})$$

$$
\begin{aligned}
Y_{j,t}|G &\sim \mathrm{Pois}(m\lambda_t \sum_{l=1}^{N_v} Z_{j,t-1} 1[e_{jl} \in E(G^*)] + \lambda_t Z_{i,t-1}) \\
&\sim \mathrm{Pois}(m\lambda_t \sum_{l=1}^{N_v} Z_{j,t-1} 1[e_{jl} \in E(G)] + \lambda_t Z_{i,t-1})
\end{aligned}
$$

Then the likelihood ratio can be computed as (dropping conditioning notation for clarity)

$$
\frac{P(Z^*_{i,t})P(Z^*_{k,t})}{P(Z_{i,t})P(Z_k,t)} \frac{\prod_j P(Z_j)}{\prod_j P(Z_j)} = \frac{P(Z^*_{i,t})P(Z^*_{k,t})}{P(Z_{i,t})P(Z_k,t)}
$$

This lets the computational complexity stay constant with respect to the size of the graph. This is currently implemented for the `add_edge_op()` and `del_edge_op()` operators.

5.2 `change_gamma()`, `change_lambda()`

The gamma and lambda parameters are updated in blocks via a standard Metropolis-Hastings step (Gelman et al. 2013,p. 280). The gamma and lambda proposals are each drawn from Normal distributions centered at the current parameter values. That is the proposed parameter vectors γ^* and λ^* are drawn from $\mathrm{Norm}(\gamma, \sigma_\gamma)$ and $\mathrm{Norm}(\lambda, \sigma_\lambda)$. The variance of the distributions is scaled as σ_γ and σ_λ which are hand selected to improve mixing.

Since the Normal distribution is symmetric, the Hastings ratio is 1 so the acceptance ratio is the ratio of the likelihoods between the current and proposed steps.

One potential issue is that gamma and lambda are used to compute the parameter of a Poisson distribution, but the Normal distribution proposals could potentially propose invalid parameter values. To remedy this, the proposal operator checks to see whether the proposed parameter value is valid (in this case positive) and automatically rejects the proposed state (staying in the current state). This could be side-stepped by using a non-symmetric proposal distribution such as log-normal or a truncated normal and an adjustment of the Hastings ratio to compensate for the asymmetry. However this does not seem to be an issue in the simulated cases, the initial values are positive and the jump (σ) size is small enough not to propose negative values.

5.3 `change_theta()`

This operator proposes a new location for one of the current change points $\theta_1, \ldots, \theta_K$. A change point $\theta_k, k \in \{1, \ldots, K\}$ is selected uniformly at random from all current change points. Then the proposed location θ^*_k is selected from values between $\{\theta_{k-1} + 1, \ldots, \theta_{k+1} - 1\}$ uniformly at random. For θ_1 and θ_k the proposed values are from $\{0, \ldots, \theta_1 - 1\}$ and $\{\theta_K + 1, \ldots, N\}$ respectively.

The λ are then updated to match the newly proposed location as:

$$\lambda_t = \begin{cases} \lambda^{(1)} & t < \theta_0 \\ \lambda^{(k^*)}, & \theta_{k^*} \leq t < \theta_{(k^*-1)} \\ \lambda^{(K+1)}, & t \geq \theta_K \end{cases}$$

Since the proposal is generated uniformly at random between θ_{k-1} and θ_{k+1} the proposal probability is

$$q(\theta_{k^*}|\theta_k) = \frac{1}{\theta_{k+1} - \theta_{k-1} - 2} = q(\theta_k|\theta_k^*)$$

. Furthermore the λ values are deterministically generated from current λ_k values and the θ_k^* values. The Hastings ratio is then 1 and the acceptance rate is the ratio of the likelihoods.

5.4 `birth/death_theta()` and Reversible Jump MCMC

The `birth_theta()` and `death_theta()` operators allow for an increase and decrease in the number of change points. Then the parameter space can jump between a collection of possible models $\{M_K, K \in \{0, 1, \cdots, n-1\}\}$ where K indexes the number of change points. However models from different M_K's have different dimensions of $\vec{\theta}$ and the likelihoods are not directly comparable (since they are not defined on the same probability space). To handle this we use a Reversible Jump MCMC (RJMCMC) as proposed in Green (1995).

5.4.1 `birth_theta()`

For a birth step a new change point is chosen uniformly at random from all possible time steps $\{1, \ldots, N\}$ that aren't currently change points $\{\theta_1, \ldots, \theta_K\}$ and then added to the current set of change points

1. Draw $u \sim Unif(0, 1)$
2. The following proposals for the new λ_1 and λ_2 values are from

$$\lambda_1 = \lambda_0 * (\frac{u}{1-u})^{(\theta_1-\theta_0)/(\theta_2-\theta_0)}$$

$$\lambda_2 = \lambda_0 * (\frac{1-u}{u})^{(\theta_2-\theta_1)/(\theta_2-\theta_0)}$$

   That is the new λ values are a compromise between the original value λ_0 of the interval that was split. This compromise is a function of the location of the split in the interval.

3. In order to determine the acceptance probability for the proposal, the corresponding death move must also be determined. In death move a current change point is selected uniformly at random and then removed. Then the λ values are changed deterministically (as described later).

The $P(birth)$ and $P(death)$ are the probabilities of proposing a birth and death step and are selected such that $P(birth) = P(death)$. The $q_{(K+1 \to K)}(\theta|\theta^*)$ term is transition probability from a parameter state with K to $K+1$ change points. The probability of being in the proposed state θ^* and proposing jumping back to the current state θ is the probability of selecting the newly added change point θ_{k^*} and deleting it. This occurs with probability $1/(K+1)$ since the change points are selected uniformly at random and there are $K+1$ change points in the proposed state. Similarly the probability of the current proposal state is $1/(N-K)$ since there are $N-K$ possible timesteps to add. Since $u \sim U(0,1)$ then $P(u)=1$ and the reverse move is deterministic so its probability is also 1 (and omitted from the equation). Finally the Jacobian is given by

$$|J_{birth}| = \frac{(\lambda_1+\lambda_2)^2}{\lambda_0}$$

.

Then the new acceptance ratio is

$$\alpha_{birth}(\theta^*|\theta) = \frac{P(death)}{P(birth)} \frac{q_{(K+1\to K)}(\theta|\theta^*)}{q_{(K\to K+1)}(\theta^*|\theta) * P(u)} |J_{birth}|,$$

where

$$\frac{P(death)}{P(birth)} \frac{q_{(K+1\to K)}(\theta|\theta^*)}{q_{(K\to K+1)}(\theta^*|\theta) * P(u)}$$
$$= 1 * \frac{\frac{1}{K+1}}{\frac{1}{N-K} * 1} = \frac{N-K}{K+1}$$

5.4.2 `death_theta()`

For the death proposal we randomly select any of the current change points uniformly at random and remove it. Then the two λ values associated with the removed change point (call them λ_1 and λ_2 to match the above notation) are recombined deterministically as

$$\lambda_0 = \lambda_1^{\frac{\theta_m - \theta_{m-1}}{\theta_{m+1}-\theta_{m-1}}} * \lambda_2^{\frac{\theta_{m+1}-\theta_m}{\theta_{m+1}-\theta_{m-1}}}$$

where θ_m is the theta value that was removed and λ_0 is the new λ value for the merged interval.

The probability of proposing the death of change point θ_m is $1/K$ since it is chosen uniformly at random from $\boldsymbol{\theta}$ which has dimension K. The $q_{(K-1\to K)}(\theta|\theta^*)$ term is the probability of adding back the change point. Since the change points are birthed uniformly at random from all change point not currently in the $\boldsymbol{\theta}^*$ vector, the probability of birthing the θ_m that was deleted which is $1/(N-(K-1)) = 1/(N-K+1)$ And the Jacobian is

$$|J_{death}| = \frac{1}{|J_{birth}|} = \frac{\lambda_0}{(\lambda_1+\lambda_2)^2}$$

since the function is invertible.

Then the acceptance rate is computed as

$$\alpha_{death}(\theta^*|\theta) = \frac{P(birth)}{P(death)} \frac{q_{(K-1\to K)}(\theta|\theta^*) * P(u)}{q_{(K\to K-1)}(\theta^*|\theta)} \frac{1}{|J_{death}|}$$

where

$$\frac{P(birth)}{P(death)} \frac{q_{(K-1\to K)}(\theta|\theta^*) * P(u)}{q_{(K\to K-1)}(\theta^*|\theta)} = 1 * \frac{\frac{1}{N-K+1}}{\frac{1}{K}} = \frac{K}{N-K+1}.$$

5.5 Graph Proposals

The data structure for the graph is an adjacency list and is implemented using a list of numeric vectors in R. While not a true hashmap, numerically indexing the list allows for near $O(1)$ look-ups (Horner 2015). To make proposals in the graph space the following operators are used.

5.6 `add_edge_op()` and `delete_edge_op()`

The `add_edge_op()` functions proposes adding an edge to the graph by sampling uniformly at random from all edges not currently in the graph. Let N_v be the number of vertices (cities) in the graph and N_e the current number of edges. Then the probability of selecting an edge to add is $2/(N_v(N_v-1)-2N_e)$ by symmetry. This is determined as follows:

1. $N_v(N_v-1)$ is the total possible size of the adjacency list and N_e is the number of unique edges currently in the list. Then $N_v(N_v-1)-2N_e$ is the remaining "slots" in the adjacency list.
2. We draw uniformly at random $A \in \{1,\dots,N_v(N_v-1)-2N_e\}$ which represents an index of the edges not present in the graph.
3. Now we interate along the adjacency list `adj` whose length is the number of vertices N_v. Starting from $i=1$ if $A_i <= N_v - \text{length}(\text{adj}[i])$ then we know that the edge to be added is in $\text{adj}[i]$ and we can search for the correct edge in $O(|V|)$.
4. Otherwise if $A_i > N_v - \text{length}(\text{adj}[i])$ then we recompute $A_{i+1} = A_i - N_v + \text{length}(\text{adj}[i])$ increment $i = i+1$ and repeat.

Then $q(G^*_{N_e+1}|G_{N_e})$ the probability of proposing adding that particular edge occurs with probability $2/(N_v(N_v-1)-2N_e)$, since A_0 is chosen uniformly at random from $\{1,\dots,N_v(N_v-1)-2N_e\}$ and each edge e_{ij} is represented twice (once in adj[i]: $\{\dots$,j,$\dots\}$) and again in $\text{adj}[j] : \{\dots,i,\dots\}$.

An edge is deleted by randomly sampling from the adjancency list. The probability of $q(G_{N_e}|G^*_{N_e+1})$ of deleting the edge (given the graph where the proposed edge was added) occurs with probability $2/2(N_e+1) = 1/(N_e+1)$. Then the Hastings ratio for adding an edge is

$$\frac{q(G_{N_e}|G^*_{N_e+1})}{q(G^*_{N_e+1}|G_{N_e})} = \frac{\frac{1}{N_e+1}}{\frac{2}{N_v(N_v-1)-2N_e}} = \frac{N_v(N_v-1)-2N_e}{2(N_e+1)},$$

and the Hastings ratio for removing an edge is given by

$$\begin{aligned}\frac{q(G_{N_e}|G^*_{N_e-1})}{q(G^*_{N_e-1}|G_{N_e})} &= \frac{\frac{2}{N_v(N_v-1)-2(N_e-1)}}{\frac{1}{N_e}} \\ &= \frac{2N_e}{N_v(N_v-1)-2(N_e-1)}.\end{aligned}$$

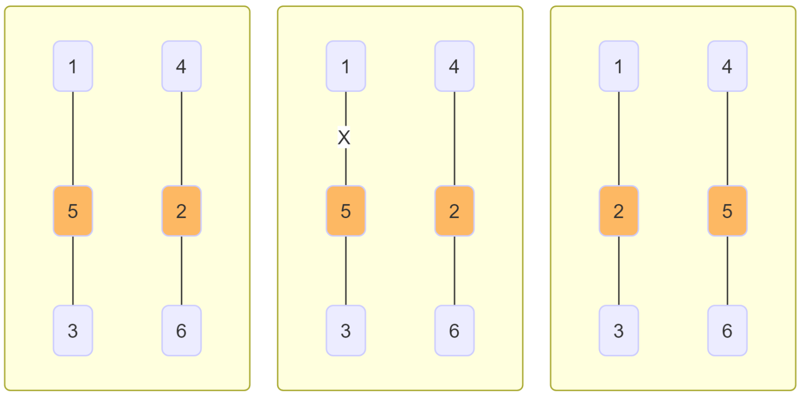

Figure 3: An example graph configuration that might necessitate a degree preserving swap. **Left** and **right**: two configurations of graphs whose vertices have the same degree, **middle**: a potential intermediary proposal. With one-step edge adding and deleting the likelihood of switching between these two graph is low if the migration rate is high enough.

