Getting Started with spaero

Eamon O'Dea

2016-04-04

The spaero package (pronounced sparrow) currently supports the estimation of statistics along rolling windows of time series. Such estimates may in some cases provide signals that the system generating the data is approaching a critical transition. Examples of critical transitions include the eutrophication of lakes, changes in climate, and the emergence or eradication of infectious diseases. The spaero package will be developed to further support statistical methods to anticipate critical transitions in infectious disease systems. Because these methods will be based on generic properties of dynamical systems, they have the potential to apply to a broad range of models. Spaero also provides functions to support computational experiments designed to evaluate these methods for applications relevant to infectious disease systems. This document provides a rudimentary demonstration of the application of such methods to simulated data.

Our simulated data is a time series produced by a stochastic SIR simulator included in the spaero package. See Keeling and Rohani (2008) for an introduction to the SIR model. The simulator is capable of including time dependent parameters. Gillespie's direct method is used to update the model variables during the simulation. Transitions between states occurs according to the rules given in Table~1, which makes use of the symbols defined in Table~2. Because some of the transition rates may change continuously with time and because the simulation algorithm updates the rates only at points of time when the model's state variables are updated, these simulations are not in general exact. However, for many realistic scenarios birth and death updates occur frequently enough that the simulation should be highly accurate.

Table 1: Transition rules for our stochastic SIR model

Event	$(\Delta S, \Delta I, \Delta R)$	Rate
birth of a susceptible	$(1, 0, 0)$	$N_0(\mu + \mu_t)$
death of a susceptible	$(-1, 0, 0)$	$S(d + d_t)$
infection	$(-1, 1, 0)$	$(\beta + \beta_t)IS + (\eta + \eta_t)S$
death of an infective	$(0, -1, 0)$	$I(d + d_t)$
recovery of an infective	$(0, -1, 1)$	$I(\gamma + \gamma_t)$
death of a removed	$(0, 0, -1)$	$R(d + d_t)$

Table 2: Model symbol definitions. Time-dependent rates have a t subscript.

Symbol	Definition
η, η_t	rate of infection from outside of population (i.e., sparking rate)
β, β_t	rates of transmission from within population contacts
μ, μ_t	birth rates
d, d_t	death rates
S	number of susceptible individuals
I	number of infective individuals
R	number of removed individuals
N	total population size, $S + I + R$
N_0	initial total population size

Before demonstrating the statistical analysis functions of spearo, we provide an overview of the simulation functions. These functions essentially provide a convenient interface to the general simulation capabilities of the pomp package. The user calls the `create_simulator` function to create an object of class `pomp`. This object contains the model structure as well as default parameters for the simulation. Simulations of the model may then be run using the simulate method in the pomp package:

```
library(spaero)
sim <- create_simulator()
simout <- pomp::simulate(sim)
```

This code creates a new pomp object and runs a simulation with the default parameters. The variable simout contains a second pomp object that contains the simulation results. These results may be extracted like so:

```
as(simout, "data.frame")
```

```
##    time reports      S I R      N cases gamma_t mu_t d_t eta_t beta_t
## 1     0       0 100001 0 0 100001     0       0    0   0     0      0
## 2     1       0 100022 0 2 100024     2       0    0   0     0      0
## 3     2       0 100049 0 2 100051     0       0    0   0     0      0
## 4     3       1 100072 0 5 100077     3       0    0   0     0      0
## 5     4       0 100139 0 6 100145     1       0    0   0     0      0
## 6     5       0 100179 0 7 100186     1       0    0   0     0      0
## 7     6       0 100153 0 9 100162     2       0    0   0     0      0
## 8     7       0 100151 0 9 100160     0       0    0   0     0      0
## 9     8       0 100121 0 9 100130     0       0    0   0     0      0
## 10    9       0 100042 0 9 100051     0       0    0   0     0      0
```

Alternatively, one can run the simulator like this to output the results as a data frame.

```
simout <- pomp::simulate(sim, as.data.frame=TRUE)
```

In addition to simulating the dynamics of disease spread, the pomp object also simulates imperfect observation of the dynamics. A cases variable is included in the output and it counts the total number of recoveries that occurred in the preceding interval between observations. A corresponding number of reports is simulated by sampling from a binomial probability mass function with a number of trials equal to the number of cases and a reporting probability equal to a user-supplied parameter, ρ.

