Infectious Disease Transmission Network Modelling with Julia

A Preprint

Justin Angevaare
Department of Mathematics and Statistics
University of Guelph
Guelph, Ontario, Canada
jangevaa@uoguelph.ca

Zeny Feng
Department of Mathematics and Statistics
University of Guelph
Guelph, Ontario, Canada
zfeng@uoguelph.ca
https://zfeng.uoguelph.ca

Rob Deardon
Faculty of Veterinary Medicine
Department of Mathematics and Statistics
University of Calgary
Calgary, Alberta, Canada
robert.deardon@ucalgary.ca
https://people.ucalgary.ca/~robert.deardon

February 17, 2020

Abstract

Julia is a modern programming language that increases accessibility of high performance computing. We leverage Julia's features in the creation of a high performance package for computationally intensive epidemic models. Specifically, we introduce **Pathogen.jl** for simulation and inference of transmission network individual level models (TN-ILMs), which are an extension of the individual level model framework of Deardon *et al.* (2010). TN-ILMs can be used to jointly infer transmission networks, event times, and model parameters within a Bayesian framework via MCMC. We detail our specific strategies for conducting MCMC for TN-ILMs, our implementation of these strategies in **Pathogen.jl**, and finally provide an example using **Pathogen.jl** to simulate an epidemic following a susceptible-infectious-removed (SIR) TN-ILM, then performing inference using observations that were generated from that epidemic.

***K*eywords** Epidemic · individual level models and transmission networks · Bayesian · Julia

1 Introduction

In the last 10 to 15 years, the computing needs of research and industry have rapidly expanded. These growing needs can be largely attributed to the mainstream collection, processing, and analysis of big data. Julia is a high level, high performance language, that has been motivated by the needs of modern scientific computing (Bezanson *et al.*, 2018). Until Julia, use of lower level compiled languages, such as Fortran or

C++, has been typically required to implement high performance features of packages in high level scientific computing languages, such as R (Bezanson *et al.*, 2018).

In 2018, 6 years after Julia's first public release in 2012, the language reached its 1.0 version release milestone. This release milestone signalled maturity in Julia's syntax and behaviour, and since, Julia has experienced accelerated adoption in teaching, academic research, and industry. One growth indicator has been the expansion of Julia's package ecosystem, and development of high profile packages such as **MLJ.jl** for machine learning (Blaom *et al.*, 2020); **DifferentialEquations.jl** for systems described by differential equations (Rackauckas and Nie, 2017b); and **Flux.jl** for Neural Networks (Innes, 2018), which all leverage language features to provide high performance.

Epidemic modelling is one among many research areas that can benefit from increased accessibility to high performance numerical computing, through using Julia. Individual level models (ILMs) are epidemic models that incorporate individual specific risk factor information to describe infectious disease dynamics (Deardon *et al.*, 2010). In accounting for population heterogeneity with ILMs, more realistic disease dynamics can be captured, and with that, control strategies can be evaluated in greater detail. As both the number of individuals, and the amount of potential risk factor data for each of these individuals increases, the computational requirements for simulation and inference of ILMs increases steeply. With epidemic modelling in support of decision making in ongoing epidemics, time, whether for development or computation, is at premium. Time constraints aside, the epidemiologists that utilize these models may not have the training to support development of a high performance implementation of a specialized ILM in a lower level language.

While there has been no packages specific for epidemic modelling published on the Julia General Registry, there are Julia packages that could be used for this purpose. **DifferentialEquations.jl** is a comprehensive, general purpose package for differential equations (Rackauckas and Nie, 2017b), and is well suited for population-level epidemic modelling through its discrete stochastic differential equation functionality Rackauckas and Nie (2017a). Use of these kinds of models can be complementary to the use of ILMs (Webb, 2017). **DifferentialEquations.jl** can simulate from population-level epidemic models, and provides several methods of Bayesian parameter estimation through **DiffEqBayes.jl**. The **BioSimulator.jl** package provides simulation methods for interacting populations (Landeros *et al.*, 2018), and could also be used for population level epidemic models. While there are multiple simulation algorithms and full flexibility into the interaction network of the simulated entities in **BioSimulator.jl**, there are no inference methods provided.

For modelling heterogeneous populations, **Agents.jl** (Vahdati, 2019), provides a general purpose package for stochastic simulation of agents on grid systems in discrete time. Such experimental Agent Based Models (ABMs) can incorporate agent movement and behaviour into the epidemics they generate. They prove as powerful tools in the consideration of individual decision making, especially in the context of learning and behavioural shifts by individuals during an epidemic, and relating those individual actions to higher level disease dynamics, such as seen in Abdulkareem *et al.* (2020). Such dynamics are not readily modelled by other methods, but ABMs are also limited in their ability to fit with traditional inference procedures. Approximate Bayesian Computation (ABC) is one avenue for ABM parametrization, where prior beliefs on realistic parameter values are refined using measures of similarity between flexibly summarized ABM simulations and experimental or observational data (Ross *et al.*, 2017). Which is to say, ABMs, like ILMs, and stochastic differential equation models all have their place in epidemics research.

Outside of Julia, there are several packages for epidemic modelling on R's CRAN (R Core Team, 2017). Amongst these epidemic modelling packages, there are two for working with ILMs specifically: **EpiILM** (Warriyar K. V. and Deardon, 2018) and **EpiILMCT**(Almutiry *et al.*, 2020).

EpiILMCT is a package for network and spatial continuous time ILMs (CT-ILMS). This package offers a high performance low level implementation in Fortran for performing MCMC for CT-ILMs. To maintain performance, functions of risk factors are limited to the form of coefficient and power-parameter. This package offers data augmentation for epidemics in which event times are assumed to be unknown. **EpiILM**, on the other hand, supports network and spatial ILMs in discrete time, and does not have data augmentation functionality. **EpiILM** similarly has its core functionality programmed in Fortran for performance reasons.

Beyond the R epidemic modelling packages that utilize the ILM framework of Deardon *et al.* (2010), there is also **epinet** which provides functionality to perform inference of transmission networks and the parameters

of exponential-family random graph models. **epinet** does not provide event data augmentation functionality, and observational data must be assumed to be complete and without error (Groendyke and Welch, 2018).

In the following, we describe ILMs (Deardon *et al.*, 2010), and then an extension of ILMs, Transmission Network Individual Level Models (TN-ILMs). We present a high performance package for simulation and inference of TN-ILMs with data augmentation capability, called **Pathogen.jl**. **Pathogen.jl** is written entirely in Julia. It may be downloaded from `https://github.com/jangevaare/Pathogen.jl`, or in Julia version 1.1 or higher with
`pkg> add Pathogen`.

2 Individual level models

In an ILM, each individual is considered to be in one of several disease states at any time, and the rates governing their transition through the disease states are a function of both individual specific risk factors and the disease states of other individuals in the population at that time. As a whole, the disease state transitions, and subsequent disease state transition rate updates constitute a time-heterogeneous Poisson process.

ILMs may be continuous or discrete with respect to time. In continuous time ILMs, we think of the temporal data structure as consisting of variable time periods, and these representing the length of time between events; *i.e.*, inter-event periods. Here, events are considered to be the disease state transitions in the population. A discrete time ILM approximates its continuous time counterpart, and generally uses time periods that are equal in duration. The likelihoods and strategies for inference for these models are different, however the disease state transition rates each uses can be defined in the same way.

There is flexibility in the disease states that are considered, with susceptible and infectious states being the only necessary states. In the following, the structure of a more complex framework, the susceptible-exposed-infectious-removed (SEIR) ILM, will be described.

