Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

lgcp – An R Package for Inference with Spatial and Spatio-Temporal Log-Gaussian Cox Processes

Benjamin M. Taylor
Lancaster University, UK

Tilman M. Davies
University of Otago, NZ

Barry S. Rowlingson
Lancaster University, UK

Peter J. Diggle
Lancaster University, UK

Abstract

This paper introduces an R package for spatial and spatio-temporal prediction and forecasting for log-Gaussian Cox processes. The main computational tool for these models is Markov chain Monte Carlo (MCMC) and the new package, **lgcp**, therefore also provides an extensible suite of functions for implementing MCMC algorithms for processes of this type. The modelling framework and details of inferential procedures are first presented before a tour of **lgcp** functionality is given via a walk-through data-analysis. Topics covered include reading in and converting data, estimation of the key components and parameters of the model, specifying output and simulation quantities, computation of Monte Carlo expectations, post-processing and simulation of data sets.

Keywords: Cox process; R; spatio-temporal point process.

1. Introduction

This article introduces a new R (R Development Core Team 2011) package, **lgcp**, for inference with spatial and spatio-temporal log-Gaussian Cox processes (LGCP). The work was motivated by applications in disease surveillance, where the major focus of scientific interest is on whether, and if so where and when, cases form unexplained clusters within a spatial region W and time-interval $[0, T]$ of interest. It will be assumed that both the location and time of each case is known, at least to a sufficiently fine resolution that a point process modelling framework is natural. In general, the aims of statistical analysis include model formulation, parameter estimation and spatio-temporal prediction. The **lgcp** package includes some functionality for parameter estimation and diagnostic checking, mostly by linkages with other

R packages for spatial statistics. However, and consistent with the scientific focus being on disease surveillance, the current version of the package places particular emphasis on real-time predictive inference. Specifically, using the modelling framework of a Cox process with stochastic intensity $R(s,t) = \lambda(s)\mu(t)\exp\{\mathcal{Y}(s,t)\}$, where $\mathcal{Y}(s,t)$ is a Gaussian process and λ and μ are offset terms supplied by the user. The package enables the user to draw samples from the joint predictive distribution of $R(s,[t_1,t_2])$ for some time interval $[t_1,t_2] \subseteq [0,T]$ as well as forecast beyond the last observation at time T, i.e., to determine $R(s,T+k)$ given data between times $T-l$ and T, where l is a user-specified lag time for any set of locations $s \in W$ and forecast lead time $k \geq 0$. In the surveillance setting, these samples would typically be used to evaluate the predictive probability that the intensity at a particular location and time exceeds a specified intervention threshold; see, for example, Diggle, Rowlingson, and Su (2005).

The **lgcp** package makes extensive use of **spatstat** functions and data structures (Baddeley and Turner 2005). Other important dependencies are: the **sp** package, which also supplies some data structures and functions (Pebesma and Bivand 2005; Bivand, Pebesma, and Gomez-Rubio 2008); the suite of covariance functions provided by the **RandomFields** package (Schlather 2001); the **rpanel** package to facilitate minimum-contrast parameter estimation routines (Bowman, Crawford, Alexander, and Bowman 2007; Bowman, Gibson, Scott, and Crawford 2010); and the **ncdf** package for rapid access to massive datasets for post-processing (Pierce 2011).

In Section 2 the log-Gaussian Cox process is introduced. Section 3 gives a review on methods of inference for log-Gaussian Cox processes. Section 4 is an overview of the package by way of a walk-through example, covering: reading in data (Section 4.2); estimating components of the model and associated parameters (Sections 4.3 and 4.4); setting up and running the model (Sections 4.5 and 4.6); and post-processing of command outputs (Section 4.7). Some possible extensions of the package are given in Section 5. The appendices give further information on the rotation of observation windows (Appendix B), simulation of data (Appendix C) and information about the `spatialAtRisk` class of objects (Appendix D), which may be useful for reference purposes. Appendix E gives some tips on handling ESRI shapefiles and Appendix F shows the user how to create adaptive MCMC schemes.

2. Spatio-Temporal log-Gaussian Cox processes

A purely spatial intensity function provides the expected number of points per unit area across a given spatial region W when situated at $x \in W$. This becomes a *spatio-temporal* intensity when the observations are time-dependent, such that we now seek to capture the expected number of points per unit area when situated at location $x \in W$, and time $t \in T$. The spatio-temporal LGCP is extremely flexible in that it enables the presence of both fixed and random effects in capturing this space-time behaviour. Here, we describe some fundamental technical details of this modelling framework.

Let $W \subset \mathbb{R}^2$ be an observation window in space and $T \subset \mathbb{R}_{\geq 0}$ be an interval of time of interest. Cases occur at spatio-temporal positions $(x,t) \in W \times T$ according to an inhomogeneous spatio-temporal Cox process, i.e., a Poisson process with a stochastic intensity $R(x,t)$. The number of cases, $X_{S,[t_1,t_2]}$, arising in any $S \subseteq W$ during the interval $[t_1,t_2] \subseteq T$ is then Poisson

distributed conditional on R,

$$X_{S,[t_1,t_2]} \sim \text{Poisson}\left\{\int_S \int_{t_1}^{t_2} R(s,t)\mathrm{d}s\mathrm{d}t\right\}. \tag{1}$$

Following Diggle *et al.* (2005), the intensity is decomposed multiplicatively as,

$$R(s,t) = \lambda(s)\mu(t)\exp\{\mathcal{Y}(s,t)\}. \tag{2}$$

In Equation 2, the *fixed spatial component*, $\lambda : \mathbb{R}^2 \mapsto \mathbb{R}_{\geq 0}$, is a user-supplied function, proportional to the population at risk at each point in space and scaled so that,

$$\int_W \lambda(s)\mathrm{d}s = 1, \tag{3}$$

whilst the *fixed temporal component*, $\mu : \mathbb{R}_{\geq 0} \mapsto \mathbb{R}_{\geq 0}$, is also a user-supplied function such that,

$$\mu(t) = \lim_{|\delta t|\to 0}\left\{\frac{\mathsf{E}[X_{W,\delta t}]}{|\delta t|}\right\}. \tag{4}$$

The function $\mathcal{Y}$ is a Gaussian process, continuous in both space and time. In the nomenclature of epidemiology, the components λ and μ determine the *endemic* spatial and temporal component of the population at risk; whereas $\mathcal{Y}$ captures the residual variation, or the *epidemic* component.

The Gaussian process, $\mathcal{Y}$, is second order stationary with minimally-parametrised covariance function,

$$\text{cov}[\mathcal{Y}(s_1,t_1),\mathcal{Y}(s_2,t_2)] = \sigma^2 r(||s_2 - s_1||;\phi)\exp\{-\theta(t_2 - t_1)\}, \tag{5}$$

where $||\cdot||$ is a suitable norm on $\mathbb{R}^2$, for instance the Euclidean norm, and $\sigma, \phi, \theta > 0$ are known parameters. In the **lgcp** package, the isotropic spatial correlation function, r, may take one of several forms and possibly require additional parameters (in **lgcp**, this can be selected from any of the compatible models in the function `CovarianceFct` from the **RandomFields** package). The parameter σ scales the log-intensity, whilst the parameters ϕ and θ govern the rates at which the correlation function decreases in space and in time, respectively. The mean of the process $\mathcal{Y}$ is set equal to $-\sigma^2/2$ so as to give $\mathsf{E}[\exp\{\mathcal{Y}\}] = 1$, hence the endemic/epidemic analogy above.

3. Inference

As in Møller, Syversveen, and Waagepetersen (1998), Brix and Diggle (2001) and Diggle *et al.* (2005), a discretised version of the above model will be considered, defined on a regular grid over space and time. Observations, X, are then treated as cell counts on this grid. The discrete version of $\mathcal{Y}$ will be denoted Y; since Y is a finite collection of random variables, the properties of $\mathcal{Y}$ imply that Y has a multivariate Gaussian density with approximate covariance matrix Σ, whose elements are calculated by evaluating Equation 5 at the centroids of the spatio-temporal grid cells. Without loss of generality, unit time-increments are assumed and events can be thought of as occurring "at" integer times t. Let X_t denote an observation over the spatial grid at time t, and $X_{t_1:t_2}$ denote the observations at times $t_1, t_1 + 1, \ldots, t_2$. For predictive inference about Y, samples from the conditional distribution of the latent field,

Y_t, given the observations to date, $X_{1:t}$ would be drawn, but this is infeasible because the dimensionality of the required integration increases without limit as time progresses. An alternative, as suggested by Brix and Diggle (2001), is to sample from $Y_{t_1:t_2}$ given $X_{t_1:t_2}$,

$$\pi(Y_{t_1:t_2}|X_{t_1:t_2}) \propto \pi(X_{t_1:t_2}|Y_{t_1:t_2})\pi(Y_{t_1:t_2}), \tag{6}$$

where $t_1 = t_2 - p$ for some small positive integer p. The justification for this approach is that observations from the remote past have a negligible effect on inference for the current state, Y_t.

In order to evaluate $\pi(Y_{t_1:t_2})$ in Equation 6, the parameters of the process Y must either be known or estimated from the data. Estimation of σ, ϕ and θ may be achieved either in a Bayesian framework, or by one of a number of more *ad hoc* methods. The methods implemented in the current version of the **lgcp** package fall into the latter category and are described in Brix and Diggle (2001) and Diggle *et al.* (2005). Briefly, this involves matching empirical and theoretical estimates of the second-moment properties of the model. For the spatial covariance parameters σ and ϕ, the inhomogeneous K-function, or g function are used (Baddeley, Møller, and Waagepetersen 2000). The autocorrelation function of the total event-counts per unit time-interval is used for estimating the temporal correlation parameter θ. The estimated parameter values can then be used to implement plug-in-prediction for the latent field Y_t.

The package **lgcp** provides only very basic tools for the estimation of $\lambda(s)$ and $\mu(t)$, as described in Section 4.3. The rationale for this is that there are many possible parametric and non-parametric choices implemented in other **R** packages that could be used to infer the fixed spatial or temporal components of the model: for example in Diggle *et al.* (2005), the authors use a generalised linear model for $\mu(t)$. What **lgcp** *does provide* is a flexible framework to allow interface with other packages: $\mu(t)$ can be defined as an arbitrary function, for example a function returning predicted values from the output of a statistical model e.g., `glm()` or `gam()`; and $\lambda(s)$ can either be defined as a continuous function of two variables, or as the output of a model onto a fine grid. In Appendix A, we provide an example in which $mu(t)$ and $\lambda(s)$ are estimated through generalised linear models.

The in-package estimation routines for $\lambda(s)$ and $\mu(t)$ are both non-parametric, respectively a simple bivariate kernel density estimate and a lowess smoother, and not intended to provide a rigorous solution to the needs of all users. Use of these automatic/interactive procedures is subjective and application specific. Generally speaking, under the 'global' treatment of the fixed components λ and μ, we would intuitively err on the side of over-smoothing rather than under-smoothing these trends. Some numerical evidence in Davies and Hazelton (2012) supports this notion in terms of the subsequent performance of the minimum contrast parameter estimation techniques (see Section 4.4).

3.1. Discretising and the fast-Fourier transform

The first barrier to inference is computation of the covariance matrix, Σ, which even for relatively coarse grids is very large. Fortunately, for stationary covariance functions defined on regular spatial grids of size $2^m \times 2^n$, there exist fast methods for computing this based on the discrete Fourier transform (Wood and Chan 1994; Rue and Held 2005; Taylor and Diggle 2012). The general idea is to embed Σ in a symmetric circulant matrix, $C = Q\Lambda Q^*$, where Λ is a diagonal matrix of eigenvalues of C, Q is a unitary matrix and * denotes the Hermitian

transpose. The entries of Q are given by the discrete Fourier transform. Computation of $C^{1/2}$, which is useful for both simulation and evaluation of the density of Y, is then straightforward using the fact that $C^{1/2} = Q\Lambda^{1/2}Q^*$.

3.2. The Metropolis-adjusted Langevin algorithm

Monte Carlo simulation from $\pi(Y_{t_1:t_2}|X_{t_1:t_2})$ is made more efficient by working with a linear transformation of Y, partially determined by the matrix C as described below. The **lgcp** package returns results pertaining to Y on a grid of size $M \times N \equiv 2^m \times 2^n$ for positive integers m and n, which is extended to a grid of size $2M \times 2N$ for computation (Møller *et al.* 1998). Writing $\Gamma_t = \Lambda^{-1/2}Q(Y_t - \mu)$, the target of interest is given by

$$\pi(\Gamma_{t_1:t_2}|X_{t_1:t_2}) \propto \left[\prod_{t=t_1}^{t_2} \pi(X_t|Y_t)\right]\left[\pi(\Gamma_{t_1})\prod_{t=t_1+1}^{t_2} \pi(\Gamma_t|\Gamma_{t-1})\right] \tag{7}$$

where the first term on the right hand side of Equation 7 corresponds to the first bracketed term on the right hand side of Equation 6 and the second bracketed term is the joint density, $\pi(\Gamma_{t_1:t_2})$, which so-factorises due to the Markov property. Since Y, and hence Γ, is an Ornstein-Uhlenbeck process in time, the transition density, $\pi(\Gamma_t|\Gamma_{t-1})$, has an explicit expression as a Gaussian density; see Brix and Diggle (2001).