5.7 `degree_preserving()`

The degree preserving swap was implemented to allow changes to the graph structure while maintaining the degree of each vertex. This is because with large migration rates, the degree of the cities strongly influences the likelihood. Let's assume that the migration rate $m = 1$, then in Figure 3, city 5 and 2 have double the counts of cities 1, 3, 4, and 6. Let's also assume that the configuration on the right is the true graph. If the MCMC algorithm proposes the graph on the left first, to get to the graph on the right would require say, disconnecting $1 - 5$ as seen in the middle. With such a high migration rate, the cities 5 and 2 would never be swapped (it would require transitioning to a graph where city 5 would have degree 2 and city 1 degree 0). The degree preserving swap handles directly swapping between the left and right graphs without passing through the middle.

This was implemented by selecting two edges (v11, v12) and (v21, v22) and attempting to form the edges (v11, v22) and (v12, v21). If both edges are not already in the graph then the swap is made. If either or both edges exists, then the proposed swap is rejected. Since the proposed swap is reversed by randomly selecting (v11, v22) and (v12, v21), the Hastings ratio is 1 and the acceptance ratio is the ratio of the likelihoods.

5.8 `rewire()`

The `rewire()` operator's goal is similar to that of the `degree_preserving()` operator. It tries to select an edge, delete it, and then reattach one of the vertices randomly. The operator selects uniformly at random an edge (v1, v2) and deletes it from the adjacency list. It then randomly samples a vertex v3 and forms the edge (v2, v3). Since the reverse move is selecting the edge (v2, v3) and then rewiring (v1, v2), the Hastings ratio is 1.

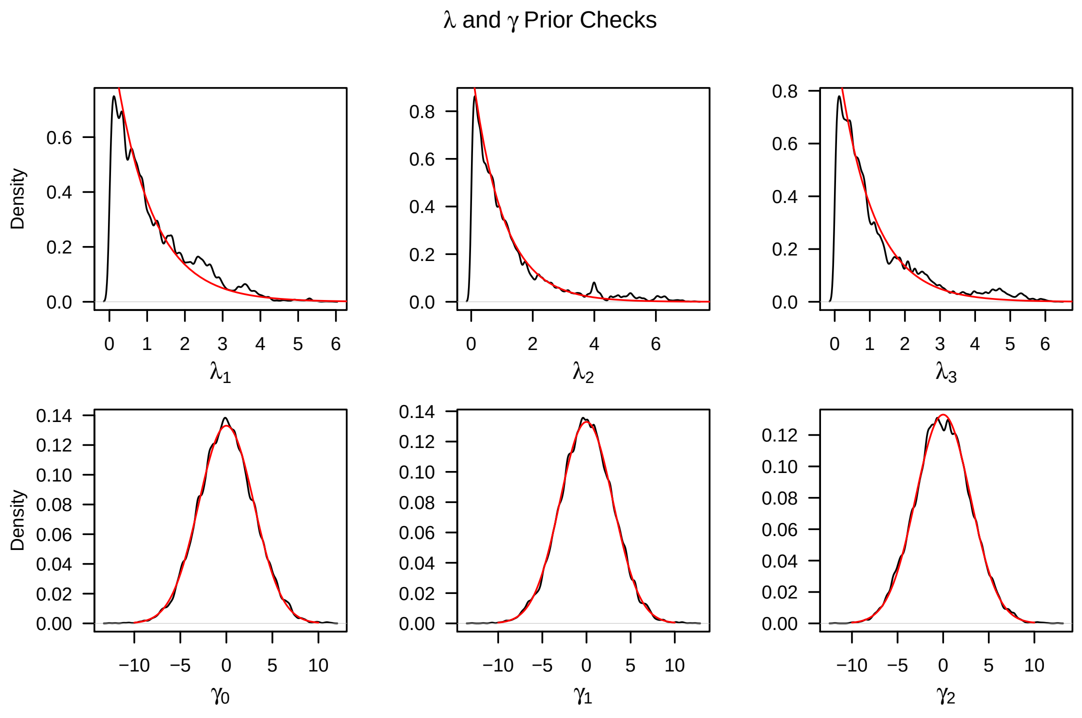

Figure 4: Prior Checks for γ and λ. Only 3 λ values are displayed. The KDE smoothed distribution of values are plotted against the true density values of the priors (that is $P(\lambda) \sim \text{Gamma}(1,1)$ and $P(\gamma) \sim \text{Norm}(0,3)$. The differences appear to be due to sampling and smoothing. λ_1 shows the least agreement with the prior.

5.9 `change_migration()`

The migration proposal is drawn from a Unif $\sim (m-0.01, m+0.01)$. Since the Uniform distribution is symmetric, the Hastings ratio is 1 so the acceptance ratio is the ratio of the likelihoods between the current and proposed steps. The Uniform was chosen over the Normal, since the variance of the Normal is larger and can propose much larger jumps. This appears to cause many rejections which slows the exploration of the space.

6 Results and Discussion

6.1 Prior Checks

Here the likelihood ratio was set to 1 so the only factor in accepting or rejecting a state was the priors. As such we would expect to see the posterior distributions of each parameter to be their respective prior distributions. This helps check that the Hastings ratio is calculated correctly and the implementation of the operators (excluding the likelihood) are correct. The following data was collected by running 100,000 iterations (thinned to 10,000 samples) of proposing to update either the λ vector, γ vector, adding a change point $(K \to K+1)$ or removing a change point $(K \to K-1)$.

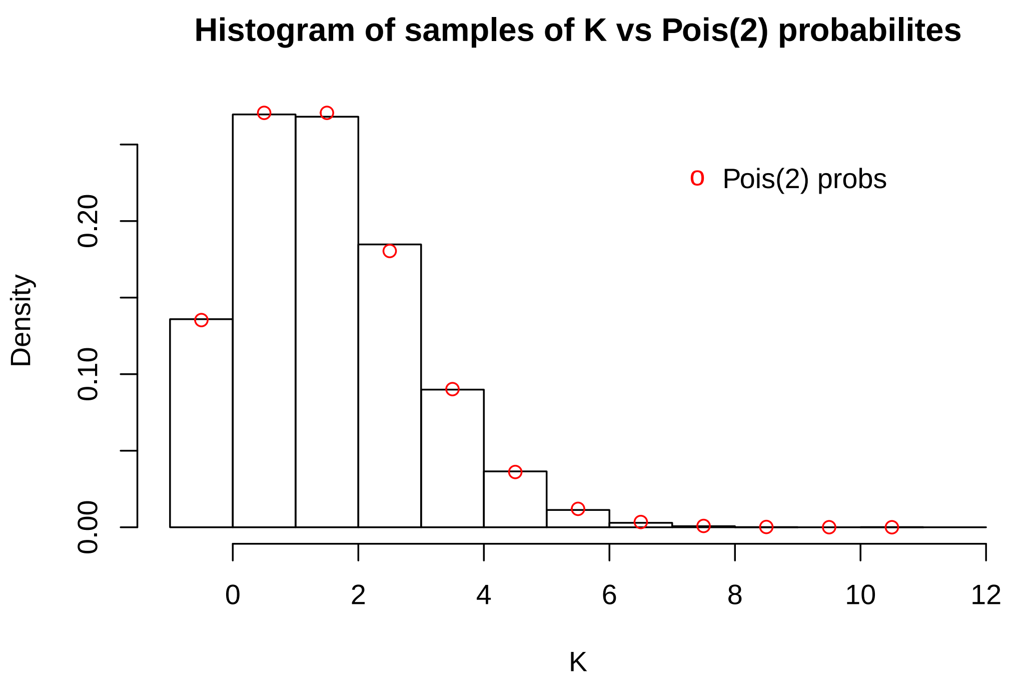

Figure 5: Sample probabilities of K overlayed with true probabilities. Overall this shows a good agreement between the sample probabilities and the true probabilties.

The priors seen here are

$$\vec{\lambda} \sim \text{Gamma}(1,1),$$

$$\vec{\gamma} \sim \text{Norm}(0,3),$$

$$K \sim \text{Unif}(0,N).$$

In Figure 4 we have plots of the samples of the λ values (first 3) and the γ values. Each is overlayed with the density of their respective priors in red. Overall there appears to be a good match between the sampled values and their prior distributions.

In Figure 5 we have a frequency histogram of the sampled values of K (the number of change points) overlayed with a red circle. The red circle represents the appropriate density value from the prior distribution of K. Overall we see a good qualitative fit between the values sampled from the posterior and the prior distribution.

6.2 Inference on Simulated Data

The following simulation parameters are taken from Held et al. (2006)'s simulations.

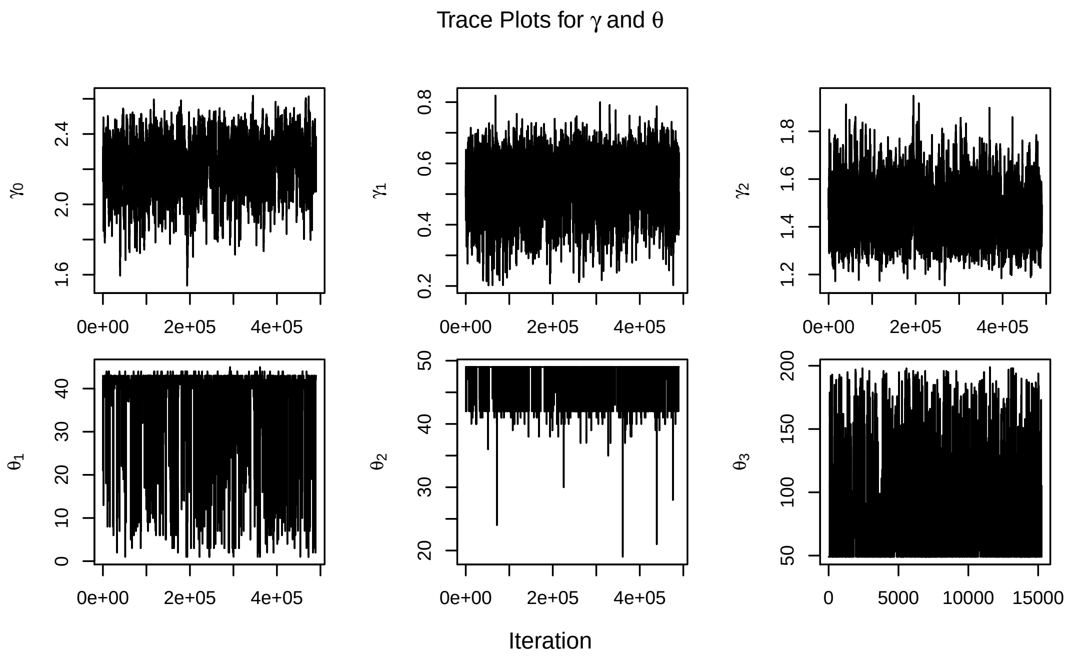

Figure 6: Trace plots for $\boldsymbol{\gamma}$ and $\boldsymbol{\theta}$. The trace plots seem relatively well mixed. The autocorrelation is likely a function of the correlation between the each set of $\boldsymbol{\gamma}$ and $\boldsymbol{\theta}$ respectively. Thinning could potentially reduce the autocorrelation between sample draws.

parameters	simulation values
K: number of changespoints	2
$\boldsymbol{\lambda}$: epidemic parameters	0.7, 1.2, 0.7
$\boldsymbol{\gamma}$: endemic parameters	log(10), 0.5, 1.5
$\boldsymbol{\theta}$: change points	39, 49

Figure 6 are trace plots, which are time series of the parameter draws. Typically a “good” trace plot shows no obvious autocorrelation (which can indicate a proposal step that is too small) or flat portions (proposal steps that are too large). Trace plots with these patterns indicate that the sampler has not converged to the posterior distribution of the parameter. “Tuning” or adjusting the proposal jump size can help fix these patterns, with the end goal of being able to efficiently explore the posterior distribution of the parameter (Gelman et al. 2013,p. 296). The trace plots for the γ parameters seem to have a slight pattern particularly in the γ_0 trace plot.

