SimInf: An R Package for Data-Driven Stochastic Disease Spread Simulations

Stefan Widgren
National Veterinary Institute
and Uppsala University
Sweden

Pavol Bauer
Uppsala University
Sweden

Robin Eriksson
Uppsala University
Sweden

Stefan Engblom
Uppsala University
Sweden

Abstract

We present the R package **SimInf** which provides an efficient and very flexible framework to conduct data-driven epidemiological modeling in realistic large scale disease spread simulations. The framework integrates infection dynamics in subpopulations as continuous-time Markov chains using the Gillespie stochastic simulation algorithm and incorporates available data such as births, deaths and movements as scheduled events at predefined time-points. Using C code for the numerical solvers and divide work over multiple processors ensures high performance when simulating a sample outcome. One of our design goals was to make **SimInf** extendable and enable usage of the numerical solvers from other R extension packages in order to facilitate complex epidemiological research. In this paper, we provide a technical description of the framework and demonstrate its use on some basic examples. We also discuss how to specify and extend the framework with user-defined models.

Keywords: computational epidemiology, discrete-event simulation, multicore implementation, stochastic modeling.

Note

This vignette is based on the paper by Widgren, Bauer, Eriksson, and Engblom (2019) but has been updated in some places to describe functionality in **SimInf** that has been developed since the paper was published. See the package NEWS file for recent changes. The results reported here are based on **SimInf** version 11.1.0.9000 unless otherwise stated.

1. Introduction

Cattle can act as a reservoir for *Salmonella* and verotoxin-producing *Escherichia coli* (VTEC), two important examples of zoonotic food-borne pathogens. In order to develop effective control strategies, it is necessary to understand the spread of zoonotic diseases in the cattle

population (Newell, Koopmans, Verhoef, Duizer, Aidara-Kane, Sprong, Opsteegh, Langelaar, Threfall, Scheutz *et al.* 2010). Since cattle are aggregated into spatially segregated farms, it is natural to use a metapopulation framework and partition the cattle population into interacting subpopulations (Grenfell and Harwood 1997; Keeling, Danon, Vernon, and House 2010). Furthermore, livestock data is commonly available, with information on births, deaths and movements (Brooks-Pollock, de Jong, Keeling, Klinkenberg, and Wood 2015). Consequently, detailed spatiotemporal demographic data and the transportation network are available and can be used for epidemiologically relevant factors when simulating the infection process within each subpopulation, coupled with spread among subpopulations governed by spatial proximity and livestock movements. However, incorporating large amounts of data in simulations is computationally challenging and requires efficient algorithms.

In this work, we present the R (R Core Team 2017) package **SimInf**, a flexible framework for data-driven spatio-temporal disease spread modeling, designed to efficiently handle population demographics and network data. The framework integrates infection dynamics in each subpopulation as continuous-time Markov chains (CTMC) using the Gillespie stochastic simulation algorithm (SSA) (Gillespie 1977) and incorporates available data such as births, deaths or movements as scheduled events. A scheduled event is used to modify the state of a subpopulation at a predefined time-point. Using compiled C (Kernighan and Ritchie 1988) code, rather than interpreted R code, for the numerical solvers ensures high performance when simulating a model. To further improve performance, OpenMP (OpenMP Architecture Review Board 2008) is used to divide work over multiple processors and perform computations in parallel. Furthermore, the framework has a well-defined interface to incorporate data that is shared among all subpopulations (global) and data that is specific to each subpopulation (local), allowing sophisticated models to be straightforwardly formulated. The proposed approach was used to study spread and control of VTEC in the complete Swedish cattle population, incorporating almost ten years of scheduled events data in a network of about 40,000 subpopulations (Bauer, Engblom, and Widgren 2016; Widgren, Engblom, Bauer, Frössling, Emanuelson, and Lindberg 2016; Widgren, Engblom, Emanuelson, and Lindberg 2018). Even if development of **SimInf** was inspired by livestock diseases and models driven by available data, the design is of completely general character and applies to arbitrary metapopulation models. One of our design goals was to make **SimInf** extendable and enable usage of the numerical solvers from other R extension packages in order to facilitate complex epidemiological research. To support this, **SimInf** has functionality to generate the required C and R code from a model specification.

Various packages available at the Comprehensive R Archive Network (CRAN) implement SSA to simulate a continuous-time stochastic process. The **GillespieSSA** package (Pineda-Krch 2008), on CRAN since 2007, implements both the direct method and three approximate methods. The **hybridModels** package (Marques, Amaku, and Grisi-Filho 2017) uses **GillespieSSA** internally to simulate infections using a metapopulation model coupled with spread among subpopulations. Since each outcome of a stochastic process is different, it is (generally) necessary to study many realisations of the process to see the distribution of outcomes consistent with the model structure and parameterization. Therefore, performance of the simulator becomes critical when using these methods in an applied context. Because the algorithms in **GillespieSSA** are implemented in R, the computational efficiency is limited in comparison with implementations in a compiled language, for example, C or C++. The **adaptivetau** package (Johnson 2016) uses a hybrid R/C++ strategy to implement the direct method and adaptive

tau leaping (Cao, Gillespie, and Petzold 2007).

There exists several related R packages for epidemiological analysis on CRAN. For example, the package **amei** (Merl, Johnson, Gramacy, and Mangel 2010), designed for finding optimal intervention strategies to minimize total expected cost to control a disease outbreak. Another package is **surveillance**, a framework for monitoring, modeling, and regression analysis of infectious diseases, (Meyer, Held, and Höhle 2017). **EpiModel** (Jenness, Goodreau, and Morris 2017) is an R package that includes a framework for modeling spread of diseases on networks. In **EpiModel** the individual is the unit and transmission between individuals is modelled through a contact network in discrete time.

The remainder of this paper is organized as follows. In Section 2 we summarize the mathematical foundation for our framework. Section 3 gives a technical description of the simulation framework. In Section 4 we illustrate the use of the package by some worked examples. In Section 5 we demonstrate how to extend **SimInf** with user-defined models. Finally, in Section 6 we provide a small benchmark between various R packages of the run-time to simulate SSA trajectories.

2. Epidemiological modeling

In mathematical modeling of the dynamics of an infectious disease in a population, the population under study is commonly divided into compartments representing discrete health states, together with assumptions about the transition rates for individuals to move from one compartment to another (Kermack and McKendrick 1927; Andersson and May 1991; Keeling and Rohani 2007). In order to capture spatial characteristics of a disease process, a compartment model can be further partioned into metapopulations (cities, households, farms), i.e., subpopulations with its own infection dynamics (Grenfell and Harwood 1997). Since the population size of each subpopulation is small, it is often necessary to use stochastic models, e.g., to account for the random event that an infection will become extinct (Bartlett 1957). A stochastic compartment model is naturally formulated as a CTMC using SSA to simulate the number of individuals within each compartment through time (Keeling and Rohani 2007). Consequently, SSA is often used when modeling various infectious diseases, for example, Ebola virus disease outbreak (King, Domenech de Cellès, Magpantay, and Rohani 2015), seasonality of influenza epidemics (Dushoff, Plotkin, Levin, and Earn 2004), avian influenza virus in bird populations (Breban, Drake, Stallknecht, and Rohani 2009), paratuberculosis infection in cattle (Smith, Schukken, and Gröhn 2015).

In order to model disease spread on a larger scale, the infection process within each subpopulation must be coupled with spread among subpopulations. For example, livestock movements are an important transmission route for many infectious diseases and can transfer infectious individuals between farms over large distances (Danon, Ford, House, Jewell, Keeling, Roberts, Ross, and Vernon 2011). The livestock movements create complex dynamic interactions among farms that can be represented as a directed temporal network (Kempe, Kleinberg, and Kumar 2002; Bajardi, Barrat, Natale, Savini, and Colizza 2011; Dutta, Ezanno, and Vergu 2014). In network terminology, each farm is represented by a node (also called a vertex). Moreover, each movement forms an edge (also called a link) between two nodes and following all edges through time, an infection may "flow" in the network and spread from node to node. Let N_{nodes} denote the number of nodes in a population, and let $i, j, k \in \{1, \ldots, N_{\text{nodes}}\}$

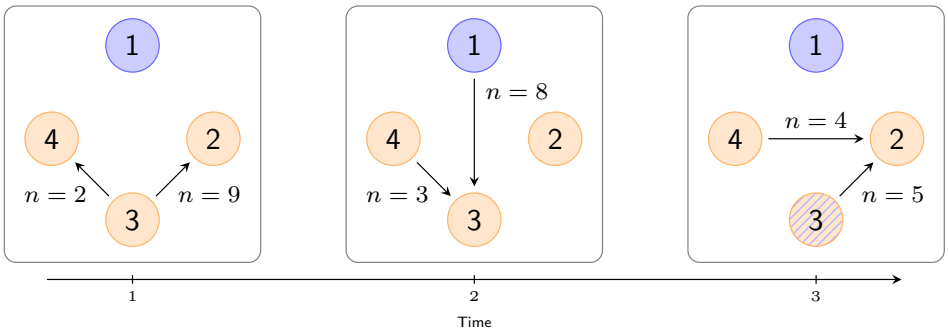

Figure 1: Illustration of movements as a temporal network. Each time step depicts movements during one time unit, for example, a day. The network has $N = 4$ nodes where node 1 is infected and nodes 2–4 are non-infected. Arrows indicate movements of individuals from a source node to a destination node and labels denote the size of the shipment. Here, infection may spread from node 1 to node 3 at $t = 2$ and then from node 3 to node 2 at $t = 3$.

denote three distinct nodes. As illustrated in Figure 1, just because there exists an edge from i to j does not mean that there exists an edge from j to i. Moreover, the existence of an edge from i to j and one from j to k does not imply there exists a path from i to k. Furthermore, note that the order of the edges matter, consider swapping the first and second time step in Figure 1, then another path for spread is possible, namely from node 3 to node 4 and then to node 2.

In Sections 2.1–2.3, we provide an overview of the epidemiological modeling framework employed in **SimInf**. The overall approach consists of CTMCs as a general model of the dynamics of the epidemiological state. Importantly, we also allow for variables obeying ordinary differential equations (ODEs). For example, this readily supports modeling infections that have an indirect transmission route, e.g., via shedding of a pathogen to the environment (Ayscue, Lanzas, Ivanek, and Gröhn 2009; Breban *et al.* 2009). Additionally, the framework handles externally defined demographic and movement events. In Sections 2.1–2.2 below we distinguish between the *local* dynamics that describes the evolution of the epidemiological state at a single node, and the *global* dynamics, which describes the system at the network level. The overall numerical approach underlying **SimInf** is described in Section 2.3. We draw much of the material here from (Bauer *et al.* 2016; Engblom and Widgren 2017).

2.1. Local dynamics

We describe the state of a single node with a *state vector* $X(t) \in \mathbf{Z}_+^{N_{\text{comp}}}$, which counts the number of individuals at each of N_{comp} compartments at time t. The transitions between these compartments are stochastic and are described by the transition matrix $\mathbb{S} \in \mathbf{Z}^{N_{\text{comp}} \times N_{\text{trans}}}$ and the transition intensities $R : \mathbf{Z}_+^{N_{\text{comp}}} \to \mathbf{R}_+^{N_{\text{trans}}}$, assuming N_{trans} different transitions. We then form a random counting measure $\mu_k(dt) = \mu(R_k(X(t-)); dt)$ that is associated with a Poisson process for the kth intensity $R_k(X(t-)$, which in turn depends on the state prior to any transition at time t, that is, $X(t-)$.

The local dynamics can then compactly be described by a pure jump stochastic differential

equation (SDE),

$$dX(t) = \mathbb{S}\boldsymbol{\mu}(dt), \tag{1}$$

where $\boldsymbol{\mu}(dt)$ is a vector measure built up from the scalar counting measures $\boldsymbol{\mu}(dt) = [\mu_1(dt), \ldots, \mu_{N_{\text{trans}}}(dt)]^\top$. If at time t, transition k occurs, then the state vector is updated according to

$$X(t) = X(t-) + \mathbb{S}_k, \tag{2}$$

with $\mathbb{S}_k$ the kth column of $\mathbb{S}$. In (1) the N_{trans} different epidemiological state transitions are competing in the sense of independent Poisson processes. The "winning" process decides what event happens and changes the state according to (2). The simulation then proceeds under the Markov assumption where previous events are remembered via the state variable X only.

To make this abstract notation a bit more concrete we consider a traditional example as follows. In an SIS-model the transitions between a susceptible and an infected compartment can be written as

$$\left.\begin{array}{rl} S + I & \xrightarrow{\beta} 2I \\ I & \xrightarrow{\gamma} S \end{array}\right\}. \tag{3}$$

With a state vector consisting of two compartments $X = [S, I]$, i.e., the number of susceptible and infected individuals, respectively, we can then write the transition matrix and intensity vector as

$$\mathbb{S} = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{4}$$

$$R(x) = [\beta x_1 x_2, \gamma x_2]^\top. \tag{5}$$

To connect this with traditional ODE-based models, note that, replacing the random measure in (1) with its mean drift, we arrive at

$$\frac{dx(t)}{dt} = \mathbb{S}R(x), \tag{6}$$

where now the state variable $x \in \mathbf{R}^{N_{\text{comp}}}$. The differences between (1) and (6) are that the randomness and discreteness of the state variable are not present in the latter formulation. If these features are thought to be important, then (1) is an accurate stochastic alternative to (6), relying only on the Markovian "memoryless" assumption.

There are, however, situations where we would like to mix the discrete stochastic model with a concentration-type ODE model. In a multi-scale description there are typically variables for which a continuous description is more natural: a typical example is the concentration of bacteria in an infectious environment for which individual counting would clearly not be feasible.

Assuming an additional concentration state vector $Y \in \mathbf{R}^{N_{\text{conc}}}$ a general model which augments (1) is

$$\left.\begin{array}{rcl} dX(t) & = & \mathbb{S}\boldsymbol{\mu}(dt) \\ Y'(t) & = & f(X(t), Y(t)) \end{array}\right\}, \tag{7}$$

where now the random measure depends also on the concentration variable,

$$\boldsymbol{\mu}(dt) = \boldsymbol{\mu}(R(X(t-), Y(t)), dt). \tag{8}$$

The overall combined state vector is then $[X;\ Y] \in [\mathbf{Z}^{N_{\text{comp}}};\ \mathbf{R}^{N_{\text{conc}}}]$.