Since the gradient of the transition density can also be written down explicitly, a natural and efficient MCMC method for sampling from the predictive density of interest (Equation 7), is a Metropolis-Hastings algorithm with a Langevin-type proposal (Roberts and Tweedie 1996),

$$q(\Gamma, \Gamma') = \mathrm{N}\left(\Gamma'; \Gamma + \frac{1}{2}\nabla \log\{\pi(\Gamma|X)\}, h^2\mathbb{I}\right)$$

where $\mathrm{N}(y; m, v)$ denotes a Gaussian density with mean m and variance v evaluated at y, $\mathbb{I}$ is the identity matrix and $h > 0$ is a scaling parameter (Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller 1953; Hastings 1970).

Various theoretical results exist concerning the optimal acceptance probability of the MALA (Metropolis-Adjusted Langevin Algorithm) – see Roberts and Rosenthal (1998) and Roberts and Rosenthal (2001) for example. In practical applications, the target acceptance probability is often set to 0.574, which would be approximately optimal for a Gaussian target as the dimension of the problem tends to infinity. An algorithm of Andrieu and Thoms (2008) for the automatic choice of h, so that this acceptance probability is achieved without disturbing the ergodic property of the chain is implemented in **lgcp**.

4. Introducing the lgcp package

4.1. An overview of this section

In this section, we present a brief tour of package functionality by means of a walk-through spatio-temporal example. In Section 4.2, reading in and converting data into the required format for inference is discussed. Section 4.3 addresses the issue of estimating $\lambda(s)$ and $\mu(t)$ using **lgcp**'s built in capabilities; the reader should note at the outset that these two quantities

would normally be estimated via some other means, as will be discussed. In Section 4.4, we discuss parameter estimation: **lgcp** provides simple minimum-contrast procedures for this task. Then in Section 4.5 and 4.6, we detail respectively the setting up and running of the MCMC algorithm, including: specifying the model, adaptive MCMC, computation of Monte Carlo expectations 'on-line' and possible rotation of the observation window. Lastly, in Section 4.7, we show how to: extract information from a finished MCMC run; plot the results; forecast into the future; handle data dumped to disk; obtain MCMC diagnostics and plot exceedance probabilities.

4.2. Reading-in and converting data

The generic data-format of interest is $(x_i, y_i, t_i) : i = 1, ..., n$, where the (x_i, y_i) are the locations and t_i the times of occurrence of events in $W \times (A, B)$, where W is a polygonal observation window and (A, B) the time-interval within which events are observed. In the following example `x`, `y` and `t` are R objects giving the location and time of events and `win` is a **spatstat** object of class `owin` specifying the polygonal observation window (Baddeley and Turner 2005). An example of constructing an appropriate `owin` object from ESRI shapefiles is given in Appendix E.

```
R> data <- cbind(x,y,t)
R> tlim <- c(0,100)
R> win
```

```
window: polygonal boundary
enclosing rectangle: [381.7342, 509.7342] x [64.14505, 192.14505] units
```

The first task for the user is to convert this into a space-time planar point pattern object i.e., one of class `stppp`, provided by **lgcp**. An object of class `stppp` is easily created:

```
R> xyt <- stppp(list(data=data,tlim=tlim,window=win))
```

The second task is to choose a time-scale, and to convert the real-valued times into integer-valued times. These steps are important, as they can affect our ability to estimate the temporal correlation parameter θ, and hence to borrow strength from observations over time: too coarse a discretisation means that data from successive aggregated time points are effectively independent, and the estimation of θ is not possible using the simple methods we suggest in this article. Rescaling time is a simple operation involving multiplying the real-valued `xyt$t` by another real number. The most intuitive way to subsequently convert these rescaled real-valued times into integer-valued times is for the user to manually do this using whatever operations are deemed necessary. Indeed, the safest way of constructing an `stppp` object is to already have time, as well as `tlim` as integer-valued vectors before first defining the `stppp` object, which would render the next step redundant.

For those who prefer an automated approach, the package **lgcp** uses `as.integer` to convert real-valued times into integer values, these real-valued times will be rounded-down to the integer below. The function `integerise.stppp` takes care of this rounding process, and also amends `xyt$tlim`, if necessary.

```
R> xyt <- integerise(xyt)
R> xyt

Space-time point pattern
 planar point pattern: 10069 points
window: polygonal boundary
enclosing rectangle: [381.7342, 509.7342] x [64.14505, 192.14505] units
   Time Window : [ 0 , 99 ]
```

Since in this example, the data were simulated on a continuous time-line from time 0 up to 100, `integerise.stppp` has rounded the upper limit (`xyt$tlim[2]`) down: both `xyt$t` and `xyt$tlim` are now `integer` vector objects. This ensures that subsequent estimates of $\mu(t)$ will be correctly scaled.

4.3. Estimating the spatial and temporal component

There are many ways to estimate the fixed spatial and temporal components of the log-Gaussian Cox process. The fixed spatial component, $\lambda(s)$, represents the spatial intensity of events, averaged over time and scaled to integrate to 1 over the observation window W. In epidemiological settings, this typically corresponds to the spatial distribution of the population at risk, although this information may not be directly available. The fixed temporal component, $\mu(t)$, is the mean number of events in W per unit time. Where the relevant demographic information is unavailable to specify $\lambda(s)$ and $\mu(t)$ directly, **lgcp** provides basic functionality to estimate them from the data.

A user may wish to specify parametric models for λ and μ, perhaps estimating them by using spatially and temporally referenced covariates. By outputting the results of such parametric models onto a grid in space, which is the easiest way to handle the fixed spatial component (by creating a `spatialAtRisk` object directly, see `?spatialAtRisk.fromXYZ`), or as a function on the real line, in the case of the fixed temporal component, the user is able to utilise a wide variety of other statistical models in other R packages to estimate λ and μ.

The function λ in particular will sometimes have the interpretation of something proportional to population density, and hence may not have been estimated by a formal statistical procedure. Other times, spatially-resolved covariate data will be available along with a population offset, and it will be of interest to adjust the risk surface to account for this information. In Appendix A, we give an example where both the fixed spatial and fixed temporal components are estimated using covariate information.

The important points to note in the estimation of $\lambda(s)$ and $\mu(t)$ are: (1) it is preferable to estimate $\lambda(s)$ using population and possibly covariate information (as described above and in Appendix A), rather than via a density estimate constructed from the cases; (2) the estimate of $\lambda(s)$ should not be zero in any area, or more specifically grid cell, where there are events in the dataset – this can happen when using external sources of information for population density; (3) it is preferable to estimate $\mu(t)$ using a parametric model, rather than via a non-parametric estimate. In the example to follow, we chose to estimate λ and μ non-parametrically for illustrative purposes only.

The package **lgcp** uses bi-linear interpolation (via the **spatstat** function `interp.im`) to transfer a user supplied $\lambda(s)$ onto the FFT grid ready for analysis. Therefore, for best results, the user

should output their estimate of λ onto the grid that will be used in the MALA algorithm. As we will be using a 2km grid in the example, this can be computed with,

```
R> OW <- selectObsWindow(xyt,cellwidth=2)
```

the objects `OW$xvals` and `OW$yvals` now contain the grid on which the fitting of λ should be projected for best results.

The function `lambdaEst` is an interactive implementation of a kernel method for estimating $\lambda(s)$ from the data as in the following example.

```
R> den <- lambdaEst(xyt,axes=TRUE)
R> plot(den)
```

This calls an **rpanel** tool (Bowman *et al.* 2007) for estimating λ (see Figure 1); once the user is happy with the result, clicking on "OK" closes the panel and the kernel density estimate is stored in the R object `den` of class `im` (a **spatstat** pixel image object). The estimate of λ can then be plotted in the usual way. The parameters `bandwidth` and `adjust` in this graphical user interface (GUI) relate to the arguments from the **spatstat** function `density.ppp`; the former corresponds to the argument `sigma` and the latter to the argument of the same name.

The function `lambdaEst` is built directly on the `density.ppp` function and as such, implements a bivariate Gaussian smoothing kernel. The bandwidth is initially that which is automatically chosen by the default method of `density.ppp`. Since `image` plots of these kernel density estimates may not have appropriate colour scales, the ability to adjust this is given with the slider 'colour adjustment'. With colour adjustment set to 1, the default `image.plot` for the equivalent pixel image object is shown and for values less than 1, the colour scheme is more spread out, allowing the user to get a better feel for the density that is being fitted. The colour adjustment parameter raises each cell-wise entry to that power. NOTE: colour adjustment does not affect the returned density and the user should be aware that the returned density will visually appear exactly as displayed when colour adjustment is set equal to 1. `lambdaEst` *does not* output to the FFT grid used in the MALA algorithm.

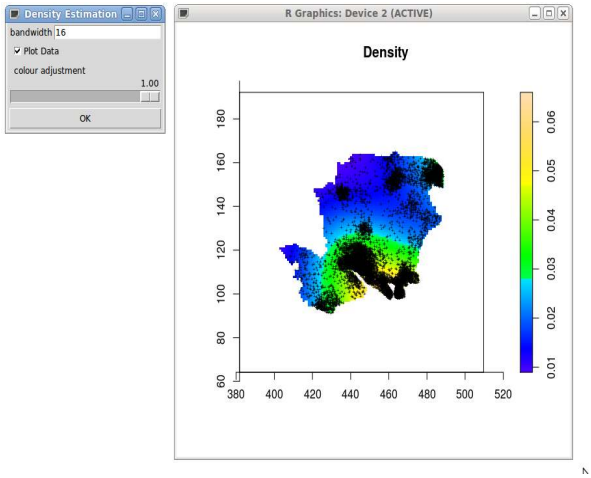

Figure 1: Choosing a kernel density estimate of $\lambda(s)$.

The **lgcp** package provides methods for coercing pixel-images like `den` to objects of class `spatialAtRisk`, which can then be used in parameter estimation and in the MALA algorithm to be discussed later; further useful information on the `spatialAtRisk` class is provided in Appendix D.

```
R> sar <- spatialAtRisk(den)

SpatialAtRisk object
   X range : [382.2342066569,509.2342066569]
   Y range : [64.645045372051,191.645045372051]
   dim     : 128 x 128
```

For the temporal component, $\mu(t)$, the user must provide an object that can be coerced into one of class `temporalAtRisk`.

Objects of class `temporalAtRisk` are non-negative functions of time over an observation time-window of interest, which must be the same as the time-window of the `stppp` data object, `xyt`. In some applications (Diggle *et al.* 2005), $\mu(t)$ might represent the fitted values of a parametric model for the case counts over time. As it is not possible to provide generic functionality for parametric $\mu(t)$, a simple non-parametric estimate of μ can be generated using the function `muEst`:

```
R> mut1 <- muEst(xyt)
R> mut1

temporalAtRisk object
function(t){
        if (!any(as.integer(t)==tvec)){
            return(NA)
        }
        return(obj[which(as.integer(t)==tvec)] * scale)
    }
<environment: 0xbb68b08>
attr(,"tlim")
[1]  0 99
attr(,"class")
[1] "temporalAtRisk" "function"
   Time Window : [ 0 , 99 ]
```

In order to retain positivity, `muEst` fits a locally-weighted polynomial regression estimate (the R function `lowess`) to the square root of the interval counts and returns the square of this smoothed estimate (see Figure 2). The amount of smoothing is controlled by the `lowess` argument `f` which specifies the proportion of points in the plot which influence the smoothed estimate at each value (see `?lowess`), for example `muEst(xyt,f=0.1)`. If the user wishes to specify a constant time-trend, $\mu(t) = \mu$, the command

```
R> mut <- constantInTime(xyt)
```

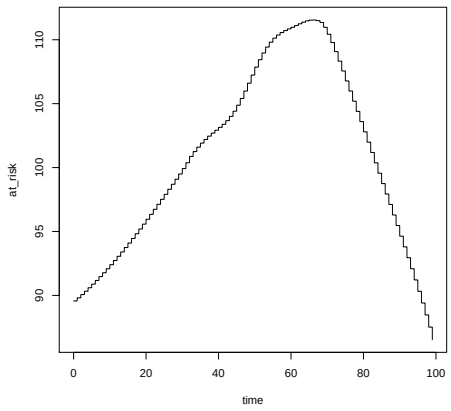

Figure 2: Estimating $\mu(t)$: the output from `plot(mut1)`.

returns the appropriate `temporalAtRisk` object, correctly scaled as in Equation 4. The fixed temporal component can be also be supplied either as a vector or as a function that is automatically coerced to a `temporalAtRisk` object and scaled to achieve the condition in Equation 4.

4.4. Estimating parameters

After estimating $\lambda(s)$ and $\mu(t)$, the next step in the analysis is to estimate the covariance parameters of the process $\mathcal{Y}$. The **lgcp** package provides basic moment-based methods for this in the form of **rpanel** GUIs that allow the user to choose σ, ϕ and θ by eye (Bowman *et al.* 2007). Parameter estimation by eye is both fast and reasonably robust and moreover emphasises the fact that the underlying methods are *ad hoc*. As mentioned above, it is possible to implement principled Bayesian parameter estimation for this model by integrating over the discretised latent-field, Y; this is a planned extension to the package (see Section 5).