This is likely due to the correlation (see Figure 7) between γ_0, γ_1 and γ_2 due to their nature of parameterizing a cyclical function. The same is true for the θ parameters; they are constrained such that $\theta_1 < \theta_2 < \theta_3$. The dependence can cause inefficient sampling by proposing moves that are outside these narrow regions/combinations of parameter values (Turner et al. 2013). This autocorrelation seen in the trace plots could possibly be reduced via thinning i.e. taking every k samples instead of every sample. Gelman et al. (2013) mentions that they have found it useful to store no more than a total of 1000 iterations.

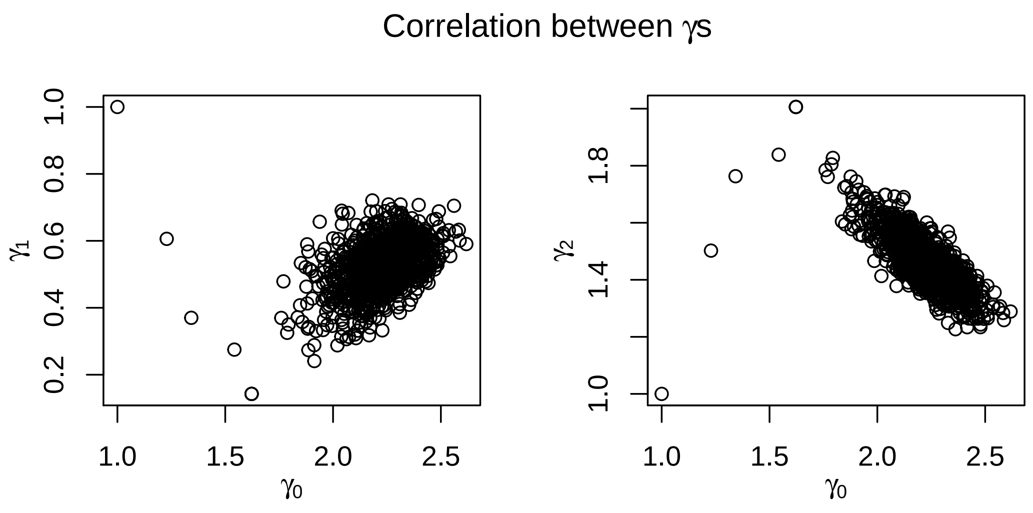

Figure 7: Plots of γ_0 vs γ_1 and γ_0 vs γ_2. Each point represents the corresponding parameter value at the same sample draw, i.e., the values of each parameter when the sample was accepted. The strong correlation between the parameters can lead to poor mixing. Note the tails are due to the initial values of the MCMC chain.

The θ_3 trace plot is of note because it is only valid for when $K = 3$; the θ_3 does not exist otherwise.

The $\boldsymbol{\lambda}$ trace plots show that the λ_2 and λ_3 traces appear to jump tightly around 1.2 and 0.7 respectively while also jumping to 0.7 and 1.2 respectively. This is an artifact of the dimension change in the model. When $K = 2$, λ_2 is tightly fixed around 1.2 and λ_3 at 0.7 but when $K = 2$, λ_3 can take the role of fitting the outbreak while λ_4 models to the return 0.7, the baseline.

Figure 10 shows the histograms of the λ and γ samples

parameters	lower	upper
K	2	2
$\boldsymbol{\lambda_1}$	0.67	0.82
$\boldsymbol{\lambda_2}$	1.09	1.29
$\boldsymbol{\lambda_3}$	0.69	0.80
$\boldsymbol{\gamma_0}$	1.95	2.50
$\boldsymbol{\gamma_1}$	0.36	0.67
$\boldsymbol{\gamma_2}$	1.28	1.68
$\boldsymbol{\theta_1}$	39	49
$\boldsymbol{\theta_2}$	39	49

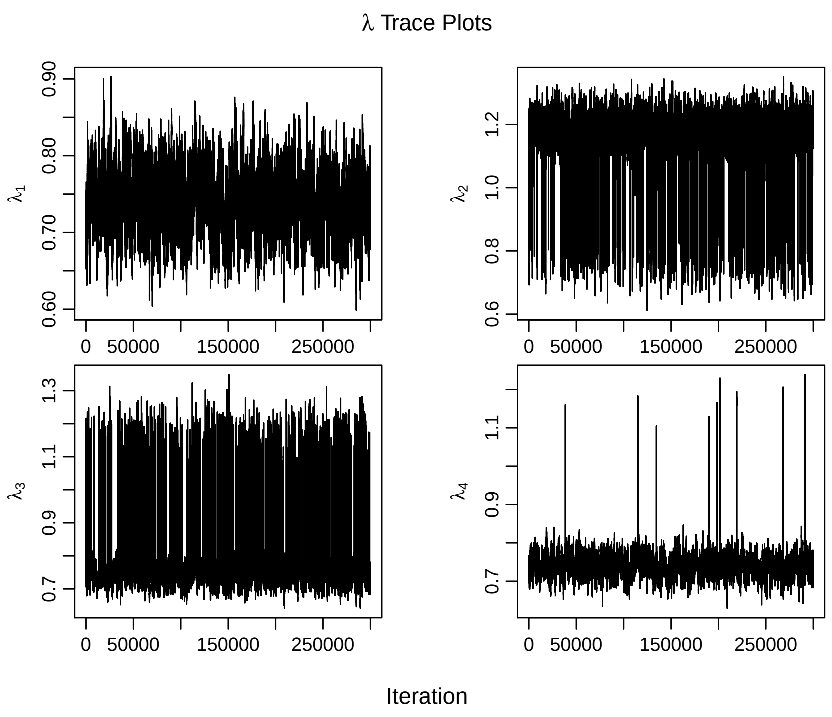

Figure 8: Trace plots for all λ values. Overall the trace plots show that the samplers are mixing well with some minor autocorrelation that could potentially be resolved with thinning or more samples. The raw trace plots include the λ values when there are 3 and 4 separate λ's and is the cause of the random jumps.

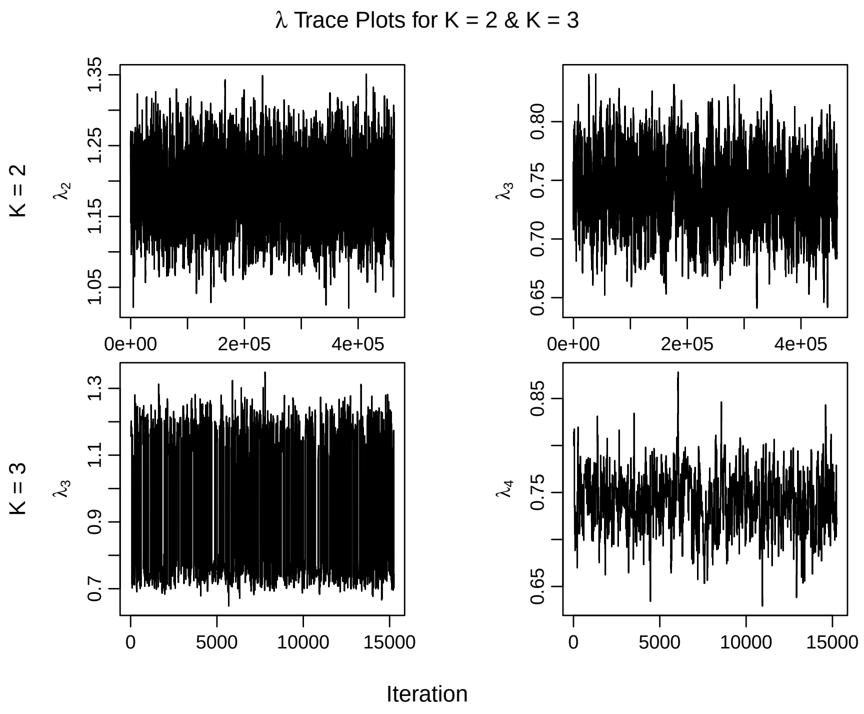

Figure 9: Trace plots filtered for K = 2 and K = 3 showing how the different number of change points affects the traces of the individual λ values.

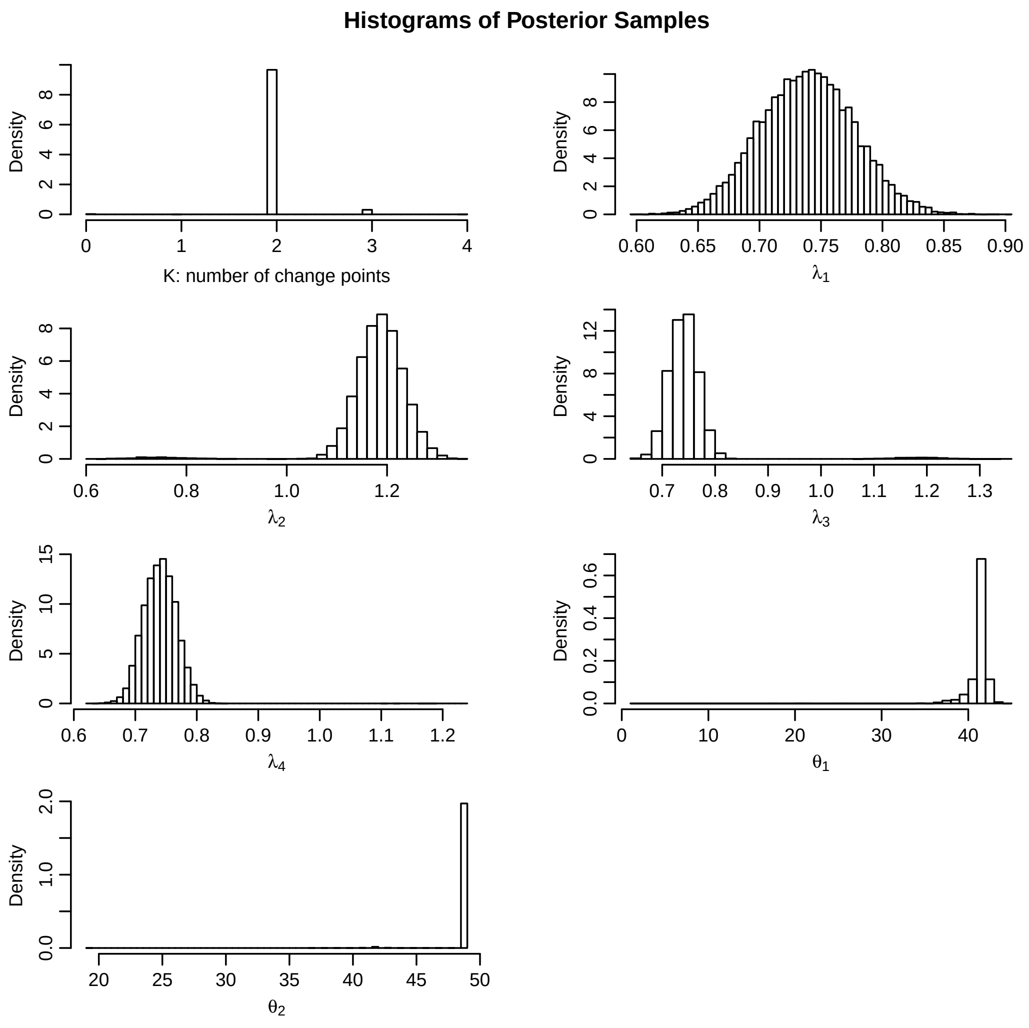

Figure 10: Histogram of posterior samples from K, λ_1, λ_2, λ_3, λ_4, θ_1 and θ_2.