2.2. Global dynamics

To extend the local dynamics to a network model consisting of N_{nodes} nodes we first define the overall state matrices $\mathbb{X} \in \mathbf{Z}_+^{N_{\text{comp}} \times N_{\text{nodes}}}$ and $\mathbb{Y} \in \mathbf{R}^{N_{\text{conc}} \times N_{\text{nodes}}}$ and then extend (7) to

$$d\mathbb{X}^{(i)}(t) = \mathbb{S}\boldsymbol{\mu}^{(i)}(dt), \tag{9}$$

$$\frac{d\mathbb{Y}^{(i)}(t)}{dt} = f(\mathbb{X}^{(i)}, \mathbb{Y}^{(i)}), \tag{10}$$

where $i \in \{1, \ldots, N_{\text{nodes}}\}$ is the node index.

We then consider the N_{nodes} nodes being the vertices of an undirected graph $\mathcal{G}$ with interactions defined in terms of the counting measures $\boldsymbol{\nu}^{(i,j)} = \boldsymbol{\nu}^{(i,j)}(dt)$ and $\boldsymbol{\nu}^{(j,i)}$. Here $\boldsymbol{\nu}^{(i,j)}$ represents the state changes due to an inflow of individuals from node i to node j, and $\boldsymbol{\nu}^{(j,i)}$ represents an inflow of individuals from node j to node i, assuming node j being in the connected component $C(i)$ of node i, and vice versa. We denote the connected components of the graph $\mathcal{G}$ as the matrix $\mathbb{C} \in \mathbf{Z}_+^{N_{\text{comp}} \times N_{\text{nodes}}}$.

The network dynamics is then written as

$$d\mathbb{X}_t^{(i)} = -\sum_{j \in C(i)} \mathbb{C}\boldsymbol{\nu}^{(i,j)}(dt) + \sum_{j;\, i \in C(j)} \mathbb{C}\boldsymbol{\nu}^{(j,i)}(dt), \tag{11}$$

$$\frac{d\mathbb{Y}^{(i)}(t)}{dt} = -\sum_{j \in C(i)} g(\mathbb{X}^{(i)}, \mathbb{Y}^{(i)}) + \sum_{j;\, i \in C(j)} g(\mathbb{X}^{(j)}, \mathbb{Y}^{(j)}). \tag{12}$$

In (12), g is similarly the "flow" of the concentration variable $\mathbb{Y}$ between the nodes in the network. For example, this could be the natural modeling target for concentration variables $\mathbb{Y}$ which are transported via surface water or air.

Combining this with (9)–(10) we obtain the overall dynamics

$$d\mathbb{X}^{(i)}(t) = \mathbb{S}\boldsymbol{\mu}^{(i)}(dt) - \sum_{j \in C(i)} \mathbb{C}\boldsymbol{\nu}^{(i,j)}(dt) + \sum_{j;\, i \in C(j)} \mathbb{C}\boldsymbol{\nu}^{(j,i)}(dt), \tag{13}$$

$$\frac{d\mathbb{Y}^{(i)}(t)}{dt} = f(\mathbb{X}^{(i)}, \mathbb{Y}^{(i)}) - \sum_{j \in C(i)} g(\mathbb{X}^{(i)}, \mathbb{Y}^{(i)}) + \sum_{j;\, i \in C(j)} g(\mathbb{X}^{(j)}, \mathbb{Y}^{(j)}). \tag{14}$$

Note that $\boldsymbol{\nu}^{(i,j)}$ and $\boldsymbol{\nu}^{(j,i)}$ may be equivalently employed for externally scheduled events given by data using an equivalent construction in terms of Dirac measures. This is the case, for example, when intra-nodal transport data of individuals are available.

2.3. Numerical method

In **SimInf**, we solve (13)–(14) by splitting the local update scheme (9)–(10) from the global update scheme (11)–(12). We discretize time as $0 = t_0 < t_1 < t_2 < \cdots$, which is partially

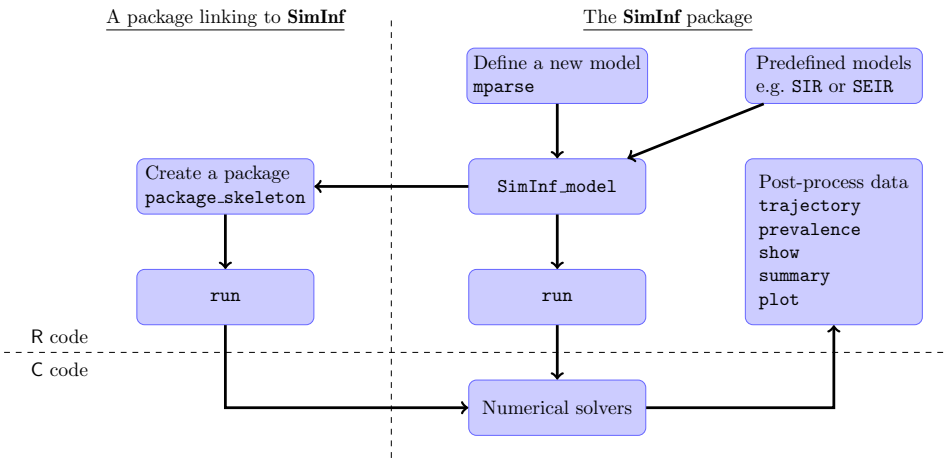

Figure 2: Schematic overview of the functionality in **SimInf**. The central object is the S4 class `SimInf_model` which contains the specification and data for a model. A new model object is created using `mparse` or a predefined template, for example, `SIR` or `SEIR`. A stochastic trajectory is simulated from a model using `run`. For computational efficiency, the numerical solvers are implemented in C code. There are several functions in **SimInf** to facilitate analysis and post-processing of simulated data, for example, `trajectory`, `prevalence` and `plot`. **SimInf** supports usage of the numerical solvers from other R packages via the `LinkingTo` feature in R.

required as external data has to be incorporated at some finitely resolved time stamps. The numerical method of **SimInf** can then be written per node i as

$$\tilde{\mathbb{X}}_{n+1}^{(i)} = \mathbb{X}_n^{(i)} + \int_{t_n}^{t_{n+1}} \mathbb{S}\boldsymbol{\mu}^{(i)}(ds), \tag{15}$$

$$\mathbb{X}_{n+1}^{(i)} = \tilde{\mathbb{X}}_{n+1}^{(i)} - \int_{t_n}^{t_{n+1}} \sum_{j\in C(i)} \mathbb{C}\boldsymbol{\nu}^{(i,j)}(ds) + \int_{t_n}^{t_{n+1}} \sum_{j;\, i\in C(j)} \mathbb{C}\boldsymbol{\nu}^{(j,i)}(ds), \tag{16}$$

$$\begin{aligned}\mathbb{Y}_{n+1}^{(i)} &= \mathbb{Y}_n^{(i)} + f(\tilde{\mathbb{X}}_{n+1}^{(i)}, \mathbb{Y}_n^{(i)})\,\Delta t_n \\ &\quad - \sum_{j\in C(i)} g(\tilde{\mathbb{X}}_{n+1}^{(i)}, \mathbb{Y}_n^{(i)})\Delta t_n + \sum_{j;\, i\in C(j)} g(\tilde{\mathbb{X}}_{n+1}^{(j)}, \mathbb{Y}_n^{(j)})\Delta t_n.\end{aligned} \tag{17}$$

In this scheme, (15) forms the local stochastic step, that is in practice simulated by the Gillespie method (Gillespie 1977). Equation (16) is the data step, where externally scheduled events are incorporated. Note that the stochastic step evolves in continuous time in the interval $[t_n, t_{n+1}]$, and the data step operates only on the final state $\tilde{\mathbb{X}}$ at t_{n+1}. The final step (17) is just the Euler forward method in time with time-step $\Delta t_n = t_{n+1} - t_n$ for the concentration variable $\mathbb{Y}$.

3. Technical description of the simulation framework

The overall design of **SimInf** was inspired and partly adapted from the Unstructured Mesh Reaction-Diffusion Master Equation (URDME) framework (Engblom, Ferm, Hellander, and

Lötstedt 2009; Drawert, Engblom, and Hellander 2012). **SimInf** uses object oriented programming and the S4 class (Chambers 2008) `SimInf_model` is central and provides the basis for the framework. A `SimInf_model` object supplies the state-change matrix, the dependency graph, the scheduled events, and the initial state of the system. Briefly, the state-change matrix defines the effect of the disease transitions on the state of the system while the dependency graph indicates the transition rates that need to be updated after a given disease transition. Additionally, model specific code written in C specifies the transition rate functions for the disease transitions in the system. All predefined models in **SimInf** have a generating function (Chambers 2008), with the same name as the model, to initialize the data structures for that specific model, see the examples in Section 4. A model can also be created from a model specification using the `mparse` method, further described in Section 5. After a model is created, a simulation is started with a call to the `run` method and if execution is successful, it returns a modified `SimInf_model` object with a single stochastic solution trajectory attached to it. **SimInf** provides several utility functions to inspect simulated data, for example, `show`, `summary` and `plot`. To facilitate custom analysis, **SimInf** provides the `trajectory` and `prevalence` methods that both return a `data.frame` with simulated data. Figure 2 shows a schematic overview of the functionality in **SimInf**. The overall modular design makes extensions easy to handle.

3.1. Installation

The most recent stable version of **SimInf** is available from the Comprehensive R Archive Network (CRAN) at (`https://CRAN.R-project.org/package=SimInf`) and may, depending on your platform, be available in source form or compiled binary form. The development version is available on GitHub (`https://github.com/stewid/SimInf`). A binary form of **SimInf** for macOS or Windows can be installed directly from CRAN. However, if you install **SimInf** from source (from CRAN or a `.tar.gz` file), the installation process requires a C compiler, and that the GNU Scientific Library (GSL) (Galassi, Davies, Theiler, Gough, Jungman, Alken, Booth, and Rossi 2009) is installed on your system and is on the path. On a Windows machine you first need to download and install Rtools from `https://cran.r-project.org/bin/windows/Rtools/`. Note that GSL (`https://www.gnu.org/software/gsl/`) is not an R add-on package, but needs to be installed separately, for example, from a terminal using: `sudo apt-get install libgsl0-dev` on Debian and Ubuntu, `sudo yum install gsl-devel` on Fedora, CentOS or RHEL, or `brew install gsl` on macOS with the Homebrew package manager. On Windows, the GSL files are downloaded, if needed, from `https://github.com/rwinlib/gsl` during the installation of **SimInf**. Furthermore, when you install **SimInf** from source, depending on features of the compiler, the package is compiled with support for OpenMP. To find out more about installing R add-on packages in general, the *R Installation and Administration* (`https://cran.r-project.org/manuals.html`) manual describes the process in detail. After installing the package

```
R> install.packages("SimInf")
```

it is loaded in R with the following command

```
R> library("SimInf")
```

Slot	Description
`S`	Each column corresponds to a state transition, and execution of state transition j amounts to adding the `S[, j]` column to the state vector `u[, i]` of node i where the transition occurred. Sparse matrix ($N_{\text{comp}} \times N_{\text{trans}}$) of object class `dgCMatrix`.
`G`	Dependency graph that indicates the transition rates that need to be updated after a given state transition has occurred. A non-zero entry in element `G[i, j]` indicates that transition rate i needs to be recalculated if the state transition j occurs. Sparse matrix ($N_{\text{trans}} \times N_{\text{trans}}$) of object class `dgCMatrix`.
`tspan`	A vector of increasing time points where the state of each node is to be returned.
`U`	The result matrix with the number of individuals in each compartment in every node. `U[, j]` contains the number of individuals in each compartment at `tspan[j]`. `U[1:`N_{comp}`, j]` contains the number of individuals in each compartment in node 1 at `tspan[j]`. `U[(`N_{comp}` + 1):(2 *` N_{comp}`), j]` contains the number of individuals in each compartment in node 2 at `tspan[j]` etc. Integer matrix ($N_{\text{nodes}} N_{\text{comp}} \times \text{length(tspan)}$). Since version 10, the internal format of `U` has been expanded to also allow replicates of each node. Each replicate adds new columns to `U` so that the data for each replicate is in blocks of `length(tspan)` columns.
`U_sparse`	It is possible to run the simulator and write the number of individuals in each compartment to the `U_sparse` sparse matrix (`dgCMatrix`), which can save a lot of memory if the model contains many nodes and time-points, but where only a few of the data points are of interest. If `U_sparse` is non-empty when `run` is called, the non-zero entries in `U_sparse` indicates where the number of individuals should be written to `U_sparse`. The layout of the data in `U_sparse` is identical to `U`. Please note that the data in `U_sparse` is numeric and that the data in `U` is integer. Also note that replicates (see above) cannot be used when data is written to a sparse matrix.
`u0`	The initial number of individuals in each compartment in every node. Integer matrix ($N_{\text{comp}} \times N_{\text{nodes}}$).
`V`	The result matrix for the real-valued continuous state. `V[, j]` contains the real-valued state of the system at `tspan[j]`. Numeric matrix ($N_{\text{nodes}} N_{ld} \times \text{length(tspan)}$). Since version 10, the internal format of `V` has been expanded to also allow replicates of each node. Each replicate adds new columns to `V` so that the data for each replicate is in blocks of `length(tspan)` columns.
`V_sparse`	It is possible to run the simulator and write the real-valued continuous state to the `V_sparse` sparse matrix (`dgCMatrix`), which can save a lot of memory if the model contains many nodes and time-points, but where only a few of the data points are of interest. If `V_sparse` is non-empty when `run` is called, the non-zero entries in `V_sparse` indicates where the real-valued continuous state should be written to `V_sparse`. The layout of the data in `V_sparse` is identical to `V`. Please note that replicates (see above) cannot be used when data is written to a sparse matrix.
`v0`	The initial value for the real-valued continuous state. Numeric matrix ($N_{ld} \times N_{\text{nodes}}$).
`ldata`	A numeric matrix with local data specific to each node. The column `ldata[, j]` contains the local data vector for node j. The local data vector is passed as an argument to the transition rate functions and the post time step function.
`gdata`	A numeric vector with global data that is common to all nodes. The global data vector is passed as an argument to the transition rate functions and the post time step function.
`events`	Scheduled events to modify the discrete state of individuals in a node at a pre-defined time t. S4 class `SimInf_events`, see Section 3.3 and Table 2.
`C_code`	Character vector with optional model C code, see Section 5. If non-empty, the C code is written to a temporary file when the `run` method is called. The temporary file is compiled and the resulting DLL is dynamically loaded.