The spatial correlation parameters σ and ϕ can be estimated either from the inhomogeneous pair correlation function, g, or the inhomogeneous K function (Baddeley *et al.* 2000). Following Brix and Diggle (2001) and Diggle *et al.* (2005), the corresponding functions in **lgcp** estimate versions of these two functions by averaging temporally localised versions. The respective commands for doing so are:

```
R> gin <- ginhomAverage(xyt,spatial.intensity=sar,temporal.intensity=mut)
R> kin <- KinhomAverage(xyt,spatial.intensity=sar,temporal.intensity=mut)
```

The choice of smoothing parameter (`bw`) in the inhomogeneous pair correlation function is important. By default, this is estimated automatically by the **spatstat** function `pcfinhom` (on which `ginhomAverage` is built), but it is also possible to set the amount of smoothing manually by supplying a value for the argument `bw`, which is then passed directly to `pcfinhom`. The parameters of the spatial correlation function are then estimated using either of the following:

```
R> sigmaphi1 <- spatialparsEst(gin,sigma.range=c(0,10),phi.range=c(0,10),
      spatial.covmodel="exponential")
```

```
R> sigmaphi2 <- spatialparsEst(kin,sigma.range=c(0,10),phi.range=c(0,10),
      spatial.covmodel="exponential")
```

These invoke another call to **rpanel**, which produces the plots in Figure 3. The user's task is to match the orange theoretical function with the black empirical counterpart. To get a good

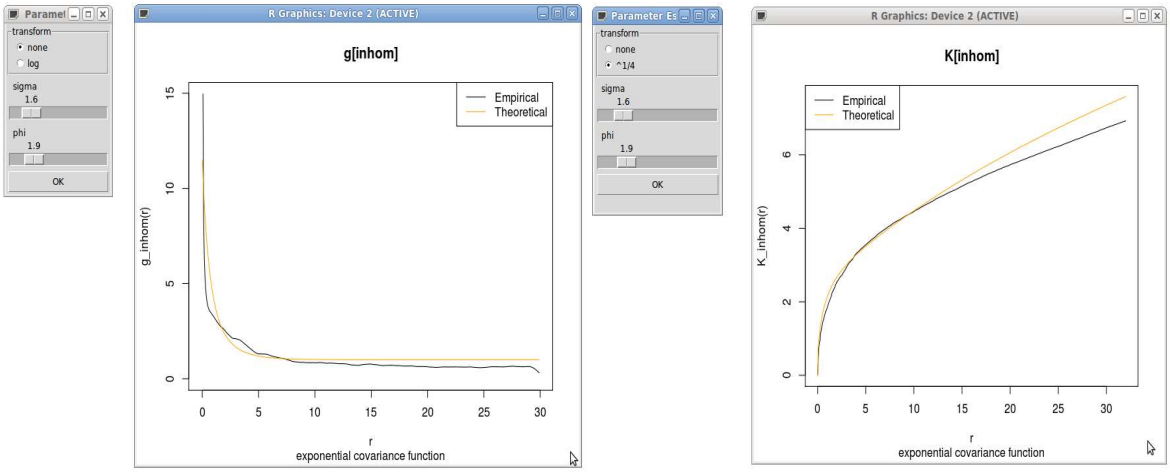

Figure 3: Estimating σ and ϕ: left via the pair correlation function and right via the inhomogeneous K function.

choice of parameters, it is likely that the routine will have to be called several times in order to refine the choice of `sigma.range` and `phi.range`. The list of correlation functions available to the user is given under the help file for the function `CovarianceFct` from the **RandomFields** package. For certain classes of covariance function, for example the Matérn or Whittle families, additional parameters are specified by the user via the `covpars` argument and supplied to `spatialParsEst` in the same order as they appear in the help file for `CovarianceFct`. These additional parameters are treated as known constants, and not estimated via a formal nor informal procedure. One reason for this is because some of these parameters are notoriously difficult to estimate, for example, the parameter ν in the Matérn family. A recommended strategy in these cases is to choose between a discrete set of candidate values for the parameter of interest. For example, in the Matérn family the integer part of ν gives the number of times the underlying Gaussian process is mean-square differentiable. The resulting estimated parameters are returned in list objects (e.g., `sigmaphi1` or `sigmaphi2`) with `sigmaphi1$sigma` and `sigmaphi1$phi` returning the required values of σ and ϕ. In the code below, these values have been input as respectively 1.6 and 1.9 as estimated above. The user has additional control over the minimum contrast estimation, for example the range of evaluation, though sensible defaults are provided automatically by the embedded **spatstat** functions. The initial parameter values appearing in the GUIs are obtained through a simple weighted least-squares optimisation procedure, but note that this is not robust, and the user has the option to turn this functionality off via `guess=FALSE` in the argument list to `spatialParsEst`.

The temporal correlation parameter, θ, can be estimated using the function `thetaEst`; this requires σ, ϕ and $\mu(t)$ to have been estimated beforehand. For example, the call

```
R> theta <- thetaEst(xyt,spatial.intensity=sar,
                     temporal.intensity=mut,sigma=1.6,phi=1.9)
```

gives the GUI shown in Figure 4. Note that again, in the code below the estimated value of 1.4 has been input manually.

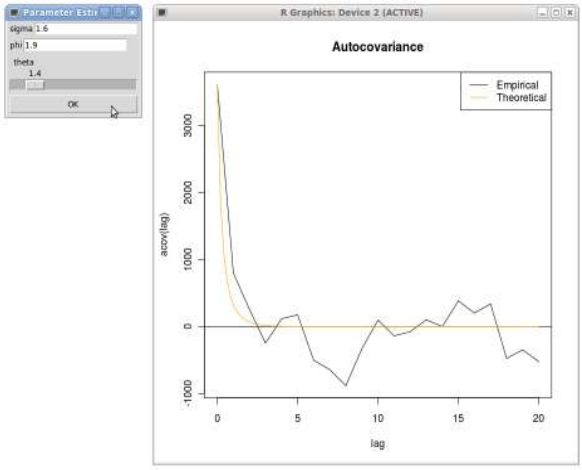

Figure 4: Estimating θ.

4.5. The commands `lgcpPredict` and `lgcpPredictSpatial`

The main functions in the **lgcp** package are `lgcpPredict` and `lgcpPredictSpatial`. In this article, we focus on **lgcp**'s spatio-temporal functionality and hence `lgcpPredict`, the rationale being that the 'purely spatial' provided by `lgcpPredictSpatial` is straightforward to understand once familiar with the former. A comparative evaluation of **lgcp**'s spatial capabilities, together with a definition of the statistical model in this case is given in Taylor and Diggle (2012).

This uses an MCMC method to produce samples and summary statistics from the predictive distribution of the discretised process Y, treating previously obtained estimates of $\lambda(s)$, $\mu(t)$, and the covariance parameters as known quantities. The MCMC algorithm is invoked by the command `lgcpPredict`, whose arguments are as follows:

```
R> args(lgcpPredict)

function (xyt, T, laglength, model.parameters = lgcppars(),
    spatial.covmodel = "exponential",
    covpars = c(), cellwidth = NULL, gridsize = NULL, spatial.intensity,
    temporal.intensity, mcmc.control, output.control = setoutput(),
    missing.data.areas = NULL, autorotate = FALSE, gradtrunc = NULL,
    ext = 2)
```

A pre-emptive note on computation times

With the options set as below: working on a 128×128 FFT grid with six time-points worth of data, and with 120,000 iterations of the MCMC algorithm, the total computation time is around $2\frac{3}{4}$ hours on a 3.2GHz Intel(R) Core(TM) i5 desktop PC with 4Gb RAM. Computation times increase approximately linearly with the addition of additional time-points.

In Taylor and Diggle (2012), using the same PC, the authors compare computation times for the function `lgcpPredictSpatial` with a computationally equivalent implementation from the **INLA** package, and they found that 100,000 iterations on a 128×128 grid (which gave good predictive performance) ran in around 20 minutes, similar to a call to `inla` with the option `strategy="laplace"` outputting results on the same predictive grid as MCMC. The results presented in Taylor and Diggle (2012) question the commonly held notion that for this class of problems, MCMC is slow, hard to tune and mixes poorly.

Data and model specification

The argument `xyt` is the `stppp` object that contains the data, `T` is the time-point of interest for prediction (c.f., the time t_2 in Section 3) and `laglength` tells the algorithm the number of previous time-points whose data should be included, that is the time-interval [`T-laglength`,`T`].

Model parameters are set using the `model.parameters` argument; for example,

```
R> lgcppars(sigma=1.6,phi=1.9,theta=1.4)
```

has the obvious interpretation. The mean of the latent field Y is set to $-\sigma^2/2$ by default. The spatial covariance model and any additional parameters are specified using the `spatial.covmodel` and `covpars` arguments; these may come from any of the compatible covariance functions detailed in `?CovarianceFct` from the **RandomFields** package. The physical dimensions of the grid cells can be set using either the `cellwidth` or `gridsize` arguments, the second of which gives the number of cells in the x and y directions (these numbers are automatically extended to be a power of two for the fast-Fourier transform). The `spatial.intensity` and `temporal.intensity` arguments specify the previously obtained estimates of $\lambda(s)$ and $\mu(t)$, respectively.

It remains to set the MCMC parameters and output controls; these will now be discussed.

Controlling MALA and performing adaptive MCMC

The `mcmc.control` argument of `lgcpPredict` specifies the MCMC implementation and is set using the `mcmcpars` function:

```
R> args(mcmcpars)

function (mala.length, burnin, retain, inits = NULL, adaptivescheme)
```

Here, `mala.length` is the number of iterations to perform, `burnin` is the number of iterations to throw away at the start and `retain` is the frequency at which to store or perform computations; for example, `retain=10` performs an action every 10th iteration. The optional argument `inits` can be used to set initial values of Γ for the algorithm, and is intended for advanced use. The initial values are stored in a list object of length `laglength`+1, each element being a matrix of dimension $2M \times 2N$. For MCMC diagnostics, discussed in the sequel, the user must dump information from the chain to disk using the `dump2dir`, discussed below.

The MALA proposal tuning parameter h in Section 3.2 must also be chosen. The most straightforward way to do this is to set `adaptivescheme=constanth(0.001)`, which gives $h = 0.001$. Without a lengthy tuning process, the value of h that optimises the mixing of the

algorithm is not known. One solution to the problem of having to choose a scaling parameter from pilot runs is to use adaptive MCMC (Roberts and Rosenthal 2009; Andrieu and Thoms 2008). Adaptive MCMC algorithms use information from the realisation of an MCMC chain to make adjustments to the proposal kernel. The Markov property is therefore no longer satisfied and some care must be taken to ensure that the correct ergodic distribution is preserved. An elegant method, introduced by Andrieu and Thoms (2008) uses a Robbins-Munro stochastic approximation update to adapt the tuning parameter of the proposal kernel (Robbins and Munro 1951); see below for suggested parameter values for this scheme. The idea is to update the tuning parameter at each iteration of the sampler according to the iterative scheme,

$$h^{(i+1)} = h^{(i)} + \eta^{(i+1)}(\alpha^{(i)} - \alpha_{\text{opt}}), \tag{8}$$

where $h(i)$ and $\alpha^{(i)}$ are the tuning parameter and acceptance probability at iteration i and α_{opt} is the target acceptance probability. For Gaussian targets, and in the limit as the dimension of the problem tends to infinity, an appropriate target acceptance probability for MALA algorithms is $\alpha_{\text{opt}} = 0.574$ (Roberts and Rosenthal 2001). The sequence $\{\eta^{(i)}\}$ is chosen so that $\sum_{i=1}^{\infty} \eta^{(i)}$ is infinite but for some $\epsilon > 0$, $\sum_{i=1}^{\infty} \left(\eta^{(i)}\right)^{1+\epsilon}$ is finite. These two conditions ensure that any value of h can be reached, but in a way that maintains the ergodic behaviour of the chain. One class of sequences with this property is,

$$\eta^{(i)} = \frac{C}{i^{\zeta}}, \tag{9}$$

where $\zeta \in (0, 1]$ and $C > 0$ (Andrieu and Thoms 2008).

The tuning constants for this algorithm are set with the function `andrieuthomsh`.

```
R> args(andrieuthomsh)
```

```
function (inith, alpha, C, targetacceptance = 0.574)
```

In the above, `inith` is the initial value of h and the remaining arguments correspond to their counterparts in the text above. In our experience of using this algorithm, we have found that it works well. We have used the values `inith=1`, `alpha=0.5` and `C=1` successfully across a variety of scenarios without any difficulty, but cannot comment as to whether these values will work well in general.

The advanced user can also write their own adaptive scheme, detailed examples of which are provided in Appendix F. Briefly, writing an adaptive MCMC scheme involves writing two functions to tell R how to initialise and update the values of h. This may sound simple, but it is crucial that these functions preserve the correct ergodic distribution of the MCMC chain, an appreciation of these subtleties is **essential** before any attempt is made to code such schemes.