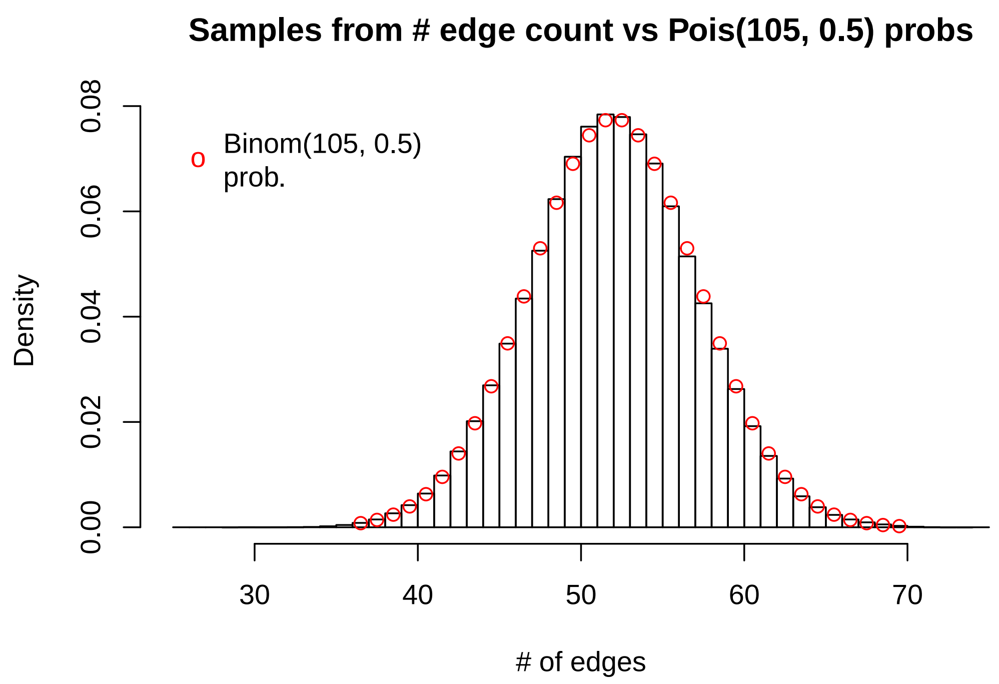

Figure 11: Posterior probabilities of the number of edge counts, overlayed with true probabilities. Overall this shows a good agreement between the posterior probabilities and the true probabilties.

6.3 Graph Inference on Fixed Two-Component Model

6.3.1 Posterior distribution of number of edges matches prior distribution

Here we set the prior ratio and the likelihood ratio to always equal to 1. In this case the posterior distribution should only be driven by the proposal mechanism/operator and correctness of the operator can be determined. The `add_edge_op()/del_edge_op()` operators, should propose each graph with equal probability. In this case the distribution of the number of edges should be $\text{Binom}(105, 0.5)$ since there are $\binom{N_v}{N_e}$ graphs with N_e many edges.

6.3.2 Inference on the number of edges/p and migration rate

Next I simulated a dataset on a graph. Inference is carried out on the graph and the migration rate, while the other parameters are fixed to their true values. A single interation of the sampler proposes adding or removing an edge (with equal probability) 25 times before proposing a migration rate change. This allows the sampler to better explore the highly correlated posterior distribution. For the following data 1,000,000 samples were drawn and thinned to every 200 samples (for a total for 5000 samples). This was run for 8 chains which were used for convergence diagnostics and then pooled together.

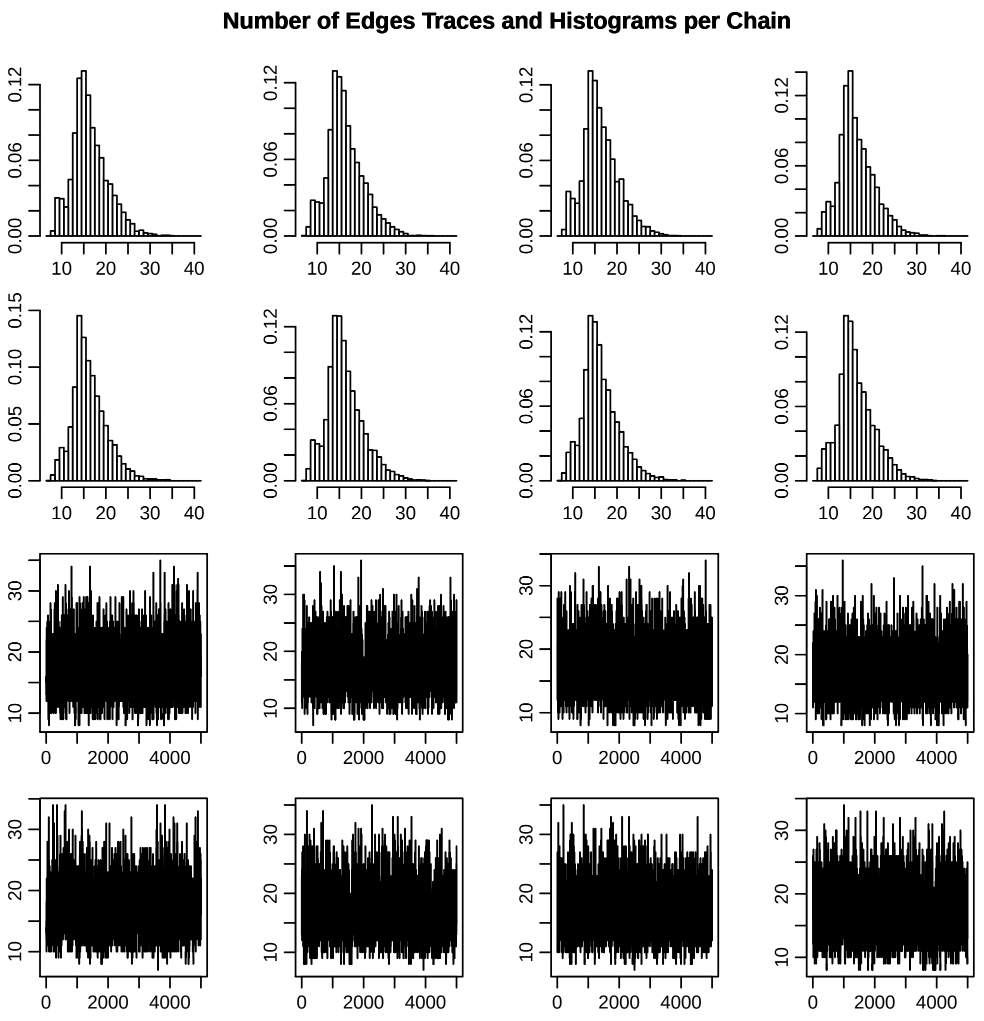

Figure 12: Histogram and trace of posterior samples of the number of edges in the graph.

parameters	simulation values
K: number of changespoints	2
$\boldsymbol{\lambda}$: epidemic parameters	0.35, 0.6, 0.35
$\boldsymbol{\gamma}$: endemic parameters	log(10)/2, 0.25, 0.75
$\boldsymbol{\theta}$: change points	39, 49
p: Erdos-Renyi parameter	0.15
true number of edges	16
N_v: number of vertices	15
m: migration rate	0.15

The individual trace plots in Figure 12 and Figure 13 shows that each chain appears to mix rather slowly. This could be due to the inherent difficulty in exploring the

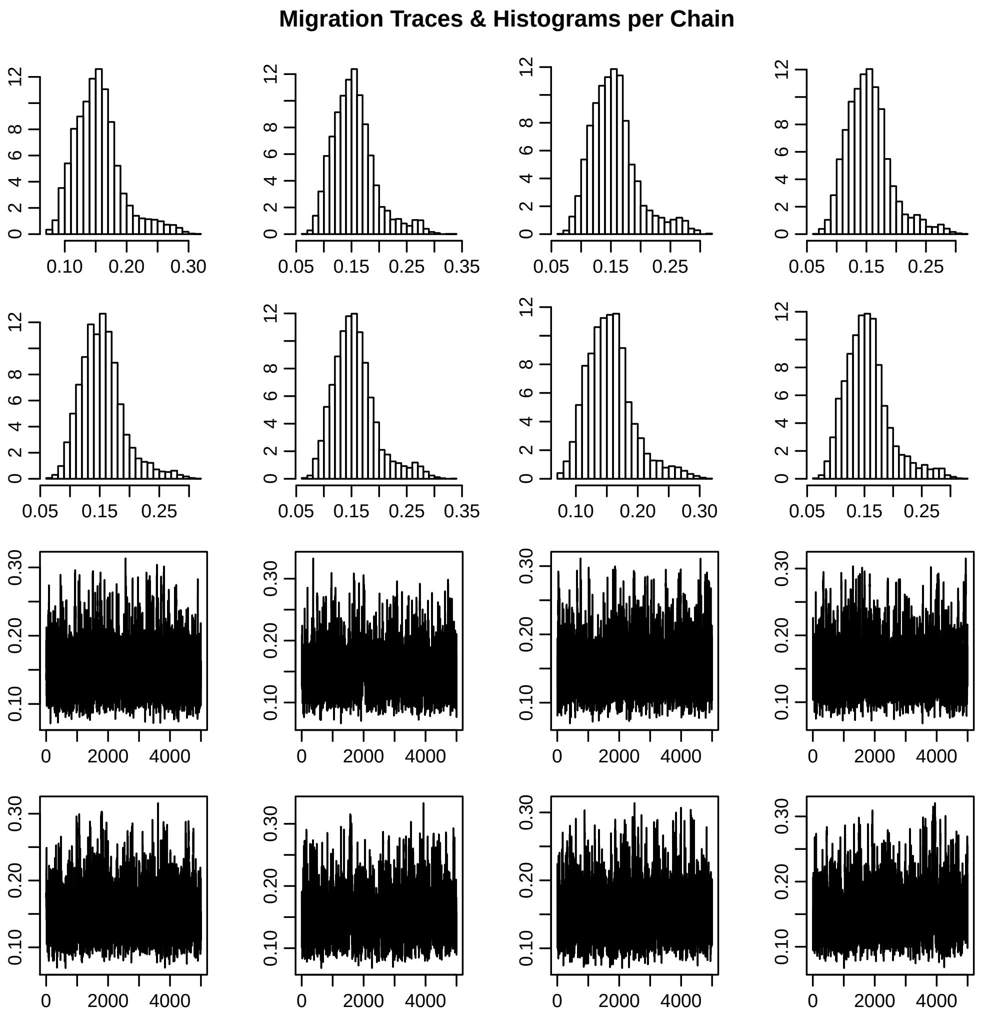

Figure 13: Histogram and trace of posterior samples of the migration rate. This shows relatively slow mixing. This could be from a combination of the inherent difficulty of exploring the space as well as poor tuning of the migration proposal.

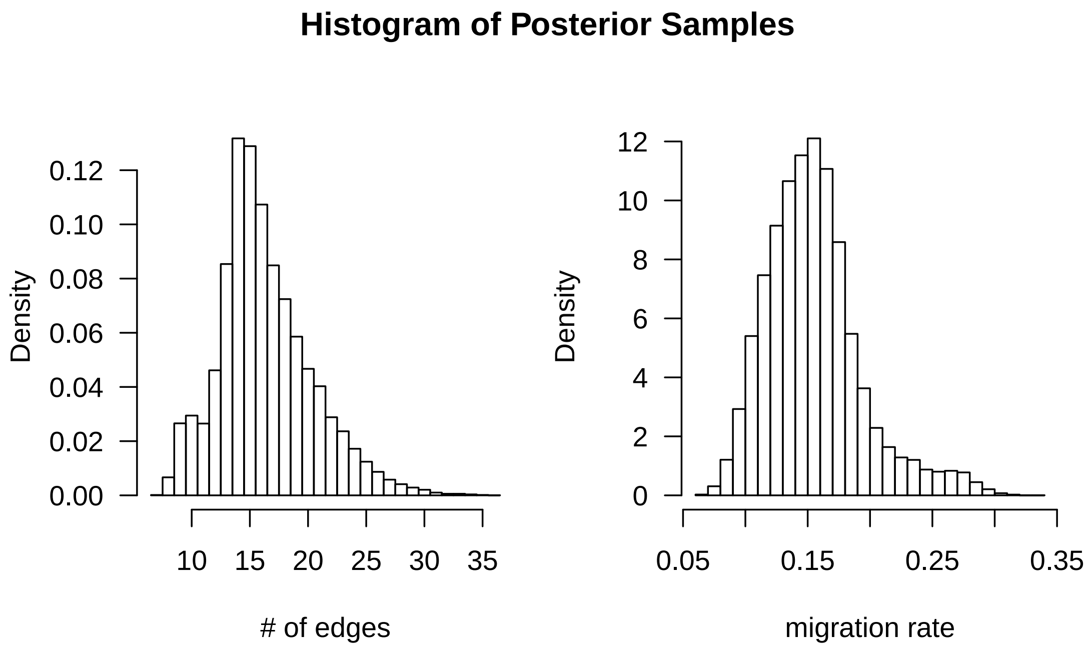

Figure 14: Histograms of pooled samples.

correlated migration rate/graph space with the current operators.