Table 1: Description of the slots in the S4 class `SimInf_model` that defines the epidemiological model. N_{trans} is the number of state transitions in the model. N_{comp} is the number of compartments in the model. N_{nodes} is the number of nodes in the model. N_{ld} is the number of local data specific to each node and equals `dim(ldata)[1]`.

3.2. Specification of an epidemiological model

The within-node disease spread model in **SimInf** is specified as a compartment model with the individuals divided into compartments defined by discrete disease statuses. The model is defined by the slots in the S4 class `SimInf_model` (Table 1). The compartments contains the number of individuals in each of the N_{comp} disease states in every N_{nodes} nodes.

Equation (17), the stochastic step, contains N_{trans} state transitions and is processed using the two slots `S` and `G`. The `S` slot is the state-change matrix ($N_{\text{comp}} \times N_{\text{trans}}$) that determines how to change the number of individuals in the compartments of a node when the jth state transition occurs, where $1 \leq j \leq N_{\text{trans}}$. Each row corresponds to one compartment and each column to a state transition. Let `u[, i]` be the number of individuals in each compartment in node i at time t_i. To move simulation time forward in node i to $t_i = t_i + \tau_i$, the vector `u[, i]` is updated according to the jth transition by adding the state-change vector `S[, j]` to `u[, i]`. After updating `u[, i]`, the transition rates must be recalculated to obtain the time to the next event. However, a state transition might not need all transition rates to be recalculated. The dependency graph `G` is a matrix ($N_{\text{trans}} \times N_{\text{trans}}$) that determines which transition rates that need to be recalculated. A non-zero entry in element `G[k, j]` indicates that transition rate `k` needs to be recalculated if the jth state transition occurs, where $1 \leq \texttt{k} \leq N_{\text{trans}}$. Furthermore, the final step (17) is incorporated using a model specific post time step callback to allow update of the concentration variable $\mathbb{Y}$.

Model-specific data that is passed to the transition-rate functions and the post time-step function are stored in the two slots `ldata` and `gdata` in the `SimInf_model` object. The `ldata` matrix holds local data for each node where `ldata[, i]` is the data vector for node i. Data that is global, i.e., shared between nodes, is stored in the `gdata` vector.

The `events` slot in the `SimInf_model` holds data to process the scheduled events, further described in Section 3.3.

During simulation of one trajectory, the state of the system is written to the two matrices `U` and `V`. This happens at each occasion the simulation time passes a time point in `tspan`, a vector of increasing time points. The first and last element in `tspan` determines the start- and end-point of the simulation. The column `U[, m]` contains the number of individuals in each compartment in every node at `tspan[m]`, where $1 \leq \texttt{m} \leq \texttt{length(tspan)}$. The first N_{comp} rows in `U` contains the compartments of the first node. The next N_{comp} rows contains the compartments of the second node etc. The `V` matrix contains output from continuous state variables. The column `V[, m]` contains the values at `tspan[m]`. The rows are grouped per node and the number of rows per node is determined by the number of continuous state variables in that specific model. It is also possible to configure the simulator to write the state of the system to the sparse matrices `U_sparse` and `V_sparse`, which can save a lot of memory if the model contains many nodes and time-points, but where only a few of the data points are of interest. In order to use this feature, call the `U` and `V` methods (before running a trajectory) with a `data.frame` that specify the nodes, time-points and compartments where the simulator should write the state of the system. The initial state in each node is specified by the two matrices `u0` and `v0` where `u0[, i]` is the initial number of individuals in each compartment at node i and `v0[, i]` is the initial continuous state in node i.

3.3. Specification of scheduled events

The scheduled events are used to modify the discrete state of individuals in a node at a

Slot	Description
`E`	Each row corresponds to one compartment in the model. The non-zero entries in a column indicate which compartments to sample individuals from when processing an event. Which column to use for each event is specified by the `select` vector (see below). `E` is a sparse matrix of class `dgCMatrix`.
`N`	Determines how individuals in enter, internal transfer and external transfer events are shifted to enter another compartment. Each row corresponds to one compartment in the model. The values in a column are added to the current compartment of sampled individuals to specify the destination compartment, for example, a value of `1` in an entry means that sampled individuals in this compartment are moved to the next compartment. Which column to use for each event is specified by the `shift` vector (see below). `N` is an integer matrix.
`event`	Four event types are supported by the current solvers: exit, enter, internal transfer, and external transfer. When assigning the events from a `data.frame`, they can either be coded as a numerical value or a character string: exit; `0` or `'exit'`, enter; `1` or `'enter'`, internal transfer; `2` or `'intTrans'`, and external transfer; `3` or `'extTrans'`. Internally in **SimInf**, the event type is coded as a numerical value.
`time`	Time of when the event occurs i.e., the event is processed when time is reached in the simulation. `time` is an integer vector.
`node`	The node that the event operates on. Also the source node for an external transfer event. `node` is an integer vector, where $1 \leq$ `node[i]` $\leq N_{\text{nodes}}$.
`dest`	The destination node for an external transfer event i.e., individuals are moved from `node` to `dest`, where $1 \leq$ `dest[i]` $\leq N_{\text{nodes}}$. Set `event = 0` for the other event types. `dest` is an integer vector.
`n`	The number of individuals affected by the event. `n` is an integer vector, where `n[i]` ≥ 0.
`proportion`	If `n[i]` equals zero, the number of individuals affected by `event[i]` is calculated by summing the number of individuals in the compartments determined by `select[i]` and sampling from a binomial distribution with `proportion[i]`. `proportion` is a numeric vector, where $0 \leq$ `proportion[i]` ≤ 1.
`select`	To process an `event[i]`, the compartments affected by the event are specified with `select[i]` together with the matrix `E`, where `select[i]` determines which column in `E` to use. The specific individuals affected by the event are sampled from the compartments corresponding to the non-zero entries in the specified column in `E[, select[i]]`, where `select` is an integer vector.
`shift`	Determines how individuals in enter, internal transfer and external transfer events are shifted to enter another compartment. The sampled individuals are shifted according to column `shift[i]` in matrix `N` i.e., `N[, shift[i]]`, where `shift` is an integer vector. See above for a description of `N`.

Table 2: Description of the slots in the S4 class `SimInf_events` that holds data to process scheduled events to modify the discrete state of individuals in a node at a pre-defined time t. Each index, `i`, of the vectors represent one event. N_{nodes} is the number of nodes in the model.

Argument	Description
`v_new`	If a continuous state vector is used by a model, this is the new continuous state vector in the node after the post time step. Exists only in `PTSFun`.
`u`	The compartment state vector in the node.
`v`	The current continuous state vector in the node.
`ldata`	The local data vector for the node.
`gdata`	The global data vector that is common to all nodes.
`node`	The node index. Note the node index is zero-based, i.e., the first node is 0.
`t`	Current time in the simulation.

Table 3: Description of the arguments to the transition rate functions (`TRFun`) and the post time step function (`PTSFun`).

pre-defined time t. There are four different types of events; enter, internal transfer, external transfer and exit. The enter event adds individuals to a node, for example, due to births. The internal transfer event moves individuals between compartments within one node. For example, to simulate vaccination and move individuals to a vaccinated compartment (see example in Section 5.1.2). Or ageing according to birth records in an age-structured model (Widgren *et al.* 2016). The external transfer event moves individuals from compartments in one node to compartments in a destination node. Finally, the exit event removes individuals from a node, for example, due to death. The event types are classified into those that operate on the compartments of a single node $E_1 = \{\text{enter, internal transfer, exit}\}$ and those that operate on the compartments of two nodes $E_2 = \{\text{external transfer}\}$. The parallel algorithm processes these two classes of events differently, see Appendix A for pseudo-code of the core simulation solver. The scheduled events are processed when simulation time reaches the time for any of the events. Events that are scheduled at the same time are processed in the following order: exit, enter, internal transfer and external transfer.

The S4 class `SimInf_events` contains slots with data structures to process events (Table 2). The slots `event`, `time`, `node`, `dest`, `n`, `proportion`, `select` and `shift`, are vectors of equal length. These vectors hold data to process one event: `e`, where $1 <$ `e` $\leq$ `length(event)`. The event type and the time of the event are determined by `event[e]` and `time[e]`, respectively. The compartments that `event[e]` operates on, are specified by `select[e]` together with the slot `E`. Each row $\{1, 2, \ldots, N_{\text{comp}}\}$ in the sparse matrix `E`, represents one compartment in the model. Let `s <- select[e]`, then each non-zero entry in the column `E[, s]` includes that compartment in the `event[e]` operation. The definitions of all of these operations are a bit involved and to quickly get an overview, schematic diagrams illustrating all of them have been prepared, we refer to Figures 10, 11, 12, 13, 14 in Appendix B.

Processing of an enter event

The enter event adds `n[e]` individuals to one or more compartments in `node[e]`, where the possible compartments are specified by a non-zero entry in the row for a compartment in column `E[, s]`. If `n[e]` equals zero, the number of individuals to add is calculated by sampling from a binomial distribution with `proportion[e]` and the total number of individuals in the compartments represented by the non-zero entries in column `E[, s]`. If the column `E[, s]` contains several non-zero entries, the compartment to add an individual is sampled in such a way that the probability is proportional to the weight in `E[, s]`. Before the individuals

are added to the compartments, it is possible to use the `shift[e]` feature (described below) to further control in which compartments they are added. The value of `dest[e]`, described below, is not used when processing an enter event. See Figure 11 in Appendix B for an illustration of a scheduled enter event.

Processing of an internal transfer event

The internal transfer event moves `n[e]` individuals into new compartments within `node[e]`. However, if `n[e]` equals zero, the number of individuals to move is calculated by sampling from a binomial distribution with `proportion[e]` and the total number of individuals in the compartments represented by the non-zero entries in column `E[, s]`. The individuals are then sampled, one by one, without replacement from the compartments specified by `E[, s]` in such a way that the probability that a particular individual is sampled at a given draw is proportional to the weight in `E[, s]`. This sampling follows an hypergeometric distribution when all compartments have the same weight, and a Wallenius' noncentral hypergeometric distribution when the weights are different. (Fog 2008). The next step is to move the sampled individuals to their new compartment using the matrix `N` and `shift[e]`, where `shift[e]` specifies which column in `N` to use. Each row $\{1, 2, \ldots, N_{\text{comp}}\}$ in `N`, represents one compartment in the model and the values determine how to move sampled individuals before adding them to `node[e]` again. Let `q <- shift[e]`, then each non-zero entry in `N[, q]` defines the number of rows to move sampled individuals from that compartment i.e., sampled individuals from compartment `p` are moved to compartment `N[p, q] + p`, where $1 \leq$ `N[p, q] + p` $\leq N_{\text{comp}}$. The value of `dest[e]`, described below, is not used when processing an internal transfer event. See Figure 13 in Appendix B for an illustration of a scheduled internal transfer event.

Processing of an external transfer event

The external transfer event moves individuals from `node[e]` to `dest[e]`. The sampling of individuals from `node[e]` is performed in the same way as for an internal transfer event. The compartments at `node[e]` are updated by subtracting the sampled individuals while adding them to the compartments at `dest[e]`. The sampled individuals are added to the same compartments in `dest[e]` as in `node[e]`, unless `shift[e]` > 0. In that case, the sampled individuals change compartments according to `N` as described in processing an internal transfer event before adding them to `dest[e]`. See Figures 12 and 14 in Appendix B for illustrations of scheduled external transfer events.

Processing of an exit event

The exit event removes individuals from `node[e]`. The sampling of individuals from `node[e]` is performed in the same way as for an internal transfer event. The compartments at `node[e]` are updated by subtracting the sampled individuals. The values of `dest[e]` and `shift[e]` are not used when processing an exit event. See Figure 10 in Appendix B for an illustration of a scheduled exit event.

3.4. Core simulation solvers

The **SimInf** package uses the ability to interface compiled code from R Chambers (2008).

The solvers are implemented in the compiled language C and is called from R using the `.Call()` interface (Chambers 2008). Using C code rather than interpreted R code ensures high performance when running the model. To improve performance further, the numerical solvers use OpenMP to divide work over multiple processors and perform computations in parallel. Two numerical solvers are currently supported. The default solver is a split-step method named `ssm`, that uses direct SSA, but once every unit of time, it also processes scheduled events and calls the post time step function. The other solver implements an "all events method" (Bauer and Engblom 2015) and is named `aem`. Similarly, it also processes scheduled events and calls the post time step function once every unit of time. A core feature of the `aem` solver is that transition events are carried out in channels which access private streams of random numbers, in contrast to the `ssm` solver where one uses only one stream for all events.

Function pointers

The flexibility of the solver is partly achieved by using function pointers (Kernighan and Ritchie 1988). A function pointer is a variable that stores the address of a function that can be used to invoke the function. This provides a simple way to incorporate model specific functionality into the solver. A model must define one transition rate function for each state transition in the model. These functions are called by the solver to calculate the transition rate for each state transition in each node. The output from the transition rate function depends only on the state of the system at the current time. However, the output is unique to a model and data are for that reason passed on to the function for the calculation. Furthermore, a model must define the post time step function. This function is called once for each node each time the simulation of the continuous-time Markov chain reaches the next day (or, more generally, the next unit of time) and after the E_1 and E_2 events have been processed. The main purpose of the post time step function is to allow for a model to update continuous state variables in each node.