Observation times and parameters can be set at simulation run time:

```
pars <- sim@params
pars["rho"] <- 0.5
pomp::simulate(sim, params=pars, times=seq(1, 4), as.data.frame=TRUE)
```

```
##   time reports      S I R      N cases gamma_t mu_t d_t eta_t beta_t sim
## 1    1       4  99971 0 6  99977     6       0    0   0     0      0   1
## 2    2       0  99948 0 6  99954     0       0    0   0     0      0   1
## 3    3       0 100035 0 6 100041     0       0    0   0     0      0   1
## 4    4       1 100086 0 7 100093     2       0    0   0     0      0   1
```

However, the covariate table that determines the time dependence of rates cannot be set at simulation time.

One can also set the number of replicates.

```
pomp::simulate(sim, nsim=2, times=seq(1, 2), as.data.frame=TRUE)
```

```
##   time reports      S I  R      N cases gamma_t mu_t d_t eta_t beta_t sim
## 1    1       0  99980 0  0  99980     0       0    0   0     0      0   1
## 2    2       0 100081 0  0 100081     0       0    0   0     0      0   1
## 3    1       0 100029 0  0 100029     0       0    0   0     0      0   2
## 4    2       3 100061 0 11 100072    11       0    0   0     0      0   2
```

The random number seed allows simulations to be reproduced.

```
pomp::simulate(sim, seed=342, as.data.frame=TRUE)
```

```
##    time reports      S I  R      N cases gamma_t mu_t d_t eta_t beta_t sim
## 1     0       0  99999 0  0  99999     0       0    0   0     0      0   1
## 2     1       0  99992 0  0  99992     0       0    0   0     0      0   1
## 3     2       0 100020 0  1 100021     1       0    0   0     0      0   1
## 4     3       0 100004 0  1 100005     0       0    0   0     0      0   1
## 5     4       0  99984 0  1  99985     0       0    0   0     0      0   1
## 6     5       0 100082 0  2 100084     1       0    0   0     0      0   1
## 7     6       2  99971 0 25  99996    23       0    0   0     0      0   1
## 8     7       0  99943 0 25  99968     0       0    0   0     0      0   1
## 9     8       0  99925 0 26  99951     2       0    0   0     0      0   1
## 10    9       1  99921 0 29  99950     3       0    0   0     0      0   1
```

```
pomp::simulate(sim, seed=342, as.data.frame=TRUE)
```

```
##    time reports      S I  R      N cases gamma_t mu_t d_t eta_t beta_t sim
## 1     0       0  99999 0  0  99999     0       0    0   0     0      0   1
## 2     1       0  99992 0  0  99992     0       0    0   0     0      0   1
## 3     2       0 100020 0  1 100021     1       0    0   0     0      0   1
## 4     3       0 100004 0  1 100005     0       0    0   0     0      0   1
## 5     4       0  99984 0  1  99985     0       0    0   0     0      0   1
## 6     5       0 100082 0  2 100084     1       0    0   0     0      0   1
## 7     6       2  99971 0 25  99996    23       0    0   0     0      0   1
## 8     7       0  99943 0 25  99968     0       0    0   0     0      0   1
## 9     8       0  99925 0 26  99951     2       0    0   0     0      0   1
## 10    9       1  99921 0 29  99950     3       0    0   0     0      0   1
```

An SIS model is also available. This model is identical to the SIR model except that the recovery event in Table~1 results in an infective individual becoming a susceptible.