In an SEIR ILM, the rate that an individual, i, transitions from the susceptible state to the exposed state during the t^{th} time period is given as:

$$\lambda_{SE}(i,t) = \left[\Omega_S(i) \sum_{k \in I_{(t)}} \Omega_T(k)\kappa(i,k)\right] + \epsilon(i,t) \text{ for } i \in S_{(t)}, \tag{1}$$

where,

- $I_{(t)}$ is the set of infectious individuals during the t^{th} time period,
- $S_{(t)}$ is the set of susceptible individuals during the t^{th} time period,
- $\Omega_S(i)$ is a function of risk factors associated with the risk of susceptible individual i contracting the disease (susceptibility),
- $\Omega_T(k)$ is a function of risk factors associated with the risk of infection transmission from the k^{th} individual (transmissibility),
- $\kappa(i,k)$ is an infection kernel, a function of risk factors involving both the i^{th} and k^{th} individuals, which often describes the connectivity between these individuals, and,
- $\epsilon(i,t)$ is a function of risk factors associated with exposure to the i^{th} individual during the t^{th} time period that the model otherwise fails to explain. Typically this refers to exposure from a non-specified source outside of the observered population. This is also referred to as the *sparks function*.

Transition between exposed and infectious states for the j^{th} individual during the t^{th} time period occurs with rate:

$$\lambda_{EI}(j,t) = \Omega_L(j) \text{ for } j \in E_{(t)} \tag{2}$$

where,

- $E_{(t)}$ is the set of exposed individuals during the t^{th} time period, and
- $\Omega_L(j)$ is a function of risk factors associated with the latent period, the length of time between exposure and onset of infectious in the j^{th} individual.

Lastly, the k^{th} individual transitions from the infectious to the removed state during the t^{th} time period with rate:

$$\lambda_{IR}(k,t) = \Omega_R(k) \text{ for } k \in I_{(t)} \tag{3}$$

where,

- $\Omega_R(k)$ is a function of risk factors associated with the removal of the k^{th} individual from an infectious state. Removal can refer to recovery with acquired immunity, death, quarantine, *etc.*

In combination, the rates $\lambda_{SE}(i,t)$, $\lambda_{EI}(j,t)$, and $\lambda_{IR}(k,t)$ describe the spread of disease in an SEIR ILM.

In a continuous time SEIR ILM, no change occurs to the sets $S_{(t)}, E_{(t)}, I_{(t)}$, and $R_{(t)}$ during an inter-event period. It follows that the rates $\lambda_{SE}(i,t)$, $\lambda_{EI}(j,t)$, and $\lambda_{IR}(k,t)$ remain constant during each of these time periods, and that the occurrence of any single event (disease state transition) marks the end of the t^{th} inter-event time period. For the remainder of this paper we focus exclusively on continuous time ILMs.

2.1 Transmission Network ILM extension

In ILMs, a susceptible individual's risk of being infected by an infectious disease is based on the culmination of various risk factors, such that the influence of specific sources of exposure are masked. We introduce Transmission Network ILMs (TN-ILMs), that instead are explicit with respect to exposure sources. In an SEIR TN-ILM, a set of competing transition rates for each susceptible individual, i, to the exposed state are defined as

$$\lambda^*_{SE}(i,k,t) = \Omega_S(i)\Omega_T(k)\kappa(i,k) \text{ for } i \in S_{(t)}, \ k \in I_{(t)} \tag{4}$$

describing transition rates specific to each infectious individual, k,and with,

$$\lambda^*_{SE}(i,t) = \epsilon^*(i,t) \text{ for } i \in S_{(t)} \tag{5}$$

describing the transition rate specific to any exogenous exposure source, during the t^{th} time period. While $\epsilon(i,t)$ in an ILM is not necessarily specific to exogenous sources in ILMs, that assumption is made with $\epsilon^*(i,t)$ in TN-ILMs.

3 Methods

3.1 Continuous time-to-event simulation

Epidemics can be stochastically simulated from a continuous time ILM using the Gillespie (1977) algorithm. Inter-event periods are generated from an exponential distribution, with rate

$$\upsilon(t) = \sum_{i \in S_t} \lambda_{SE}(i,t) + \sum_{j \in E_t} \lambda_{EI}(j,t) + \sum_{k \in I_t} \lambda_{IR}(k,t). \tag{6}$$

The specific event that occurs at this time is generated from a multinomial distribution, with probability vector

$$\pi_1(t) = \left[\frac{\lambda_{SE}(1,t)}{\upsilon(t)}, \ \ldots, \ \frac{\lambda_{SE}(N,t)}{\upsilon(t)}, \ \frac{\lambda_{EI}(1,t)}{\upsilon(t)}, \ \ldots, \ \frac{\lambda_{EI}(N,t)}{\upsilon(t)}, \ \frac{\lambda_{IR}(1,t)}{\upsilon(t)}, \ \ldots, \ \frac{\lambda_{IR}(N,t)}{\upsilon(t)} \right]^\top, \tag{7}$$

for a population of size N. If the generated event is the transition of a susceptible individual to the exposed state, and a TN-ILM is used, the transmission source is generated from a multinomial distribution, with probability vector

$$\pi_2(t) = \left[\frac{\lambda^*_{SE}(i,1,t)}{\lambda_{SE}(i,t)}, \quad \ldots, \quad \frac{\lambda^*_{SE}(i,N,t)}{\lambda_{SE}(i,t)}, \quad \frac{\lambda^*_{SE}(i,t)}{\lambda_{SE}(i,t)} \right]^\top. \tag{8}$$

Following the generation of an inter-event interval, and generation of the specific event occurrence, the population is then updated (*i.e.*, membership in the sets of S_t, E_t, I_t, and R_t), and the process repeats until either no further events are possible (*i.e.*, $\pi_1(t)$ is a vector of zeros), or until some earlier stop condition is met.

3.2 Continuous time ILM likelihood

The likelihood of the continuous time-to-event SEIR ILM is the product of likelihoods at time periods indexed by $t = 2, \ldots T$, where T is the total number of events that have occurred. With the length of time since the beginning of the t^{th} time period denoted as Δ_t, the likelihood function for the associated parameters, $\boldsymbol{\theta}$, is given as:

$$L(\boldsymbol{\theta}) = \prod_{t=2}^{T} \psi(t) \upsilon(t) \exp\left\{-\upsilon(t)\Delta_t\right\}, \tag{9}$$

where,

$$\psi(t) = \begin{cases} \frac{\lambda_{SE}(i,t)}{\upsilon(t)} \text{ if } i \in (S_{(t)} \cap E_{(t+1)}), \\ \frac{\lambda_{EI}(j,t)}{\upsilon(t)} \text{ if } j \in (E_{(t)} \cap I_{(t+1)}), \\ \frac{\lambda_{IR}(k,t)}{\upsilon(t)} \text{ if } k \in (I_{(t)} \cap R_{(t+1)}), \end{cases} \tag{10}$$

and,

$$N = |S_{(t)}| + |E_{(t)}| + |I_{(t)}| + |R_{(t)}| \quad \forall t. \tag{11}$$

Modification is required for TN-ILMs to account for specific transmissions sources, with the TN-ILM the likelihood function is given as:

$$L(\boldsymbol{\theta}) = \prod_{t=2}^{T} \psi^*(t) \upsilon(t) \exp\left\{-\upsilon(t)\Delta_t\right\}, \tag{12}$$

where,

$$\psi^*(t) = \begin{cases} \frac{\lambda^*_{SE}(i,k,t)}{\upsilon(t)} \text{ if } i \in (S_{(t)} \cap E_{(t+1)}) \text{ by endogenous exposure from individual } k, \\ \frac{\lambda^*_{SE}(i,t)}{\upsilon(t)} \text{ if } i \in (S_{(t)} \cap E_{(t+1)}) \text{ by exogenous exposure}, \\ \frac{\lambda_{EI}(j,t)}{\upsilon(t)} \text{ if } j \in (E_{(t)} \cap I_{(t+1)}), \\ \frac{\lambda_{IR}(k,t)}{\upsilon(t)} \text{ if } k \in (I_{(t)} \cap R_{(t+1)}). \end{cases} \tag{13}$$

3.3 Bayesian inference

In the Bayesian framework, beliefs about a parameter are described by a posterior distribution; a probability distribution for model parameter values, θ, conditioned on observational data D. From a posterior distribution, credible intervals and point estimates for parameters can be obtained, and hypothesis testing can be conducted. Monte Carlo methods, such as Markov chain Monte Carlo (MCMC) methods are typically used to generate a sufficient quantity of samples from the posterior distribution, such that the properties of the posterior distribution can be estimated through the generated samples.