After diagnostics, samples from each of the chains were pooled together. The effective sample size for the pooled chains are `11759` and `6667` for the number of edges and the migration rate respectively. Figure 14 shows histograms of the pooled posterior samples of the number of edges and the migration rate. The mean of the distribution of the number of edges is 16.31 edges and median is 16 edges. The mean and median are close to the true number of edges in the graph, 16. The mean migration rate is 0.155 and the median migration rate is 0.151.

Figure 15 shows a hex plot of the joint posterior samples of p (which is computed by $\frac{\text{num. edges}}{105}$ where 105 is the maximum number of edges in a graph with 15 vertices) and migration rate. It shows a strong negative correlation, which is expected. If the migration rate is smaller than the true value m (say $0.5m$) then the a city connected to d cities would need to connect to $2d$ many cities to have the "correct" counts. Similarly if the migration rate is higher than the true migration, then the number of edges would need to be lower than the true number of edges.

The 90% HPDI for the number of edges and the migration rate are (9, 24) and (0.08, 0.24) respectively and cover the true values.

parameters	lower	upper
# of edges	9	24
migration	0.08	0.24

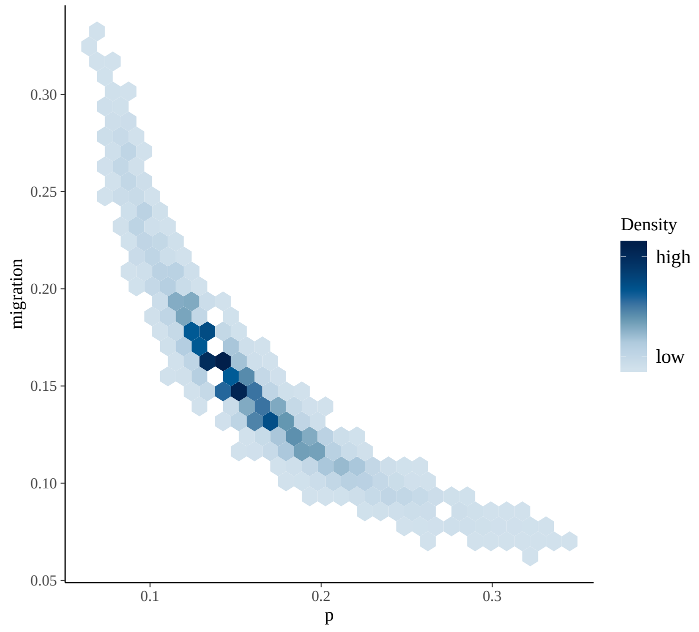

Figure 15: Joint posterior hex plot between migration rate and p the parameter for the Erdos-Renyi graph. The p value is directly computed from the number of edges by dividing by the total possible edges which is 105. The plot shows a strong negative correlation between p and the migration rate.

7 Conclusion

In this thesis, I've re-implemented a Metropolis-Hastings version Held et al. (2006)'s two-component model and tested the model on simulated data. The MH version was able to recover the true parameters, with the 90% HPDI covering the true parameters. I then extended the model to allow for multiple time series and a graph informed dependency structure which was implemented as an adjacency list. To perform inference on the graph, several graph operators were implemented and tested. Special attention was payed to computational efficiency of the operators and the likelihood computation in order to allow the model to scale to larger, more realistic data sets like the Germany *E. coli* dataset which contains data from ~ 400 districts each with ~ 600 time points each.

7.1 Code Additions and Limitations

Besides some basic refactoring, some features need to be added to the code. One major feature is storing and comparing sampled graphs. During testing against simulated datasets, while the posterior mean of the number of edges matches the true number of edges, there's the possiblity that none of the sampled graphs will actually match the true graph;it just happened that the number of edges is correct, but the sampled graphs are completely different from the true graph. While this is easy to check manually on small graphs, for realistically sized graphs this is unfeasible. To check this, a method to check that the sampled graphs (with the true number of edges) matches the true graph, graphs with less than the true number of edges are subgraphs of the true graph and graphs with more than the true number of edges are supergraphs of the true graph.

This however may be computationally prohibitive.

Another feature is automated testing. With the ability to run multiple chains, a system to experiment with different parameters should be coded. For example, a well-calibrated study requires sampling several sets of parameters and testing the coverage of the credible intervals. With consumer grade processors having up to 32 threads, an automated well-calibrated study should be able to run 32 sets simultaneously. Additionally experiments testing different operators, samplers, etc. could be setup and run in an efficient and easily reproducible manner. While parallel chains have been implemented, due to time and resource constraints, a well-calibrated study was unable to be done in time.

7.2 Data Analysis of Real Data

While the two-component parameters were held fixed to make test the graph extension, this strategy could also work on a real data set. The multiple time series can be compressed down to a univariate time series by summing the data across each city at a particular time step. Then the two-component parameters can be estimated on this compressed time series. Then holding these parameters fixed, we could estimate the graph on the full, uncompressed data. This should give identifical estimates for the endemic γ parameters, the change points K and the location of the change points $\vec{\theta}$ since these are independent of the connections between cities. Then only the λ epidemic parameters need to estimated simulataneously with the graph and migration rates. This would greatly simplify inference compared with estimating all the parameters at once.

7.3 Future Extensions

While this thesis only covers a constant migration rate, the end goal would be to model the city (vertex) specific and between city (edge) attributes that could affect the migration rate. For example the size of the cities (city specific) and the distance between cities (between city attribute) could be important predictors of migration rates. In the case of the 2011 outbreak of *E. coli* distance to the Autobahn and shipping routes might be important since the *E. coli* was carried on contaminated produce. Determining what factors are most important in determining migration rates could give clues to e.g., public health officials, what locations are susceptible to outbreaks and importantly, what are potential interventions to reduce outbreaks.

The migration rate m could be modelled as a logistic function of these city parameters $m \sim logit(\beta_0 + \beta_1 * X_1 + \beta_2 * X_2 + \dots)$ where the X_i's can be both city specific attributes or edge specific attributes. One simple example could be

$$m_{i,j} \sim \text{logit}(\text{pop. city } i * X_1 + \text{pop. city } j * X_2 + \frac{\text{pop. city } i * \text{pop. city } j}{d(\text{city } i, \text{city } j)} * X_3),$$

where $d(i,j)$ is some distance metric (e.g. euclidean, public transportation distance). The populations of city i and city j are city specific and $\frac{\text{pop. city } i * \text{pop. city } j}{d(\text{city } i, \text{city } j)}$ is an edge specfic attribute. Then m_{ij} would be the migration rate from city i to city j and $m_{j,i}$ the rate from city j to city i.

Furthermore this would allow the graph to be changed from an undirected, unweighted graph to a fully-connected directed graph with weighted edges, where the weight of $e_{i,j}$ is $m_{i,j}$. If the number of change points K and the location of the change points $\vec{\theta}$ is held fixed (e.g., by being previously estimated), then all the parameters in the model are continuous and computational issues regarding the discrete nature of the undirected graph can be side-stepped. For example, having all the parameters be continuous could allow for gradient-based sampling methods such as Hamitonian Monte Carlo which holds the promise of efficient sampling with little to no tuning required.

8 Appendix/Code

```
pkgs = c("compiler", "profvis", "igraph","microbenchmark", "Rcpp")
inst = lapply(pkgs, library, character.only = TRUE)

#rcpp functions used to speed up parts of the likelihood computation

#cpp version of an unnormalized dpois where log = TRUE
#pre: two numeric vectors of the same length
#post: unnormalized loglikelihood

cppFunction("double mypois(NumericVector data, NumericVector lambdas) {
  int n = data.size();
  NumericVector plik(n);
  plik = -lambdas + data*log(lambdas);

  double tlik = 0;

  for(int i = 0; i < n; ++i){
    tlik += plik[i];
  }

  return tlik;
  }")

cppFunction("NumericVector diff(NumericVector x){
   return diff(x);
}")

cppFunction("NumericVector sort(NumericVector x) {
   NumericVector y = clone(x);
   std::sort(y.begin(), y.end());
   return y;
}")

#cpp version of an unnormalized dpois where log = TRUE
#pre: data vector of counts, lambdas a vector of lambda values with
# length equal to the number of time points
#post: unnormalized loglikelihood
mypois <- function(data, lambdas) {
  sum(-lambdas + data*log(lambdas))
}
```

8.1 Data Simulation

```
#this code block simulates data used in the MCMC
#----------graph initialization-------#
```

```
# edgelist for handrawn graph
el = matrix(c(1, 2,
              1, 3,
              1, 4,
              1, 5,
              1, 6,
              2, 7,
              2, 8,
              7, 8,
              6, 9,
              9, 10),
            byrow = TRUE,
            ncol = 2)

g_city = graph_from_edgelist(el, directed = FALSE)
############################################

#helpers
#unclass from igraph for later use
unclass_adjlist <- function(adjlist) {
  adjlist <- unclass(adjlist)
  for (i in 1:length(adjlist)) {
    adjlist[[i]] = unclass(adjlist[[i]])
  }
  return(adjlist)
}

# ER random graph generation
# g_city = sample_gnp(15, 0.175)
# plot(g_city)

adjlist = as_adj_list(g_city, mode = 'all')
adjlist <- unclass_adjlist(adjlist)

#---simulate endemic component based on paper's values----#
N = 199     #total epochs/time steps
init = 10   #Z_0 value is the initial number of people infected

#------- setting endemic parameters-----------------------#
gamma  <- c(log(10), 0.5, 1.5) / 2
rho <- 2 * pi / 52
t = 1:N
INDEP <-
  as.matrix(data.frame(
    x0 = 1,
    x1 = sin(rho * t),
    x2 = cos(rho * t * 1.5)
  ))
nu_t <- as.matrix(INDEP) %*% gamma
ende_lambda = rep(0, N)
ende_lambda = exp(nu_t)

#-----setting epidemic lambda values -----------
K <- 2              #changepoints
theta  <- c(39, 49) #location of changepoints
migration <- 0.15

lambda <- c(0.5, 1.5, 0.5) / 2   #lambda values
epi_lambda = rep(lambda, diff(c(0, theta, N))) #lambda vector

#-----------------counts matrix ---------#
#separate counts from the ende and epi components
ende_c = matrix(0, nrow = gorder(g_city), ncol = N)
epi_c = matrix(0, nrow = gorder(g_city), ncol = N)

counts = matrix(0, nrow = gorder(g_city), ncol = N)
counts[, 1] = init



#get list of all adjacent vertices
adj = as_adj_list(g_city)

#-----------compute data-----------------#

for (i in 1:(N - 1)) {
  #all rows/cities get same value from the ende
  ende_c[, i] = rpois(gorder(g_city), ende_lambda[i])

  for (j in 1:gorder(g_city)) {
    counts_j = sum(counts[adj[[j]], i]) * migration
    #sum all adjacent counts
    counts_j = counts_j + counts[j, i]
    #add self count
    epi_c[j, i] = rpois(1, epi_lambda[i] * counts_j)
    #generate new counts
    counts[j , i + 1] = ende_c[j, i] + epi_c[j, i]
    #append new counts
  }
}
###########################################
#list of actual parameter values; useful for testing

actual <- list(
  K = K,
  theta = theta,
  lambda = lambda,
  LL = -Inf,
  gamma = gamma,
  INDEP = INDEP,
  graph = g_city,
  adj = adjlist,
  counts = counts,
  migration = migration
)
```