Specifying output

By default, `lgcpPredict` computes the Monte Carlo mean and variance of Y and the mean and variance of $\exp\{Y\}$ (the relative risk) for each of the grid cells and time intervals of interest. Additional storage and on-line computations are specified by the `output.control` argument and the `setoutput` function:

```
R> args(setoutput)

function (gridfunction = NULL, gridmeans = NULL)
```

The option `gridfunction` is used to declare general operations to be performed during simulation (for example, dumping the simulated Ys to disk), whilst user-defined Monte Carlo averages are computed using `gridmeans`. A complete run of the MALA chain can be saved using the `dump2dir` function:

```
R> args(dump2dir)

function (dirname, lastonly = TRUE, forceSave = FALSE)
```

The user supplies a character string, `dirname`, giving the name of a directory in which the results are to be saved. The other arguments to `dump2dir` are, respectively, an option to save only the last grid (i.e., the time `T` grid) and to bypass a safety message that would otherwise be displayed when `dump2dir` is invoked. The safety message warns the user of disk space requirements for saving. For example, on a 128×128 output grid using 5 days of data, 1000 simulations from the MALA will take up approximately 625 megabytes.

The option `lastonly` in the functions `dump2dir` and `MonteCarloAverage` (see below) is set to `TRUE` by default, this means that only information from the last time point is saved or manipulated. The main reason for doing this is that by definition of the model, statistical interest is focused on inference for the last time point, `T`. It is assumed that the information from lagged time points contributes to predictions at time `T`, but that information from further in the past has only negligible effect. Setting `lastonly=TRUE` also has the advantage that the algorithm runs faster, but should it be of interest to examine predictions at lagged time points, the option should be set to `FALSE`.

Another option is to compute Monte Carlo expectations,

$$\mathsf{E}_{\pi(Y_{t_1:t_2}|X_{t_1:t_2})}[g(Y_{t_1:t_2})] = \int_W g(Y_{t_1:t_2})\pi(Y_{t_1:t_2}|X_{t_1:t_2})\mathrm{d}Y_{t_1:t_2}, \tag{10}$$

$$\approx \frac{1}{n}\sum_{i=1}^{n} g(Y_{t_1:t_2}^{(i)}) \tag{11}$$

where g is a function of interest, $Y_{t_1:t_2}^{(i)}$ is the ith retained sample from the target and n is the total number of retained iterations. For example, to compute the mean of $Y_{t_1:t_2}$, set $g(Y_{t_1:t_2}) = Y_{t_1:t_2}$. The output from such a Monte Carlo average would then be a set of $t_2 - t_1$ grids, each cell of which is equal to the mean over all retained iterations of the algorithm. In the context of setting up the `gridmeans` option to compute the Monte Carlo mean, the user would define a function `g` as

```
R> gfun <- function(Y){
       return(Y)
   }
```

and input this to the MALA run using the function `MonteCarloAverage`,

```
R> args(MonteCarloAverage)

function (funlist, lastonly = TRUE)
```

Here, `funlist` is either a list or a character vector giving the names of the function(s) g. The specific syntax for the example above would be a call of the form `MonteCarloAverage("gfun")`. The functions of interest (e.g., `gfun` above) are assumed to act on each of the individual grids, Y_{t_i}, and return a grid of the same dimension.

A second example arises in epidemiological studies where it is of clinical interest to know whether, at any location s, the ratio of current to expected risk exceeded a pre-specified intervention threshold; see, for example, Diggle *et al.* (2005), where real-time predictions of relative risk are presented as maps of exceedance probabilities, $\mathrm{P}\{\exp(Y_t) > k | X_{1:t}\}$ for a pre-specified threshold k. Any such exceedance probability can be expressed as an expectation,

$$\mathsf{P}[\exp(Y_{t_1:t_2}) > k] = \mathsf{E}_{\pi(Y_{t_1:t_2}|X_{t_1:t_2})}\{\mathbb{I}[\exp(Y_{t_1:t_2}) > k]\} = \frac{1}{n}\sum_{i=1}^{n}\mathbb{I}[\exp(Y^{(i)}_{t_1:t_2}) > k],$$

where $\mathbb{I}$ is the indicator function, and a Monte Carlo approximation can therefore be computed on-line using `MonteCarloAverage`.

The corresponding function g is

$$g(Y_{t_1:t_2}) = \mathbb{I}[\exp(Y_{t_1:t_2}) > k].$$

Exceedance probabilities are made available directly within **lgcp** by the function `exceedProbs`.

To make use of this facility, the user specifies the thresholds of interest, for example 1.5, 2 and 3, then creates a function to compute the required exceedances:

```
R> exceed <- exceedProbs(c(1.5,2,3))
```

The object `exceed` is now a function that returns the exceedance probabilities as an array object of dimension $M \times N \times 3$. This function can be passed through to the `gridmeans` option, together with the previously defined `gfun`, via `gridmeans=MonteCarloAverage(c("gfun","exceed")`. The `lgcpPredict` function then returns point-wise predictive means and three sets of exceedance probabilities. Note that, the example function `gfun` is included for illustrative purposes only and is in fact redundant, as `lgcpPredict` automatically returns the predictive mean (as well as the variance) of Y.

Rotation

Testing whether estimation can proceed more efficiently in a rotated space is described in detail in Appendix B. Note that if the data and observation window are rotated, then λ must also be rotated to retain compatibility. If λ was estimated in the original frame of reference and `autorotate=TRUE`, then `lgcpPredict` will automatically rotate λ if it is computationally worthwhile to do so. For λ specified as a grid, either directly or via an object of class `im`, then a small amount of information loss occurs in the rotation because the square cells in the original orientation become misaligned with the axes in the rotated space and vice-versa. If λ is specified by a continuous function, then no such loss occurs.

Gradient truncation

One undesirable property of the Metropolis-adjusted Langevin algorithm is that the chain is prone to taking very long excursions from the mode of the target; this behaviour can have a detrimental effect on the mixing of the chain and consequently on any results. The tendency to make long excursions is caused by instability in the computation of the gradient vector, but the issue is relatively straightforward to rectify without affecting convergence properties (Møller *et al.* 1998). The key is to truncate the gradient vector if it becomes too large. If `gradtrunc=NULL`, then an appropriate truncation is automatically selected by the code. With `gradtrunc=Inf`, no gradient truncation occurs.

As far as the authors are aware, there are no published guidelines for selecting this truncation parameter. The current version of the `lgcp` package uses the maximum achieved gradient over a set of 100 independent realisations of $\Gamma_{t_1:t_2}$.

4.6. Running

When all of the above options have been specified, the MALA algorithm can be called as follows:

```
R> tmpdr <- tempdir()
R> lg <- lgcpPredict(xyt=xyt,
                     T=50,
                     laglength=5,
                     model.parameters=lgcppars(sigma=1.6,phi=1.9,theta=1.4),
                     cellwidth=2,
                     spatial.intensity=sar,
                     temporal.intensity=mut,
                     mcmc.control=mcmcpars(mala.length=120000,burnin=20000,
                        retain=100,
                        adaptivescheme=andrieuthomsh(inith=1,alpha=0.5,C=1,
                        targetacceptance=0.574)),
                     output.control=setoutput(gridfunction=
                        dump2dir(dirname=tmpdr),
                        gridmeans=MonteCarloAverage("exceed")))
```

The above call assumes that the spatial covariance model is exponential, that no rotation is to be performed and that the user wishes to have `lgcpPredict` compute an appropriate gradient truncation automatically. The arguments `spatial.intensity` and `temporal.intensity` relate to the spatial and temporal intensities, estimated in Section 4.3; note that the chosen temporal model is constant in time. Recall that the option `lastonly` is by default set to `TRUE` in both `MonteCarloAverage` and `dump2dir`.

The simulated example uses data from times 45 to 50 inclusive, 120,000 iterations, of which the first 20,000 are treated as burn-in, and retains every 100th sample. The observation window is approximately 100km square, so the specified cell width of 2km (given in the same units as used in the object `xyt`) gives an output grid of size 64×64, i.e., computation is carried out on a 128×128 grid. The complete run is saved to disk and exceedance probabilities are computed for the last time-point only.

During simulation, a progress bar is displayed giving the percentage of iterations completed.

It is possible at the start of a run that the user may be confronted by a warning message like `Time 48:  5 data points lost due to discretisation`. This means that there are points close to the edge of the polygonal observation window that lie outside the computational grid. Critical information is only lost in any subsequent conditional predictions if we were specifically interested in behaviour close to these areas, in which case there are two options. The easiest solution assuming that prediction in these areas is important, but bears additional computational expense and may not fix the warnings, is to use a finer grid resolution. The package **lgcp** provides facilities for the user to design both the observation window as well as the `spatialAtRisk` object, and therefore loss of information at the edges can be adjusted for manually by using an appropriate choice of these two objects.

4.7. Post-processing

The stored output `lg` is an object of class `lgcpPredict`. Typing `lg` into the console prints out information about the run:

```
R> lg


lgcpPredict object.

  General Information
  -------------------
      FFT Gridsize: [ 128 , 128 ]

              Data:
       Time |       45       46       47       48       49       50
     Counts |       98      345      106      100       73       67

         Parameters: sigma=1.6, phi=1.9, theta=1.4
    Dump Directory: /tmp/RtmpeaOS3o

     Grid Averages:
         Function Output Class
           exceed        array

         Time taken: 2.77 hours

  MCMC Information
  ----------------
    Number Iterations: 120000
              Burn-in: 20000
             Thinning: 100
      Mean Acceptance: 0.574
      Adaptive Scheme: andrieuthomsh
               Last h: 0.00904236148499419
```

Information returned includes the FFT grid size used in computation; the count data for each day; the parameters used; the directory, if specified, to which the simulation was dumped; a list of `MonteCarloAverage` functions together with the R class of their returned values; the time taken to do the simulation; and information on the MCMC run.

Extracting information

The cell-wise mean and variance of Y computed via Monte Carlo can always be extracted using `meanfield(lg)` and `varfield(lg)`, respectively. The calls `rr(lg)`, `serr(lg)`, `intens(lg)` and `seintens(lg)` return respectively the Monte Carlo mean relative risk (the mean of $\exp\{Y\}$), the standard error of the relative risk, the estimated cell-wise mean Poisson intensity and the standard error of the Poisson intensity. The x and y coordinates for the grid output are obtained via `xvals(lg)` and `yvals(lg)`. The returned object from calls like `meanfield`, `varfield` or `intens` are objects of class `lgcpgrid`, so the command,

```
R> intensity <- intens(lg)
```

creates an `lgcpgrid` object containing the mean Poisson intensities. Then, for example `intensity$grid[[1]]` returns the Poisson intensities relating to the first time aggregated point, with an appropriate (x, y) grid available using `xvals(lg)` and `yvals(lg)`. Similarly, `intensity$grid[[2]]` returns the intensities from the second aggregated time point. `lgcpPredict` not only produces predictions for grid locations inside the observation window, but also it produces predictions for cells outside the observation window too. This is because we define $\lambda(s) = 0$ for $s \notin W$: we expect no counts, and observe no counts in these cells. The values returned by `intensity$grid[[1]]` are therefore stored in a rectangular matrix. To see which of these cells lies inside the observation window, use the following command,

```
R> fftgr <- discreteWindow(lg)
```

which returns a logical matrix of the same dimension as `intensity$grid[[1]]`. These commands give the user the freedom to manipulate the outputs from an MCMC run and produce their own plots, for example. Should the user prefer `list` or `array` versions of the `lgcpgrid` objects, conversion is possible via the functions `as.list` and `as.array`.

In geographic information system (GIS) applications, where the discrete grid on which Monte Carlo computation is performed pertains to a geographical area, **lgcp** provides methods for coercing `lgcpgrid` objects into `raster` objects (Hijmans and van Etten 2012) and also `SpatialPixelsDataFrame` objects via for example:

```
R> rastlg <- raster(intens(lg))
R> spdflg <- as.SpatialPixelsDataFrame(rr(lg))
```

The resulting objects, respectively a `raster` image and a `SpatialPixelsDataFrame`, can then be exported and viewed in GIS software.

If invoked, the commands `gridfun(lg)` and `gridav(lg)` return respectively the `gridfunction` and `gridmeans` options of the `setoutput` argument of the `lgcpPredict` function, whilst `window(lg)` returns the observation window.

Note that the structure produced by `gav <- gridav(lg)` is a `list` of length 2. The first element of `gav`, retrieved with `gav$names`, is a list of the function names given in the call

to `MonteCarloAverage`. The second element, `gav$output`, is a list of the function outputs; the ith element in the list being the output from the function corresponding to the ith element of `gav$names`. To return the output for a specific function, use the syntax `gridav(lg,fun="exceed")`, which in this case returns the exceedance probabilities, for example.

Plotting

Plots of the Monte Carlo mean relative risk and standard errors can be obtained with the commands:

```
R> plot(lg,xlab="x coordinate",ylab="y coordinate")
R> plot(lg,type="serr",xlab="x coordinate",ylab="y coordinate")
```

These commands produce a series of plots corresponding to each time step under consideration; the plots shown in Figure 5 are from the last time step, time 50.

To plot the mean Poisson intensity instead of the relative risk, the optional argument `type` can be set in the above:

```
R> plot(lg,type="intensity",xlab="x coordinate",ylab="y coordinate")
```

The cases for each time step are also plotted by default.