3.4 Markov chain Monte Carlo

To perform Bayesian parameter estimation for continuous time ILMs, the Metropolis-Hastings (M-H) MCMC algorithm is used. The M-H algorithm generates a Markov chain consisting of a sequence of samples within the parameter space, the distribution of which converges to the targeted distribution - the posterior distribution of the parameters in this case. To generate such an Markov chain, some initial values, here denoted as $\boldsymbol{\theta}_1$, must be first selected or generated. Following initialization of the Markov chain, new samples are proposed with a transition kernel. For a symmetric transition kernel, a sample proposed for the w^{th} iteration, $\boldsymbol{\theta}_w{}'$, is accepted with probability

$$\alpha(\boldsymbol{\theta}'_w) = P(\boldsymbol{\theta}_w = \boldsymbol{\theta}'_w) = \min\left(1, \frac{L(\boldsymbol{\theta}'_w)P(\boldsymbol{\theta}'_w)}{L(\boldsymbol{\theta}_{w-1})P(\boldsymbol{\theta}_{w-1})}\right). \tag{14}$$

If the proposal $\boldsymbol{\theta}'_{w+1}$ is rejected, $\boldsymbol{\theta}_w$ remains at $\boldsymbol{\theta}_{w-1}$ (Hastings, 1970; Robert and Casella, 2013).

Sampling from the posterior distribution of continuous time ILMs using the M-H algorithm is straightforward when an epidemic is fully observed; *i.e.*, with event times that are known exactly with certainty. While for certain epidemics, complete or nearly complete case identification may occur, exact event times are rarely known. Here, unknown event times are imputed using a *data augmentation* process, in which they are treated as additional parameters to be estimated. This tends to result in a massive increase in the dimensionality of the parameter space, and with that computational challenges arise. With TN-ILMs these challenges are exacerbated by the transmission network, which is latent and requires imputation. A consistency must be enforced in the generation of MCMC proposals between the transmission network and event times. Without this incorporated into the proposal mechanism, the rejection rate using M-H would be unreasonably high. As such, a specialized MCMC algorithm is required for inference of TN-ILMs.

Initialization strategy

Initialization is an important step to facilitate efficient MCMC in these models. High dimensionality and event time interdependence results in vast areas of the parameter space having near-zero posterior density, which is approximated to zero in computation. With MCMC, it may take a long time to move into, and sample from areas of the parameter space that have non-zero computationally approximated posterior mass. This would also impact adaptive tuning of transition kernel variance, and combined represent a bottleneck in achieving MCMC convergence. Our strategy is to generate many potential sets of initial values, and to select the set with the highest posterior density. Multiple Markov chains, and a higher number of initialization generations per Markov chain are recommended for higher dimensional applications of TN-ILMs. These independently initialized Markov chains allow for a more substantiated assessment of Markov chain convergence, in comparison to assessing the behaviour of a single Markov chain in isolation.

The initialization generated for each Markov chain begins with sampling a set of model parameters from their corresponding prior distributions. A set of event times is also generated based upon observation times and prior distributions for observation delays, and latent periods if applicable. Then, the posterior density is calculated for a standard ILM using these generated event times and model parameters. The posterior density calculation is iterative, and abandoned at the point in which it is no longer possible for that initialization to represent the maximum in the initialization process. The transmission source specificity introduced for TN-ILMs is not accounted for in the posterior density calculation, instead a transmission network is generated from its conditional distribution, that is, a transmission network is generated using a Gibbs sampler.

Iteration strategy

Each MCMC iteration can be broken down into several sub-steps: event time sampling, parameter sampling, and transmission network sampling.

Event times A random walk Metropolis-Hastings sampling procedure is used to sample event times from the TN-ILM posterior distribution. Specifically a bounded normal distribution is used as the transition kernel, with its mean set to the event time in the previous iteration, and with a pre-specified variance. These bounds are determined by the times of the events that are dependent through the transmission network or model structure.

The event times relevant to determining the bounds of event time proposal distributions include: the times of other state transitions by the involved individual (*e.g.*, an individual may not transition to a removed state prior to being in an infected state); the infection and, if applicable, removal times of their transmission source (*e.g.*, a susceptible individual may only transition to the exposed state while their transmission source is in an infected state). Similarly, if an individual transmits the disease themselves, they must be in the infected state prior to their earliest transmission, and must not be removed until after their final transmission. This ensures the transmission network remains consistent with the set of event times.

Due to event time interdependence, event time proposals are generated one at a time. The order in which event times are updated is randomized at each iteration. A decision to accept or reject proposed event times can be conducted in batches such that fewer likelihood calculations occur. Tuning of both the batch size and the variance of the bounded normal transition kernel is required for each TN-ILM application.

TN-ILM parameters Following updates to the event times, a new set of TN-ILM parameters is proposed, and the proposed values subject to the acceptance rule for M-H MCMC algorithms shown in Equation 14. For these proposals, a multivariate normal transition kernel is used. The covariance matrix of the transition kernel can be automatically tuned during sampling, using the adaptive sampling method of Roberts and Rosenthal (2007).

Transmission network Finally, a new transmission network is updated using the Gibbs sampler. Each transmission source is sampled from a multinomial distribution corresponding to its conditional distribution. The probability vector of each multinomial distribution is generated following Equation 8, for each applicable time period using the current set of event times and model parameters.

4 Software implementation with Julia

We have implemented simulation and inference methods for TN-ILMs as described in Section 3 in Julia with the **Pathogen.jl** package. **Pathogen.jl** is open source and released with an MIT (Expat) license. Our description of **Pathogen.jl** is consistent with its `v0.4.3` release.

Pathogen.jl leverages several of the features of the Julia Language. First is the ability for the user to define functions for disease state transition rates with a high level of flexibility without sacrificing performance. While Julia is an interactive language, with a familiar Read-Evaluate-Process-Loop (REPL) interface, behind that, Julia code is Just-In-Time (JIT) compiled to highly optimized machine code. In a set of benchmarks performed by Bezanson *et al.* (2017), Julia was usually found to be within a factor of two in regards to computation time of equivalent C code.

Type systems are used by programming languages to structure data. Julia's type system is particularly powerful. New user-defined types are implemented without performance penalty in comparison to the basic types provided by the language. This is made no clearer than by the fact that much of Julia's features, are written in Julia itself. Julia's *multiple dispatch* enables both generalist and specialist function methods that are invoked based on argument type. The type system results in code that can often be common across a set of types. **Pathogen.jl** supports SEIR, SEI, SIR, and SI TN-ILMs, and the types provided by the **Pathogen.jl** package are implemented with regards to these disease model classes. With only some basic methods defined for these types or across unions of these types, higher level code such as that involved in epidemic simulation, likelihood calculations, and performing MCMC is able to be kept common across all disease model classes. This same functionality in theory would reduce the development time required to implement additional disease model classes. The types implemented by **Pathogen.jl**, and their hierarchy are shown in Figure 1.