The transition rate function is defined by the data type `TRFun` and the post time step function by the data type `PTSFun`. These data types are defined in the header file *src/SimInf.h* and shown below. The arguments `v_new`, `u`, `v`, `ldata`, `gdata`, `node`, and `t` of the functions are described in Table 3.

```
typedef double (*TRFun)(const int *u, const double *v, const double *ldata,
                        const double *gdata, double t);

typedef int (*PTSFun)(double *v_new, const int *u, const double *v,
                      const double *ldata, const double *gdata,
                      int node, double t);
```

Overview of the solvers

Here follows an overview of the steps a solver performs to run a trajectory, see Appendix A for pseudo-code for the `ssm` solver and *src/solvers/ssm/SimInf_solver.c* for the source code. The simulation starts with a call to the `run` method with the model as the first argument. This method will first call the validity method on the model to perform error-checking and then call a model specific C function to initialize the function pointers to the transition rate functions and the post time step function of the model. Subsequently, the simulation solver is called to run one trajectory using the model specific data, the transition rate functions,

and the post time step function. If the `C_code` slot is non-empty, the C code is written to a temporary file when the `run` method is called. The temporary file is compiled using `'R CMD SHLIB'` and the resulting DLL is dynamically loaded. This is further described in Section 5.

The solver simulates the trajectory in parallel if OpenMP is available. The default is to use all available threads. However, the user can specify the number of threads to use with `set_num_threads()`. The solver divides data for the N_{nodes} nodes and the E_1 events over the number threads. All E_1 events that affect node i is processed in the same thread as node i is simulated in. The E_2 events are processed in the main thread.

The solver runs the continuous-time Markov chain for each node i. For every time step τ_i, the count in the compartments at node i is updated according to the state transition that occurred (Section 3.2). The time to the next event is computed, after recalculating affected transition rate functions (Section 3.2). When simulated time reaches the next day in node i the E_1 events are processed for that node (Section 3.3). The E_2 events are processed when all nodes reaches the next day (Section 3.3). Thereafter, the post time step function is called to allow the model to incorporate model specific actions. When simulated time passes the next time in `tspan`, the count of the compartments and the continuous state variables are written to `U` and `V`.

4. Model construction and data analysis: Basic examples

4.1. A first example: The SIR model

Specification of the `SIR` *model without scheduled events*

This section illustrates the specification of the predefined `SIR` model, which contains the three compartments susceptible (`S`), infected (`I`) and recovered (`R`). The transmission route of infection to susceptible individuals is through direct contact between susceptible and infected individuals. The `SIR` model has two state transitions in each node i,

$$\begin{array}{ccc} S_i & \xrightarrow{\beta S_i I_i/(S_i+I_i+R_i)} & I_i, \\ I_i & \xrightarrow{\gamma I_i} & R_i, \end{array} \tag{18}$$

where β is the transmission rate and γ is the recovery rate. To create an `SIR` model object, we need to define `u0`, a `data.frame` with the initial number of individuals in each compartment when the simulation starts. Let us consider a node with 999 susceptible, 1 infected and 0 recoverd individuals. Since there are no between-node interactions in this example, the stochastic process in one node does not affect any other nodes in the model. Consequently, it is straightforward to run many realizations of this model, simply by replicating a node in `u0`, for example, `n = 1000` times.

```
R> n <- 1000
R> u0 <- data.frame(S = rep(999, n), I = rep(1, n), R = rep(0, n))
```

Next, we define the time period over which we want to simulate the disease spread. This is a vector of integers in units of time or a vector of dates. You specify those time points that

you wish the model to return results for. The model itself does not run in discrete time steps, but in continuous time, so this does not affect the internal calculations of disease transitions through time. In this example we will assume that the unit of time is one day and simulate over 180 days returning results every 7th day.

```
R> tspan <- seq(from = 1, to = 180, by = 7)
```

We are now ready to create an `SIR` model and then use the `run()` routine to simulate data from it. For reproducibility, we first call the `set.seed()` function and also specify the number of threads to use for the simulation. To use all available threads, you only have to remove the `set_num_threads()` call.

```
R> model <- SIR(u0 = u0, tspan = tspan, beta = 0.16, gamma = 0.077)
R> set.seed(123)
R> set_num_threads(1)
R> result <- run(model = model)
```

The return value from `run()` is a `SimInf_model` object with a single stochastic solution trajectory attached to it. The `show()` method of the `SimInf_model` class prints some basic information about the model, such as the global data parameters and the extremes, the mean and the quartiles of the count in each compartment across all nodes.

```
R> result

Model: SIR
Number of nodes: 1000
Number of transitions: 2
Number of scheduled events: 0

Local data
----------
 Parameter Value
 beta      0.160
 gamma     0.077

Compartments
------------
    Min. 1st Qu. Median  Mean 3rd Qu.  Max.
 S 108.0   368.0  993.0 755.4   999.0 999.0
 I   0.0     0.0    1.0  30.4    38.0 235.0
 R   0.0     1.0    5.0 214.2   484.0 891.0
```

The `plot()` method of the `SimInf_model` class can be used to visualize the simulated trajectory. The default plot will display the median count in each compartment across nodes as a colored line together with the inter-quartile range using the same color, but with transparency. To display the outcome for individual nodes, specify the subset of nodes to plot

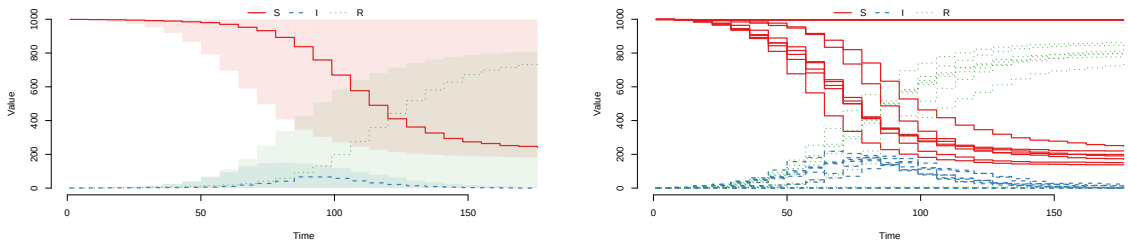

Figure 3: Output from a stochastic `SIR` model in 1000 nodes starting with 999 susceptible, 1 infected and 0 recovered individuals in each node ($\beta = 0.16$, $\gamma = 0.077$). There are no between-node interactions. Left: The default plot shows the median and inter-quartile range of the count in each compartment through time across all nodes. Right: Realizations from a subset of 10 nodes.

using the `index` parameter and set `range = FALSE`. In this example, an outbreak is likely to occur in an infected node, but sometimes the infectious disease will become extinct before it causes an epidemic, as shown in Figure 3.

```
R> plot(result)
R> plot(result, index = 1:10, range = FALSE)
```

Most modeling and simulation studies require custom data analysis once the simulation data has been generated. To support this, **SimInf** provides the `trajectory()` method to obtain a `data.frame` with the number of individuals in each compartment at the time points specified in `tspan`. Below is an excerpt of the simulated data from the first node that clearly shows there was an outbreak there. To extract all data from every node, you only have to remove the index argument. Consult the help page for other `trajectory()` parameter options.

```
R> head(trajectory(model = result, index = 1))
```

```
  node time   S  I  R
1    1    1 999  1  0
2    1    8 998  1  1
3    1   15 991  8  1
4    1   22 973 21  6
5    1   29 935 42 23
6    1   36 886 61 53
```

Specification of scheduled events in the `SIR` *model*

In this example, we will continue to work with the predefined `SIR` model to illustrate how demographic data can be incorporated into a simulation. In order for the numerical solver to process a scheduled event, the compartments that are involved in the event must be specified. This is done by each event specifies one column in the select matrix `E` using the `select`

attribute of the event. The non-zero entries in the selected column in `E` specify the involved compartments. For the predefined `SIR` model, `E` is defined as

$$\mathbf{E} = \begin{array}{c} \\ S \\ I \\ R \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

This means that we can specify a scheduled event to operate on a single compartment (S, I or R) as well as an event that involves all three compartments. When several compartments are involved in an event, the individuals affected by the event will be sampled without replacement from the specified compartments. The numerical solver performs an extensive error checking of the event before it is processed. And an error will be raised if the event is invalid, for example, if the event tries to move more individuals than exists in the specified compartments.

Consider we have 4 scheduled events to include in a simulation. Below is a `data.frame`, that contains the events.

```
R> events

     event time node dest n proportion select shift
1    enter    2    3    0 5        0.0      1     0
2 extTrans    3    1    3 7        0.0      4     0
3     exit    4    2    0 0        0.2      4     0
4    enter    4    1    0 1        0.0      2     0
```

Interpret it as follows:

1. In time step 2 we add 5 susceptible individuals to node 3.
2. In time step 3 we sample 7 individuals without replacement among the S, I and R compartments in node 1 and move them to the corresponding compartments in node 3.
3. In time step 4 we sample 20% of all individuals without replacement among the S, I and R compartments in node 2 and remove them from node 2.
4. In time step 4 we add 1 infected individual to node 1.

Now, let us illustrate with a small example, consisting only of five nodes, how scheduled events can alter the composition within nodes during simulation. Let us start with empty nodes and then create some enter events to add susceptible individuals to each node during the first ten time-steps.

```
R> u0 <- data.frame(S = rep(0, 5), I = rep(0, 5), R = rep(0, 5))
R> add <- data.frame(event = "enter", time = rep(1:10, each = 5),
+    node = 1:5, dest = 0, n = 1:5, proportion = 0, select = 1, shift = 0)
```

We then create one enter event to introduce an infected individual to the 5th node at $t = 25$.

```
R> infect <- data.frame(event = "enter", time = 25, node = 5,
+    dest = 0, n = 1, proportion = 0, select = 2, shift = 0)
```

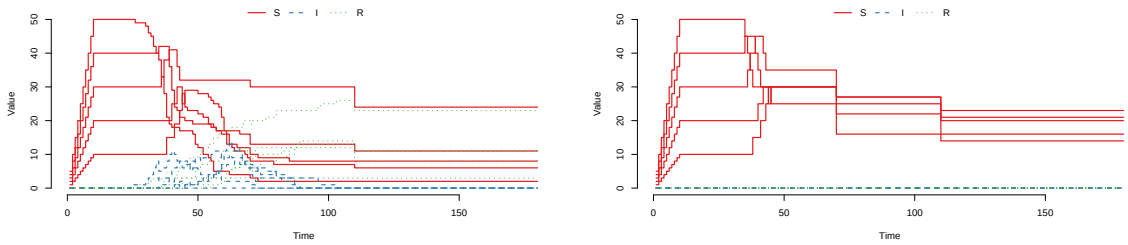

Figure 4: An `SIR` model in five nodes with scheduled events. Left: One realization where susceptible individuals were added during the first ten time-steps. Then, one infected individual was introduced at $t = 25$. Further, some movements occurred between $t = 35$ and $t = 45$. Finally, at $t = 70$ and $t = 110$, twenty percent of the individuals were removed. Right: For comparison, the deterministic dynamics when not introducing an infected individual.

Additionally, we create some external transfer events to form interactions among the nodes with movements between $t = 35$ and $t = 45$. Each shipment contains $n = 5$ individuals.

```
R> move <- data.frame(event = "extTrans", time = 35:45, node = c(5, 5, 5,
+    5, 4, 4, 4, 3, 3, 2, 1), dest = c(4, 3, 3, 1, 3, 2, 1, 2, 1, 1, 2),
+    n = 5, proportion = 0, select = 4, shift = 0)
```

Finally, we create exit events to remove 20% of the individuals from each node at $t = 70$ and $t = 110$.

```
R> remove <- data.frame(event = "exit", time = c(70, 110),
+    node = rep(1:5, each = 2), dest = 0, n = 0, proportion = 0.2,
+    select = 4, shift = 0)
```

Figure 4 shows one realization of a model incorporating the events. Here we observe two transmission processes on different scales. First, the stochastic transmission process within each node. Secondly, the between-node transmission due to movements. Also stochastic, because of the sampling process that select susceptible, infected or recovered individuals to move between the nodes.

```
R> events <- rbind(add, infect, move, remove)
R> model <- SIR(u0 = u0, tspan = 1:180, events = events, beta = 0.16,
+    gamma = 0.077)
R> set.seed(3)
R> set_num_threads(1)
R> result <- run(model)
R> plot(result, index = 1:5, range = FALSE)
```

We can use `replicate` (or similar) to generate many realizations from a `SimInf_model` object, together with some custom analysis of each trajectory. Here, we find that infection spread from the 5th node in about half of `n = 1000` trajectories.

```
R> set.seed(123)
R> set_num_threads(1)
R> mean(replicate(n = 1000, {
+    nI <- trajectory(run(model = model), index = 1:4)$I
+    sum(nI) > 0
+  }))

[1] 0.477
```

C code for the `SIR` *model*

The C code for the `SIR` model is defined in the source file *src/models/SIR.c*. This file contains the `SIR_run` function to initialize the core solver (Listing 1 in Appendix C), the transition rate functions (Listing 2 in Appendix C) and the post time step function (Listing 3 in Appendix C).