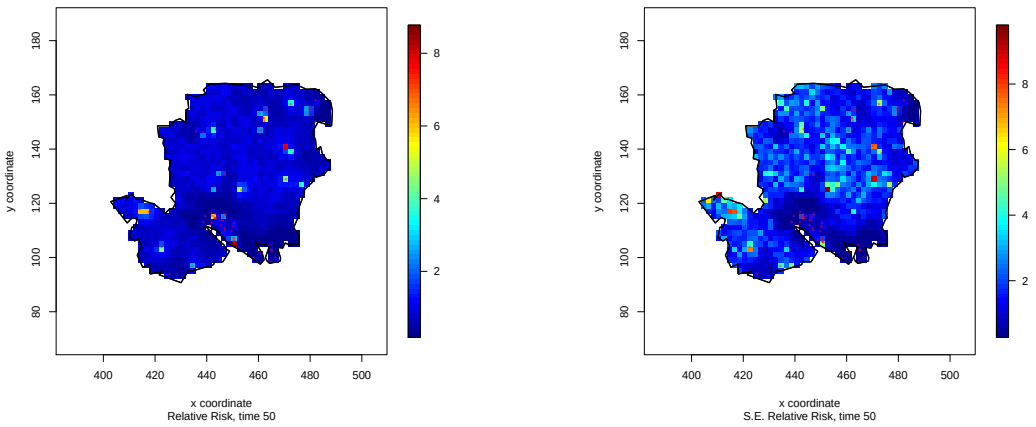

Figure 5: Plots of the Monte Carlo mean relative risk (left) and associated standard errors (right).

It was also suggested by an anonymous referee, that a user could make use of the **spacetime** and **xts** packages for plotting and manipulating data and model outputs from **lgcp** (Pebesma 2012; Ryan and Ulrich 2012).

Forecasting

It is of statistical and epidemiological interest to be able to forecast beyond the time frame of an analysis, that is to be able to forecast the Poisson intensity,

$$A\lambda(s)\mu(t_2+k)\exp\{Y(s,t_2+k)\}$$

where A is the cell area. The package **lgcp** provides functionality to be able to construct estimates of this together with its approximate variance via the function `lgcpForecast`.

Our choice of modelling framework implies that $\{Y(\cdot, t)\}$ is an Ornstein-Uhlenbeck process (Brix and Diggle 2001). Hence,

$$Y(\cdot, t_2 + k) \sim \mathrm{N}[\xi(k)Y(\cdot, t_2) + (1 - \xi(k))\mu, (1 - \xi(k))^2\Sigma], \tag{12}$$

where $\xi(k) = \exp(-\theta k)$. Our task is to infer the forecast distribution of $Y(\cdot, t_2 + k)$ given the observed data $X_{t_1:t_2}$. The conditional independence properties of the model imply,

$$\pi[Y(\cdot, t_2 + k)|X_{t_1:t_2}] = \int \pi[Y(\cdot, t_2 + k)|Y(\cdot, t_2)]\pi[Y(\cdot, t_2)|X_{t_1:t_2}]\mathrm{d}Y(\cdot, t_2).$$

The mean and variance can now be derived as,

$$\begin{aligned} \mathsf{E}[Y(\cdot, t_2 + k)|X_{t_1:t_2}] &= \xi(k)\mathsf{E}[Y(\cdot, t_2)|X_{t_1:t_2}] + (1 - \xi(k))\mu, \\ \mathsf{Var}[Y(\cdot, t_2 + k)|X_{t_1:t_2}] &= \xi(k)^2\mathsf{Var}\{Y(\cdot, t_2)|X_{t_1:t_2}\} + (1 - \xi(k))^2\Sigma. \end{aligned}$$

Unfortunately, $\mathsf{Var}\{Y(\cdot, t_2)|X_{t_1:t_2}\}$ is unavailable in practice, so we instead replace this by a diagonal approximation, which is returned by default by `lgcpPredict` in `lg$y.var`.

Forecast distributions are returned using the following command:

```
R> fcast <- lgcpForecast(lg,c(51,53,55,60),spatial.intensity=sar,
                                  temporal.intensity=function(x){return(100)})
```

The object `fcast` contains the predicted means and variance of Y, the relative risk and Poisson intensities for times 51, 53, 55 and 60. Note that a `temporal.intensity` object must be provided as the `temporalAtRisk` object used in the MCMC step may not be valid for time points beyond time `T`. In the above example, we assume that for the forecast time-frame there will be 100 cases per time point on average.

We note that should information have been dumped to disk, then it would possible to produce a Monte Carlo estimate of the forecast distribution of Y, the relative risk and Poisson intensities, or indeed any quantity of interest that could be expressed as an expectation. Such a solution would be asymptotically exact (as the number of samples tends to infinity) but computationally intensive to compute. Suppose we wish to estimate $\mathsf{E}\{f[Y(\cdot, t_2 + k)]|X_{t_1:t_2}\}$ for some function, f, the Monte Carlo estimate would be obtained as follows. Let $Y_{t_2}^{(j)}$ denote the jth sample dumped to disk in the NetCDF file (see below). Assuming the chain has achieved stationarity, this can be treated as a draw from $\pi[Y(\cdot, t_2)|X_{t_1:t_2}]$. For each $Y_{t_2}^{(j)}$, produce a draw from $\pi[Y(\cdot, t_2 + k)|X_{t_1:t_2}]$ by sampling $Y_{t_2+k}^{(j)}$ according to Equation 12 – note that `rgauss` can be used to simulate from a $\mathrm{N}(0, \Sigma)$ density. The sample mean of $f(Y_{t_2+k}^{(j)})$ is an unbiased estimate of the quantity of interest, $\mathsf{E}\{f[Y(\cdot, t_2 + k)]|X_{t_1:t_2}\}$.

NetCDF

The **lgcp** package provides functions for accessing and performing computations on MCMC runs dumped to disk. Because this can generate very large files, **lgcp** uses the cross-platform `NetCDF` file format for storage and rapid data access, as provided by the package **ncdf** (Pierce 2011). Access to subsets of these stored data is via a file indexing system, which removes the need to load the complete data into memory.

Subsets of data dumped to disk can be accessed with the `extract` function:

```
R> subsamp <- extract(lg,x=c(4,10),y=c(32,35),t=1,s=-1)
```

which returns an array of dimension $7 \times 4 \times 1 \times 1000$ (recall there were 1000 retained iterations). The arguments `x` and `y` refer to the range of x and y *indices* of the grid of interest whilst `t` specifies the time points of interest. Note, however, that in this example times 45 through 50 were used for prediction, and `t=1` here in fact refers to the sixth of these time-points, i.e., time 50, since the option `lastonly` was set to `TRUE` by default. Finally, `s=-1` stipulates that all simulations are to be returned. More generally, each argument of `extract` can be specified either as a range or set equal to -1, in which case all of the data in that dimension are returned. The `extract` function can also extract MCMC traces from individual cells using, for example, `extract(lg,x=37,y=12,t=1)`.

Should the user wish to extract data from a polygonal sub-region of the observation window, this can be achieved with the command

```
R> subsamp2 <- extract(lg,inWindow=win2,t=1)
```

where `win2` is a polygonal observation window defined below. Here, `win2` had been selected using the following commands:

```
R> plot(window(lg))
R> win2 <- clickpoly(add=TRUE)
```

The first of the above commands plots the observation window, whilst the second is a **spatstat** function for drawing polygonal `owin` objects manually. The user could also specify the `extract` argument `inWindow` directly using a **spatstat** `owin` object.

If the user decides that some other summary than those specified by the `gridmeans` option is of interest, this can easily be computed from the stored data (c.f., Section 4.5.4) The syntax is then slightly different, as in the following example that computes the same exceedances in Section 4.5.4:

```
R> ex <- expectation(obj=lg,fun=exceed)
```

Alternatively, cell-wise quantiles of *functions* of the stored data can also be retrieved and plotted:

```
R> qt <- quantile(lg,c(0.5,0.75,0.9),fun=exp)
R> plot(qt,xlab="X coords",ylab="y coords")
```

As for the extract function above, quantiles can also be computed for smaller spatial observation windows. The indices of any cells of interest in these plots can be retrieved by typing `identify(lg)`. Cells are then selected via left mouse clicks in the graphics device, selection being terminated by a right click.

Lastly, Linux users can benefit from the software Ncview, available from http://meteora.ucsd.edu/~pierce/ncview_home_page.html, which provides fast visualisation of NetCDF files. Figure 7 shows a screen-shot, with the control panel (left), an image of one of the sampled grids (top right) and several MCMC chains (bottom right), which are obtained by clicking on the sampled grids; up to five chains can be displayed at a time. There are

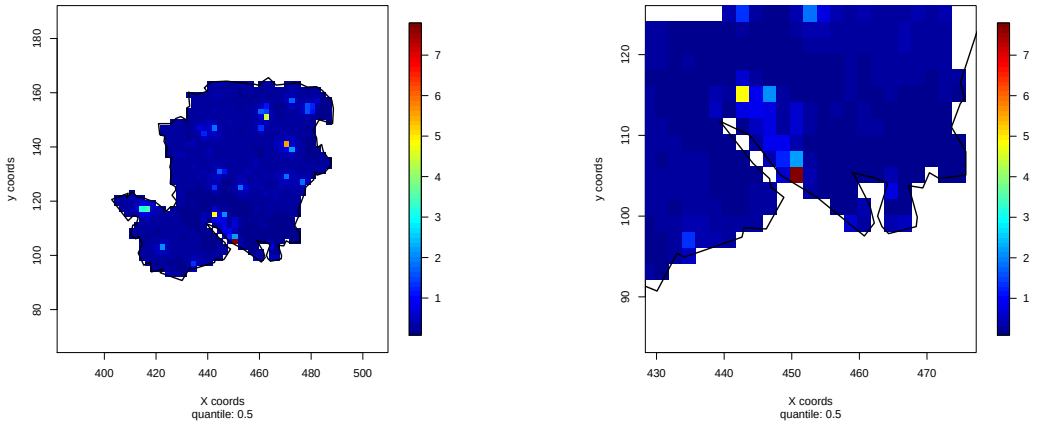

Figure 6: Plot showing the median of relative risk (obtained using `fun=exp` as in the text) computed from the simulation. Left: quantiles computed for whole window. Right: zooming in on the lower area of the map, representing the cities of Southampton and Portsmouth. Greater detail is available by initially performing the simulation on a finer grid.

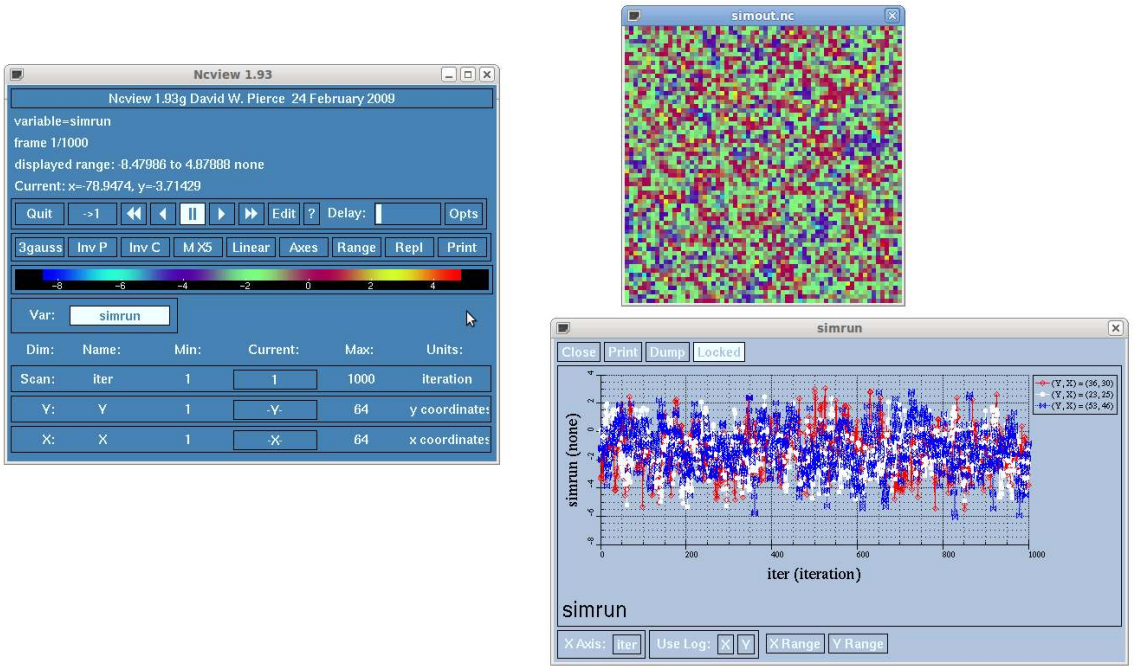

Figure 7: Viewing a MALA run with software `netview`.

equivalent tools for Windows users e.g., Intel® Array Visualiser (http://www.intel.com/cd/software/products/asmo-na/eng/compilers/226277.htm).

MCMC diagnostics

MCMC diagnostics for the chain are based on the full output from data dumped to disk (see Section 4.7.4). The `hvals` command returns the value of h used at each iteration in the algorithm, the left hand plot in Figure 8 shows the values of h for the non-burn-in period of the chain; the adaptive algorithm was initialised with $h = 1$, which very quickly converged to around $h = 0.009$.

```
R> plot(hvals(lg)[20000:120000],type="l",xlab="Iteration",ylab="h")
R> tr <- extract(lg,x=6,y=32,t=1,s=-1)
R> plot(tr,type="l",xlab="Iteration",ylab="Y")
```

A trace plot using data from the $[6, 32]$–cell is shown in the right-hand panel of Figure 8 (recall that `t=1` in the above corresponds to the last time point in this case). Note that this is a trace plot for Y, as opposed to Γ. To plot the auto-correlation function, the standard R function can be used, for example, `acf(tr)` gives the acf of the first extracted chain.