Distributed computing functionality is highly relevant, if not essential for large data applications. For TN-ILMs, this would be for modelling large populations and/or higher complexity models. Julia has been designed for simple, but powerful distributed computing. Some of this is leveraged in **Pathogen.jl** for conducting MCMC for TN-ILMs. Independent Markov chains are easily initialized, and ran across multiple cores, or on a high performance cluster in parallel.

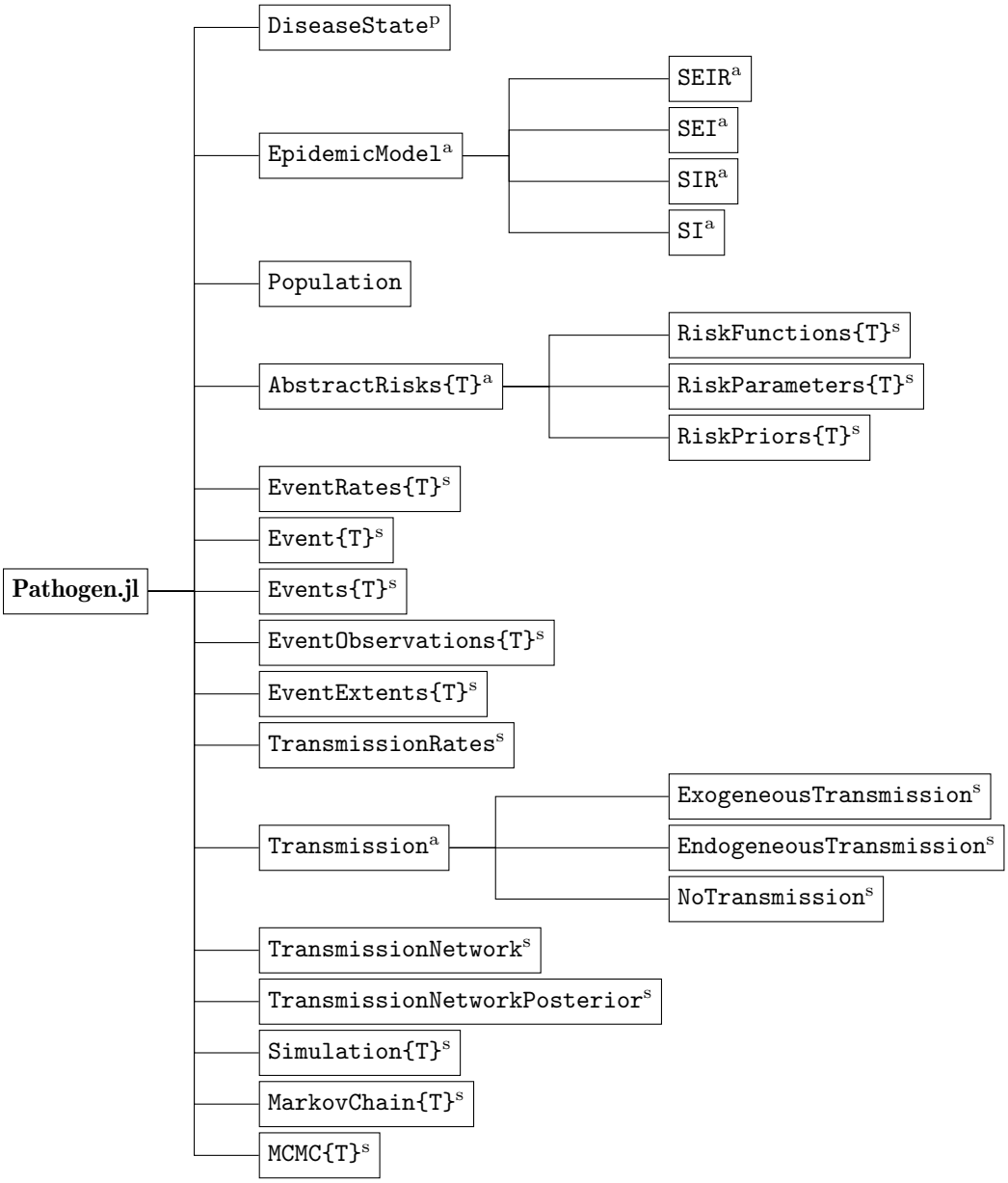

Figure 1: The hierarchy of the types provided by **Pathogen.jl** are shown above. Type names followed by `{T}` indicate that that type is parametric - in all cases the type is parametrized by a subtype of `EpidemicModel`. Subscripts are used to indicate the kind of type: [p]primitive: elementary data representations; [a]abstract: types used for organization purposes or to invoke specific function methods but do not contain data; [s]struct: types that have fields containing other types.

4.1 Installing Pathogen.jl

The latest version of Pathogen is available for Julia version 1.1 and higher. It is listed on Julia's General Registry, and as such the latest stable release can be added to one's library with `Pkg> add Pathogen`, or the development version with `Pkg> add Pathogen#master` or by cloning the repository located at `https://github.com/jangevaare/Pathogen.jl`. This will also install any packages required by **Pathogen.jl** not already installed by the user. Once installed, the package can be immediately used with `using Pathogen`. This entire process takes only a few seconds.

4.2 Pathogen.jl basics

Whether using **Pathogen.jl** for simulation or inference of TN-ILM epidemics, a population must be first defined. We represent populations in our package with a `Population` type. More specifically, our `Population` type is a *struct*. In Julia, a struct is a type that is composed of other types - perhaps other structs, or *primitive* types such as `Float64` or `Bool`.

To construct a `Population`, a `DataFrame` containing individual specific risk factors must be specified. `DataFrame` is a type provided by the **DataFrames.jl** package, which is a dependency of **Pathogen.jl**. In this `DataFrame`, each row will represent an individual. There is also an option to provide a distance matrix when describing a population. Distance measures are common components of infectivity kernels. The distance matrix can be used to avoid repeated calculation of these distances from individual specific risk factor information. `Population` in **Pathogen.jl** is declared as

```
struct Population
  risks::DataFrame
  distances::Union{Nothing, AbstractArray}
  individuals::Int64
end
```

The structure of TN-ILMs is described with `RiskFunctions{T}`, which is a collection of functions that calculate individual specific disease state transition rates. `RiskFunctions{T}` is a *parametric struct*, declared as shown below, with slightly different construction and behaviour for different values of `T`.

```
struct RiskFunctions{T<: EpidemicModel} <: AbstractRisk{T}
  sparks::Union{Nothing, Function}
  susceptibility::Union{Nothing, Function}
  infectivity::Union{Nothing, Function}
  transmissibility::Union{Nothing, Function}
  latency::Union{Nothing, Function}
  removal::Union{Nothing, Function}
end
```

Parametric structs are used throughout **Pathogen.jl** to provide specialization to different disease model classes. This allows for modified functionality where it is needed for the various model class implementations in otherwise common code. For instance, construction of `RiskFunctions{SIR}` when compared to `RiskFunctions{SEIR}` does not involve specification of a function describing the transition rate between exposed and infectious classes (*i.e.*, Ω_L in Equation 2).