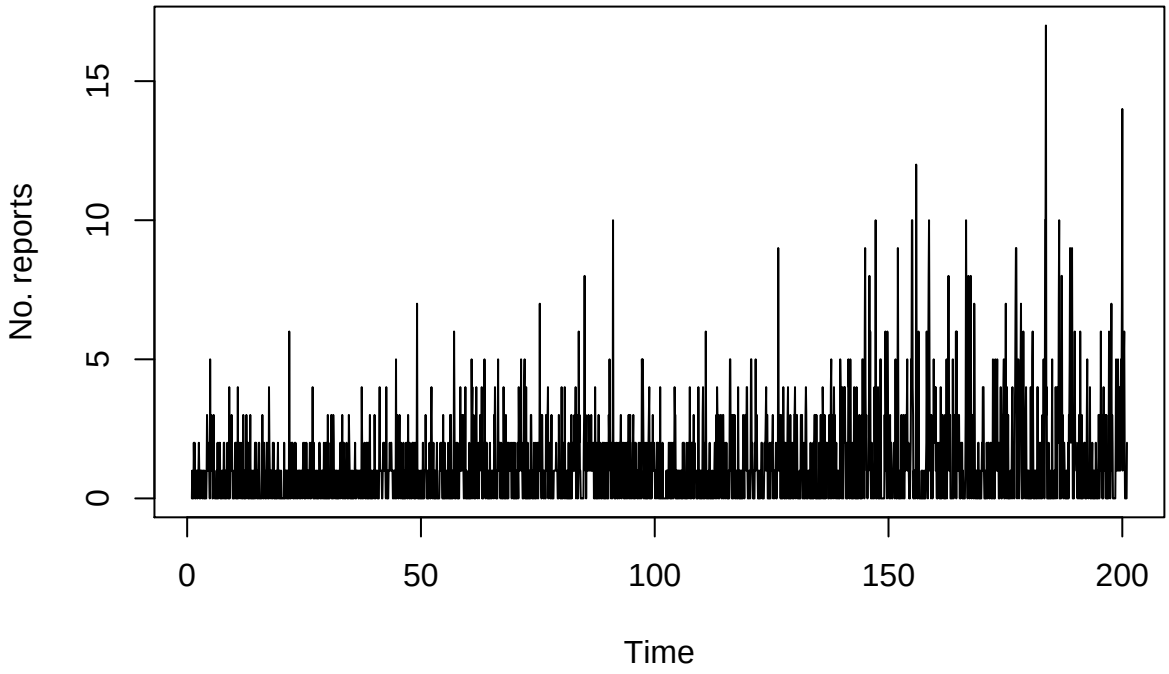
```
sim_sis <- create_simulator(process_model="SIS")
pomp::simulate(sim_sis, as.data.frame=TRUE)
```

```
##    time reports      S I R      N cases gamma_t mu_t d_t eta_t beta_t sim
## 1     0       0  99999 0 0  99999     0       0    0   0     0      0   1
## 2     1       1 100056 0 0 100056     2       0    0   0     0      0   1
## 3     2       0 100063 0 0 100063     0       0    0   0     0      0   1
## 4     3       0 100037 0 0 100037     1       0    0   0     0      0   1
## 5     4       1 100096 2 0 100098     2       0    0   0     0      0   1
## 6     5       0 100105 0 0 100105     2       0    0   0     0      0   1
## 7     6       0 100056 0 0 100056     0       0    0   0     0      0   1
## 8     7       0  99982 0 0  99982     1       0    0   0     0      0   1
## 9     8       0  99971 0 0  99971     1       0    0   0     0      0   1
## 10    9       0 100034 0 0 100034     1       0    0   0     0      0   1
```

Now let's simulate data according to the SIR model where the transmission rate starts out well below the threshold value and gradually increases. Note that parameters corresponding to the initial conditions (i.e., S_0, I_0, and R_0) are normalized to sum to N_0. Thus we can specify that the simulation begins with a population of 100,000 susceptibles as follows.