8.2 compute_log_prior()

```
#' Computes and returns the log prior of the parameters
#'
#' @param input a list containing the parameters
#' @return the log prior probability of the parameters

compute_log_prior <- function(input){
  K = input$K
  theta = input$theta
  lambda = input$lambda
  gamma = input$gamma

  adj = input$adj
  edge_count = input$edge_count

  prior_star <- -log( choose( choose(length(adj), 2), edge_count))
   -log(choose(N, K)) +
     sum(dgamma(lambda, 1, 1, log = TRUE)) +
     sum(dnorm(gamma, 0, 3, log = TRUE)) +
     dpois(K, 2, log = TRUE)

  return(prior_star)
}
```

8.3 Likelihood Function

8.3.1 compute_log_like_diff()

```
#' Computes the differences in the log-likelihoods between the new graph
#' and the old graph and return the difference
#' @param input a list containing the parameters
#'  (including the proposed adjancency list)
#' the old adjacency list and the two vertices that are changed
#' @return the log prior probability of the parameters
```

```r
compute_log_like_diff <- function(input, old_adj, v1, v2) {
  K = input$K
  theta = input$theta
  lambda = input$lambda
  gamma = input$gamma
  INDEP = input$INDEP
  LL = input$LL
  adj = input$adj
  counts = input$counts
  migration = input$migration

  #replicate lambda values to the
  #correct length according to the changepoints
  if (K == 0) {
    vepi_star <- rep(lambda, N)
  } else {
    vepi_star <- rep(lambda, diff(c(0, theta, N)))
  }

  #generate the infection counts matrix according to which
  # vertices/cities are adjacent


  #---------------new data------------------------------#
  inf_counts = matrix(0, nrow = 2, ncol = N)

  inf_counts[1, ] = .colSums(counts[adj[[v1]], , drop = FALSE],
                             length(adj[[v1]]), N) *
    migration + counts[v1, ]

  inf_counts[2, ] = .colSums(counts[adj[[v2]], , drop = FALSE],

                             length(adj[[v2]]), N) *
    migration + counts[v2, ]

  #######################################################

  #------------------old data-----------------------------#

  old_inf_counts = matrix(0, nrow = 2, ncol = N)

  old_inf_counts[1, ] = .colSums(counts[old_adj[[v1]], , drop = FALSE]
                                 , length(old_adj[[v1]]), N) *
    migration + counts[v1,]

  old_inf_counts[2, ] = .colSums(counts[old_adj[[v2]], , drop = FALSE]
                                , length(old_adj[[v2]]), N) *
    migration + counts[v2, ]
  #######################################################


  # ------- endemic and epidemic parameter matrices -----
  nu <- exp(INDEP %*% gamma)  #generate nu parameter for endemic
  mat_ende_lambda <- matrix(rep(nu, 2), nrow = 2, byrow = TRUE)

  #creates matrix of lambdas from estimated lambda row
  mat_lambdas <-
    matrix(rep(vepi_star, 2), nrow = 2, byrow = TRUE)
  #######################################################

  #-----------compute the log-likelihoods---------------#
  #data is for 2:N, first column is an initial value
  new_log_lik <-
    mypois(counts[c(v1, v2), 2:N],
           inf_counts[, 1:(N - 1)] * mat_lambdas[, 1:(N - 1)]
           + mat_ende_lambda[, 1:(N - 1)])

  old_log_lik <-
    mypois(counts[c(v1, v2), 2:N],
           old_inf_counts[, 1:(N - 1)] * mat_lambdas[, 1:(N - 1)]
           + mat_ende_lambda[, 1:(N - 1)])
  ##################################################S#####

  return(new_log_lik - old_log_lik)
}
```

8.3.2 compute_log_like()

```
#' Computes and returns the unnormalized graph
#' log likelihood of the parameters. Recomputes all the data.
#' @param input a list containing the parameters
#' @return the log likelihood probability of the parameters

compute_log_like <- function(input) {
  K = input$K
  theta = input$theta
  lambda = input$lambda
  gamma = input$gamma
  INDEP = input$INDEP
  LL = input$LL
  adj = input$adj
  counts = input$counts
  migration = input$migration
  #replicate lambda values to the correct
  #length according to the changepoints
  if (K == 0) {
    vepi_star <- rep(lambda, N)
  } else {
    vepi_star <- rep(lambda, diff(c(0, theta, N)))
  }

  #generate the infection counts matrix according to which
  # vertices/cities are adjacent
  ord = length(adj)  #number of vertices in the graph
  inf_counts = matrix(0, nrow = ord, ncol = N)

  for (i in 1:ord) {
    #inf_counts[i, ] = colSums(counts[adj[[i]], , drop = FALSE])*
    #migration + counts[i, ]
    inf_counts[i, ] = .colSums(counts[adj[[i]], , drop = FALSE]
                      , length(adj[[i]]), N)*migration + counts[i, ]
  }

  # ------- endemic and epidemic parameter matrices ----
  nu <- exp(INDEP %*% gamma)         #generate nu parameter for endemic
  #creates matrix of repeated rows
  mat_ende_lambda <- matrix(rep(nu, ord), nrow = ord, byrow = TRUE)

  #creates matrix of lambdas from estimated lambda row
  mat_lambdas <-
    matrix(rep(vepi_star, ord), nrow = ord, byrow = TRUE)

  #compute the log-likelihood
  #data is for 2:N, first column is an initial value
  log_lik <-
    mypois(counts[, 2:N],
           inf_counts[, 1:(N - 1)] * mat_lambdas[, 1:(N - 1)]
           + mat_ende_lambda[, 1:(N - 1)])

  return(log_lik)
}

#-------for prior testing--------

 # compute_log_like_diff <- function(input, adj, v1,v2){
 #   0
 # }
```

8.4 Non-graph Operators

8.4.1 change_migration()

```
#' Proposes a new migration value using a MRW with a uniform
#' distribution
#' @param input a list containing the parameters
#' @return a list of either the newly proposed
#' state (input_star) or the state

change_migration <- function(input) {
  LL <- input$LL
  migration <- input$migration

  #propose new state from normal
  migration_star <- migration + runif(1, -0.01, 0.01)

  if (migration_star > 1 || migration_star < 0){
    return(input)
  }

  #repackage list
  input_star <- input
  input_star$migration <- migration_star

  LL_star <- compute_log_like(input_star) #compute log likelihood
  prior_star <-  compute_log_prior(input_star) #compute LOG prior
  post_star <- LL_star + prior_star #new log posterior

  U = runif(1)

  if (log(U) < post_star - LL) {
    input_star$LL <- post_star
    return(input_star)
  } else {
    return(input)
  }

}
```

8.4.2 change_gamma(), death_theta(), birth_theta()

```
#' Proposes a new gamma parameter vector from a random normal
#'  Metropolis Random Walk (MRW)
#' @param input a list containing the current parameter state
#' @return a list of either the newly proposed state (input_star) or the
#' current Metropolis Random Walk (MRW)
```

```
change_gamma <- function(input) {
  LL <- input$LL
  gamma <- input$gamma

  #propose new state from normal
  gamma_star <- gamma + rnorm(3, 0, 0.01)

  #repackage list
  input_star <- input
  input_star$gamma <- gamma_star

  LL_star <- compute_log_like(input_star) #compute log likelihood
  prior_star <-  compute_log_prior(input_star) #compute LOG prior
  post_star <- LL_star + prior_star #new log posterior

  U = runif(1)

  if (log(U) < post_star - LL) {
    input_star$LL <- post_star
    return(input_star)
  } else {
    return(input)
  }

}


#' Proposes removing a changepoint. Uses a Reversible Jump MCMC (RJMCMC)
#'as seen in Green 1995
#' @param input a list containing the current parameter state
#' @return a list of either the newly proposed state (input_star) or the
#' current state (input)


death_theta <- function(input) {
  K = input$K
  theta = input$theta
  lambda = input$lambda
  LL = input$LL

  #if K = 0 (no change points) then automatically reject this proposal
  if (K == 0) {
    return(input)
  }

  #randomly sample from the current list of changepoints
  m <- ceiling(runif(1, 0, length(theta)))
```

```
theta_star <- theta[c(-m)]  # remove that from the list

#adjusting lambda values due to removal of changepoint
#find lambda values that were adjacent to changepoint m
lambda1 <- lambda[m]
lambda2 <- lambda[m + 1]

#adds endpoints to the vector of theta values
temp_theta = c(0, theta, N)
j = m + 1

#crate new value as a
#geometric weighted sum of the previous two lambda values
lambda0 <-
  lambda1 ^ ((temp_theta[j] - temp_theta[j - 1]) /
               (temp_theta[j + 1] - temp_theta[j - 1])) *
  lambda2 ^ ((temp_theta[j + 1] - temp_theta[j]) /
               (temp_theta[j + 1] - temp_theta[j - 1]))

#create new lambda proposal vector
lambda_star <- lambda
lambda_star[m] <-
  lambda0 #insert the lambda value into the correct location
lambda_star <- lambda_star[-c(m + 1)] #delete the extra value

#repackage values into input_star
input_star <- input
input_star$lambda <- lambda_star
input_star$theta <- theta_star
input_star$K <- K - 1

#compute log likelihood
LL_star <- compute_log_like(input_star)
#compule log prior
prior_star <-  compute_log_prior(input_star)
post_star <- LL_star + prior_star #new log posterior

#hastings ratio
hastings <- log(K) - log(N - (K - 1))

#value preserving jacobian for RJMCMC
log_jacobian <- -2 * log(lambda1 + lambda2) + log(lambda0)
#final log acceptance
alpha <- post_star  - LL + log_jacobian + hastings

if (alpha > log(runif(1))) {
  input_star$LL <- post_star
```

```
    return(input_star)
  }  else {
    return(input)
  }
}


#' Proposes adding a changepoint. Uses a Reversible Jump MCMC (RJMCMC)
#' as seen  in Green 1995
#' @param input a list containing the current parameter state
#' @return a list of either the newly proposed state (input_star)
#' or the current  state (input)


birth_theta <- function(input) {
  K = input$K
  theta = input$theta
  lambda = input$lambda

  LL = input$LL

  #if K = N, the max number of time
  #steps then automatically reject this propos
  if (K == N) {
    return(input)
  }


  #theta_m <- sample(setdiff(1:N, theta), 1)  # birth

  #randomly sample from all possible time steps
  theta_m <- ceiling(runif(1, 0, N))
  #resample if the time step is already in the changepoint list
  while (theta_m %in% theta) {
    theta_m <- ceiling(runif(1, 0, N))
  }

  #create a theta vector with the newly proposed changepoint
  theta_star <- sort(c(theta, theta_m))
  #determine the location of the changepoint in the vector
  m <- which(theta_star == theta_m)

  #add endpoints
  temp_theta_star = c(0, theta_star, N)

  #---update lambda vectors; RJMCMC
  u <- runif(1)
```

```r
j = m + 1 #indexing help
#take original lambda value and split is as a geoemtric average
lambda0 <- lambda[m]
lambda1 <-
  lambda0 * (u / (1 - u)) ^
  ((temp_theta_star[j] - temp_theta_star[j - 1]) /

  (temp_theta_star[j + 1] - temp_theta_star[j - 1]))

lambda2 <-
  lambda0 * ((1 - u) / u) ^
  ((temp_theta_star[j + 1] - temp_theta_star[j]) /
  (temp_theta_star[j + 1] - temp_theta_star[j - 1]))

#edge case when there is no changepoint
if (K == 0) {
  lambda0 <- lambda[1]
  lambda1 <-
    lambda0 * (u / (1 - u)) ^
    ((temp_theta_star[j] - temp_theta_star[j - 1]) /
   (temp_theta_star[j + 1] - temp_theta_star[j - 1]))

  lambda2 <-
    lambda0 * ((1 - u) / u) ^
    ((temp_theta_star[j + 1] - temp_theta_star[j]) /
    (temp_theta_star[j + 1] - temp_theta_star[j - 1]))
}

# updating the lambda vector
#according to the location of the changepoint
if (m == 1) {
  lambda_star <- c(lambda1, lambda2, lambda[-1])
}
else if (m == length(theta_star)) {
  lambda_star <- c(lambda[1:K], lambda1, lambda2)
}
else {
  lambda_star <-
    c(lambda[1:(m - 1)], lambda1, lambda2, lambda[(m + 1):(K + 1)])
}

#repackage the new proposed state
input_star <- input
input_star$lambda <- lambda_star
input_star$theta <- theta_star
input_star$K <- K + 1
```

```r
  LL_star <- compute_log_like(input_star)
  prior_star <-  compute_log_prior(input_star)
  post_star <- LL_star + prior_star #new log posterior


  #hastings ratio
  hastings <- -log(K + 1) + log(N - K)
  log_jacobian <- 2 * log(lambda1 + lambda2) - log(lambda0)

  alpha <- post_star  - LL + log_jacobian + hastings

  if (alpha > log(runif(1))) {
    input_star$LL <- post_star
    return(input_star)
  }
  else {
    return(input)
  }
}


#' Moves a changepoint. Moves it anywhere between the previous
#' changepoint and the next changepoint
#'
#' @param input a list containing the current parameter state
#' @return a list of either the newly proposed state
#' (input_star) or the current state (input)

change_theta <- function(input) {
  K = input$K
  theta = input$theta
  LL = input$LL

  #edge case; K = 0, nothing to move; K = N no where to move
  if (K == 0 || K == N) {
    return(input)
  }

  #location of the selected changepoint in the temp_theta vector
  j <- sample(1:K, size = 1) + 1
  #j <- ceiling(runif(1,1,K+1))

  temp_theta <- c(0, theta, N + 1)

  #if there's no space to move the changepoint
  if (temp_theta[j + 1] - temp_theta[j - 1] < 2) {
    return(input)
  }

  #newly proposed location
  s_star <-
    ceiling(runif(1, temp_theta[j - 1], temp_theta[j + 1] - 1))
  #s_star <- temp_theta[j] + sample(c(-1,1),1)

  #proposed theta vector
  theta_star <- theta
  theta_star[j - 1] <- s_star

  input_star <- input
  input_star$theta <- theta_star

  LL_star <-
    compute_log_like(input_star)
  prior_star <-  compute_log_prior(input_star)
  post_star <- LL_star + prior_star

  U <- runif(1)

  if (log(U) < post_star - LL) {
    input_star$LL <- post_star
    return(input_star)
  } else {
    return(input)
  }
}


#' Proposes a new lambda parameter vector from a random normal
#'  Metropolis Random Walk (MRW)
#' @param input a list containing the current parameter state
#' @return a list of either the newly proposed state (input_star)
#' or the current (input)

change_lambda <- function(input, componentwise = FALSE) {
  K = input$K
  theta = input$theta
  lambda = input$lambda
  gamma = input$gamma
  INDEP = input$INDEP
  LL = input$LL
  graph = input$graph
  adj = input$adj

  # #component wise
  # j <- sample(1:(K + 1), size = 1)
  # lambda_star[j] <- lambda_star[j] + rnorm(1,0,0.01)

  lambda_star <- lambda
  lambda_star <- lambda_star + rnorm(K + 1, 0, 0.0025)

  input_star <- input
  input_star$lambda <- lambda_star

  LL_star <- compute_log_like(input_star)
  prior_star <-   compute_log_prior(input_star)
  post_star <- LL_star + prior_star #new log posterior

  #edge case
  if (post_star == -Inf && LL == -Inf) {
    return(input)
  }

  U = runif(1)

  if (log(U) < post_star - LL) {
    input_star$LL <- post_star
    return(input_star)
  } else {
    return(input)
  }
}
```