There is full flexibility in the form of TN-ILM risk functions used in `RiskFunctions{T}`, as long as these risk functions follow an expected signature for their arguments. Each risk function must accept a `Population`, a parameter vector (`AbstractVector`), and an `Int64` individual identifier, as arguments. Infectivity kernels are an exception to this, and must accept two `Int64` individual identifiers - for infection source and infection recipient. Each risk function should return a `Float64`. A complete example of constructing `RiskFunctions{SIR}`, including the construction of the risk functions that compose it, is provided in Section 5.

The parametrization of TN-ILM risk functions is represented by a separate type, `RiskParameters{T}`. For each of the risk functions required by a TN-ILM, a parameter vector must be provided in `RiskParameters{T}`, which has been declared as:

```
struct RiskParameters{T<: EpidemicModel} <: AbstractRisk{T}
  sparks::Union{Nothing, AbstractVector}
  susceptibility::Union{Nothing, AbstractVector}
  infectivity::Union{Nothing, AbstractVector}
  transmissibility::Union{Nothing, AbstractVector}
  latency::Union{Nothing, AbstractVector}
  removal::Union{Nothing, AbstractVector}
end
```

Both `RiskFunctions{T}` and `RiskParameters{T}` are subtypes of the abstract type `AbstractRisk{T}`, as can be seen in Figure 1. If there are unused fields in any `AbstractRisk{T}` for specific model classes, those are automatically populated with `nothing` during construction.

4.3 Simulation with Pathogen.jl

To simplify the simulation interface, we have provided a `Simulation{T}` struct with **Pathogen.jl**. `Simulation{T}`s include all of the information required to iterate and track the progression of an epidemic. Once constructed, the `simulate!` function is used to run the simulation, updating event times, individual disease states, and the transmission network within the `Simulation{T}` as appropriate. The `simulate!` function will iterate until a specified stop condition is met (processing time, simulation time, and/or number of iterations) or if there are no further events possible. `Simulation{T}` is a *parametric mutable struct*, which is like a parametric struct, except it allows for its values to be changed or updated. For instance, the `Int64` value for `iterations`, shown in the type declaration below, is incremented at each iteration, which is possible because of this mutability. Explicitness about struct mutability in Julia allows for certain optimizations by the compiler when dealing with structs that do not change in composition.

```
mutable struct Simulation{T <: EpidemicModel}
  time::Float64
  iterations::Int64
  population::Population
  risk_functions::RiskFunctions{T}
  risk_parameters::RiskParameters{T}
  disease_states::Vector{DiseaseState}
  transmission_rates::TransmissionRates
  event_rates::EventRates{T}
  events::Events{T}
  transmission_network::TransmissionNetwork
end
```

The use of `{T}` throughout the declaration of `Simulation{T}` ensures matching type parametrizations for all of the types that compose it. This also means that the parametrization of `Simulation{T}` can be inferred during its construction.

There are several ways to construct a `Simulation{T}`. The simplest construction method requires only specification of a `Population`, `RiskFunctions{T}`, and `RiskParameters{T}` for the TN-ILM. In this case, an entirely susceptible population is assumed for a starting time of 0.0 time units. The internal code of this basic `Simulation{T}` construction method is shown below. Individuals can start the simulation from other specified disease states and/or the simulation may have a different start times using the other construction methods (not shown).

```
  function Simulation(pop::Population,
                      rf::RiskFunctions{T},
                      rp::RiskParameters{T}) where T <: EpidemicModel
    states = fill(State_S, pop.individuals)
    tr = initialize(TransmissionRates, states, pop, rf, rp)
    rates = initialize(EventRates, tr, states, pop, rf, rp)
    events = Events{T}(pop.individuals)
    net = TransmissionNetwork(pop.individuals)
    return new{T}(0.0, 0, pop, rf, rp, states, tr, rates, events, net)
  end
```

Finally, observational data can be generated from a completed `Simulation` using the provided `observe()` function, with statistical distributions specified for observation delays, in the form of `UnivariateDistribution`s from the **Distributions.jl** package (Besançon *et al.*, 2019), another dependency of **Pathogen.jl**.

4.4 Inference with Pathogen.jl

As with simulation in **Pathogen.jl**, performing parameter estimation for TN-ILMs via MCMC requires the definition of a `Population` and of `RiskFunctions{T}`. `RiskParameters` are now unknown, and will be sampled from the posterior distribution through MCMC. In order to do this, specification of prior distributions for each risk parameter is first required. These priors are structured through the `RiskPriors{T}` type. This parametric struct has the same form as `RiskParameters`, but in place of each parameter value, a `UnivariateDistribution` must be provided.

We also must specify priors for event time data augmentation process. Currently Uniform prior distributions are supported and applied broadly through an `EventExtents{T}` struct. These are upper bounds on the length of time between observations of infectiousness, and the actual onset of infection, as well as the length of time between removal observations and actual removal times. For TN-ILMs that have an exposed class, a bound on the length of the latent period is also specified in use of `EventExtents{T}`.

```
struct EventExtents{T <: EpidemicModel}
  exposure::Union{Nothing, Real}
  infection::Union{Nothing, Real}
  removal::Union{Nothing, Real}
end
```

Finally, an `MCMC{T}` struct is constructed, containing all of the information required for performing MCMC, common across individual Markov chains (*e.g.*, observational data, population data, prior distributions), as well as the individual Markov chains that sample from the TN-ILM posterior distribution. The composition of `MCMC{T}` is:

```
mutable struct MCMC{T <: EpidemicModel}
  event_observations::EventObservations{T}
  event_extents::EventExtents{T}
  population::Population
  starting_states::Vector{DiseaseState}
  risk_functions::RiskFunctions{T}
  risk_priors::RiskPriors{T}
  markov_chains::Vector{MarkovChain{T}}
end
```

`MarkovChain{T}` includes vectors of event times from data augmentation, networks, as well as TN-ILM parameters, and is declared as:

```
mutable struct MarkovChain{T <: EpidemicModel}
  iterations::Int64
  events::Vector{Events{T}}
  transmission_network::Vector{TransmissionNetwork}
  risk_parameters::Vector{RiskParameters{T}}
  log_posterior::Vector{Float64}
  cov::OnlineStats.CovMatrix
end
```

To construct an `MCMC{T}` object to perform inference, `EventObservations{T}`, `EventExtents{T}`, `Population`, `RiskFunctions{T}`, and `RiskPriors{T}` must all be specified. MCMC can then be initialized, using the `start!` function. During initialization the user must specify the number of chains to initialize, and the number of initialization attempts per chain. Each chain can be initialized on different cores. After initialization, MCMC can proceed using the `iterate!` function. With this function the number iterations are specified, as well as transition kernel variance for event time data augmentation. MCMC for

each `MarkovChain{T}` can also be ran in parallel. An example of the process of performing MCMC with **Pathogen.jl** is provided in Section 5.

5 Example

In the following we present a full example using **Pathogen.jl** to:

- Generate an epidemic population,
- Simulate from a TN-ILM,
- Simulate observations from the epidemic, and,
- Use the observations to estimate event times, transmission network, and TN-ILM parameters via MCMC.

The source code to replicate this example exactly is included in `SIR.jl`.