4.2. A second example: The `SISe_sp` model

Here we will illustrate the use of local data (`ldata`) and continuous state variables (`V`) to formulate a more complex model with variables obeying ODEs. Moreover, we will introduce the `prevalence()` method, another important function for post-processing trajectories. Let us consider VTEC in cattle for this example. Briefly, a VTEC infection in cattle can be formulated as a susceptible-infected-susceptible (SIS) compartment model. However, previous modeling has shown that it is important to consider within and between-farm transmission via the environment (Ayscue *et al.* 2009; Zhang, Chase-Topping, McKendrick, Savill, and Woolhouse 2010), ambient temperature (Gautam, Bani-Yaghoub, Neill, Döpfer, Kaspar, and Ivanek 2011), herd size and between-farm spread from livestock movements (Zhang *et al.* 2010). Therefore, let us use the predefined `SISe_sp` model. It contains an environmental compartment to model shedding of a pathogen to the environment. Moreover, it also includes a spatial coupling of the environmental contamination among proximal nodes to capture between-node spread unrelated to moving infected individuals. Consequently, the model has two state transitions,

$$\begin{aligned} S_i &\xrightarrow{\upsilon\varphi_i} I_i, \\ I_i &\xrightarrow{\gamma} S_i, \end{aligned} \tag{19}$$

where the transition rate per unit of time from susceptible to infected is proportional to the concentration of the environmental contamination $\varphi_i(t)$ in node i. Moreover, the transition rate from infected to susceptible is the recovery rate γ, measured per individual and per unit of time. Finally, the environmental infectious pressure is evolved by

$$\frac{d\varphi_i(t)}{dt} = \frac{\alpha I_i(t)}{N_i(t)} + \sum_k \frac{\varphi_k(t)N_k(t) - \varphi_i(t)N_i(t)}{N_i(t)} \cdot \frac{D}{d_{ik}} - \beta(t)\varphi_i(t), \tag{20}$$

where α is the average shedding rate of the pathogen to the environment per infected individual and $N_i = S_i + I_i$ the size of node i. Next comes the spatial coupling among proximal nodes, where D is the rate of the local spread and d_{ik} the distance between holdings i and k. The seasonal decay and removal of the pathogen is captured by $\beta(t)$. The environmental

infectious pressure $\varphi_i(t)$ in each node is evolved in the post-time-step function by the Euler forward method (see the file 'src/models/SISe_sp.c' for the C code). The value of $\varphi_i(t)$ is saved to the V matrix at the time-points specified by tspan.

Let us use a synthetic dataset of 1600 farms located within a 50 square kilometer region. Load the data with the following commands

```
R> data("nodes", package = "SimInf")
R> u0 <- u0_SIS()
R> events <- events_SIS()
```

where the location of each farm is in nodes, u0 defines the initial cattle population and events contains four years (1460 days) of scheduled events data (births, deaths and movements). Moreover, let us define proximal neighbors as neighbors within 2500m and use the utility function distance_matrix() to estimate the distance between farms within that cutoff.

```
R> d_ik <- distance_matrix(x = nodes$x, y = nodes$y, cutoff = 2500)
```

Let us assume that 10% of the farms have 5% infected cattle at the beginning of the simulation.

```
R> set.seed(123)
R> i <- sample(x = 1:1600, size = 160)
R> u0$I[i] <- as.integer(u0$S[i] * 0.05)
R> u0$S[i] <- u0$S[i] - u0$I[i]
```

The SISe_sp model contains parameters at a global and local scale. Here, the parameter values were chosen such that the proportion of infected nodes in a trajectory is about 10% and displays a seasonal pattern. The global parameters are: the spatial coupling = 0.2 (D in Equation (20)), the shedding rate alpha = 1, the recovery rate gamma = 0.1 and the indirect transmission rate upsilon = 0.012. Moreover, the global parameter $\beta(t)$ captures decay of the pathogen in four seasons: beta_t1 = 0.1, beta_t2 = 0.12, beta_t3 = 0.12 and beta_t4 = 0.1. However, the duration of each season is local to a node and is specified as the day of the year each season ends. Here, for simplicity, we let end_t1 = 91, end_t2 = 182, end_t3 = 273 and end_t4 = 365 in all nodes. Furthermore, the distances between nodes are local data extracted from distance = d_ik. Finally, we let phi = 0 at the beginning of the simulation (becomes v0 in the model object).

```
R> model <- SISe_sp(u0 = u0, tspan = 1:1460, events = events, phi = 0,
+    upsilon = 0.012, gamma = 0.1, alpha = 1, beta_t1 = 0.10,
+    beta_t2 = 0.12, beta_t3 = 0.12, beta_t4 = 0.10, end_t1 = 91,
+    end_t2 = 182, end_t3 = 273, end_t4 = 365, distance = d_ik,
+    coupling = 0.2)
```

Let us use the prevalence() method to explore the proportion of infected nodes through time. It takes a model object and a formula specification, where the left hand side of the formula specifies the compartments representing cases i.e., have an attribute or a disease. The right hand side of the formula specifies the compartments at risk. Here, we are interested in

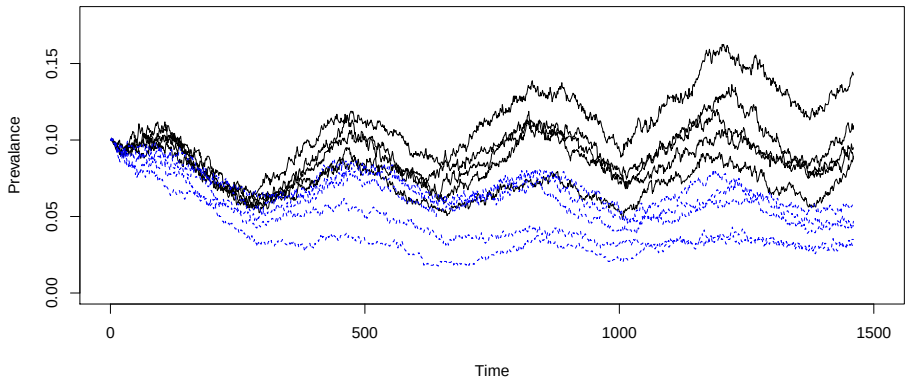

Figure 5: Exploring options for reducing disease spread. Reference trajectories showing the proportion infected nodes (black solid lines). The proportion infected nodes after reducing the spatial coupling with 50% (blue dotted lines).

the proportion of nodes with at least one infected individual, therefore, we let `formula = I ~ S + I` and specify `level = 2`. Consult the help page for other `prevalence()` parameter options.

```
R> plot(NULL, xlim = c(0, 1500), ylim = c(0, 0.18), ylab = "Prevalance",
+    xlab = "Time")
R> set.seed(123)
R> set_num_threads(1)
R> replicate(5, {
+    result <- run(model = model)
+    p <- prevalence(model = result, formula = I ~ S + I, level = 2)
+    lines(p)
+  })
```

Assume there exists some sort of treatment by which the `coupling` can be reduced by 50%. Is that sufficient for controlling the disease? We can use the function `gdata()` to change the global `coupling` parameter and then run some more trajectories.

```
R> gdata(model, "coupling") <- 0.1
R> replicate(5, {
+    result <- run(model = model)
+    p <- prevalence(model = result, formula = I ~ S + I, level = 2)
+    lines(p, col = "blue", lty = 2)
+  })
```

The results in Figure 5 indicate that reducing the spatial coupling D with 50% is not sufficient to eradicate the infection from this synthetic cattle population. Nevertheless this short example serves as a template for using the **SimInf** computational framework when implementing large scale data-driven disease spread models and exploring options for control.

5. Extending SimInf: New models

One of the design goals of **SimInf** was to make it extendable. The current design supports two ways to extend **SimInf** with new models, and this section describes the relevant steps to implement a new model. Since extending **SimInf** requires that C code can be compiled, you will first need to install a compiler. To read more about interfacing compiled code from R and creating R add-on packages, the *Writing R extensions* (https://cran.r-project.org/doc/manuals/r-release/R-exts.html) manual is the official guide and describes the process in detail. Another useful resource is the *R packages* book by Wickham (2015) (https://r-pkgs.org/).

5.1. Using the model parser to define a new model

The simplest way to define a new model for **SimInf** is to use the model parser method `mparse`. It takes a character vector of transitions in the form of `"X -> propensity -> Y"` and generates the C and R code for the model. The left hand side of the first '`->`'-sign is the initial state, the right hand side of the last '`->`'-sign is the final state, and the propensity is written between the '`->`'-signs. The special symbol '`@`' is reserved for the empty set $\emptyset$. We suggest to first draw a schematic representation of the model that includes all compartments and arrows for all state transitions.

Introductory examples of using `mparse`

In a first example we will consider the SIR model in a closed population i.e., no births or deaths. If we let `b` denote the transmission rate and `g` the recovery rate, the model can be described as,

```
R> transitions <- c("S -> b*S*I/(S+I+R) -> I", "I -> g*I -> R")
R> compartments <- c("S", "I", "R")
```

We can now use the `transitions` and `compartments` variables, together with the constants `b` and `g` to build an object of class `'SimInf_model'` via a call to `mparse`. It also needs to be initialized with the initial condition `u0` and `tspan`.

```
R> n <- 1000
R> u0 <- data.frame(S = rep(99, n), I = rep(5, n), R = rep(0, n))
R> model <- mparse(transitions = transitions, compartments = compartments,
+    gdata = c(b = 0.16, g = 0.077), u0 = u0, tspan = 1:180)
```

As in earlier examples, the `model` object can now be used to simulate data and plot the results. Internally, the C code that was generated by `mparse` is written to a temporary file when the `run` method is called. The name of the temporary file is computed from the MD5 hash of the C code, using the **digest** package. If the temporary file is compiled successfully, the resulting DLL is dynamically loaded and used to run one trajectory of the model. The hash of the C code is also used to determine if a model has already been compiled and loaded, and thus the compilation step can be skipped before running a trajectory.

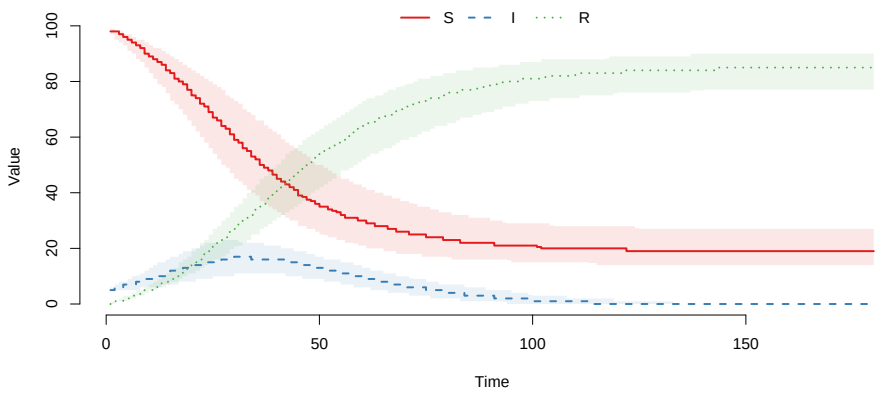

Figure 6: Showing the median and inter-quartile range from 1000 realizations of an `mparse` `SIR` model ($\beta = 0.16$, $\gamma = 0.077$), starting with 99 susceptible, 5 infected, and 0 recovered individuals.

```
R> set.seed(123)
R> set_num_threads(1)
R> result <- run(model = model)
R> plot(result)
```

The flexibility of the `mparse` approach allows for quick prototyping of new models or features. Let us elaborate on the previous example and explore the incidence cases per day. This can easily be done by adding a new compartment `'Icum'` whose sole purpose is to keep track of how many individuals who become infected over time. The right hand side `'I + Icum'` of the transition `'S -> b*S*I/(S+I+R) -> I + Icum'`, means that both `'I'` and `'Icum'` are incremented by one each time the transition happens.

```
R> transitions <- c("S -> b*S*I/(S+I+R) -> I + Icum", "I -> g*I -> R")
R> compartments <- c("S", "I", "Icum", "R")
```

Since there are no between-node movements in this example, the stochastic process in one node does not affect any other nodes in the model. It is therefore straightforward to run many realizations of this model, simply by replicating a node in the initial condition `u0`, for example, $n = 1000$ times.

```
R> n <- 1000
R> u0 <- data.frame(S = rep(99, n), I = rep(1, n), Icum = rep(0, n),
+    R = rep(0, n))
R> model <- mparse(transitions = transitions, compartments = compartments,
+    gdata = c(b = 0.16, g = 0.077), u0 = u0, tspan = 1:150)
R> set.seed(123)
R> set_num_threads(1)
R> result <- run(model = model)
```

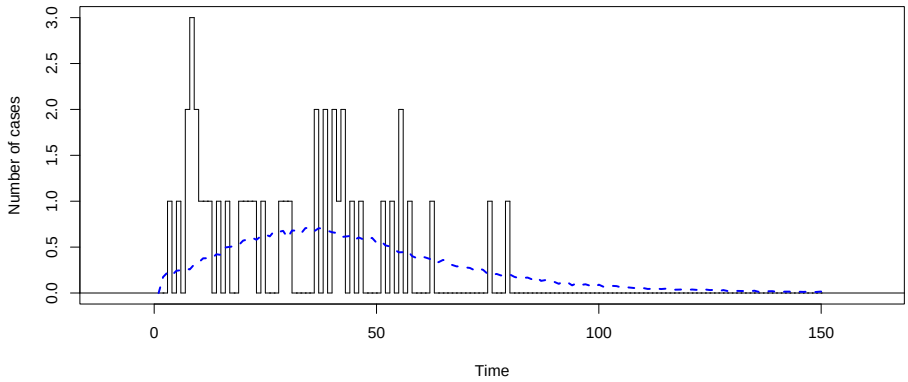

Figure 7: (black solid line) One realization of an epidemic curve displaying the number of incident cases per day in a node when simulating 150 days of an `mparse SIR` model ($\beta = 0.16, \gamma = 0.077$), starting with 99 susceptible, 1 infected and 0 recovered individuals. (blue dashed line) Average number of incident cases per day from 1000 realizations of the model.