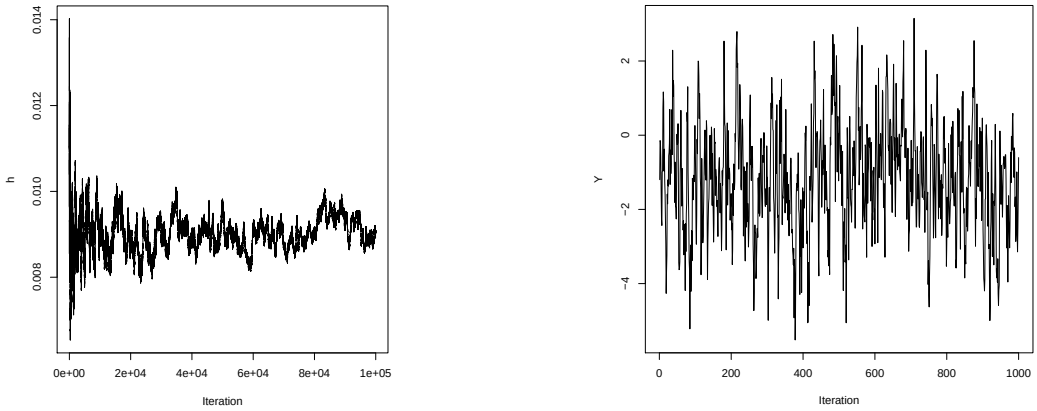

Figure 8: MCMC diagnostic plots. Left: plot of values of h taken by the adaptive algorithm. Right: trace plot of the saved chain from the $[6, 32]$–cell.

The package **lgcp** also provides a function for extracting auto-correlation information from all cells via:

```
R> acor <- autocorr(lg,lags=c(1,5,10),inWindow=NULL)
R> plot(acor,zlim=c(-1,1),xlab="x-coords",ylab="y-coords")
```

In the call to `autocorr` in the above, the option `inWindow=NULL` specifies that the auto-correlation from all cells should be computed, as by default the output is cropped to the observation window. The `plot` command in the second line produces the sequence of plots shown in Figure 9. These plots show that although the lag-1 auto-correlation is quite high, it is quite low across the whole observation window by lag 10.

Figure 9: MCMC diagnostics, left to right: plots of cell-wise lag 1, 5 and 10 auto-correlation.

There are a number of R packages for handling MCMC output, for example **coda** (Plummer, Best, Cowles, and Vines 2006). The package **lgcp** does not provide an interface for working with these other packages because the size of the output dumped to disk is potentially so large. It is possible to interface with these packages manually by extracting smaller subsets of data, and converting them to the appropriate format, but this is not dealt with here.

Plotting exceedance probabilities

Recall that the object `exceed`, defined above, was a function with an attribute giving a vector of thresholds to compute cell-wise exceedance probabilities at each threshold. A plot can be produced either directly from the `lgcpPredict` object,

```
R> plotExceed(lg, fun = "exceed")
```

or, equivalently, from the output of an `expectation` on an object dumped to disk:

```
R> plotExceed(ex[[1]], fun = "exceed",lgcppredict=lg)
```

Recall also that the option `lastonly=TRUE` was selected for `MonteCarloAverage`, hence `ex[[1]]` in the second example above corresponds to the same set of plots as for the first example. The advantage of computing expectations from files dumped to disk is flexibility. For example, if the user now wanted to plot the exceedances for day 49, this is simply achieved by replacing `ex[[6]]` with `ex[[5]]`. Also, exceedances for a new set of thresholds can be computed by creating, for example, a new function by the command `exceed2 <- exceedProbs(c(2.3,4))`. An example of the resulting output is given in Figure 10.

5. Future extensions

This article has described how **lgcp** may be used to fully specify, fit, and simulate from, i.e., predict conditional on observed data, a spatio-temporal log-Gaussian Cox process on $\mathbb{R}^2$. The package **lgcp** provides a substantial volume of novel code, such as access to fast-Fourier transform methods needed in simulation and the first open-source R implementation of the Metropolis-adjusted Langevin algorithm for this target. The package **lgcp** also has functionality not discussed in this article including methods for handling missing spatial data over

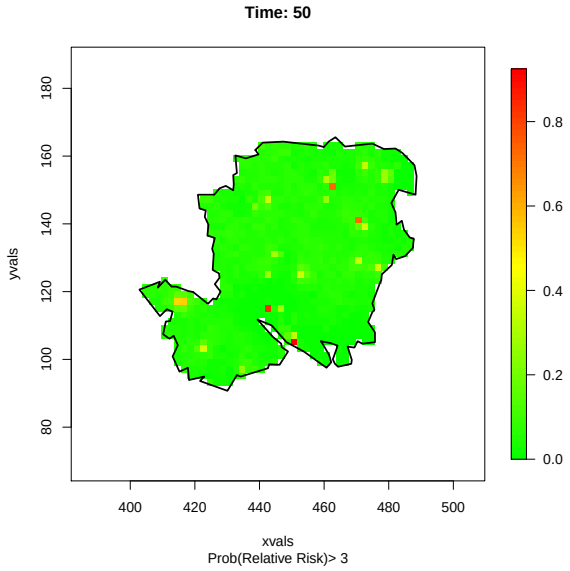

Figure 10: Plot showing the cell-wise probability (colour coded) that the relative risk is greater than 3.

time, the approximation of Gaussian fields by Gaussian Markov random fields and prediction for 'spatial-only' log Gaussian Cox processes (Taylor and Diggle 2012).

The initial motivation for this work was disease surveillance as performed by Diggle *et al.* (2005), and it is this application which has driven the core functionality for initial release (at the time of writing, **lgcp** is at version 0.9-7). A list of possible extensions to **lgcp** includes: the ability to include spatial and temporally referenced covariates into the MCMC scheme; to perform principled Bayesian parameter inference (both currently under development); and to handle applications where covariate information is available at differing spatial resolutions. Finally, in the spatio-temporal setting, it is of further interest to include the ability to handle non-separable correlation structures; see for example Gneiting (2002); Rodrigues and Diggle (2010).

6. Acknowledgements

The population data used in this article was based on real data from project AEGISS (Diggle *et al.* 2005). AEGISS was supported by a grant from the Food Standards Agency, U.K., and from the National Health Service Executive Research and Knowledge Management Directorate.

A. Using covariates to estimate the fixed components

By 'a spatially-resolved covariate', we refer to a covariate that can sensibly be defined for a spatial region. Typically, such a covariate would pertain to the environment (e.g., average rainfall), rather than to individuals (e.g., individual eye colour), unless geographical aggregation of the individual attributes makes sense. In this section, we show how to estimate the

fixed spatial and temporal components in the situation where we wish to use spatially-resolved covariate information to adjust for effects of interest.

The procedure is as follows: (1) choose the size of the computational grid (i.e., the size of the cells) and compute the FFT grid using

```
R> OW <- selectObsWindow(xyt,cellwidth=2)
```

as in Section 4.3; (2) compute the total number of events in each of the cells using the function `getCellCounts`,

```
R> cellcounts <- getCellCounts(x=xyt$x,y=xyt$y,xgrid=OW$xvals,xgrid=OW$yvals)
```

(3) resolve all available covariate information onto this grid i.e., extract the value of each covariate for each of the grid cells; and (4) separately fit spatial and temporal models to these data as described below. Note that the **sp** function `overlay` is very useful for resolving covariate data onto grids.

In the following code, we first simulate some population and covariate data on a 128×128 grid defined by centroids `xvals` and `yvals`; for illustration, we pretend that this is the FFT grid on which inference will take place. Then we generate the covariate data: both continuous, `ctsvar`, as well as a binary, `binvar`, variables. We next define a function, `mut`, that will cause seasonal peaks and troughs in the number of events. Lastly, Poisson counts are simulated on a grid over a time period of $[0, 100]$, in this case ignoring extra-Poisson spatial and temporal correlation. For the purpose of estimating the fixed spatial and temporal components, all that is required are cell counts aggregated over time, `simd`, and the total number of events over space at each time point, `count`. The function `rgauss` simulates correlated Gaussian fields on lattices. Figure 11 illustrates the covariate and simulated data.

```
R> set.seed(666)
R> ncells <- 128

R> population <- exp(rgauss(ncells=ncells,
                  model.parameters=lgcppars(sigma=2,phi=0.1))$grid[[1]])
R> ctsvar <- rgauss(ncells=ncells,
                  model.parameters=lgcppars(sigma=1,phi=0.1))$grid[[1]]
R> binvar <- matrix(as.numeric(rgauss(ncells=ncells,
                  model.parameters=lgcppars(sigma=2,phi=0.05))$grid[[1]]>1),
                  ncells,ncells)

R> xvals <- seq(0,1,length.out=ncells)
R> yvals <- seq(0,1,length.out=ncells)

R> mut <- function(t){
R>     return(exp(-7 + sin(2*pi*t/25)+cos(2*pi*t/25)))
R> }

R> param <- c(1,2)
```

```
R> time <- seq(0:100)

R> simd <- matrix(0,ncells,ncells)
R> count <- c()
R> for(t in 1:length(time)){
R>     rate <- mut(time[t])*population*exp(ctsvar*param[1]+binvar*param[2])
R>     dat <- matrix(rpois(ncells^2,rate),ncells,ncells)
R>     simd <- simd + dat
R>     count <- c(count,sum(dat))
R> }
```

We can now fit covariate-adjusted spatial and temporal models to these data to obtain estimates of λ and μ.

Firstly, we 'postulate' a Poisson log-linear model for the cell counts given the covariate data, which can be fitted using the `glm` function from the **stats** package; `log(population)` is included as an offset.

```
R> spatmod <- glm(as.vector(simd)~as.vector(ctsvar)+as.vector(binvar),
                  family="poisson",offset=as.vector(log(population)))
R> summary(spatmod)$coefficients

                   Estimate Std. Error   z value Pr(>|z|)
(Intercept)       -1.964796 0.02551681 -77.00005        0
as.vector(ctsvar)  1.020210 0.02283510  44.67727        0
as.vector(binvar)  1.999494 0.03895494  51.32838        0
```

As should be expected, the covariate effects stored in the object `param` defined above are retrieved by this procedure. For the temporal component, we again postulate a Poisson generalised linear model framework including harmonic regression terms, which is also fitted using `glm`.

```
R> tempmod <- glm(count~sin(2*pi*time/25)+cos(2*pi*time/25),family="poisson")
R> summary(tempmod)$coefficients

                       Estimate Std. Error   z value      Pr(>|z|)
(Intercept)           2.9203005 0.02750970 106.15532  0.000000e+00
sin(2 * pi * time/25) 0.9618636 0.03281484  29.31185 7.326175e-189
cos(2 * pi * time/25) 0.9576516 0.03248664  29.47832 5.459656e-191
```

The resulting objects, `spatmod` and `tempmod`, can be converted to `spatialAtRisk` and `temporalAtRisk` objects for use in a run of `lgcpPredict` by using the following:

```
R> sar <- spatialAtRisk(list(X=xvals,Y=yvals,
                             Zm=matrix(fitted(spatmod),ncells,ncells)))
R> tar <- temporalAtRisk(fitted(tempmod),tlim=c(0,100),warn=FALSE)
```