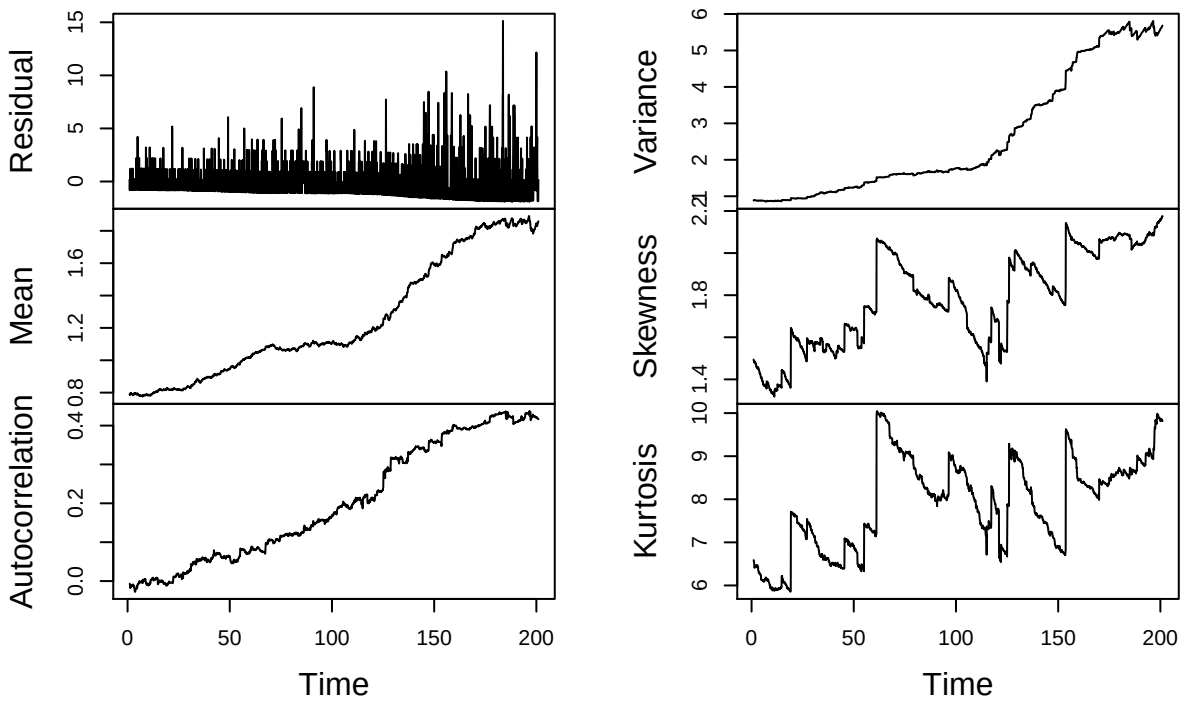
```
params <- c(gamma=24, mu=0.014, d=0.014, eta=1e-4, beta=0,
            rho=0.9, S_0=1, I_0=0, R_0=0, N_0=1e5)
covar <- data.frame(gamma_t=c(0, 0), mu_t=c(0, 0), d_t=c(0, 0), eta_t=c(0, 0),
                    beta_t=c(0, 24e-5), time=c(0, 300))
times <- seq(0, 200, by=1/12)

sim <- create_simulator(params=params, times=times, covar=covar)
so <- pomp::simulate(sim, as.data.frame=TRUE, seed=272)
plot(ts(so[, "reports"], freq=12), ylab="No. reports")
```



By eye, we can see the distribution of reports seems to change over time. We can summarize these changes by computing statistics over moving windows.

```
st1 <- get_stats(so[, "reports"], center_kernel="uniform",
                 center_trend="local_constant", center_bandwidth=360,
                 stat_bandwidth=360)
plot_st <- function(st) {
  plot_vars <- ts(cbind(Residual=st$centered$x[, 1], Mean=st$stats$mean,
                  Autocorrelation=st$stats$autocor, Variance=st$stats$var,
                  Skewness=st$stats$skew, Kurtosis=st$stats$kurt), freq=12)
  plot(plot_vars, main="")
}
plot_st(st1)
```



The increasing trends in the statistics are a potential warning signals that the system is approaching the epidemic threshold. Readers interested in this type of analysis can find guidelines in Dakos et al. (2012) and may also want to consider performing it with the `generic_ews` function in the earlywarnings package described in that paper. We'll next review the input parameters and implementation of get_stats.

Two key parameters that the user must provide to `get_stats` are the shape and size of the rolling window. There is a rolling window for an estimate of the mean and for an estimate of statistics within the window. Arguments controlling these windows are prefixed with the "center_" and "stat_" respectively. An estimate of the mean is necessary because the calculation of the statistics involve deviations from the mean. `get_stats` supports estimation of the mean via several methods and users may also estimate the mean using other methods, subtract it from the input time series, and then set the "center_trend" argument to "assume_zero". Regarding the shapes of windows, a rectangular window function and a Gaussian-shaped function are available by providing either "uniform" or "gaussian" to the kernel arguments. The rectangular function may be preferred for ease of interpretation while the Gaussian function may be preferred for obtaining a smoother series of estimates. The width of the window is controlled by the bandwidth arguments. For a window centered on a particular index, the absolute difference between that index and all other indices in the time series is divided by the bandwidth to determine a distance to all other observations. This distance is then plugged into a kernel function corresponding to the window type. For the gaussian window, the kernel function is a Gaussian probability density function with a standard deviation of one. For the rectangular window, the kernel function equals one if the distance is less than one and zero otherwise. The output of the kernel function is a weight for each observation. These weights are used in the estimators described next. Note that these bandwidth conventions are different from those of `generic_ews`.