Let us post-process the simulated trajectory to compare the incidence cases in the first node with the average incidence cases among all realizations by extracting the trajectory data and calculate successive differences of `'Icum'` at each time-point.

```
R> traj <- trajectory(model = result, compartments = "Icum")
R> cases <- stepfun(result@tspan[-1], diff(c(0, traj$Icum[traj$node == 1])))
R> avg_cases <- c(0, diff(by(traj, traj$time, function(x) sum(x$Icum))) / n)
```

Finally, plot the result as an epidemic curve (Figure 7). In this example, the number of incident cases in the first node exceeds what is expected on average.

```
R> plot(cases, main = "", xlab = "Time", ylab = "Number of cases",
+    do.points = FALSE)
R> lines(avg_cases, col = "blue", lwd = 2, lty = 2)
```

Incorporate scheduled events in an `mparse` *model*

To illustrate how models generated using `mparse` can incorporate scheduled events, consider an epidemic in a population consisting of 1600 nodes, for example, cattle herds, that are connected to each other by livestock movements. Assume an outbreak is detected on day twenty-one after introduction of an infection in one node and that we wish to explore how vaccination could limit the outbreak, if resources for vaccination can handle 50 herds per day and 80% of the animals in each herd. Let us add a new compartment V to the model to represent vaccinated individuals, so that the model now contains the $\{S, I, I_{cum}, R, V\}$ compartments. As before, let `b` denote the transmission rate and `g` the recovery rate.

```
R> transitions <- c("S -> b*S*I/(S+I+R+V) -> I + Icum", "I -> g*I -> R")
R> compartments <- c("S", "I", "Icum", "R", "V")
```

Load the example data for an `SIR` model in a population of 1600 nodes (cattle herds) with its associated scheduled events: births, deaths, and livestock movements. Moreover, let $I_{cum} = 0$ and $R = 0$.

```
R> u0 <- u0_SIR()
R> u0$Icum <- 0
R> u0$V <- 0
R> events <- events_SIR()
```

Now generate vaccination events i.e., internal transfer events. Use `select = 3` and `shift = 1` to move animals from the susceptible, infectious and recovered compartments to the vaccinated compartment, see the definitions of E and N below. Let us start the vaccinations in nodes 1–50 on day twenty-one, and continue until all herds are vaccinated on day fifty-two. Moreover, use `proportion = 0.8` to vaccinate 80% of the animals in each herd. We assume, for the sake of simplicity, that vaccinated individuals become immune and non-infectious immediately.

```
R> vaccination <- data.frame(event = "intTrans", time = rep(21:52,
+    each = 50), node = 1:1600, dest = 0, n = 0, proportion = 0.8,
+    select = 3, shift = 1)
```

To simulate from this model, we have to define the select matrix E to handle which compartments to sample from when processing a scheduled event. Let the first column in E handle enter events (births); add newborn animals to the susceptible compartment S. The second column is for exit events (deaths) and external transfer events (livestock movements); sample animals from the S, I, R and V compartments. Finally, the third column is for internal transfer events (vaccination); sample individuals from the S, I and R compartments. We must also define the shift matrix N to process internal transfer events (vaccination); move sampled animals from the S compartment four steps forward to the V compartment. Similarly, move sampled animals from the I compartment three steps forward to the V compartment, and finally, move sampled individuals from the R compartment one step forward.

$$\mathbf{E} = \begin{array}{c} \\ S \\ I \\ I_{cum} \\ R \\ V \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array} \qquad \mathbf{N} = \begin{array}{c} \\ S \\ I \\ I_{cum} \\ R \\ V \end{array} \begin{array}{c} 1 \\ \begin{pmatrix} 4 \\ 3 \\ 0 \\ 1 \\ 0 \end{pmatrix} \end{array}$$

```
R> E <- matrix(c(1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0), nrow = 5,
+    ncol = 3, dimnames = list(c("S", "I", "Icum", "R", "V"),
+    c("1", "2", "3")))
R> N <- matrix(c(4, 3, 0, 1, 0), nrow = 5, ncol = 1,
+    dimnames = list(c("S", "I", "Icum", "R", "V"), "1"))
```

Additionally, we have to redefine the `select` column in the events, since the data is from the predefined `SIR` model with another E matrix. Let us change `select` for movements.

```
R> events$select[events$select == 4] <- 2
```

Now let us create an `epicurve` function to estimate the average number of new cases per day from `n = 1000` realizations in a `for`-loop and simulate one trajectory at a time. To clear infection that was introduced in the previous trajectory, animals are first moved to the susceptible compartment. Then, one infected individual is introduced into a randomly sampled node from the population. Note that we use the 'L' suffix to create an integer value rather than a numeric value. Run the model and accumulate `Icum`. For efficiency, use `format = "matrix"`, the internal matrix format, to extract `Icum` in every node at each time-point in `tspan`.

```
R> epicurve <- function(model, n = 1000) {
+    Icum <- numeric(length(model@tspan))
+    for (i in seq_len(n)) {
+      model@u0["S", ] <- model@u0["S", ] + model@u0["I", ]
+      model@u0["I", ] <- 0L
+      j <- sample(seq_len(n_nodes(model)), 1)
+      model@u0["I", j] <- 1L
+      model@u0["S", j] <- model@u0["S", j] - 1L
+      result <- run(model = model)
+      traj <- trajectory(model = result, compartments = "Icum",
+        format = "matrix")
+      Icum <- Icum + colSums(traj)
+    }
+    stepfun(model@tspan[-1], diff(c(0, Icum / n)))
+  }
```

Generate an epicurve with the average number of cases per day for the first three hundred days of the epidemic without vaccination.

```
R> model_no_vac <- mparse(transitions = transitions,
+    compartments = compartments, gdata = c(b = 0.16, g = 0.077),
+    u0 = u0, tspan = 1:300, events = events, E = E, N = N)
R> cases_no_vac <- epicurve(model_no_vac)
```

Similarly, generate an epicurve after incorporating the `vaccination` events.

```
R> model_vac <- mparse(transitions = transitions,
+    compartments = compartments, gdata = c(b = 0.16, g = 0.077),
+    u0 = u0, tspan = 1:300, events = rbind(events, vaccination),
+    E = E, N = N)
R> cases_vac <- epicurve(model_vac)
```

As expected, the number of cases decrease rapidly after vaccination, while the outbreak is ongoing for a longer time in the unvaccinated population (Figure 8).

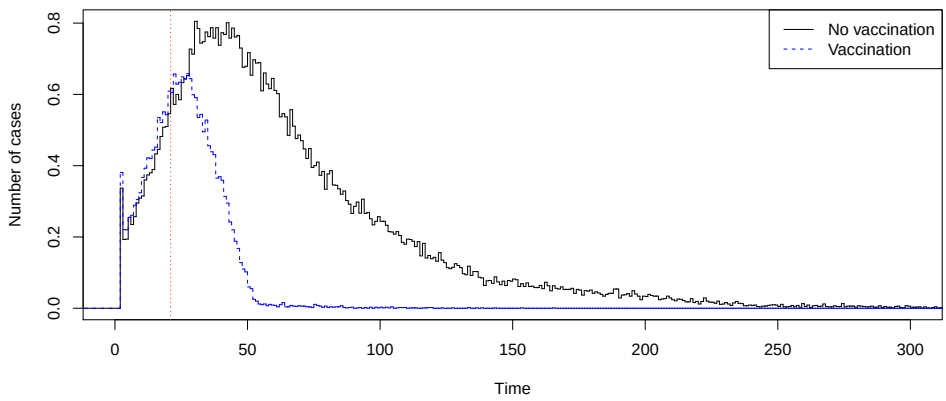

Figure 8: Comparison between the number of cases per day of an outbreak in an unvaccinated population of cattle herds (black solid line) and after vaccination of animals (blue dashed line). The vertical line (dotted) indicates when vaccination was initiated.

```
R> plot(cases_no_vac, main = "", xlim = c(0, 300), xlab = "Time",
+    ylab = "Number of cases", do.points = FALSE)
R> lines(cases_vac, col = "blue", do.points = FALSE, lty = 2)
R> abline(v = 21, col = "red", lty = 3)
R> legend("topright", c("No vaccination", "Vaccination"),
+    col = c("black", "blue"), lty = 1:2)
```

5.2. Use the SimInf framework from another package

Another possibility is to extend **SimInf** by creating an R add-on package that uses **SimInf** by linking to its core solver native routine. To facilitate this, the **SimInf** package includes the `package_skeleton` method to automate some of the setup for a new source package. It creates directories, saves R and C code files to appropriate places, and creates skeleton help files.

Even if **SimInf** was designed to study the dynamics of infectious diseases, it is not limited to that use case but can be used to study the dynamics of other systems. Consider we wish to create a new add-on package **PredatorPrey** based on the Rosenzweig-MacArthur predator-prey model demonstrated in the **GillespieSSA** package (Rosenzweig and MacArthur 1963; Pineda-Krch 2008). The model has a density-dependent growth in the prey and and a nonlinear Type-2 functional response in the predator (Rosenzweig and MacArthur 1963). Let R and F denote the number of prey and predators, respectively. The model consists of five transitions (Equation (21)): i) prey birth, ii) prey death due to non-predatory events, iii)

prey death due to predation, iv) predator birth, and v) predator death

$$\left.\begin{array}{ccc} \emptyset & \xrightarrow{b_R \cdot R} & R \\ R & \xrightarrow{(d_R+(b_R-d_R)\cdot R/K)\cdot R} & \emptyset \\ R & \xrightarrow{\alpha/(1+w\cdot R)\cdot R\cdot F} & \emptyset \\ \emptyset & \xrightarrow{b_F\cdot\alpha/(1+w\cdot R)\cdot R\cdot F} & F \\ F & \xrightarrow{d_F\cdot F} & \emptyset \end{array}\right\}, \tag{21}$$

where b_R, d_R, b_F, and d_F are the per capita birth and death rate of the prey and predator, respectively. Moreover, K is the carrying capacity of the prey, α is the predation efficiency, and w is the degree of predator saturation (Pineda-Krch 2008). Using parameter values from Pineda-Krch (2008), we define the model as

```
R> transitions <- c("@ -> bR*R -> R", "R -> (dR+(bR-dR)*R/K)*R -> @",
+    "R -> alpha/(1+w*R)*R*F -> @", "@ -> bF*alpha/(1+w*R)*R*F -> F",
+    "F -> dF*F -> @")
R> compartments <- c("R", "F")
R> parameters <- c(bR = 2, bF = 2, dR = 1, K = 1000, alpha = 0.007,
+    w = 0.0035, dF = 2)
```

Assume the initial population consists of $R = 1000$ prey and $F = 100$ predators and we are interested in simulating $n = 1000$ replicates over 100 days. Since there are no between-node movements in this example, we can generate replicates simply by starting with n identical nodes.

```
R> n <- 1000
R> u0 <- data.frame(R = rep(1000, n), F = rep(100, n))
R> model <- mparse(transitions = transitions, compartments = compartments,
+    gdata = parameters, u0 = u0, tspan = 1:100)
```

Now instead of running the model to generate data, let us use it to create an R add-on package.

```
R> path <- tempdir()
R> package_skeleton(model = model, name = "PredatorPrey", path = path)
```

Where the first argument is the `SimInf_model` object generated by the `mparse` method, the second argument is the name of the package to create a skeleton for and the third argument is the path to the new package. Note that a temporary directory is used here for illustration of the functionality. We refer to the **SimInf** documentation for other arguments that can be supplied to the `package_skeleton` method. The created R file (*R/models.R*) defines the S4 class `PredatorPrey` that contains the `SimInf_model` and a generating function to create a new object of the `PredatorPrey` model. The generating function is a template that might need to be extended to meet the specific requirements for the model.

The C file (*src/models.c*) defines one function for each state transition, the post time step function and the model specific run function. The file is automatically compiled when installing

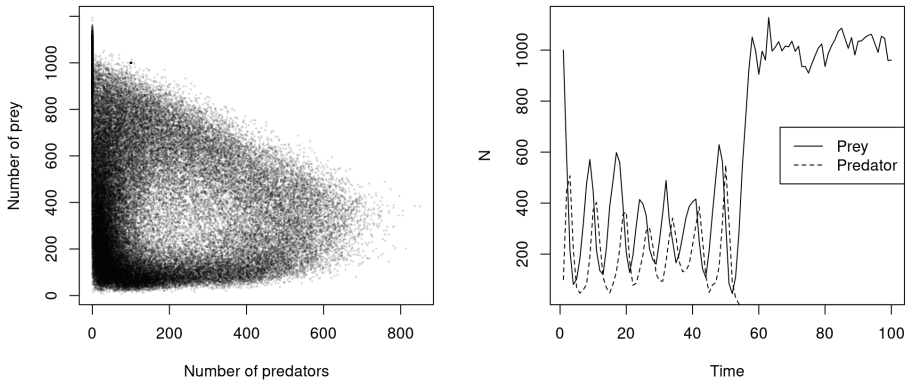

Figure 9: Left: Phase plane trajectories from 1000 realizations of the Rosenzweig-MacArthur predator-prey model. Right: One realization of the Rosenzweig-MacArthur predator-prey model, where the predators go extinct and then the prey population fluctuates around a plateu of 1000 individuals.

the package. The header file `"SimInf.h"` contains the declarations for these functions and must be included. The `SimInf_model_run` function is the interface from R to the core solver in C and list all function pointers to the transition rate functions in a vector in the order the state transitions appear in the dependency graph `G`, see Listings 1 and 2 in Appendix C for an example from the predefined `SIR` model in **SimInf** and the use of the address of operator `'&'` to obtain the address of a function. The `SimInf_model_run` function must return the result from the call to the core solver with `SimInf_run`. The arguments to `SimInf_run` are the arguments passed to the `SimInf_model_run` function plus the vector of function pointers to the transition rate functions and the function pointer to the post time step function. The add-on **PredatorPrey** source package can now be built and installed with the following commands.

```
R> pkg <- file.path(path, "PredatorPrey")
R> install.packages(pkg, repos = NULL, type = "source")
```

Here we let `repos = NULL` to install from local files and use `type = "source"` to compile the files. If the installation was successful, the newly installed package **PredatorPrey** can be loaded in R with the following command.

```
R> library("PredatorPrey")
```

Now create a model and run it to generate data.

```
R> model <- PredatorPrey(u0 = u0, tspan = 1:100, gdata = parameters)
R> set.seed(123)
R> set_num_threads(1)
R> result <- run(model)
```

Because a `PredatorPrey` object contains the `SimInf_model` class, it can make use of all utility functions provided in the **SimInf** package, for example, `show()`.

```
R> result

Model: PredatorPrey
Number of nodes: 1000
Number of transitions: 5
Number of scheduled events: 0

Global data
-----------
 Parameter Value
 bR        2.0e+00
 bF        2.0e+00
 dR        1.0e+00
 K         1.0e+03
 alpha     7.0e-03
 w         3.5e-03
 dF        2.0e+00

Compartments
------------
   Min. 1st Qu. Median Mean 3rd Qu. Max.
 R    8     236    607  598     975 1195
 F    0       0     35  112     161  851
```

Or the `trajectory()` method, for example, to plot the phase plane from 1000 realizations or to illustrate stochastic extinction of the predators in the fourth node (Figure 9).

```
R> opar <- par(mfrow = c(1, 2))
R> plot(R ~ F, trajectory(model = result), cex = 0.3, pch = 20,
+    xlab = "Number of predators", ylab = "Number of prey",
+    col = rgb(0, 0, 0, alpha = 0.1))
R> plot(R ~ time, trajectory(model = result, index = 4), type = "l",
+    xlab = "Time", ylab = "N")
R> lines(F ~ time, trajectory(model = result, index = 4), type = "l", lty = 2)
R> legend("right", c("Prey", "Predator"), lty = 1:2)
R> par(opar)
```

This example illustrated how **SimInf** supports usage of the numerical solvers from other R packages via the `LinkingTo` feature in R.