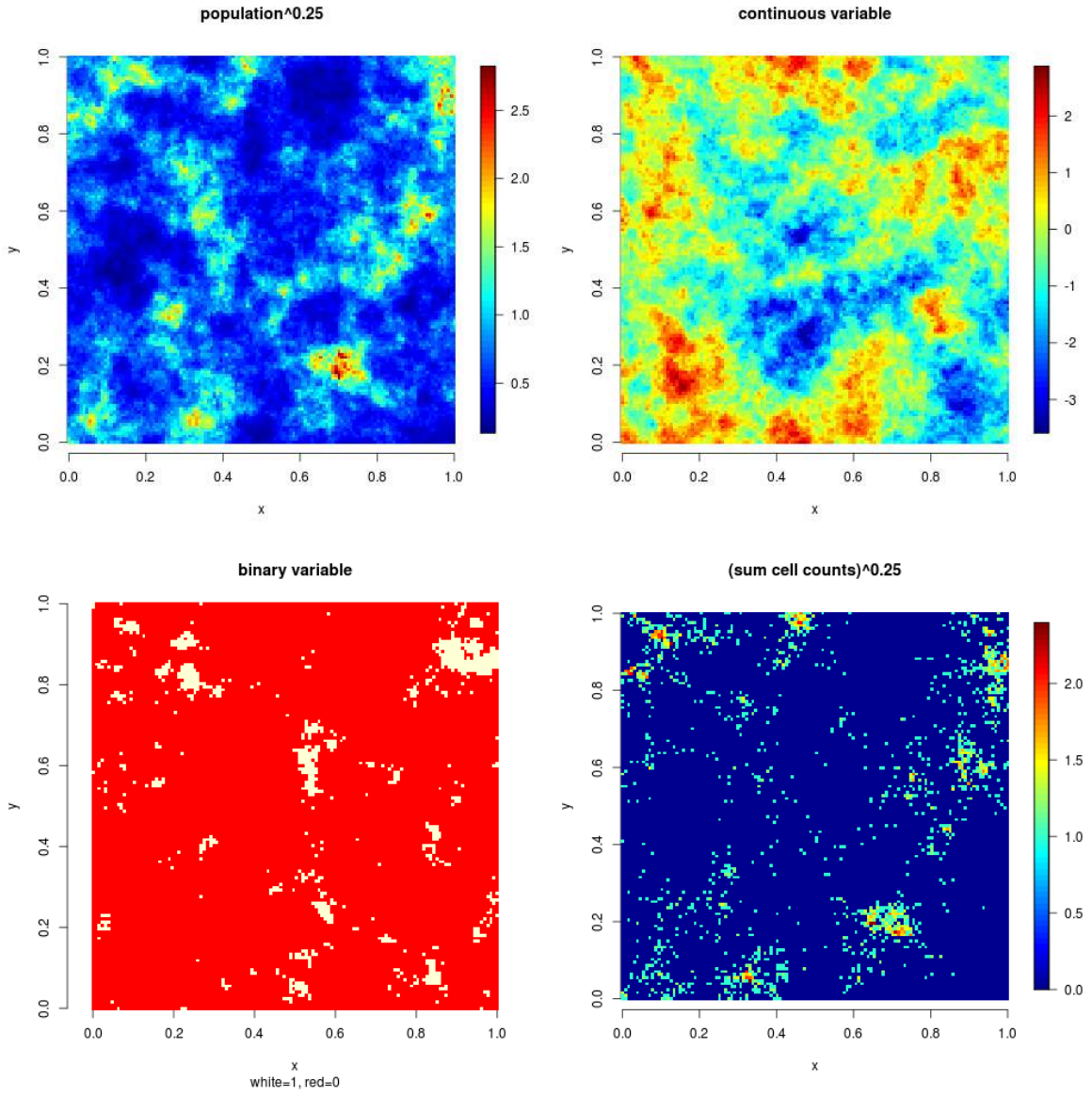

Figure 11: Top left: image plot of the simulated population, `population`, raised to the power 1/4 (to improve presentation). Top right: image plot of the continuous variable `ctsvar`. Bottom left: image plot of the binary variable, `binvar`. Bottom right: plot of simulated cell counts, `simd`, raised to the power 1/4.

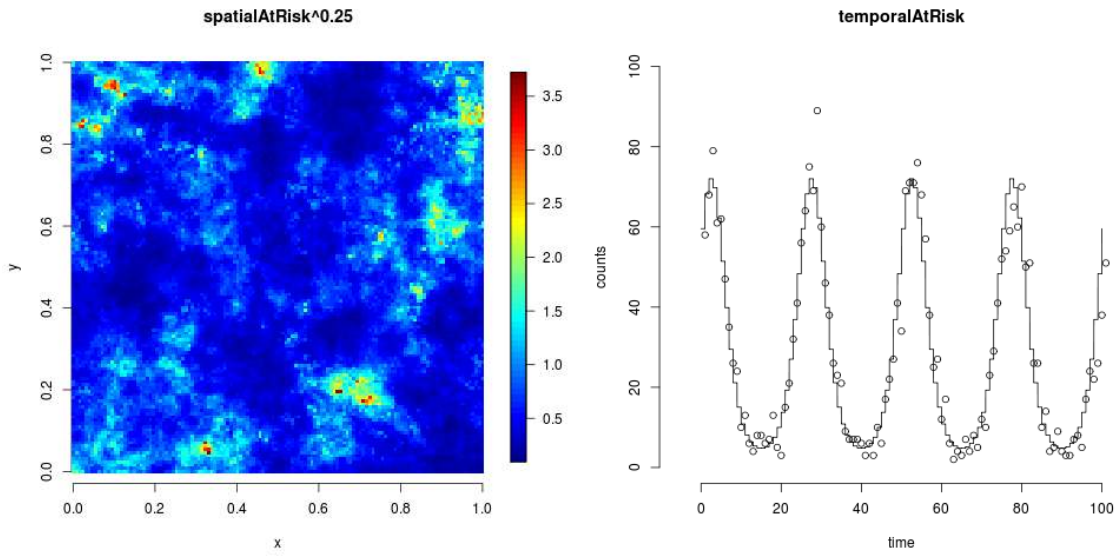

Figure 12: Left: fitted `spatialAtRisk` object, `sar`, raised to the power 1/4; compare with the simulated data in Figure 11. Right: fitted `temporalAtRisk` object, `tar`, with the actual counts appearing as dots.

In the above, the fitted values from `spatmod` are rescaled so that λ integrates to 1 over the observation window. The object `tar` acts as a look-up table: for each integer time point, `t`, in the range $[0, 100]$, `tar(t)` is the fitted value from the temporal generalised linear model for that time. Figure 12 shows the fitted fixed spatial and temporal components.

B. Rotation

The MALA algorithm works on a regular square grid placed over the observation window. The user is responsible for providing a physical grid size on which to perform estimation/prediction. The gridded observation window is then extended automatically to obtain a $2^m \times 2^n$ grid on which the simulation is performed. By default, the orientation of this extended grid is the same as the object `win`. If the observation window is elongated and set at a diagonal, then some loss of efficiency that would occur as a consequence of redundant computation at irrelevant locations can be recovered by rotating the coordinate axes and performing the computations in the rotated space.

To illustrate this, suppose `xyt2` is an `stppp` object with such an elongated and diagonally oriented window (see Figure 13). The function `roteffgain` displays whether any efficiency can be gained by rotation; clearly this not only depends on the observation window, but also on the size of the square cells on which the analysis will be performed. In the example below, the user wishes to perform the analysis using a cell width of 25km (corresponding to `cellwidth=25000` in the code below):

```
R> roteffgain(xyt2,cellwidth=25000)

By rotating the observation window, the efficiency gain would be: 200%,
```

```
   see ?getRotation.stppp
NOTE: efficiency gain is measured as the percentage increase in FFT
   grid cells from not rotating compared with rotating
[1] TRUE
```

The routine returns `FALSE` if there is no 'efficiency gain'. Note that the efficiency gain is not a reflection on computational speed, but rather a measure of how many fewer cells the MALA is required to use; this is illustrated in Figure 13. As a technical aside, a better measure would be a ratio of mixing times for the MCMC chains based on unrotated and rotated windows; however, as the mixing time depends on how well the MALA has been tuned, it is not clear how this can be estimated accurately.

Having ascertained whether rotation is advantageous, the optimally rotated data, observation window and rotation matrix can be retrieved using the function `getRotation`. For prediction using `lgcpPredict`, there is also an `autorotate` option: this allows the user to perform MALA on a rotated grid with minimal input so long as rotation leads to a gain in efficiency. If the model is fitted using a rotated frame, then the predictions will also be returned in this frame: this means that in the original orientation the output will be on a grid misaligned to the original axes. The **lgcp** package provides methods for the generic function `affine` so that `stppp` and `spatialAtRisk` objects can be rotated manually.

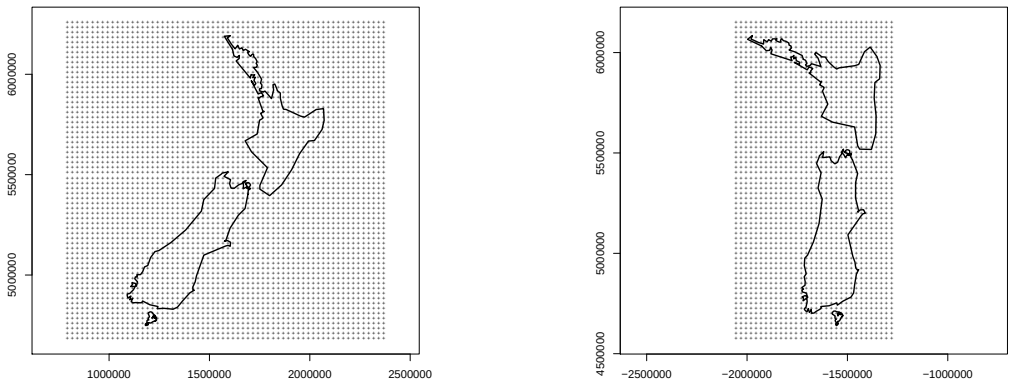

Figure 13: Illustrating the potential gain in efficiency by rotating the observation window. Left plot: the selected grid without rotation. Right plot: the optimally rotated grid.

C. Simulating data

The **lgcp** package also provides an approximate simulation tool for drawing samples from the model in Equation 1. Simulation minimally requires an observation window, a range of times over which to simulate data, spatial and temporal intensity functions λ and μ, a cell width for the discretisation and a set of spatial/temporal model parameters together with a choice of spatial covariance model.

The code below simulates data from a spatio-temporal log-Gaussian Cox process on the observation window from the example in the text above. The function `tempfun` is coerced

into a `temporalAtRisk` object and defines the temporal trend. Any appropriately defined `temporalAtRisk` object can be used here. Similarly, `spatial.intensity` can either be an object of class `spatialAtRisk` or one that can be coerced to one.

```
R> W <- xyt$window
R> tempfun <- function(t){return(100)}
R> sim <- lgcpSim(owin=W,
                  tlim=c(0,100),
                  spatial.intensity=den,
                  temporal.intensity=tempfun,
                  cellwidth = 0.5,
                  model.parameters=lgcppars(sigma=2,phi=5,theta=2))
```

Note that the finer the grid resolution, the more accurately will the process be simulated, and that smaller values of ϕ require a finer discretisation to get an accurate representation of the latent field. A warning is issued if the algorithm thinks the chosen cell width is too large. The discretisation in time is chosen automatically by the algorithm. The respective command for simulating spatial data is `lgcpSimSpatial`.

D. Handling the `SpatialAtRisk` class

This section illustrates the available commands for converting between different types of R objects that can be used to describe $\lambda(s)$. Conversion methods are provided for objects from the packages **spatstat** (Baddeley and Turner 2005), **sp** (Bivand *et al.* 2008) and **sparr** (Davies, Hazelton, and Marshall 2011). These are illustrated in Figure 14. For the purposes of parameter estimation, Figure 15 shows the different `spatialAtRisk` objects that can be converted into an appropriate format (i.e., a **spatstat** `im` object).

There is also a function to convert from `fromXYZ`-type `spatialAtRisk` objects to **sp** objects of class `SpatialGridDataFrame`: `as.SpatialGridDataFrame(obj,...)`. Lastly, `fromFunction`-type can be converted to `fromXYZ`-type `spatialAtRisk` objects using the `as.fromXYZ` function. Note that if a `spatialAtRisk` object is specified via a function, then it is the user's responsibility to ensure that the function integrates to 1 over the observation window; one way to bypass this problem is to convert the function to an `spatialAtRisk` object of `fromXYZ`-type.

E. Handling shapefiles

Here we show how to read in and convert shapefiles for use in **lgcp**. Boundary files for Wales can be obtained from the Great Britain data files from the GADM (`http://www.gadm.org/country`) website. The following script will download the level 1 (England, Scotland, Wales, Northern Ireland) and level 2 (counties and regions) data in `.RData` format, and put them in a temporary directory:

```
R> td = tempdir()
R> mapdata = c("GBR_adm1.RData","GBR_adm2.RData")
R> src = "http://www.gadm.org/data/rda/%s"
```

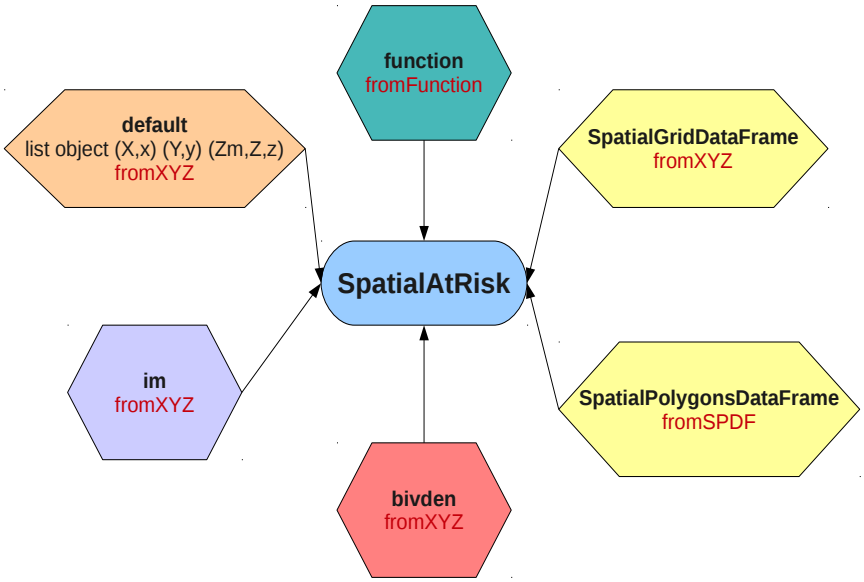

Figure 14: Conversion to `spatialAtRisk` objects. By default, `SpatialAtRisk` looks for a `list`-type object, but other objects that can be coerced include **spatstat** `im` objects, `function` objects, **sp** `SpatialGridDataFrame` and `SpatialPolygonsDataFrame` objects and **sparr** `bivden` objects. The text in red gives the type of `spatialAtRisk` object created.