We start by loading the various publicly available Julia packages used in the example:

```
using Distances,
LinearAlgebra,
DataFrames,
Distributions,
Pathogen
```

We generate risk factor data for a population containing 100 individuals. A location (x and y coordinates over a 15×30 unit area), and an arbitrary Gamma($\alpha = 1, \beta = 1$) distributed risk factor:

```
n = 100
risks = DataFrame(x = rand(Uniform(0, 15), n),
                  y = rand(Uniform(0, 30), n),
                  riskfactor1 = rand(Gamma(), n))
```

We pre-calculate Euclidean distances between individuals in a distance matrix, which is used in the specification of a `Population` object:

```
dists = [euclidean([risks[i, :x];
                    risks[i, :y]],
                   [risks[j, :x];
                    risks[j, :y]]) for i = 1:n, j = 1:n]
pop = Population(risks, dists)
```

Next, several functions of risk factors are defined with the signature expected by **Pathogen.jl**, and these structured into a `RiskFunctions{SIR}` object:

```
function _constant(params::Vector{Float64}, pop::Population, i::Int64)
  return params[1]
end

function _one(params::Vector{Float64}, pop::Population, i::Int64)
  return 1.0
end

function _linear(params::Vector{Float64}, pop::Population, i::Int64)
  return params[1] * pop.risks[i, :riskfactor1]
end

function _powerlaw(params::Vector{Float64}, pop::Population,
                   i::Int64, k::Int64)
  beta = params[1]
  d = pop.distances[k, i]
  return d^(-beta)
end

rf = RiskFunctions{SIR}(_constant, # sparks function
                        _one, # susceptibility function
                        _powerlaw, # infectivity kernel
                        _one, # transmissibility function
                        _linear) # removal function
```

These risk functions are then parametrized:

```
rparams = RiskParameters{SIR}([0.0001], # sparks
                              Float64[], # susceptibility
                              [4.0], # infectivity
                              Float64[], # transmissibility
                              [0.1]) # removal
```

We specify starting states for each of the 100 individuals. We set individual 1 as infectious at the start of the epidemic, and the remaining 99 individuals as susceptible. We then organize our population, the risk functions and their parameterizations, and these starting states into a `Simulation` struct. We then run the simulation until a simulation length ≥ 200 time units is reached.

```
starting_states = append!([State_I], fill(State_S, n-1))
sim = Simulation(pop, starting_states, rf, rparams)
simulate!(sim, tmax=200.0)
```

The simulated epidemic can be explored visually. For our example, plots are generated in Julia using **Plots.jl** (Breloff, 2015) with the **GR** plotting backend (Heinen *et al.*, 1985–2019), with *plot recipes* provided in **Pathogen.jl**. An epidemic curve can be generated from an `Events{T}` object, such as the one within our `Simulation{T}`, with

```
p1 = plot(sim.events)
```

and transmission network plots showing the disease states of individuals at specified times, with

```
p2=plot(sim.transmission_network,
        sim.population,
        sim.events,
        0.0, title="Time = 0")

...
p6=plot(sim.transmission_network,
        sim.population,
        sim.events,
        100.0, title="Time = 200")
```

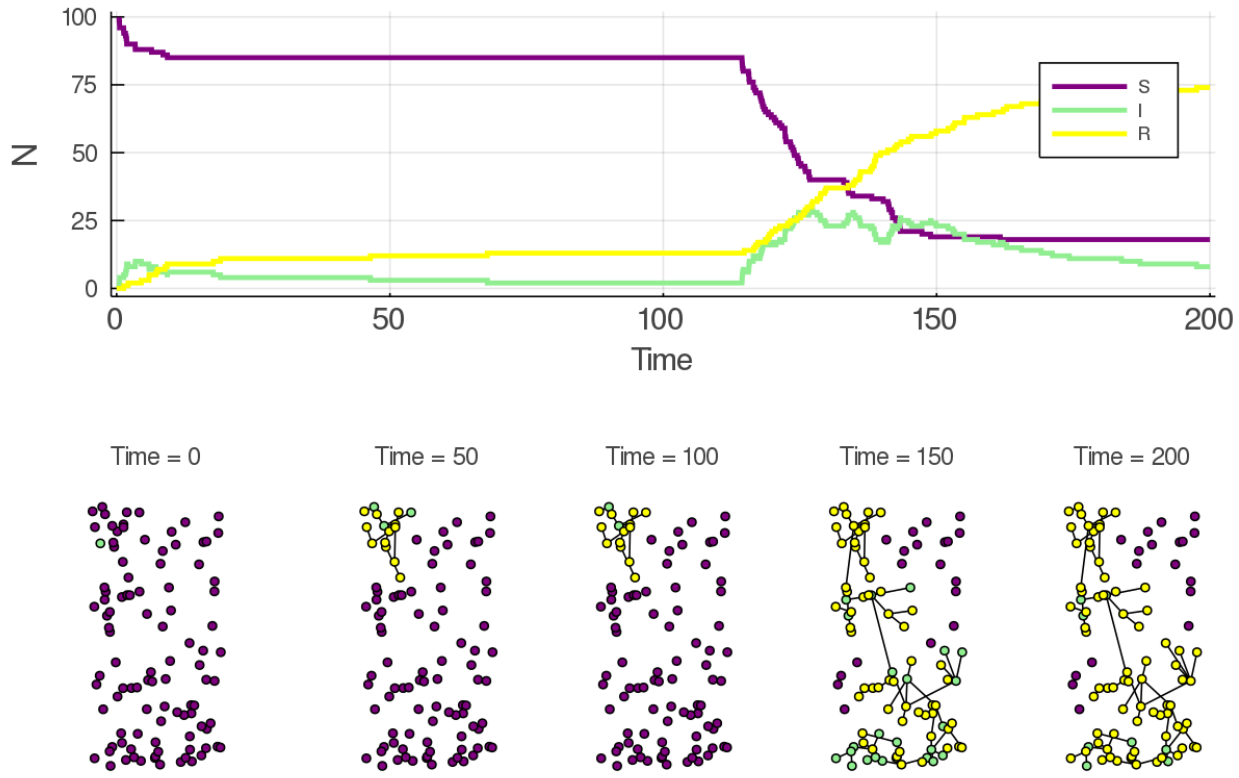

Figure 2: A epidemic simulated from a TN-ILM following code provided in Section 5. The top plot shows the overall number of individuals in each of the represented disease states over the length of the epidemic. The bottom row of plots shows individual disease state information along with the state of the transmission network at 5 time points in the epidemic.

These plots can then be combined in a layout, to obtain Figure 2 with

```
l = @layout [a;
             b c d e f]

plot(p1, p2, p3, p4, p5, p6, layout=l)
```

From the simulated epidemic, observations can be generated, with statistical distributions used for the generation of observation delay. The optional `force=true` keyword argument is used to bound the infection observation delay such that an infection observation is guaranteed (*i.e.*, an individual can't move to the removed state undetected).

```
obs = observe(sim, Uniform(0.5, 2.5), Uniform(0.5, 2.5), force=true)
```

With observational data, inference for a specified model can be conducted. For our example, we will assume the model structure is known, and reuse the set of risk functions we declared for the epidemic simulation. For each parameter value, a prior distribution must be specified before commencing MCMC. We must also specify priors for our event times through `EventExtents`.

```
rpriors = RiskPriors{SIR}([Exponential(0.0001)],
                          UnivariateDistribution[],
                          [Uniform(1.0, 7.0)],
                          UnivariateDistribution[],
                          [Uniform(0.0, 1.0)])
```

```
ee = EventExtents{SIR}(5.0, 5.0)
```

MCMC will now be initialized, following the initialization strategy detailed in Section 3.4, with:

```
mcmc = MCMC(obs, ee, pop, rf, rpriors)
start!(mcmc, attempts=50000)
```

We then perform 50k iterations (second positional argument) for the initialized Markov chain using the `iterate!` function. Here, we elected to batch event time data augmentation into 5 sets (specified with `event_batches` keyword argument), with a transition kernel variance of 1.0 for each event time (third positional argument). We also condition event time augmentation on the previous transmission network (specified with `condition_on_network` keyword argument). 50k iterations in this manner required approximately 30 minutes on a computer with an Intel 2.7 GHz i7-3740QM processor. For applications to real world data, or for simulations with real world application, longer runs by multiple chains is advised in order to validate convergence.

```
iterate!(mcmc, 50000, 1.0, condition_on_network=true, event_batches=5)
```