6. Benchmark

A comprehensive analysis of the performance of the numerical solver in **SimInf** is presented by Bauer *et al.* (2016). Here, we provide a small benchmark of the run-time of an SIR

model using three R packages on CRAN: **SimInf** version 6.1.0, **adaptivetau** version 2.2-3 and **GillespieSSA** version 0.6.1. The measurements were obtained on a ThinkPad T460p, Intel core i7-6700HQ quad-core at 2.6GHz , 32GB 2133MHz RAM, running Fedora 30 and using R version 3.6.1. Ten replicates were performed and average run-time was estimated to generate 1,000 realizations of an SIR model with parameters $\beta = 0.16$ and $\gamma = 0.077$ and initial conditions $S = 1000$, $I = 10$ and $R = 0$. As shown in Table 4, the implementation in **SimInf** appears to run faster than **adaptivtau**. This difference probably depends on **adaptivetau** using a hybrid R/C++ implementation with R code for the transition rate functions while **SimInf** uses C code. To reduce run-time further, **SimInf** has built-in support to perform computations in parallel. As expected, **GillespieSSA** has the longest run-time since it has an implementation in pure R.

R package	Method	Threads	Time [ms]
SimInf	Direct SSA	4	38
SimInf	Direct SSA	2	69
SimInf	Direct SSA	1	126
adaptivetau	Tau-leaping	1	2730
adaptivetau	Direct SSA	1	8894
GillespieSSA	Tau-leaping	1	18465
GillespieSSA	Direct SSA	1	53009

Table 4: Comparison of the average run-time for generating 1000 realizations of an SIR model.

7. Conclusion

In this paper we have introduced the R package **SimInf** which supports data-driven simulations of disease transmission over spatio-temporal networks. The package offers a very efficient and highly flexible tool to incorporate real data in simulations at realistic scales.

We hope that our package will facilitate incorporating available data, for example, livestock data, in network epidemic models to better understand disease transmission in a temporal network and improve design of intervention strategies for endemic and emerging threats. Future efforts will be concentrated on a software development driven predominantly by actual use cases.

8. Acknowledgments

This work was financially supported by the Swedish Research Council within the UPMARC Linnaeus centre of Excellence (P. Bauer, R. Eriksson, S. Engblom), the Swedish Research Council Formas (S. Engblom, S. Widgren), the Swedish Board of Agriculture (S. Widgren), and by the Swedish strategic research program eSSENCE (S. Widgren). The authors also thank two anonymous reviewers for their helpful suggestions and comments, which greatly improved the quality of both the software package and this manuscript.

References

Andersson RM, May RM (1991). *Infectious Diseases of Humans*. Oxford Science Publications, Oxford.

Ayscue P, Lanzas C, Ivanek R, Gröhn YT (2009). "Modeling On-Farm *Escherichia coli* O157:H7 Population Dynamics." *Foodborne Pathogens and Disease*, **6**(4), 461–470. doi:10.1089/fpd.2008.0235.

Bajardi P, Barrat A, Natale F, Savini L, Colizza V (2011). "Dynamical Patterns of Cattle Trade Movements." *PloS ONE*, **6**(5), e19869. doi:10.1371/journal.pone.0019869.

Bartlett MS (1957). "Measles Periodicity and Community Size." *Journal of the Royal Statistical Society. Series A (General)*, **120**(1), 48–70. doi:10.2307/2342553.

Bauer P, Engblom S (2015). "Sensitivity Estimation and Inverse Problems in Spatial Stochastic Models of Chemical Kinetics." In A Abdulle, S Deparis, D Kressner, F Nobile, M Picasso (eds.), *Numerical Mathematics and Advanced Applications - ENUMATH 2013*, pp. 519–527. Springer-Verlag, Cham. doi:10.1007/978-3-319-10705-9_51.

Bauer P, Engblom S, Widgren S (2016). "Fast Event-Based Epidemiological Simulations on National Scales." *International Journal of High Performance Computing Applications*, **30**(4), 438–453. doi:10.1177/1094342016635723.

Breban R, Drake JM, Stallknecht DE, Rohani P (2009). "The Role of Environmental Transmission in Recurrent Avian Influenza Epidemics." *PLoS computational biology*, **5**(4), e1000346. doi:10.1371/journal.pcbi.1000346.

Brooks-Pollock E, de Jong M, Keeling M, Klinkenberg D, Wood J (2015). "Eight Challenges in Modelling Infectious Livestock Diseases." *Epidemics*, **10**, 1–5. doi:10.1016/j.epidem.2014.08.005.

Cao Y, Gillespie DT, Petzold LR (2007). "Adaptive Explicit-Implicit Tau-Leaping Method With Automatic Tau Selection." *The Journal of Chemical Physics*, **126**(22), 224101. doi:10.1063/1.2745299.

Chambers J (2008). *Software for Data Analysis: Programming with R*. Springer-Verlag, New York. doi:10.1007/978-0-387-75936-4.

Danon L, Ford AP, House T, Jewell CP, Keeling MJ, Roberts GO, Ross JV, Vernon MC (2011). "Networks and the Epidemiology of Infectious Disease." *Interdisciplinary Perspectives on Infectious Diseases*, **2011**. doi:10.1155/2011/284909.

Drawert B, Engblom S, Hellander A (2012). "URDME: A Modular Framework for Stochastic Simulation of Reaction-Transport Processes in Complex Geometries." *BMC Systems Biology*, **6**(1), 1–17. doi:10.1186/1752-0509-6-76.

Dushoff J, Plotkin JB, Levin SA, Earn DJ (2004). "Dynamical Resonance Can Account for Seasonality of Influenza Epidemics." *Proceedings of the National Academy of Sciences of the United States of America*, **101**(48), 16915–16916. doi:10.1073/pnas.0407293101.

Dutta BL, Ezanno P, Vergu E (2014). "Characteristics of the Spatio-Temporal Network of Cattle Movements in France Over a 5-Year Period." *Preventive Veterinary Medicine*, **117**(1), 79–94. doi:10.1016/j.prevetmed.2014.09.005.

Engblom S, Ferm L, Hellander A, Lötstedt P (2009). "Simulation of Stochastic Reaction-Diffusion Processes on Unstructured Meshes." *SIAM Journal on Scientific Computing*, **31**(3), 1774–1797. doi:10.1137/080721388.

Engblom S, Widgren S (2017). "Data-Driven Computational Disease Spread Modeling: From Measurement to Parametrization and Control." In ASRS Rao, S Pyne, CR Rao (eds.), *Disease Modeling and Public Health*, volume 36 of *Handbook of Statistics*. Elsevier, Amsterdam. doi:10.1016/bs.host.2017.05.005.

Fog A (2008). "Sampling methods for Wallenius' and Fisher's noncentral hypergeometric distributions." *Communications in Statistics—Simulation and Computation*, **37**(2), 241–257. doi:10.1080/03610910701790236.

Galassi M, Davies J, Theiler J, Gough B, Jungman G, Alken P, Booth M, Rossi F (2009). *GNU Scientific Library Reference Manual*. Network Theory Limited, 3rd edition. ISBN: 0-9546120-7-8, URL https://www.gnu.org/software/gsl/.

Gautam R, Bani-Yaghoub M, Neill WH, Döpfer D, Kaspar C, Ivanek R (2011). "Modeling the Effect of Seasonal Variation in Ambient Temperature on the Transmission Dynamics of a Pathogen With a Free-Living Stage: Example of *Escherichia coli* O157:H7 in a Dairy Herd." *Preventive Veterinary Medicine*, **102**(1), 10–21. doi:10.1016/j.prevetmed.2011.06.008.

Gillespie DT (1977). "Exact Stochastic Simulation of Coupled Chemical Reactions." *The Journal of Physical Chemistry*, **81**(25), 2340–2361. doi:10.1021/j100540a008.

Grenfell B, Harwood J (1997). "(Meta) Population Dynamics of Infectious Diseases." *Trends in ecology & evolution*, **12**(10), 395–399. doi:10.1016/S0169-5347(97)01174-9.

Jenness S, Goodreau SM, Morris M (2017). ***EpiModel**: Mathematical Modeling of Infectious Disease*. R package version 1.5.0, URL https://CRAN.R-project.org/package=EpiModel.

Johnson P (2016). ***adaptivetau**: Tau-Leaping Stochastic Simulation*. R package version 2.2-1, URL https://CRAN.R-project.org/package=adaptivetau.

Keeling MJ, Danon L, Vernon MC, House TA (2010). "Individual Identity and Movement Networks for Disease Metapopulations." *Proceedings of the National Academy of Sciences*, **107**(19), 8866–8870. doi:10.1073/pnas.1000416107.

Keeling MJ, Rohani P (2007). *Modeling Infectious Diseases in Humans and Animals*. First edition. Princeton University Press, New Jersey.

Kempe D, Kleinberg J, Kumar A (2002). "Connectivity and Inference Problems for Temporal Networks." *Journal of Computer and System Sciences*, **64**(4), 820–842. doi:10.1006/jcss.2002.1829.

Kermack WO, McKendrick AG (1927). "A Contribution to the Mathematical Theory of Epidemics." *Proceedings of the Royal Society, London A*, **115**, 700–721. doi:10.1098/rspa.1927.0118.

Kernighan BW, Ritchie DM (1988). *C Programming Language*. Prentice Hall Press, Upper Saddle River, NJ.

King AA, Domenech de Cellès M, Magpantay FMG, Rohani P (2015). "Avoidable Errors in the Modelling of Outbreaks of Emerging Pathogens, with Special Reference to Ebola." *Proceedings of the Royal Society of London B: Biological Sciences*, **282**(1806). doi:10.1098/rspb.2015.0347.

Marques FS, Amaku M, Grisi-Filho JHH (2017). ***hybridModels**: An R Package for Stochastic Simulation of Disease Spreading in Dynamic Networks*. R package version 0.2.15, URL https://CRAN.R-project.org/package=hybridModels.

Merl D, Johnson LR, Gramacy RB, Mangel M (2010). "**amei**: An R Package for the Adaptive Management of Epidemiological Interventions." *Journal of Statistical Software*, **36**(6), 1–32. doi:10.18637/jss.v036.i06.

Meyer S, Held L, Höhle M (2017). "Spatio-Temporal Analysis of Epidemic Phenomena Using the R Package **surveillance**." *Journal of Statistical Software*, **77**(11), 1–55. doi:10.18637/jss.v077.i11.

Newell DG, Koopmans M, Verhoef L, Duizer E, Aidara-Kane A, Sprong H, Opsteegh M, Langelaar M, Threfall J, Scheutz F, *et al.* (2010). "Food-Borne diseases - The Challenges of 20 Years Ago Still Persist While New Ones Continue to Emerge." *International Journal of Food Microbiology*, **139**, S3–S15. doi:10.1016/j.ijfoodmicro.2010.01.021.

OpenMP Architecture Review Board (2008). "OpenMP Application Program Interface Version 3.0." URL https://www.openmp.org/wp-content/uploads/spec30.pdf.

Pineda-Krch M (2008). "**GillespieSSA**: Implementing the Gillespie Stochastic Simulation Algorithm in R." *Journal of Statistical Software*, **25**(1), 1–18. doi:10.18637/jss.v025.i12.

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/.

Rosenzweig ML, MacArthur RH (1963). "Graphical Representation and Stability Conditions of Predator-Prey Interactions." *The American Naturalist*, **97**(895), 209–223. doi:10.1086/282272.

Smith R, Schukken Y, Gröhn Y (2015). "A New Compartmental Model of *Mycobacterium avium subsp. paratuberculosis* Infection Dynamics in Cattle." *Preventive Veterinary Medicine*, **122**, 298–305. doi:10.1016/j.prevetmed.2015.10.008.

Wickham H (2015). *R Packages: Organize, Test, Document, and Share Your Code*. O'Reilly Media, Inc., Sebastopol, CA. URL https://r-pkgs.org/.

Widgren S, Bauer P, Eriksson R, Engblom S (2019). "SimInf: An R Package for Data-Driven Stochastic Disease Spread Simulations." *Journal of Statistical Software*, **91**(12), 1–42. doi:10.18637/jss.v091.i12.

Widgren S, Engblom S, Bauer P, Frössling J, Emanuelson U, Lindberg A (2016). "Data-Driven Network Modelling of Disease Transmission Using Complete Population Movement Data: Spread of VTEC O157 in Swedish Cattle." *Veterinary Research*, **47**(1), 81. doi:10.1186/s13567-016-0366-5.

Widgren S, Engblom S, Emanuelson U, Lindberg A (2018). "Spatio-Temporal Modelling of Verotoxigenic *Escherichia coli* O157 in Cattle in Sweden: Exploring Options for Control." *Veterinary Research*, **49**(1), 78. doi:10.1186/s13567-018-0574-2.

Zhang XS, Chase-Topping ME, McKendrick IJ, Savill NJ, Woolhouse MEJ (2010). "Spread of *E. coli* O157 Infection Among Scottish Cattle Farms: Stochastic Models and Model Selection." *Epidemics*, **2**(1), 11–20. doi:10.1016/j.epidem.2010.02.001.