8.5 Graph Helpers

The graph helpers maintain the invariant properties of the adjacency list and an edge list.

```
remove_edge <- function(adj_star, ep) {
  m = which(adj_star[[ep[2]]] == ep[1])
  adj_star[[ep[2]]] = adj_star[[ep[2]]][-m]

  m = which(adj_star[[ep[1]]] == ep[2])
  adj_star[[ep[1]]] = adj_star[[ep[1]]][-m]

  return(adj_star)
}

add_edge <- function(adj_star, ep) {
  adj_star[[ep[2]]] = sort(c(adj_star[[ep[2]]], ep[1]))
  adj_star[[ep[1]]] = sort(c(adj_star[[ep[1]]], ep[2]))

  return(adj_star)
```

```r
}

update_edge_c <- function(elist, v1, v2, update_t) {
  n_v1 <- as.character(v1)
  n_v2 <- as.character(v2)

  decrement_edge_c <- function(elist, n_v1, n_v2) {
    if (elist[[n_v1]] == 1) {
      elist[n_v1] <- NULL
    } else {
      elist[[n_v1]] = elist[[n_v1]] - 1
    }

    if (elist[[n_v2]] == 1) {
      elist[n_v2] <- NULL
    } else {
      elist[[n_v2]] = elist[[n_v2]] - 1
    }
    return(elist)

  }

  increment_edge_c <- function(elist, n_v1, n_v2) {
    if (is.null(elist[[n_v1]])) {
      elist[[n_v1]] <- 1
    } else {
      elist[[n_v1]] = elist[[n_v1]] + 1
    }

    if (is.null(elist[[n_v2]])) {
      elist[[n_v2]] <- 1
    } else {
      elist[[n_v2]] = elist[[n_v2]] + 1
    }
    return(elist)

  }


  if (update_t == "del") {
    elist = decrement_edge_c(elist, n_v1, n_v2)
  } else if (update_t == "add") {
    elist = increment_edge_c(elist, n_v1, n_v2)
  }

  return(elist)
}
```

8.6 Graph Operators

8.6.1 add_edge_op(), del_edge_op()

```
add_edge_op <- function(input) {
  K = input$K
  theta = input$theta
  lambda = input$lambda
  gamma = input$gamma
  INDEP = input$INDEP
  LL = input$LL

  num_edges = input$edge_count

  adj = input$adj
  ord = length(adj)
  if (num_edges == choose(ord, 2)) {
    return(input)
  }

  max = ord * (ord - 1) - 2 * num_edges
  A = sample(1:max, 1)
  pos_edges = ord - 1

  for (i in 1:ord) {
    if (A > (pos_edges - length(adj[[i]]))) {
      A = A - (pos_edges - length(adj[[i]]))
    } else {
      v1 = i
      possible_neighbors = setdiff(1:ord, i)
      to_add = setdiff(possible_neighbors, adj[[i]])
      v2 = to_add[A]
      break
    }
  }

  adj_star <- add_edge(adj, c(v1, v2))

  input_star <- input
  input_star$adj <- adj_star
  input_star$edge_count <- input$edge_count + 1

  #LL_star <- compute_log_like(input_star)
  #prior_star <- compute_log_prior(input_star)
  #post_star <- LL_star + prior_star #new log posterior
  #alpha <- post_star - LL + hastings

  hastings <-
```

```
    log(ord * (ord - 1) - 2 * num_edges) - log(2 * (num_edges + 1))
  rat <- compute_log_like_rat(input_star, adj, v1, v2)
  prior_old <- compute_log_prior(input)
  prior_star <- compute_log_prior(input_star)
  alpha <- rat  + hastings + prior_star - prior_old

  U = runif(1)

  if (log(U) < alpha) {
    input_star$graph = 1
    input_star$LL <- LL + rat + prior_star - prior_old
    return(input_star)

  } else {
    input$graph <- 0
    return(input)
  }
}

del_edge_op <- function(input) {
  K = input$K
  theta = input$theta
  lambda = input$lambda
  gamma = input$gamma
  INDEP = input$INDEP
  LL = input$LL

  num_edges = input$edge_count
  edge_count <- input$edge_count


  if (num_edges == 0) {
    return(input)
  }

  adj = input$adj
  ord = length(adj)

  counts = sapply(adj, length)
  v1s = sample(1:ord, 1, prob = counts)

  v11 <- v1s[1]

  if (length(adj[[v11]]) == 1) {
    v12 = adj[[v11]]
  } else {
    v12 = sample(adj[[v11]], 1, replace = TRUE)
  }


  adj_star <- remove_edge(adj, c(v11, v12))


  input_star <- input
  input_star$adj <- adj_star
  input_star$edge_count <- input$edge_count - 1


  #LL_star <- compute_log_like(input_star)
  #prior_star <- compute_log_prior(input_star)
  #post_star <- LL_star + prior_star #new log posterior
  #alpha <- post_star - LL + hastings

  hastings <-
    log(2) + log(num_edges) - log(ord * (ord - 1) - 2 * (num_edges - 1))
  rat <- compute_log_like_rat(input_star, adj, v11, v12)
  prior_old <- compute_log_prior(input)
  prior_star <- compute_log_prior(input_star)

  alpha <- rat  + hastings + prior_star - prior_old

  U = runif(1)

  if (log(U) < alpha) {
    input_star$graph = 1
    input_star$LL <- LL + rat + prior_star - prior_old
    return(input_star)
  } else {
    input$graph <- 0
    return(input)
  }
}
```

8.6.2 rewire(), degree_preserving(), flip_edge()

```
rewire <- function (input) {
  #randomly pick edge
  #vertex weighted
  #random sample
  #v11:..... v12
  #v21: ..... if not in then move it
  #
  K = input$K
  theta = input$theta
```

```
lambda = input$lambda
gamma = input$gamma
INDEP = input$INDEP
LL = input$LL

ec_star = input$ec
adj = input$adj
ord = length(adj)

counts = sapply(adj, length)

i = 0
repeat {
  i = i + 1
  if (i == 50) {
    input$graph = -1
    return(input)
  }
  v1s = sample(1:ord, 2, prob = counts)
  v11 <- v1s[1]
  v21 <- sample(1:ord, 1)

  if (length(adj[[v11]]) == 1) {
    v12 = adj[[v11]]
  } else {
    v12 = sample(adj[[v11]], 1, replace = TRUE)
  }

  if (v12 %in% adj[[v21]]) {
    next
  }
}

adj_star <- remove_edge(adj, c(v11, v12))
adj_star <- add_edge(adj_star, c(v21, v12))

input_star <- input
input_star$adj <- adj_star


LL_star <- compute_log_like(input_star)
prior_star <- compute_log_prior(input_star)
post_star <- LL_star + prior_star #new log posterior

U = runif(1)

if (log(U) < post_star - LL) {
    input_star$graph = 1
    input_star$LL <- post_star
    return(input_star)

  } else {
    input$graph <- 0
    return(input)
  }

}

degree_preserving <- function(input) {
  K = input$K
  theta = input$theta
  lambda = input$lambda
  gamma = input$gamma
  INDEP = input$INDEP
  LL = input$LL

  ec_star = input$ec
  adj = input$adj
  ord = length(adj)

  counts = sapply(adj, length)

  i = 0
  repeat {
    i = i + 1
    if (i == 20) {
      input$graph <- -1
      return(input)
    }
    v1s = sample(1:ord, 2, prob = counts)
    v11 <- v1s[1]
    v21 <- v1s[2]

    if (length(adj[[v11]]) == 1) {
      v12 = adj[[v11]]
    } else {
      v12 = sample(adj[[v11]], 1, replace = TRUE)
    }

    if (v21 %in% c(v11, v12, adj[[v12]])) {
      next
    }
    if (length(adj[[v21]]) == 1) {
      v22 = adj[[v21]]
    } else {
      v22 = sample(adj[[v21]], 1, replace = TRUE)
    }

    if (v22 %in% c(v11, v12, adj[[v11]])) {
      next
    } else {
      break
    }
  }

  adj_star <- remove_edge(adj, c(v11, v12))
  adj_star <- remove_edge(adj_star, c(v21, v22))

  adj_star <- add_edge(adj_star, c(v11, v22))
  adj_star <- add_edge(adj_star, c(v21, v12))

  input_star <- input
  input_star$adj <- adj_star

  LL_star <- compute_log_like(input_star)
  prior_star <- compute_log_prior(input_star)
  post_star <- LL_star + prior_star #new log posterior

  U = runif(1)

  if (log(U) < post_star - LL) {
    input_star$graph = 1
    input_star$LL <- post_star
    return(input_star)

  } else {
    input$graph <- 0
    return(input)
  }
}



#' Randomly selects a possible edge and flips its current state
#'
#' @param input
#'
#' @return
#' @export
```

```r
#'
#' @examples

flip_edge <- function(input, mig = FALSE) {
  K = input$K
  theta = input$theta
  lambda = input$lambda
  gamma = input$gamma
  INDEP = input$INDEP
  LL = input$LL
  migration = input$migration

  adj = input$adj
  ord = length(adj)

  adj_star <- adj

  if (runif(1) < 1 / 2) {
    stepl = 1
  } else {
    stepl = 3
  }

  for (i in 1:stepl) {
    ep = sample(1:ord, 2, replace = FALSE)

    if (ep[1] %in% adj_star[[ep[2]]]) {
      adj_star <- remove_edge(adj_star, ep)

      if (mig) {
        migration_star <- migration + runif(1, 0, 0.05)
      }

      ec_star = update_edge_c(ec_star, ep[1], ep[2], "del")
      ec_star$count = ec_star$count - 1

    } else {
      adj_star <- add_edge(adj_star, ep)

      if (mig) {
        migration_star <- migration - runif(1, 0.000, 0.05)
      }

      ec_star = update_edge_c(ec_star, ep[1], ep[2], "add")
      ec_star$count = ec_star$count + 1
    }
  }
  input_star <- input
  input_star$adj <- adj_star
  input_star$ec <- ec_star
  if (mig) {
    input_star$migration <- migration_star
  }

  LL_star <- compute_log_like(input_star)
  prior_star <- compute_log_prior(input_star)
  post_star <- LL_star + prior_star #new log posterior

  U = runif(1)

  if (log(U) < post_star - LL) {
    input_star$graph = 1
    input_star$LL <- post_star
    return(input_star)

  } else {
    input$graph <- 0
    return(input)
  }
}
```