Once MCMC is completed, parameter and event time traces can be converted into `Array`s, and summarized using conventional array and statistics functionality in Julia. We provide convenient plotting functions for visualizing MCMC and posterior distributions, yielding the plots seen in Figures 3 and 4. The trace plot in Figure 3 is generated with:

```
p1 = plot(1:20:50001, mcmc.markov_chains[1].risk_parameters,
yscale=:log10, title="TN-ILM parameters")
```

For the remainder of this example we take every 20th iteration from iteration 10000 through 50000 as being representative samples from the TN-ILM posterior distribution. With this, the epidemic curve posterior distributions are visualized with

```
p2 = plot(mcmc.markov_chains[1].events[10000],
    State_S, linealpha=0.01, title="S")
for i=10020:20:50000
  plot!(p2, mcmc.markov_chains[1].events[i], State_S, linealpha=0.01)
end
plot!(p2, sim.events, State_S, linecolor=:black)
```

for each state. Partially transparent epidemic curves are repeatedly plotted to show the posterior density. With a layout, the plots are combined to to form Figure 3.

```
l = @layout [a; [b c d]]
plot(p1, p2, p3, p4, layout=l)
```

The `TransmissionNetworkPosterior` or `TNPosterior` type for short, is used to represent the posterior distribution of Transmission Networks, which can be plotted or analyzed by other means. Figure 4 shows such a plot, and can be generated with:

```
p1 = plot(sim.transmission_network, sim.population,
          title="True Transmission\nNetwork", framestyle=:box)

tnp = TNPosterior(mcmc.markov_chains[1].transmission_network[10000:20:50000])
p2 = plot(tnp, sim.population,
          title="Transmission Network\nPosterior Distribution", framestyle=:box)

plot(p1, p2, layout=(1, 2))
```

Credible intervals and point estimates can also be calculated for TN-ILM parameters and event times from the MCMC samples.

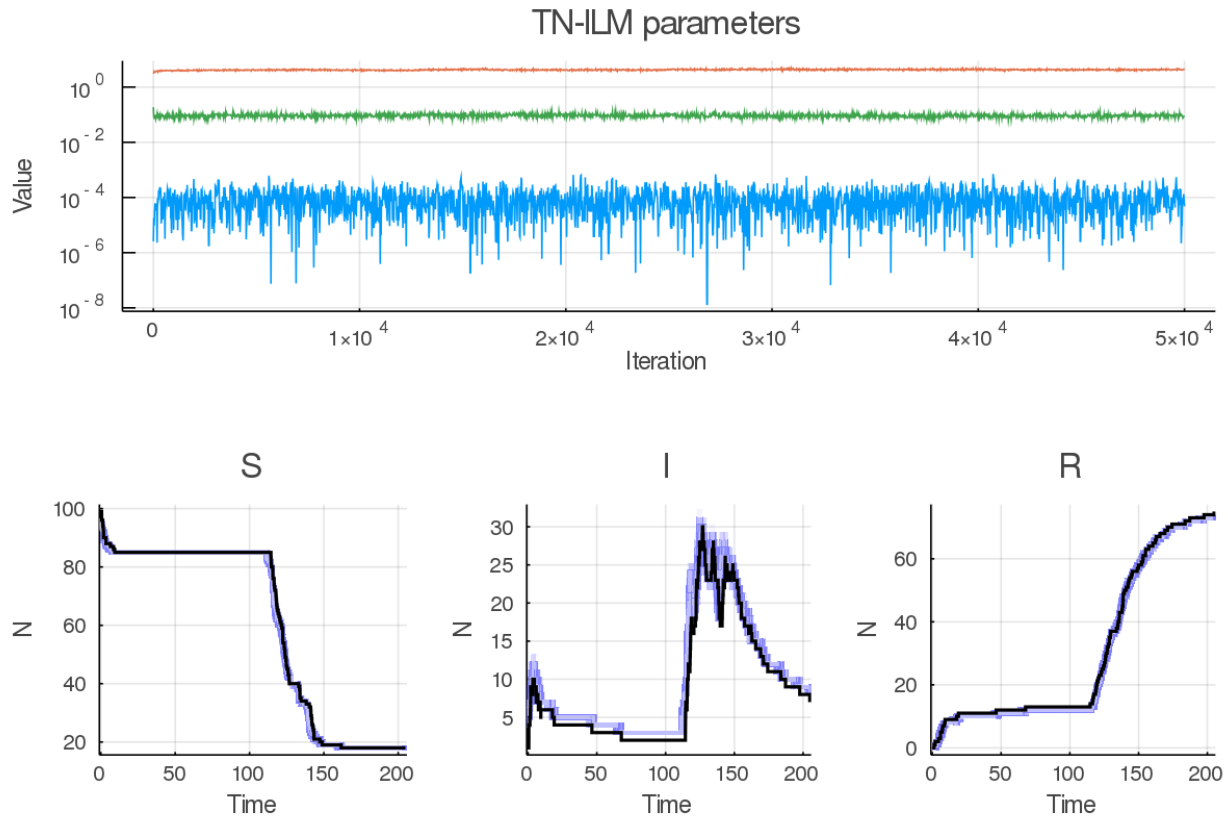

Figure 3: Here we should several plots for visualizing MCMC for an TN-ILM. The top plot shows the values of 3 TN-ILM parameters over 50k iterations. The bottom row of plots shows the epidemic curve posterior distributions in blue (posterior density estimated with iterations 10k - 50k), with the true epidemic curves indicated with black.

```
tracedata = convert(Array{Float64, 2}, mcmc.markov_chains[1].risk_parameters)
tracesummary = vcat(mean(tracedata[10000:20:50000, :], dims=1),
                    [quantile(tracedata[10000:20:50000, i], j)
                    for j = [0.025, 0.975], i = 1:3])
```

Which evaluates to:

```
3x3 Array{Float64,2}:
 9.61085e-5   4.26675  0.0928756
 2.86372e-6   3.9581   0.0720961
 0.000375989  4.61725  0.116018
```

Where the rows are the mean, and lower and upper bounds of the 95% credible interval for each of the three TN-ILM parameters. In our example, the parameter values that were used to generate the epidemic simulation were $0.0001, 4.0$, and 0.1, and are all well contained in the credible intervals.

6 Future work

In the future, we hope to increase the feature set of **Pathogen.jl** to include TN-ILMs incorporating time-varying disease state transition rates. The flexibility of **Pathogen.jl** could be explored further through the use of Artificial Neural Networks for one or more rate describing functions, similar to what has been done with Neural Differential Equations with **DiffEqFlux.jl** (Rackauckas *et al.*, 2019). Providing simulation and inference capabilities for phylodynamic ILMs is also of particular interest. Additional inference methods may also be added, such as Approximate Bayesian Computation. Beyond these, as the Julia package ecosystem continues to mature, integration with other Julia packages will be prioritized. While not only reducing the