By default, `get_stats` estimates statistics via weighted sample moments. To clarify, the estimate of the moment for the moving window centered on index i of the time series x is

$$m_i(f_j(x)) = \sum_j w_{ij} f_j(x)/N_i, \tag{1}$$

where w_{ij} is a kernel weight, $f_j(x)$ is the value of the moment at index j, and $N_i = \sum_j w_{ij}$ is a normalization constant. Table~3 provides the formulas for the statistics estimated in terms of these moment estimates. In some cases, users may obtain less biased estimates by setting the "stat_trend" argument to "local_linear".

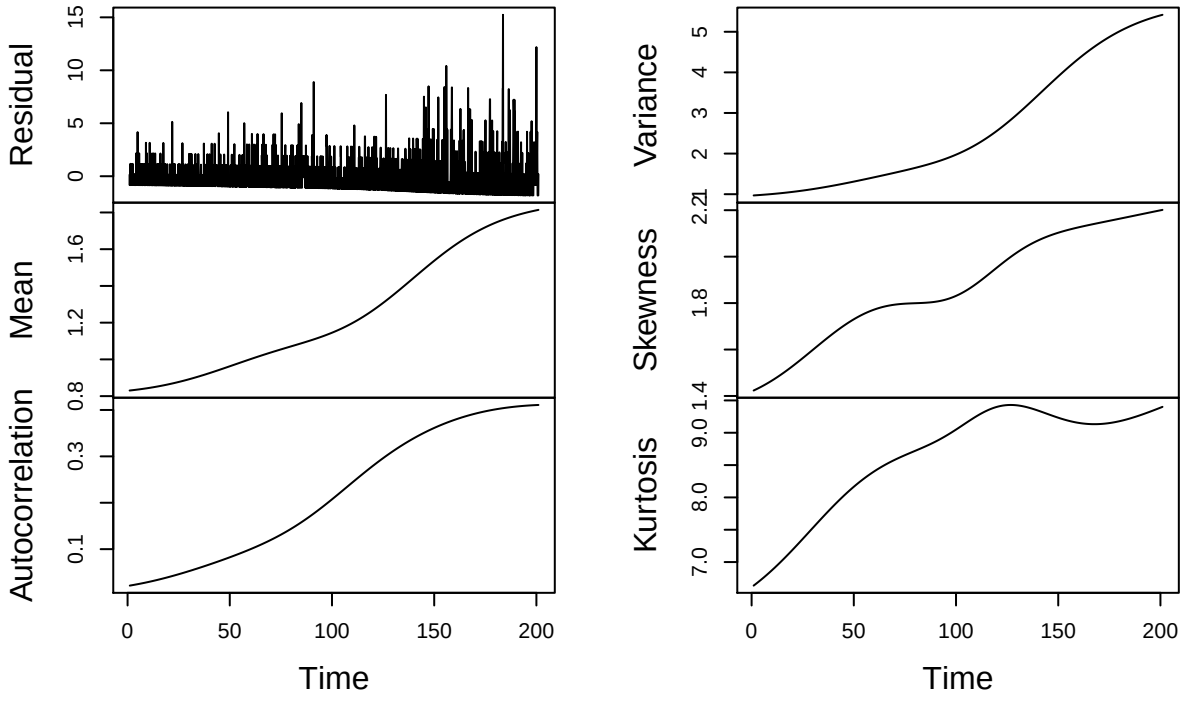
This replaces the weighted average estimate with a prediction from a local linear regression. This method can reduce bias near the ends of the time series if a trend exists such that $f_j(x)$ for j near i tend to be above or below the expected value of $f_i(x)$ across repeated realizations of a time series.

Table 3: Formulas for statistics at windows in terms of moment estimates.