A. Pseudo-code for the default simulation solver

Algorithm Pseudo-code for the default simulation solver using direct SSA

```
1:  Run trajectory: Dispatch to model specific run method.
2:  C interface: Initialize model transition rate functions and post time step function.
3:  for all nodes i = 1 to N_nodes do in parallel
4:     Compute transition rates for all transitions ω_{i,j}, j = 1, ..., N_trans.
5:  end for
6:  while t < T_End do
7:     for all nodes i = 1 to N_nodes do in parallel
8:        loop
9:           Compute sum of transition rates λ_i = Σ_{j=1}^{N_trans} ω_{i,j}
10:          Sample time to next stochastic event τ_i = −log(r_1)/λ_i where r_1
             is a uniformly distributed random number in the range (0, 1)
11:          if τ_i + t_i >= T_Next day then
12:             Move simulated time forward t_i = T_Next day
13:             go to 20
14:          end if
15:          Move simulated time forward t_i = t_i + τ_i
16:          Determine which state transition happened; by inversion,
             find n such that Σ_{j=1}^{n−1} ω_{i,j} < λr_2 ≤ Σ_{j=1}^{n} ω_{i,j} where r_2
             is a uniformly distributed random number in the range (0, 1)
17:          Update the compartments u[, i] using the state-change vector S[, n]
18:          Use the dependency graph G[, n] to recalculate affected transition rates ω_{i,j}
19:       end loop
20:       Process E_1 events
21:    end for
22:    Process E_2 events
23:    for all nodes i = 1 to N_nodes do in parallel
24:       Call post time step function and update the continuous state variable v[ ,i].
25:    end for
26:    T_Next day = T_Next day + 1
27: end while
```

B. Illustration of scheduled events

This section illustrates how the scheduled events for the `SISe3_sp` model are specified (Table 5) and how each event type is executed (Figures 10, 11, 12, 13, 14)

Action	`event`	`time`	`node`	`dest`	`n`	`proportion`	`select`	`shift`
Exit individuals in S_1 and I_1	exit	t	i	0	n	0	4	0
Exit individuals in S_2 and I_2	exit	t	i	0	n	0	5	0
Exit individuals in S_3 and I_3	exit	t	i	0	n	0	6	0
Enter individuals in S_1 and I_1	enter	t	i	0	n	0	1	0
Enter individuals in S_2 and I_2	enter	t	i	0	n	0	2	0
Enter individuals in S_3 and I_3	enter	t	i	0	n	0	3	0
Age individuals in S_1 and I_1	intTrans	t	i	0	n	0	4	1
Age individuals in S_2 and I_2	intTrans	t	i	0	n	0	5	2
Move individuals in S_1 and I_1	extTrans	t	i	j	n	0	4	0
Move individuals in S_2 and I_2	extTrans	t	i	j	n	0	5	0
Move individuals in S_3 and I_3	extTrans	t	i	j	n	0	6	0

Table 5: Examples of the specification of a single row of scheduled event data in the `SISe3_sp` model to add, move or remove individuals during the simulation, where t is the time-point for the event, i is the node to operate on, j is the destination node for a movement.

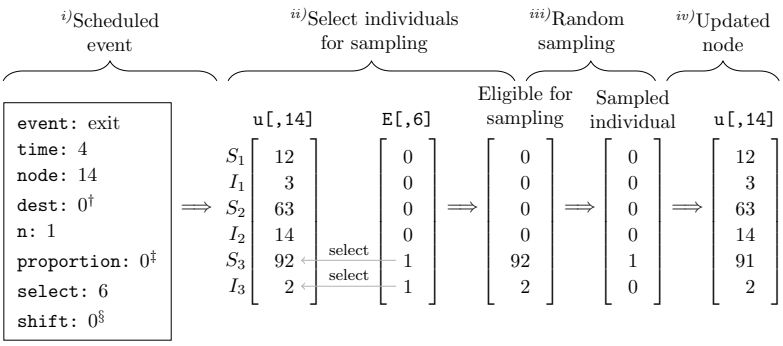

Figure 10: Illustration of a scheduled exit event in the `SISe3_sp` model at time = 4. The removal of one individual in the third age category $\{S_3, I_3\}$ from node 14. Interpreting the figure from left to right: i) A single row of the event data operating on node 14. ii) `u[, 14]` is the current state of node 14; `E[, 6]` is the 6th column in the select matrix that determines which compartments (age categories) that are eligible for sampling. iii) The operation of randomly sampling one individual (n = 1) to move from the compartments selected in step ii). iv) The resultant state of node 14 after subtracting the sampled individual in step iii from node 14. †`dest` and §`shift` are not used in a scheduled exit event. ‡`proportion` is not used when $n > 0$.

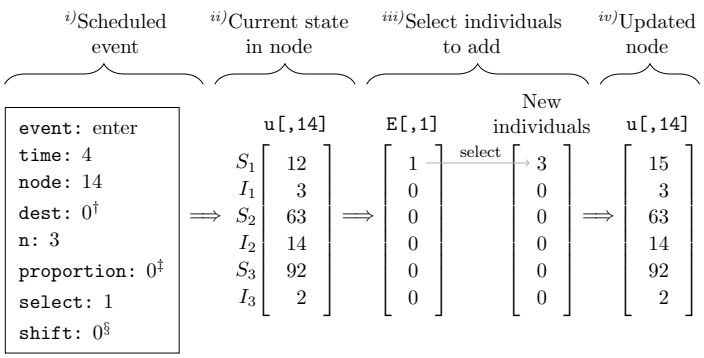

Figure 11: Illustration of a scheduled enter event in the `SISe3_sp` model at time = 4. Add three susceptible individuals to the first age category $\{S_1\}$ in node 14. Interpreting the figure from left to right: i) A single row of the event data operating on node 14. ii) `u[, 14]` is the current state of node 14. iii) `E[, 1]` is the first column in the select matrix that determines which compartments (age categories) the new individuals are added. iv) The resultant state of node 14 after adding the individuals in step iii). §`shift` can be used in a scheduled enter event, see Figure 13 for an illustration of that functionality. ‡`proportion` is not used when $n > 0$. †`dest` is not used in a scheduled enter event.

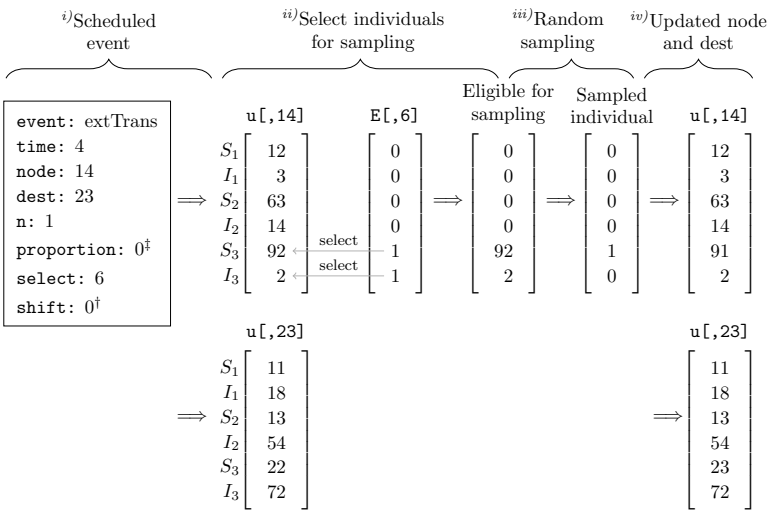

Figure 12: Illustration of a scheduled external transfer event in the `SISe3_sp` model at time = 4. The movement of one individual in the third age category $\{S_3, I_3\}$ from node 14 to destination node 23. Interpreting the figure from left to right: i) A single row of the event data operating on node 14 and destination node 23. ii) `u[, 14]` is the current state of node 14; `u[, 23]` is the current state of the destination node 23; `E[, 6]` is the 6th column in the select matrix that determines which compartments (age categories) that are eligible for sampling. iii) The operation of randomly sampling one individual (n = 1) to move from the compartments selected in step ii). iv) The resultant state of node 14 and destination node 23 after subtracting the sampled individuals in step iii) from node 14 and adding them to destination node 23. †`shift` can be used in a scheduled external transfer event, see Figure 14. ‡`proportion` is not used when $n > 0$.

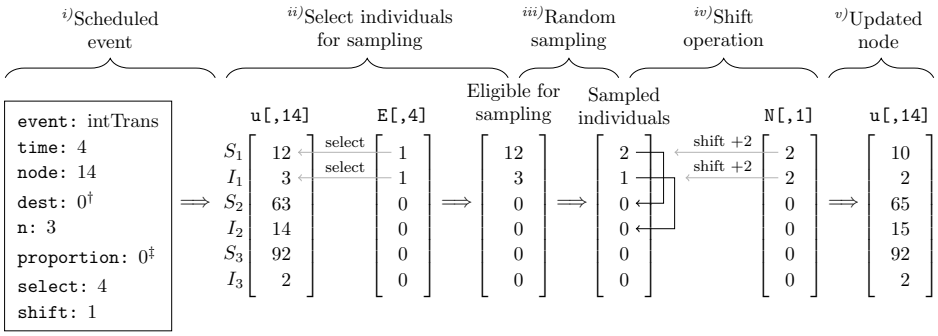

Figure 13: Illustration of a scheduled internal transfer event in the `SISe3_sp` model at time = 4. The ageing of three individuals in the first age category $\{S_1, I_1\}$. Interpreting the figure from left to right: i) A single row of the event data operating on node 14. ii) `u[, 14]` is the current state of node 14; `E[, 4]` is the 4th column in the select matrix that determines which compartments (age categories) that are eligible for sampling. iii) The operation of randomly sampling three individuals (n = 3) to age from the compartments selected in step ii). iv) The shift operation applies the shift specified in column 1 of the shift matrix (`N`) to the individuals sampled in step iii). v) The resultant state of node 14 after subtracting the sampled individuals in step iii) and adding the individuals after the shift operation in step iv). †`dest` is not used in internal transfers. ‡`proportion` is not used when $n > 0$.

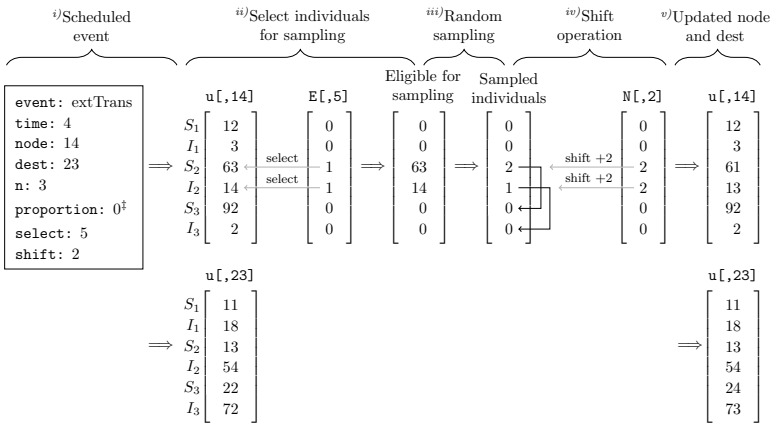

Figure 14: Illustration of a scheduled external transfer event in the `SISe3` model at time = 4. The ageing of three individuals in the second age category $\{S_2, I_2\}$ that are subsequently moved. Interpreting the figure from left to right: i) A single row of the event data operating on node 14 and destination node 23. ii) `u[, 14]` is the current state of node 14; `u[, 23]` is the current state of the destination node 23; `E[, 5]` is the 5th column in the select matrix that determines which compartments (age categories) that are eligible for sampling. iii) The operation of randomly sampling three individuals (n = 3) to move from the compartments selected in step ii). iv) The shift operation applies the shift specified in column 2 of the shift matrix (`N`) to the individuals sampled in step iii). v) The resultant state of node 14 and destination node 23 after subtracting the sampled individuals in step iii) from node 14 and adding them to the destination node 23 after the shift operation in step iv). ‡`proportion` is not used when $n > 0$.

C. C code for the SIR model

Listing 1: Implementation of the function to init and run a simulation with the SIR model

```
SEXP SIR_run(SEXP model, SEXP options)
{
    TRFun tr_fun[] = {&SIR_S_to_I, &SIR_I_to_S};

    return SimInf_run(model, options, tr_fun, &SIR_post_time_step);
}
```

Listing 2: Implementation of the transition rate functions in the SIR model for the transitions in Equation 18 between the susceptible and infected compartments. The enumeration declarations are used to name the variable offsets and facilitate extraction of the values from the various data vectors.

```
/* Offset in integer compartment state vector */
enum {S, I, R};

/* Offsets in global data (gdata) to parameters in the model */
enum {BETA, GAMMA};

/* susceptible to infected: S -> I */
double SIR_S_to_I(const int *u, const double *v, const double *ldata,
                  const double *gdata, double t)
{
    const double S_n = u[S];
    const double I_n = u[I];
    const double n = S_n + I_n + u[R];

    if (n > 0.0)
        return (gdata[BETA] * S_n * I_n) / n;
    return 0.0;
}

/* infected to susceptible: I -> S */
double SIR_I_to_S(const int *u, const double *v, const double *ldata,
                  const double *gdata, double t)
{
    return gdata[GAMMA] * u[I];
}
```

Listing 3: Implementation of the post time step function in the SIR model. The post time step function should return a value > 0 if the node needs to recalculate the transition rates for the node, a value (error code) < 0 if an error is detected, or otherwise 0. Since the post time step function for the SIR model does not make any changes to a node, it always return 0.

```
int SIR_post_time_step(double *v_new, const int *u, const double *v,
                       const double *ldata, const double *gdata,
                       int node, double t)
{
    return 0;
}
```

Affiliation:

Stefan Widgren
Department of Disease Control and Epidemiology
National Veterinary Institute
SE-751 89 Uppsala, Sweden
E-mail: stefan.widgren@sva.se
and
Division of Scientific Computing
Department of Information Technology
Uppsala University
SE-751 05 Uppsala, Sweden

Pavol Bauer
Division of Scientific Computing
Department of Information Technology
Uppsala University
SE-751 05 Uppsala, Sweden

Robin Eriksson
Division of Scientific Computing
Department of Information Technology
Uppsala University
SE-751 05 Uppsala, Sweden
E-mail: robin.eriksson@it.uu.se

Stefan Engblom
Division of Scientific Computing
Department of Information Technology
Uppsala University
SE-751 05 Uppsala, Sweden
E-mail: stefane@it.uu.se