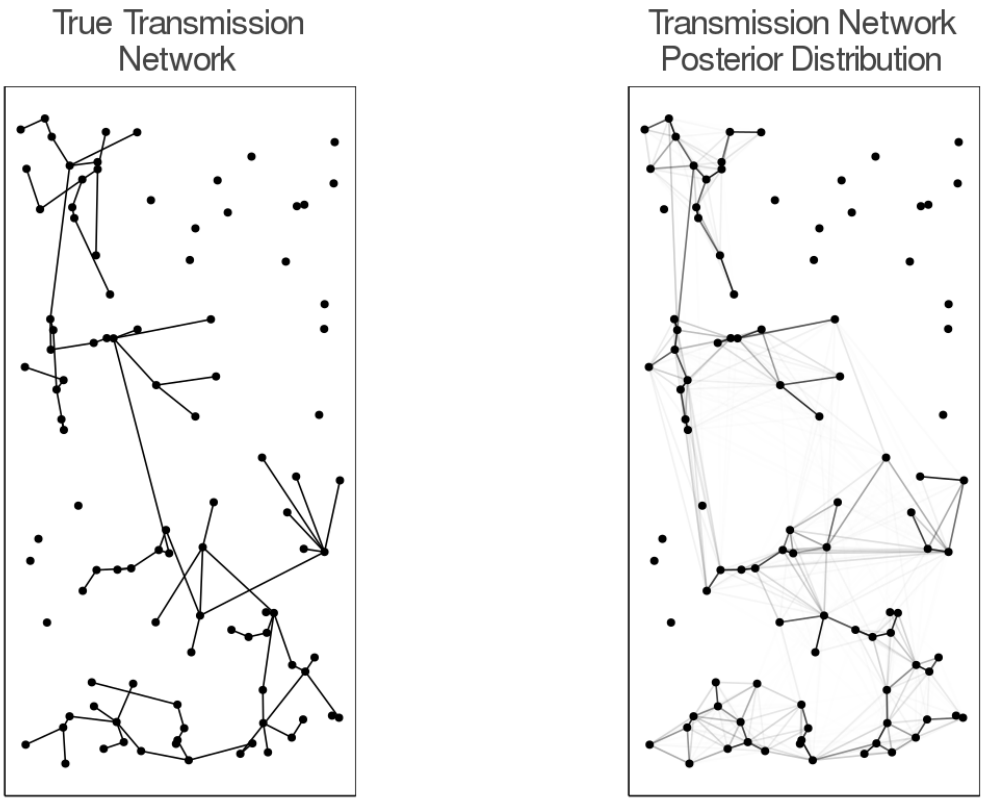

Figure 4: The true transmission network (left), in comparison to the transmission network posterior distribution (right), where the transparency of transmission pathways represents their posterior density.

codebase of **Pathogen.jl**, this process will present opportunity to add features, increase generality, improve performance, and improve usability of the package.

Acknowledgements

This research was funded via a Highly Qualified Personnel HQP scholarship from the Ontario Ministry of Agriculture, Food and Rural Affairs (OMAFRA) / University of Guelph Partnership, as well as from Dr. Feng's and Dr. Deardon's Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants.

References

Abdulkareem SA, Augustijn EW, Filatova T, Musial K, Mustafa YT (2020). "Risk perception and behavioral change during epidemics: Comparing models of individual and collective learning." *PloS one*. doi:10.1371/journal.pone.0226483.

Almutiry W, Deardon R, Warriyar K V V (2020). *EpiILMCT: Continuous Time Distance-Based and Network-Based Individual Level Models for Epidemics*. R package version 1.1.6, URL `https://CRAN.R-project.org/package=EpiILMCT`.

Besançon M, Anthoff D, Arslan A, Byrne S, Lin D, Papamarkou T, Pearson J (2019). "Distributions.jl: Definition and Modeling of Probability Distributions in the JuliaStats Ecosystem." *arXiv preprint arXiv::1907.08611*.

Bezanson J, Chen J, Chung B, Karpinski S, Shah VB, Vitek J, Zoubritzky L (2018). "Julia: Dynamism and Performance Reconciled by Design." *Proc. ACM Program. Lang.*, **2**(OOPSLA), 120:1–120:23. ISSN 2475-1421. doi:10.1145/3276490.

Bezanson J, Edelman A, Karpinski S, Shah VB (2017). "Julia: A fresh approach to numerical computing." *SIAM Review*, **59**(1), 65–98. doi:10.1137/141000671.

Blaom A, Kiraly F, Lienart T, Vollmer S (2020). "alan-turing-institute/MLJ.jl: v0.8.0." doi:10.5281/zenodo.3635093.

Breloff T (2015). "Plots.jl: Powerful convenience for visualization in Julia." URL `http://docs.juliaplots.org/`.

Deardon R, Brooks SP, Grenfell BT, Keeling MJ, Tildesley MJ, Savill NJ, Shaw DJ, Woolhouse ME (2010). "Inference for individual-level models of infectious diseases in large populations." *Statistica Sinica*, **20**(1), 239–261.

Gillespie DT (1977). "Exact stochastic simulation of coupled chemical reactions." *The Journal of Physical Chemistry*, **81**(25), 2340–2361. doi:10.1021/j100540a008.

Groendyke C, Welch D (2018). "epinet: An R Package to Analyze Epidemics Spread across Contact Networks." *Journal of Statistical Software, Articles*, **83**(11), 1–22. doi:10.18637/jss.v083.i11.

Hastings WK (1970). "Monte Carlo sampling methods using Markov chains and their applications." *Biometrika*, **57**(1), 97–109. ISSN 0006-3444. doi:10.1093/biomet/57.1.97.

Heinen J, *et al.* (1985–2019). "GR Framework." URL `https://gr-framework.org/`.

Innes M (2018). "Flux: Elegant Machine Learning with Julia." *Journal of Open Source Software*. doi:10.21105/joss.00602.

Landeros A, Stutz T, Keys KL, Alekseyenko A, Sinsheimer JS, Lange K, Sehl ME (2018). "BioSimulator.jl: Stochastic simulation in Julia." *Computer Methods and Programs in Biomedicine*, **167**, 23–35. doi:10.1016/j.cmpb.2018.09.009.

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL `https://www.R-project.org/`.

Rackauckas C, Innes M, Ma Y, Bettencourt J, White L, Dixit V (2019). "DiffEqFlux.jl - A Julia Library for Neural Differential Equations." *CoRR*, **abs/1902.02376**. `1902.02376`, URL `http://arxiv.org/abs/1902.02376`.

Rackauckas C, Nie Q (2017a). "Adaptive methods for stochastic differential equations via natural embeddings and rejection sampling with memory." *Discrete & Continuous Dynamical Systems - B*, **22**(7), 1–31. doi:10.3934/dcdsb.2017133.

Rackauckas C, Nie Q (2017b). "DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia." *Journal of Open Research Software*, **5**(1). doi:10.5334/jors.151.

Robert C, Casella G (2013). *Monte Carlo statistical methods*. Springer Science & Business Media. ISBN 978-1-4757-4145-2.

Roberts GO, Rosenthal JS (2007). "Coupling and ergodicity of adaptive Markov chain Monte Carlo algorithms." *Journal of Applied Probability*, pp. 458–475. doi:10.1239/jap/1183667414.

Ross RJ, Baker RE, Parker A, Ford M, Mort R, Yates C (2017). "Using approximate Bayesian computation to quantify cell–cell adhesion parameters in a cell migratory process." *NPJ systems biology and applications*, **3**(1), 1–10. doi:10.1038/s41540-017-0010-7.

Vahdati A (2019). "Agents.jl: agent-based modeling framework in Julia." *Journal of Open Source Software*, **4**(42). doi:10.21105/joss.01611.

Warriyar K V V, Deardon R (2018). *EpiILM: Spatial and Network Based Individual Level Models for Epidemics*. R package version 1.4.2, URL `https://CRAN.R-project.org/package=EpiILM`.

Webb GF (2017). "Individual based models and differential equations models of nosocomial epidemics in hospital intensive care units." *Discrete & Continuous Dynamical Systems-B*, **22**(3), 1145–1166. doi:10.3934/dcdsb.2017056.