8.7 Samplers

Different samplers which can be called in different combinations

8.7.1 Graph Samplers

```
update_param_mig <- function(input) {
  U = runif(1)
  if (U < 1) {
    output <- flip_edge(input)
  } else {
    output <- change_migration(input)
  }
  return(output)
}

update_del_add <- function(input) {
  U <- runif(1)
  if (U < 1 / 2) {
    output <- add_edge_op(input)
  } else {
    output <- del_edge_op(input)
  }
  return(output)
```

```r
}
update_add <- function(input) {
  output <- add_edge_op(input)
  return(output)
}

update_del_add_mig <- function(input) {
  U <- runif(1)
  if (U < 98 / 200) {
    output <- add_edge_op(input)
  } else if (U < 196 / 200) {
    output <- del_edge_op(input)
  } else {
    output <- change_migration(input)
  }
  return(output)

}
```

8.7.2 Two-Component Only Samplers

```r
update_param4 <- function(input) {
  cutoff <- 1 / 6

  p = runif(1)
  if (p < cutoff) {
    output <- birth_theta(input)
  } else if (p < 2 * cutoff) {
    output <- change_theta(input)
  } else if (p < 3 * cutoff) {
    output <- change_lambda(input)
  }  else if (p < 4 * cutoff) {
    output <- change_gamma(input)
  } else if (p < 5 * cutoff) {
    output <- degree_preserving(input)
  }
  else  {
    output <- death_theta(input)
  }
  return(output)

}

update_param3 <- function(input) {
  U = runif(1)
  if (U < 1 / 4) {
    output <- flip_edge(input, mig = FALSE)
  } else if (U < 2 / 4) {
    output <- degree_preserving(input)

  } else if (U < 1) {
    output <- rewire(input)
  } else {
    #output <- change_migration(input)
  }
  return(output)
}


update_param_nr <- function(input) {
  U <- runif(1)
  if (U < 1 / 2) {
    output <- add_edge_op(input)
  } else {
    output <- del_edge_op(input)
  }

  for (i in 1:25) {
    U <- runif(1)
    if (U < 1 / 2) {
      output <- add_edge_op(output)
    } else {
      output <- del_edge_op(output)
    }
  }
  output <- change_migration(output)
  return(output)
}

update_param_nr2 <- function(input) {
  output <- flip_edge(input, mig = FALSE)
  for (i in 1:20) {
    output <- rewire(output)
    output <- degree_preserving(output)
  }
  output <- change_migration(output)
}
```

8.8 Initialize Markov chain

```
init_list <- vector(mode = "list", 15)
edge_c_list = list()
edge_c_list$count = 0
```

```r
output <-
list(
K = 2,
theta = c(39, 49),
lambda = lambda,
LL = -Inf,
gamma = gamma,
INDEP = INDEP,
adj = init_list,
graph = 1,
edge_count = 0,
counts = counts,
migration = 0.20
)

output_LL <- compute_log_like(output) + compute_log_prior(output)

output <-
list(
K = 2,
theta = c(39, 49),
lambda = lambda,
LL = output_LL,
gamma = gamma,
INDEP = INDEP,
adj = init_list,
graph = 1,
edge_count = 0,
counts = counts,
migration = 0.20
)

actual <- list(
K = K,
theta = theta,
lambda = lambda,
LL = 0,
gamma = gamma,
INDEP = INDEP,
graph = g_city,
adj = adjlist,
counts = counts,
migration = migration,
edge_count = sum(sapply(actual$adj, length)) / 2
)
```

```r
actual <- list(
K = K,
theta = theta,
lambda = lambda,
LL = compute_log_like(actual) + compute_log_prior(actual),
gamma = gamma,
INDEP = INDEP,
graph = g_city,
adj = adjlist,
counts = counts,
migration = migration,
edge_count = sum(sapply(actual$adj, length)) / 2
)
```

8.9 Sampler Caller

This function calls different samplers and returns the result vector.

```r
run_samples <- function(output, thin, save) {
  thin <- thin
  nSim <- save

  #pre-allocate
  res <- list(
    K = numeric(nSim),
    theta = matrix(0, nSim, N),
    lambda = matrix(0, nSim, N + 1),
    gamma = matrix(0, nSim, 3),
    adj = vector(mode = "list", N),
    accept = numeric(nSim),
    edge_count = numeric(nSim),
    migration = numeric(nSim)
  )

  # sampler block

  for (i in 1:nSim) {
    if (i < 2) {
      output <-  update_add(output)
    }
    for (j in 1:thin) {
      output <-  update_param_nr(output)
    }

    #data store

    # k <- output$K
    # res$K[i] <- k

    # if (length(output$theta) == 0) {
    #   res$theta[i,] <-  0
    # } else {
    #   res$theta[i,] <-   c(output$theta, rep(output$theta[k], N - k))
    # }
    #
    # res$lambda[i,] <-
    #   c(output$lambda, rep(output$lambda[k + 1], N - k))
    # res$gamma[i,] <- output$gamma
    # res$accept[i] <- output$graph

    res$count[i] <- output$edge_count
    res$migration[i] <- output$migration


    # if (i %% 100 == 1) {
    #   plot(graph.adjlist(output$adj, mode = "all"))
    #
    #   print(output$LL)
    #   print(output$migration)
    #   # debug(add_edge_op)
    #   # debug(del_edge_op)
    # }
  }
  return(res)
}
```

8.10 Parallelization Code

```
library(doParallel)
cl <- makeCluster(11, outfile = "")
registerDoParallel(cl)

my_dat_list <-
  list(actual, output, output, output, output, output, output, output)

ptime <- foreach(output = my_dat_list) %dopar% {
  run_samples(put, thin = 200, save = 5000)
}


stopCluster(cl)
```

Citations

Albert, R., and Barabasi, A.-L. (2002), “Statistical mechanics of complex networks,” *Reviews of Modern Physics*, 74, 47–97. https://doi.org/10.1103/RevModPhys.74.47.

Albert Pfeilsticker (1863), *Beiträge zur Pathologie der Masern mit besonderer Berücksichtigung der …*, Fues.

Carpenter, B. (2017), “Bayesian Posteriors are Calibrated by Definition,” *Statistical Modeling, Causal Inference, and Social Science.*

Cook, S. R., Gelman, A., and Rubin, D. B. (2006), “Validation of Software for Bayesian Models Using Posterior Quantiles,” *Journal of Computational and Graphical Statistics*, 15, 675–692.

Corporation, M., and Weston, S. (2019), *doParallel: Foreach Parallel Adaptor for the 'parallel' Package.*

Diggle, P., Knorr-Held, L., Rowlingson, B., Su, T., Hawtin, P., and Bryant, T. (2003), “On-line Monitoring of Public Health Surveillance Data,” pp. 233–266. https://doi.org/10.1093/acprof:oso/9780195146493.003.0009.

Eddelbuettel, D., and François, R. (2011), “Rcpp: Seamless R and C++ Integration,” *Journal of Statistical Software*, 40, 1–18. https://doi.org/10.18637/jss.v040.i08.

EHEC O104:H4 Outbreak Germany 2011 (2011), report, Robert Koch-Institut; Robert Koch-Institut, Infektionsepidemiologie. https://doi.org/http://dx.doi.org/10.25646/88.

Gabry, J., and Mahr, T. (2019), *Bayesplot: Plotting for Bayesian Models.*

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2013), *Bayesian Data Analysis, Third Edition*, CRC Press.

Green, P. J. (1995), “Reversible jump Markov chain Monte Carlo computation and Bayesian model determination.” https://doi.org/10.1093/biomet/82.4.711.

Grimmett, G., and Stirzaker, D. (2004), *Probability and random processes*, Oxford Oxford University Press.

Groendyke, C., Welch, D., and Hunter, D. R. (2011), “Bayesian Inference for Contact Networks Given Epidemic Data,” *Scandinavian Journal of Statistics*, 38, 600–616. https://doi.org/10.1111/j.1467-9469.2010.00721.x.

Groendyke, C., Welch, D., and Hunter, D. R. (2012), “A Network-based Analysis of the 1861 Hagelloch Measles Data,” *Biometrics*, 68, 755–765. https://doi.org/10.1111/j.1541-0420.2012.01748.x.

Held, L., Hofmann, M., Höhle, M., and Schmid, V. (2006), “A two-component model for counts of infectious diseases,” *Biostatistics*, 7, 422–437. https://doi.org/10.1093/biostatistics/kxj016.

Horner, J. (2015), “Hash Table Performance in R: Part I,” *R-bloggers.*

Keeling, M. J., and Rohani, P. (2008), *Modeling Infectious Diseases in Humans and Animals*, Princeton University Press.

Kruschke, J. (2014), *Doing Bayesian Data Analysis: A Tutorial with R, JAGS, and Stan*, Academic Press.

Lee, C., Garbett, A., and Wilkinson, D. J. (2018), "A network epidemic model for online community commissioning data," *Statistics and Computing*, 28, 891–904. https://doi.org/10.1007/s11222-017-9770-6.

Pawitan, Y. (2001), *In All Likelihood: Statistical Modelling and Inference Using Likelihood*, OUP Oxford.

Plummer, M., Best, N., Cowles, K., and Vines, K. (2006), "CODA: Convergence Diagnosis and Output Analysis for MCMC," *R News*, 6, 7–11.

R Core Team (2019), *R: A Language and Environment for Statistical Computing*, Vienna, Austria: R Foundation for Statistical Computing.

Turner, B. M., Sederberg, P. B., Brown, S. D., and Steyvers, M. (2013), "A Method for Efficiently Sampling From Distributions With Correlated Dimensions," *Psychological methods*, 18, 368–384. https://doi.org/10.1037/a0032222.

Wasserman, S., Faust, K., and Urbana-Champaign), S. (. of I. W. (1994), *Social Network Analysis: Methods and Applications*, Cambridge University Press.

Weiser, A. A., Gross, S., Schielke, A., Wigger, J.-F., Ernert, A., Adolphs, J., Fetsch, A., Müller-Graf, C., Käsbohrer, A., Mosbach-Schulz, O., Appel, B., and Greiner, M. (2013), "Trace-Back and Trace-Forward Tools Developed Ad Hoc and Used During the STEC O104:H4 Outbreak 2011 in Germany and Generic Concepts for Future Outbreak Situations," *Foodborne Pathogens and Disease*, 10, 263–269. https://doi.org/10.1089/fpd.2012.1296.

White, L. A., Forester, J. D., and Craft, M. E. (2017), "Using contact networks to explore mechanisms of parasite transmission in wildlife," *Biological Reviews*, 92, 389–409. https://doi.org/10.1111/brv.12236.