Statistic	Formula
mean_i	$m_i(x_j)$
variance_i	$m_i((x_j - \text{mean}_j)^2)$
autocovariance_i	$m_i((x_j - \text{mean}_j)(x_{j-\text{lag}} - \text{mean}_{j-\text{lag}}))$
autocorrelation_i	$\text{autocovariance}_i / \text{variance}_i$
$(\text{decay time})_i$	$-\text{lag}/(\log \min(\max(\text{autocorrelation}_i, 0), 1))$
$(\text{index of dispersion})_i$	$\text{variance}_i / \text{mean}_i$
$(\text{coefficient of variation})_i$	$(\text{variance}_i)^{0.5} / \text{mean}_i$
skewness_i	$m_i((x_j - \text{mean}_j)^3) / (\text{variance}_i)^{1.5}$
kurtosis_i	$m_i((x_j - \text{mean}_j)^4) / (\text{variance}_i)^2$

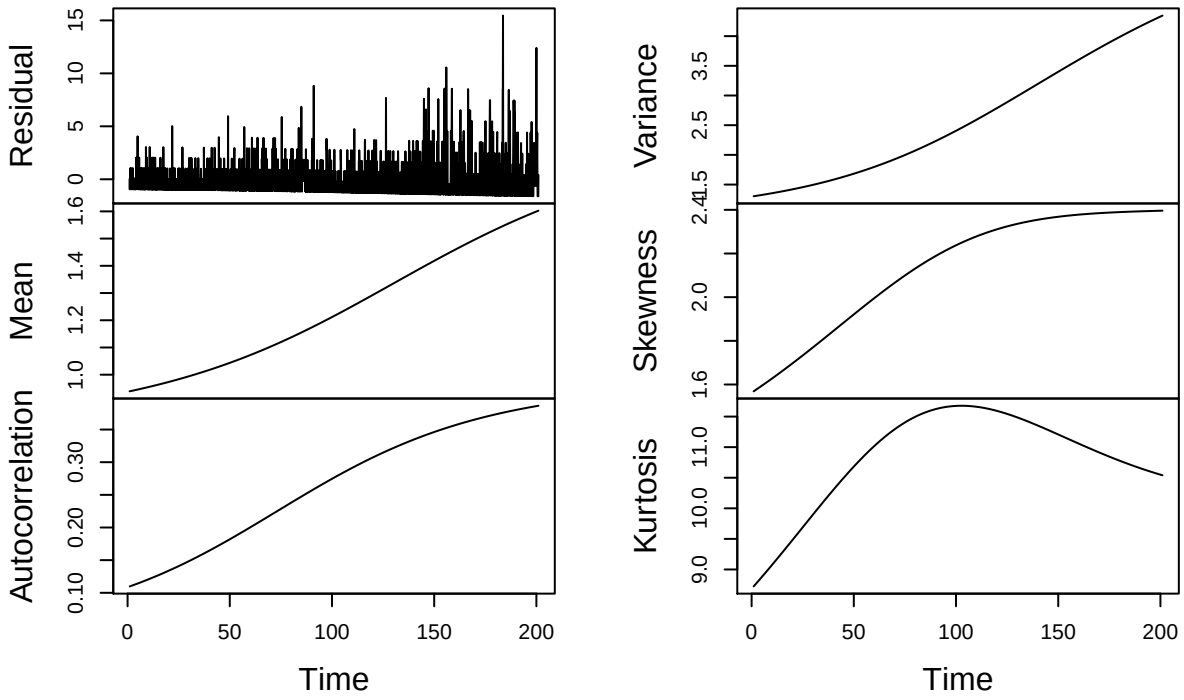
Let's look at the effect of changing some of these parameters on the estimated statistics. First we try a Gaussian window.

```
st2 <- get_stats(so[, "reports"], center_kernel="gaussian",
                 center_trend="local_constant", center_bandwidth=360,
                 stat_bandwidth=360, stat_kernel="gaussian")
plot_st(st2)
```



Next, we'll increase the bandwidths.

```
st3 <- get_stats(so[, "reports"], center_kernel="gaussian",
                 center_trend="local_constant", center_bandwidth=720,
                 stat_bandwidth=720, stat_kernel="gaussian")
plot_st(st3)
```



That concludes our initial overview of the package. The current version of spaero is just a starting point and the package will continue to be actively developed for the foreseeable future.

References

Dakos, Vasilis, Stephen R. Carpenter, William A. Brock, Aaron M. Ellison, Vishwesha Guttal, Anthony R. Ives, Sonia Kéfi, et al. 2012. "Methods for Detecting Early Warnings of Critical Transitions in Time Series Illustrated Using Simulated Ecological Data." *PLoS ONE* 7 (7): e41010. doi:10.1371/journal.pone.0041010.

Keeling, Matt J., and Pejman Rohani. 2008. *Modeling Infectious Diseases in Humans and Animals.* Princeton, New Jersey: Princeton UP.