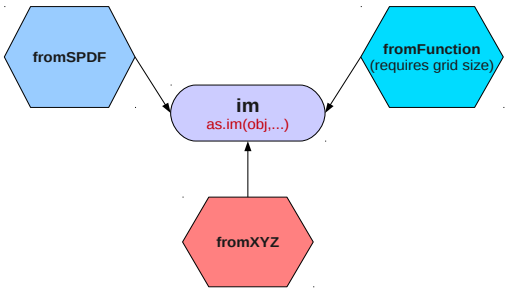

Figure 15: Conversion to **spatstat** objects of class `im`; these are useful for parameter estimation, in each case a call to the function `as.im(obj,...)` will perform the coercion.

```
R> for(mapfile in mapdata){
R>   download.file(sprintf(src,mapfile),
R>                 destfile=file.path(td,mapfile))
R> }
```

The first polygon needed is the observation window, which is the boundary of Cymru (in English, "Wales"), so it is extracted and converted with the following:

```
R> load(file.path(td,"GBR_adm1.RData"))
R> cymru_border <- gadm[gadm$NAME_1=="Wales",]
R> cymru_border <- spTransform(cymru_border,CRS("+init=epsg:27700"))
R> spatstat.options(checkpolygons = FALSE)
R> W <- as(cymru_border,"owin")
R> W <- simplify.owin(W,dmin=2000)
R> spatstat.options(checkpolygons = TRUE)
```

The transformation to Ordnance Survey (OS) Grid units is done so that the resulting locations have valid Euclidean distances, which is not the case with latitude-longitude coordinates.

Note that the `simplify.owin` step is fairly crucial in this example. Without simplifying the observation window, some routines such as `ginhomAverage` or `kinhomAverage` can run for a *long* time.

F. Writing adaptive MCMC schemes

```
Warning message:
With adaptive MCMC, the Markov property is not satisfied and GREAT
   care must be taken to ensure that the correct ergodic distribution
   is preserved. Please do not attempt to write an adaptive scheme
   unless you REALLY know what you are doing!
```

There are two generic functions to consider when writing adaptive MCMC routines, namely `initialiseAMCMC` and `updateAMCMC`, these respectively define the initialisation and the updating procedures for the adaptive scheme of interest. The task of the user is therefore to tell `lgcpPredict` what value of h to use at iteration 1, and how to update it. **lgcp** has two schemes built in: `constanth` and `andrieuthomsh` detailed below.

F.1. A simple example: `constanth`

This example shows how the scheme `constanth` was implemented. This is not really an adaptive MCMC scheme and just returns the (fixed) value of h set by the user. In `lgcpPredict`, this 'adaptive' scheme would be specified using `adaptivescheme=constanth(0.01)` in the `mcmc.control` argument. The effect is to return $h = 0.01$ at each iteration of the MCMC loop.

The user is required to write three functions: `constanth`, and for compatibility with the S3 implementation of this framework, `initialiseAMCMC.constanth` and `updateAMCMC.constanth`; these functions are detailed below.

```
R> constanth <- function(h){
R>     obj <- h
R>     class(obj) <- c("constanth","adaptivemcmc")
R>     return(obj)
R> }

R> initialiseAMCMC.constanth <- function(obj,...){
R>     return(obj)
R> }

R> updateAMCMC.constanth <- function(obj,...){
R>     return(obj)
R> }
```

When called, the first of these functions creates an object of super-class `constanth`, this is just a numeric with a class attribute attached. The other two functions simply return the value of h specified by the user at appropriate positions in the code `MALAlgcp`.

F.2. A more complex example: `andrieuthomsh`

The second example shows how to implement the rather neat method of Andrieu and Thoms (2008), detailed in Section 4.5. An `adaptivescheme` implementing this algorithm needs to know what value of h to start with, the values of α and C and also the target acceptance probability; this motivates the choice of arguments for the function `andrieuthomsh` in the code below:

```
R> andrieuthomsh <- function(inith,alpha,C,targetacceptance=0.574){
R>     if (alpha<=0 | alpha>1){
R>         stop("parameter alpha must be in (0,1]")
R>     }
R>     if (C<=0){
R>         stop("parameter C must be positive")
R>     }
R>     obj <- list()
R>     obj$inith <- inith
R>     obj$alpha <- alpha
R>     obj$C <- C
R>     obj$targetacceptance <- targetacceptance

R>     itno <- 0
R>     incrit <- function(){
R>         itno <<- itno + 1
R>     }
R>     restit <- function(){
R>         itno <<- 0
R>     }
R>     obj$incritno <- incrit
```

```
R>     obj$restartit <- restit

R>     curh <- inith

R>     hupdate <- function(){
R>         curh <<- exp(log(curh) + (C/(itno^alpha))*
R>             (get("ac",envir=parent.frame(2))-targetacceptance))
R>     }
R>     reth <- function(){
R>         return(curh)
R>     }
R>     obj$updateh <- hupdate
R>     obj$returncurh <- reth

R>     class(obj) <- c("andrieuthomsh","adaptivemcmc")
R>     return(obj)
R> }
```

This function returns an object of super-class `andrieuthomsh`, which is a list object consisting of the parameters specified by the user and additionally some internal functions, namely `incrit`, `restit`, `hupdate` and `reth` which are responsible for the updating. Note that in `updateAMCMC.andrieuthomsh`, the internal functions are simply called, and therefore it is these internal functions that actually define the adaptive scheme. The internal functions perform respectively the following tasks: increase the internal iteration counter, restart the internal iteration counter, do the actual updating of h and lastly return the current value of h.

Note that from a developmental point of view, the piece of code,

```
R> get("ac",envir=parent.frame(2))
```

retrieves the current acceptance probability in the MCMC loop, which in `MALAlgcp` is stored as an object called `ac`.

To initialise the scheme, the method for `initialiseAMCMC` simply returns the initial value of h set by the user:

```
R> initialiseAMCMC.andrieuthomsh <- function(obj,...){
R>     return(obj$inith)
R> }
```

In the update step, the internal functions created by the `andrieuthomsh` function are invoked. The procedure is as follows (1) information about the MCMC loop is retrieved using `get("mcmcloop",envir=parent.frame())`, then if the algorithm has just come out of the burn in period, the adaptation of h is restarted (this just gives h some extra freedom to explore the parameter space as compared to an algorithm that did not restart, doing this does not affect the convergence of the algorithm). Next the internal iteration counter is incremented, and lastly the value of h is updated and returned (the procedures for these internal functions are printed above in the code for `andrieuthomsh`).

```
R> updateAMCMC.andrieuthomsh <- function(obj,...){
R>     mLoop <- get("mcmcloop",envir=parent.frame())
R>     if(iteration(mLoop)==(mLoop$burnin)+1){
R>         obj$restartit()
R>     }
R>     obj$incritno()
R>     obj$updateh()
R>     return(obj$returncurh())
R> }
```

References

Andrieu C, Thoms J (2008). "A Tutorial on Adaptive MCMC." *Statistics and Computing*, **18**(4), 343–373.

Baddeley A, Turner R (2005). "**spatstat**: An R Package for Analyzing Spatial Point Patterns." *Journal of Statistical Software*, **12**(6), 1–42.

Baddeley AJ, Møller J, Waagepetersen R (2000). "Non- and Semi-Parametric Estimation of Interaction in Inhomogeneous Point Patterns." *Statistica Neerlandica*, **54**, 329–350.

Bivand RS, Pebesma EJ, Gomez-Rubio V (2008). *Applied Spatial Data Analysis with R*. Springer-Verlag. URL http://www.asdar-book.org/.

Bowman A, Crawford E, Alexander G, Bowman RW (2007). "**rpanel**: Simple Interactive Controls for R Functions Using the **tcltk** Package." *Journal of Statistical Software*, **17**(9), 1–18.

Bowman AW, Gibson I, Scott EM, Crawford E (2010). "Interactive Teaching Tools for Spatial Sampling." *Journal of Statistical Software*, **36**(13), 1–17. URL http://www.jstatsoft.org/v36/i13/.

Brix A, Diggle PJ (2001). "Spatiotemporal Prediction for Log-Gaussian Cox Processes." *Journal of the Royal Statistical Society B*, **63**(4), 823–841.

Davies T, Hazelton M (2012). "Log-Gaussian Cox Process Models for Planar Point Patterns." Submitted.

Davies TM, Hazelton ML, Marshall JC (2011). "**sparr**: Analyzing Spatial Relative Risk Using Fixed and Adaptive Kernel Density Estimation in R." *Journal of Statistical Software*, **39**(1), 1–14.

Diggle P, Rowlingson B, Su T (2005). "Point Process Methodology for On-Line Spatio-Temporal Disease Surveillance." *Environmetrics*, **16**(5), 423–434.

Gneiting T (2002). "Nonseparable, Stationary Covariance Functions for Space-Time Data." *Journal of the American Statistical Association*, **97**, 590–600.

Hastings WK (1970). "Monte Carlo Sampling Methods Using Markov Chains and Their Applications." *Biometrika*, **57**(1), 97–109.

Hijmans RJ, van Etten J (2012). "**raster**: Geographic Analysis and Modeling with Raster Data." R package version 1.9-92, URL http://CRAN.R-project.org/package=raster.

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953). "Equation of State Calculations by Fast Computing Machines." *The Journal of Chemical Physics*, **21**(6), 1087–1092.

Møller J, Syversveen AR, Waagepetersen RP (1998). "Log Gaussian Cox Processes." *Scandinavian Journal of Statistics*, **25**(3), 451–482.

Pebesma E (2012). "**spacetime**: Classes and Methods for Spatio-Temporal Data." URL http://cran.r-project.org/web/packages/spacetime/.

Pebesma EJ, Bivand RS (2005). "Classes and Methods for Spatial Data in R." *R News*, **5**(2), 9–13. URL http://CRAN.R-project.org/doc/Rnews/.

Pierce D (2011). "**ncdf** Home Page: A NetCDF Package for R." URL http://cirrus.ucsd.edu/~pierce/ncdf/.

Plummer M, Best N, Cowles K, Vines K (2006). "**CODA**: Convergence Diagnosis and Output Analysis for MCMC." *R News*, **6**(1), 7–11.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Robbins H, Munro S (1951). "A Stochastic Approximation Method." *The Annals of Mathematical Statistics*, **22**(3), 400–407.

Roberts G, Rosenthal J (2001). "Optimal Scaling for Various Metropolis-Hastings Algorithms." *Statistical Science*, **16**(4), 351–367.

Roberts GO, Rosenthal JS (1998). "Optimal Scaling of Discrete Approximations to Langevin Diffusions." *Journal of the Royal Statistical Society B*, **60**, 255–268(14).

Roberts GO, Rosenthal JS (2009). "Examples of Adaptive MCMC." *Journal of Computational and Graphical Statistics*, **18**(2), 349–367.

Roberts GO, Tweedie RL (1996). "Exponential Convergence of Langevin Distributions and Their Discrete Approximations." *Bernoulli*, **2**(4), pp. 341–363.

Rodrigues A, Diggle P (2010). "A Class of Convolution-Based Models for Spatio-Temporal Processes with Non-Separable Covariance Structure." *Scandinavian Journal of Statistics*, **37**, 553–567.

Rue H, Held L (2005). *Gaussian Markov Random Fields*. Chapman & Hall.

Ryan JA, Ulrich JM (2012). "**xts**: eXtensible Time Series." URL http://cran.r-project.org/web/packages/xts/.

Schlather M (2001). "Simulation and Analysis of Random Fields." URL http://www.stochastik.math.uni-goettingen.de/~schlather/#Software.

Taylor BM, Diggle PJ (2012). "INLA or MCMC? A Tutorial and Comparative Evaluation for Spatial Prediction in Log-Gaussian Cox Processes." Submitted, available from http://www.arxiv.org/pdf/1202.1738.

Wood ATA, Chan G (1994). "Simulation of Stationary Gaussian Processes in $[0, 1]^d$." *Journal of Computational and Graphical Statistics*, **3**(4), 409–432.

Affiliation:

Benjamin M. Taylor
Division of Medicine,
Lancaster University,
Lancaster,
LA1 4YF,
UK
E-mail: b.taylor1@lancaster.ac.uk
URL: http://www.maths.lancs.ac.uk/~taylorb1/

Tilman M. Davies
Department of Mathematics and Statistics
University of Otago
Science III
PO Box 56
Dunedin 9054
New Zealand
E-mail: tdavies@maths.otago.ac.nz
URL: http://www.maths.otago.ac.nz/home/department/staff/_staffscript.php?s=tilman_davies

Barry Rowlingson
Division of Medicine,
Lancaster University,
Lancaster,
LA1 4YF,
UK
E-mail: b.rowlingson@lancaster.ac.uk
URL: http://www.maths.lancs.ac.uk/~rowlings/

Professor Peter J. Diggle
Division of Medicine,
Lancaster University,
Lancaster,
LA1 4YF,
UK
E-mail: p.diggle@lancaster.ac.uk
URL: http://www.lancs.ac.uk/~diggle/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd