How to Use geoCount

Liang Jing

January 6, 2012

Contents

Preface

Hierarchical models are increasingly used in many of the earth sciences. A class of Generalized Linear Mixed Models was proposed by Diggle, Tawn and Moyeed (1998) for the analysis of spatial non-Gaussian data, but model estimation, checking and selection in this class of models remain difficult tasks due to the presence of an unobservable latent process. Model checking methods within this class have not been considered so far in the literature.

We considered this class of models for the analysis of geostatistical count data, and implemented robust Markov Chain Monte Carlo algorithms with the help of advanced techniques, such as group updating, Langevin-Hastings algorithms, and data-based transformations, for posterior sampling and estimation.

Then we explored the application of Bayesian model checking methods based on measures of relative predictive surprise, as those described in Bayarri and Castellanos (2007). We also proposed an alternative model checking method to diagnose incompatibility between model and data based on a kind of transformed residuals.

An R package, {`geoCount`} was developed to implement all the methods by using advanced computing techniques, such as R/C++ interfacing and parallel computing.

Chapter 1 introduces the type of data and models dealt with by this package. Chapter 2, 3, and 4 introduce how to use the functions in the package to perform the analysis and modeling. Chapter 5 introduce how to install the package and some running issues.

The details of theories, methodologies, algorithms, and techniques that are implemented in this package can be found in my dissertation.

1

Introduction

This package is designed to analyze geostatistcal count data by using generalized linear spatial models. In this chapter, we introduce the properties of the data and the structure of the models.

1.1 Geostatistical Data

Spatial data contain information about both attributes of interest and locations, and can be found in a number of disciplines, including ecology, epidemiology, geography, forestry, and meteorology. Following the description in Diggle and Ribeiro (2007), one special type of spatial data, geostatistical data, mainly has two characteristics: first, values $Y_i \ : \ i = 1, ..., n$ are observed at a discrete set of sampling locations $\boldsymbol{x}_i$ within some spatial region A; second, each observed value Y_i is either a direct measurement of, or is statistically related to, the value of an underlying continuous spatial phenomenon, $S(\boldsymbol{x}_i)$, at the corresponding sampling location $\boldsymbol{x}_i$. Given the latent continuous process $S(\boldsymbol{x})$, the observed data $Y_i \ : \ i = 1, ..., n$ are usually assumed to be independent. Also, for each location, there is usually no replication of Y_i.

In most applications with geostatistical data, the scientific goals focus on two areas, *estimation and prediction*:

- The estimation of coefficient parameters that describe the relationship between response variable and explanatory variables, and the estimation for parameters that define the covariance structure of the latent process.

- The prediction of the realized values of unobservable latent process at certain locations, and the prediction for some property of the complete realization of latent process within certain area, for example the average of the process $S(B) = \frac{1}{|B|}\int_B S(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$, where B defines an area.

An example of geostatistical data is given below.

Rongelap data

First analyzed by Diggle, Tawn and Moyeed (1998), the data were collected from Rongelap Island, the principal island of Rongelap Atoll in the South Pacific, which forms part of the Marshall Islands. U.S. nuclear weapon testing program generated heavy fallout over the island in the 1950's and it has been uninhabited since 1985. Figure 1.1 shows a map of Rongelap Island with the 157 sampling locations derived from a sampling design which consists of a primary grid covering the island at a spacing of 200 meters and four secondary 5 by 5 sub-grids at a spacing of 50 meters. For each location, photon emission counts attributable to radioactive caesium were measured. The data have the form $(\boldsymbol{x}_i, m_i, t_i)\ :\ i = 1, ..., 157$, where $\boldsymbol{x}_i$ represents spatial location, m_i is the photon emission count for that location, and t_i is the time (in seconds) over which t_i was accumulated. For further information of these data, see Diggle, Harper and Simon (1997).

If we use the observed emission counts per unit time m_i/t_i as response variable, then Rongelap data can be transformed into the basic format of geostatistical data,

$$(\boldsymbol{x}_i, y_i)\ :\ i = 1, ..., n$$

where each $y_i = m_i/t_i$ is a realization of a random variable Y_i whose distribution depends on an underlying unobservable spatially continuous stochastic process $S(\boldsymbol{x})$.

More examples of geostatistical data can be found in Diggle and Ribeiro (2007) and Christensen, Roberts and Sköld (2006).

1.2 Spatial Models with Gaussian Processes

From the characteristics of geostatistical data, geostatistical models usually consists of two elements: first, random variables $S(\boldsymbol{x}_i) : i = 1, ..., n$, which are typically a partial

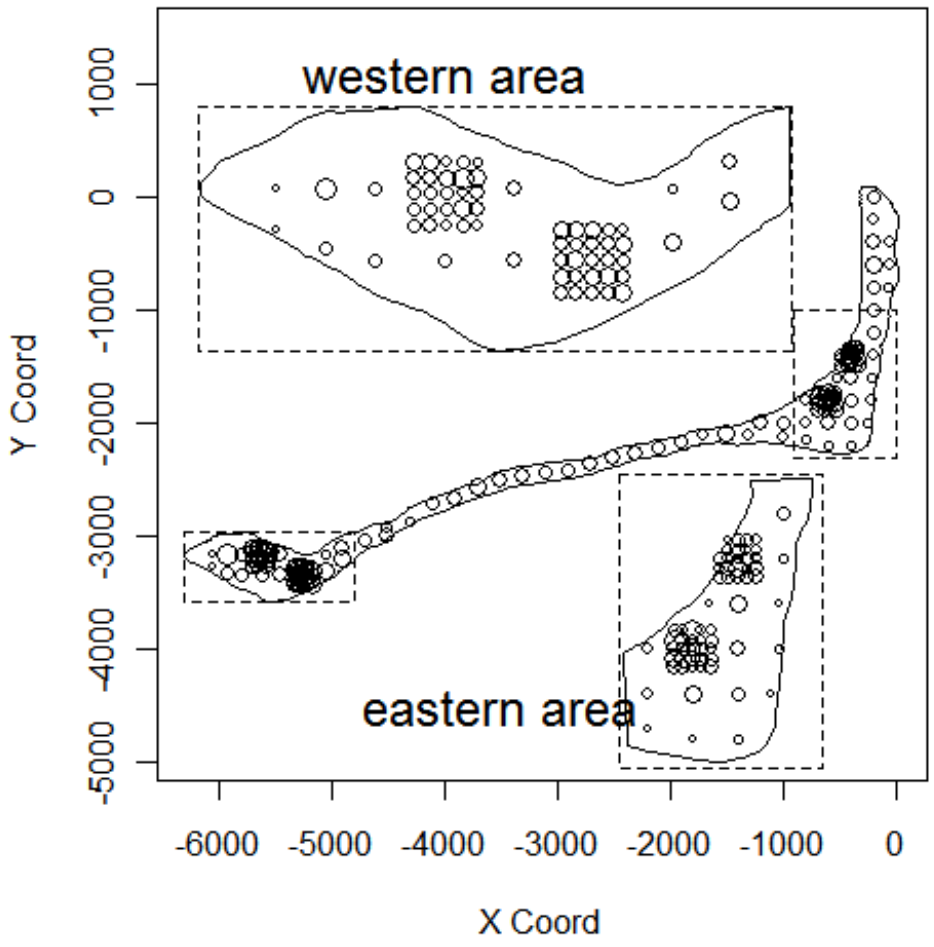

Figure 1.1: Rongelap data: photon emission counts are measured for 157 different locations on the island

realization of a stochastic process $S(\boldsymbol{x}) : \boldsymbol{x} \in R^2$ on the whole space; second, a joint distribution for random variables $Y_1, ..., Y_n$ conditional on $S(\boldsymbol{x}_1), ..., S(\boldsymbol{x}_n)$. Sometimes $S(\boldsymbol{x}_i)$ is called the *signal* and Y_i the *response*.

1.2.1 Gaussian Process Model

Considering the nature of the space, it is desirable for $S(\boldsymbol{x})$ to be continuous and possibly differentiable. To meet these requirements, the simplest and most popular choice is the *Gaussian process*, which is a stochastic process whose realizations consist of random variables associated with every point in the space with the property that every finite collection of these random variables has a multivariate normal distribution. Thus, $S(\boldsymbol{x})$ can be defined as

> $S(\boldsymbol{x}) : \boldsymbol{x} \in R^2$ is a Gaussian process with mean $\mu(\boldsymbol{x})$, variance σ^2 for all $\boldsymbol{x}$, and correlation function $\rho(u) = Corr[S(\boldsymbol{x}), S(\boldsymbol{x}')]$ depending only on $u = ||\boldsymbol{x}' - \boldsymbol{x}||$, the Euclidean distance between $\boldsymbol{x}$ and $\boldsymbol{x}'$.

In general, the response variables Y_i are assumed to be conditionally independent given the $S(\boldsymbol{x}_i)$ and considered as a noisy version of $S(\boldsymbol{x}_i)$. Specifically,

> for any set of sampling locations $\boldsymbol{x}_1, ..., \boldsymbol{x}_n$, $Y_1, ..., Y_n$ are conditionally independent given $S(\boldsymbol{x}_1), ..., S(\boldsymbol{x}_n)$ with mean $E[Y_i|S(\boldsymbol{x}_i)] = g^{-1}(S(\boldsymbol{x}_i))$ where $g(\cdot)$ is a link function.

1.2.2 Correlation Function Family

In order to define a legitimate model, the correlation function $\rho(u)$ must be positive-definite, which means for any collection of finite locations $\boldsymbol{x}_i : i = 1, ..., n$ the correlation matrix with entries $\rho(||\boldsymbol{x}_i - \boldsymbol{x}_j||)$ must be a positive definite matrix.

The common families of correlation function that are known to be positive-definite are introduced in this section.

1. The Matern family

   This family is named after Matern (1960) and has two parameters: $\kappa > 0$, called the *order*, is a shape parameter that determines the smoothness and $\phi > 0$ is a scale parameter,

   $$\rho(u) = [2^{\kappa-1}\Gamma(\kappa)]^{-1}(u/\phi)^{\kappa}K_{\kappa}(u/\phi) \tag{1.1}$$

where $K_\kappa(\cdot)$ denotes a modified Bessel function of order κ.

2. The powered exponential family
   This family also has a shape parameter $0 < \kappa \leq 2$ and a scale parameter $\phi > 0$,

$$\rho(u) = exp\{-(u/\phi)^\kappa\}. \tag{1.2}$$

3. The spherical family
   This family is widely used in applications,

$$\rho(u) = \begin{cases} 1 - \frac{3}{2}(u/\phi) + \frac{1}{2}(u/\phi)^3 & : \ 0 \leq u \leq \phi \\ 0 & : \ u > \phi \end{cases} \tag{1.3}$$

   where $\phi > 0$ is a scale parameter.

4. One example of a non-monotone correlation function, which is rarely used in practice, is

$$\rho(u) = (u/\phi)^{-1} sin(u/\phi) \tag{1.4}$$

   where $\phi > 0$ is a scale parameter.

For all of the above families κ is dimensionless and ϕ has dimensions of distance.

An example of several correlation curves is illustrated in Figure 1.2. More information about correlation functions can be found in Schlather (1999) and Gneiting (1997).

1.3 Linear Gaussian Process Model

Recall the model formulation of Gaussian process models in section 1.2.1. When the link function $g(\cdot)$ is the identity function, the model is a *linear Gaussian process model* and the model specification is

$$Y_i = \mu(\boldsymbol{d}_i) + S(\boldsymbol{x}_i) + Z_i \ : \ i = 1, ..., n \tag{1.5}$$

where $\mu(\boldsymbol{d}_i)$ is a mean effect term depending on covariates $\boldsymbol{d}_i$, $S(\boldsymbol{x})$ is a stationary Gaussian process with $E[S(\boldsymbol{x})] = 0$ and $Cov[S(\boldsymbol{x}'), S(\boldsymbol{x})] = \sigma^2\rho(\|\boldsymbol{x}' - \boldsymbol{x}\|)$, and Z_i is i.i.d. having $N(0, \tau^2)$ distribution. An equivalent formulation can be written as

$$Y_i|\mu(\boldsymbol{d}_i), S(\boldsymbol{x}_i) \sim N(\mu(\boldsymbol{d}_i) + S(\boldsymbol{x}_i), \tau^2). \tag{1.6}$$

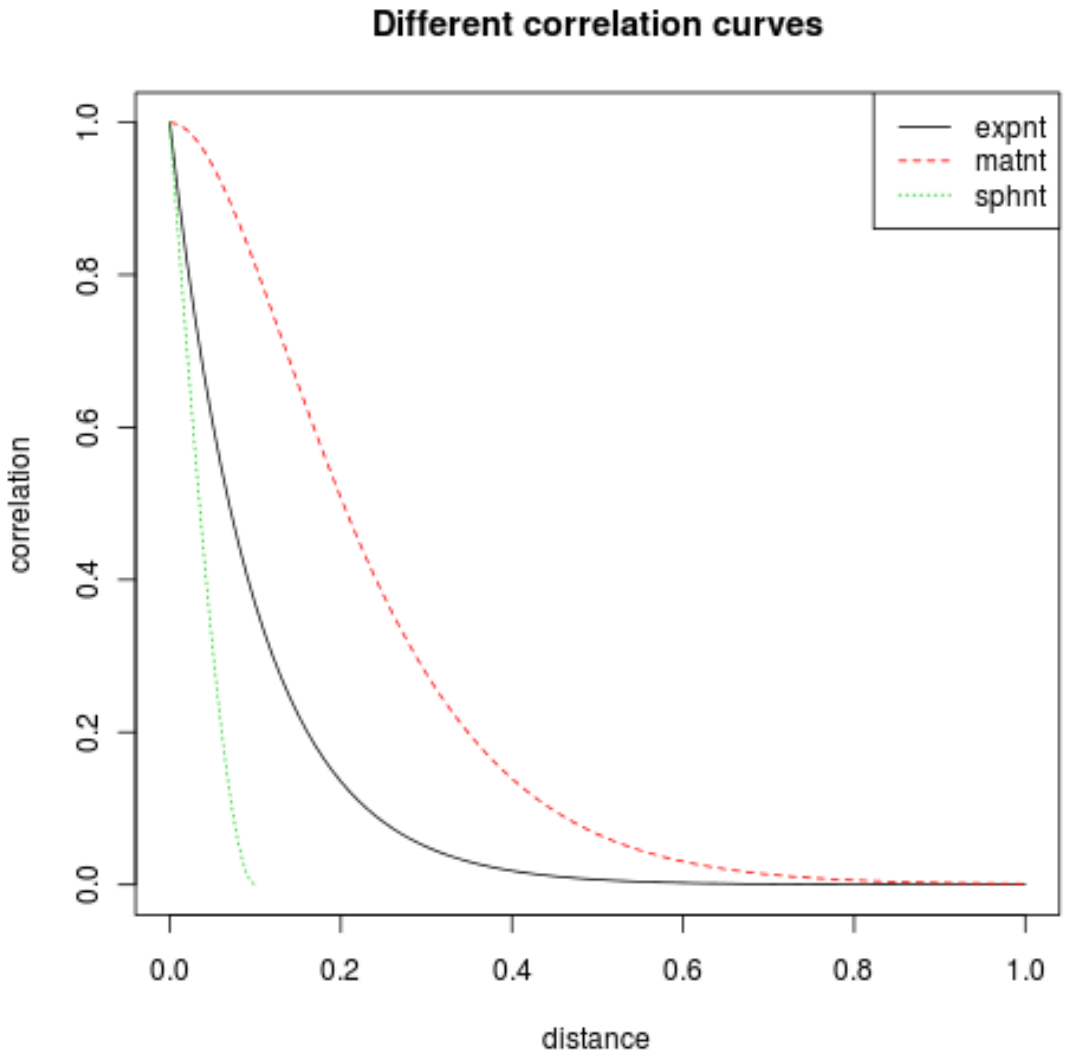

Figure 1.2: Example of different correlation curves: exponential (solid line), Matern (dashed line), and spherical (dotted line)

More practically, the mean term can be allowed to depend on location, $\boldsymbol{d}_i = \boldsymbol{d}(\boldsymbol{x}_i)$, and such location-dependent mean $\mu(\boldsymbol{x}_i)$ is called a *spatial trend*.

However, when the response data fail to follow a normal distribution, it is not appropriate to use a linear Gaussian process model. One solution to this problem is transformation of the data. The most widely used transformation (when the data are positive) is the Box-Cox family of transformation, Box and Cox (1964),

$$Y^* = \begin{cases} (Y^\lambda - 1)/\lambda & : \lambda \neq 0 \\ \log Y & : \lambda = 0 \end{cases} . \tag{1.7}$$

Then the transformed response variable is assumed to follow a linear Gaussian process model. De Oliveira, Kedem and Short (1997) explored the properties of transformed Gaussian process models for Bayesian prediction.

1.4 Generalized Linear Spatial Models

When it is important to model the non-Gaussian sampling mechanism or a non-Gaussian distribution of the response random variable of interest, a more flexible and useful model frame should be employed – *generalized linear spatial model* (GLSM), first proposed by Diggle, Tawn and Moyeed (1998). The complete model specification is

$$\begin{aligned} Y_i|S(\boldsymbol{x}_i) &\sim p(y_i|\mu_i), \quad i = 1, ..., n \\ \mu_i &= g^{-1}(S(\boldsymbol{x}_i)) \\ \boldsymbol{S} = (S(\boldsymbol{x}_1), ..., S(\boldsymbol{x}_n)) &\sim \text{MVN}(D\boldsymbol{\beta}, \boldsymbol{\Sigma}) \end{aligned} \tag{1.8}$$

where

- response variables $Y_i : i = 1, ..., n$ are conditionally independent given $S(\boldsymbol{x}_i) : i = 1, ..., n$ and follow a specific distribution $p(\cdot)$ with mean μ_i;
- $g(\cdot)$ is a known link function;
- $\boldsymbol{S} = (S(\boldsymbol{x}_1), ..., S(\boldsymbol{x}_n))$ is a stationary Gaussian process with mean structure $D\boldsymbol{\beta}$ and covariance structure $\boldsymbol{\Sigma}$;
- $D' = (\boldsymbol{d}_1, ..., \boldsymbol{d}_n)$ is a known $p \times n$ covariate matrix usually related to locations and assumed of full rank while $\boldsymbol{\beta} = (\beta_1, ..., \beta_p)'$ is the coefficients vector ($D\boldsymbol{\beta}$ together determines the "spatial trend" in the response variables, Diggle and Ribeiro (2007) section 3.6);

- $\boldsymbol{\Sigma}$ is a variance-covariance matrix with entries $\sigma_{ij} = \sigma^2\rho(u_{ij})$ where σ^2 is a unknown constant variance and $\rho(u_{ij})$ belongs to one of the common correlation function families described in section 1.2.2.

Note that this model is also known as *spatial generalized linear model* (SGLM), and it is included in *generalized linear mixed models* (GLMM) category since the latent variable $S(\boldsymbol{x}_i)$ can be interpreted as *random effects*.

In the following sections, the two most widely used GLSMs are introduced.

1.4.1 The Poisson Log-normal Spatial Model

As the name implies, this model has logarithm link function and the conditional distribution of each response variable Y_i is Poisson. The complete model specification is

$$\begin{aligned} Y_i|S(\boldsymbol{x}_i) &\sim \text{Poisson}(\mu_i) \\ \mu_i &= \exp\{S(\boldsymbol{x}_i)\} \\ \boldsymbol{S} = (S(\boldsymbol{x}_1), ..., S(\boldsymbol{x}_n)) &\sim \text{MVN}(D\boldsymbol{\beta}, \boldsymbol{\Sigma}) \end{aligned} \tag{1.9}$$

where $Y_i : i = 1, ..., n$ are conditionally independent given $S(\boldsymbol{x}_i) : i = 1, ..., n$ and $\boldsymbol{S}, D\boldsymbol{\beta}, \boldsymbol{\Sigma}$ are defined as in (1.8).

This model is naturally a good candidate for count data. For the Rongelap data in which the response variable is photon emission counts Y_i over time-period t_i at location $\boldsymbol{x}_i$, the Poisson log-normal spatial model can be easily modified as,

$$\mu_i = t_i \exp\{S(\boldsymbol{x}_i)\} \tag{1.10}$$

with powered exponential correlation function as shown in Diggle et al. (1998).

More examples of count data that are suitable for this model include: Yang, Teng and Haran (2009) studied infant mortality rates by county in the southern U.S. States of Alabama, Georgia, Mississippi, North Carolina and South Carolina (Health Re-sources and Services Administration (2003)) between 1998 and 2000; counts of weed plants on a field were recorded in 1993, 1994 and 1995, described in Olsen (1997).

1.4.2 The Binomial Logistic-normal Spatial Model

Here the response variable Y_i represents the outcome of a conditionally independent binomial variable with the number of trials n_i with probability of success p_i. The full model specification is:

$$\begin{aligned} Y_i|S(\boldsymbol{x}_i) &\sim \text{Binomial}(n_i, p_i) \\ p_i &= \frac{e^{S(\boldsymbol{x}_i)}}{1+e^{S(\boldsymbol{x}_i)}} \\ \boldsymbol{S} = (S(\boldsymbol{x}_1), ..., S(\boldsymbol{x}_n)) &\sim \text{MVN}(D\boldsymbol{\beta}, \boldsymbol{\Sigma}) \end{aligned} \tag{1.11}$$

where $Y_i : i = 1, ..., n$ are conditionally independent given $S(\boldsymbol{x}_i) : i = 1, ..., n$ and $\boldsymbol{S}, D\boldsymbol{\beta}, \boldsymbol{\Sigma}$ are defined as in (1.8).

An example of this model to the study of campylobacter infections in north Lancashire and south Cumbria appeared in Diggle et al. (1998).

2

Data Simulation and Visualization

2.1 Simulate Locations

As introduced in Chapter 1 geostatistical data have two attributes: location and response variable associated to location. To simulate geostatistical data, we first need to simulate coordinates of locations.

In this package several functions are provided to simulate locations:

`locGrid` : simulates a given number of locations distributed on a grid.
`locCircle` : simulates a given number of locations equally distributed on a circle.
`locSquad` : simulates a given number of locations equally distributed on a square.

For example,

```
loc <- locGrid(1, 2, 10, 5)
plot(loc, xlab="x", ylab="y")
loc2 <- rbind( locCircle(1, 60), locCircle(0.667, 40),
      locCircle(0.333, 20),  locCircle(0, 1) )
plot(loc2, xlab="x", ylab="y")
```

where the result is shown in Figure 2.1 and 2.2.

Sometimes (especially when doing preliminary analysis), it is more convenient to scale the locations to fit into a unit grid by using `unifLoc` function before further analysis (for example posterior sampling). Doing this could increase the speed of MCMC simulations significantly.

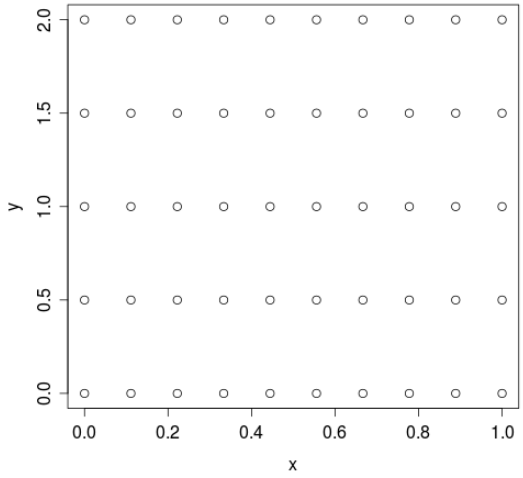

Figure 2.1: locGrid - simulates locations on a grid.

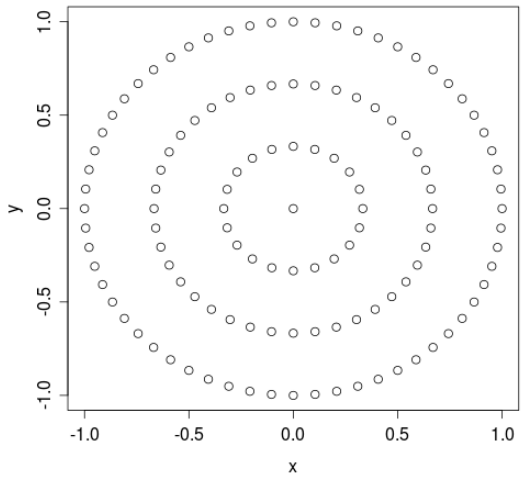

Figure 2.2: locCircle - simulates locations on a circle.

2.2 Simulate Data

Once locations are available, `simData` function can be used to simulate the response data from Poisson log-normal spatial model or binomial logistic-normal spatial model.

```
dat <- simData(loc = locGrid(1, 1, 10, 10), L = 0,
        X = NULL, beta = 1, cov.par = c(1, 0.1, 1),
        rho.family = "rhoPowerExp", Y.family = "Poisson")
```

where

- `loc:` a $n \times 2$ matrix which indicates the coordinates of given locations.
- `L:` a vector of length n; it indicates the time duration during which the Poisson counts are accumulated, or the total number of trials for Binomial response; if 0 is found in the vector, 1 will be used to replace all the values in the vector.
- `X:` a $n \times p$ covariate matrix (the default value `NULL` indicates no covariate).
- `beta:` a vector of length $(p + 1)$ that indicates the coefficients for covariates and intercept.
- `cov.par:` a vector of length 3 that indicates the value of (σ, ϕ, κ).
- `rho.family:` take the value of `"rhoPowerExp"` or `"rhoMatern"` which indicates the powered exponential or Matern correlation function is used.
- `Y.family:` take the value of `"Poisson"` or `"Binomial"` which indicates Poisson or Binomial distribution for response variables.

The output of this function is a list with two elements containing data for response and latent variables respectively.

2.3 Data Visualization

`plotData` function can be used to visualize the data, shown in Figure 2.3. The size of the "bubble" (which is the default shape and can be changed by setting `pch` parameter) represents the amount of the count on the location.

```
loc <- locGrid(1, 1, 10, 10)
dat <- simData(loc = loc, L = 0,
        X = NULL, beta = 4, cov.par = c(1, 0.2, 1),
        rho.family = "rhoPowerExp", Y.family = "Poisson")
plotData(dat$data, loc, xlab="x", ylab="y")
```

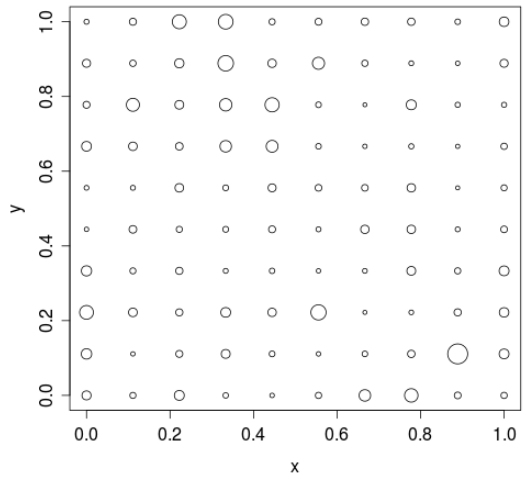

Figure 2.3: plotData - plots geostatistical data.

It is able to plot up to three data sets on one plot, shown in Figure 2.4.

```
loc <- rbind(locCircle(1, 60),
             locCircle(0.667, 40),
             locCircle(0.333, 20)
             )
dat <- simData(loc, cov.par = c(1, 0.2, 1))
Y <- dat$data
plotData(Y[1:60], loc[1:60, ], Y[61:100], loc[61:100, ],
         Y[101:120], loc[101:120, ], pch = 16,
         xlab="x", ylab="y"
         )
```

Another function `plotDataBD` is an enhanced version of `plotData`. Besides plotting the counts for the given locations, it is also able to plot the boundaries if the information is given. For example, after loading the boundary information for Texas counties from the integrated data set `TexasCounty.boundary`, we can use `plotDataBD` to plot it, shown in Figure 2.5.

```
data(TexasCounty_boundary)
plotDataBD(TexasCounty.boundary, xlab = "Longitude", ylab = "Latitude")
```

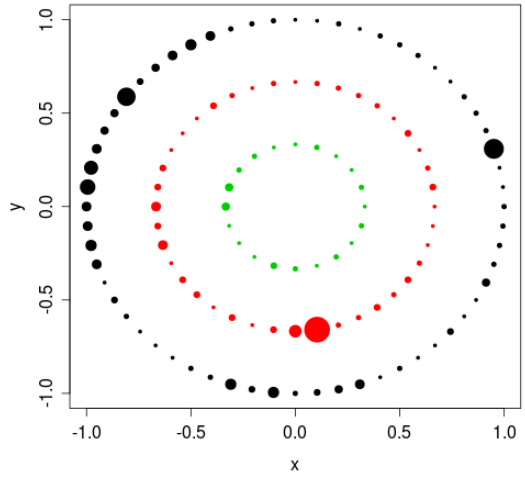

Figure 2.4: plotData2 - plots three geostatistical data sets.

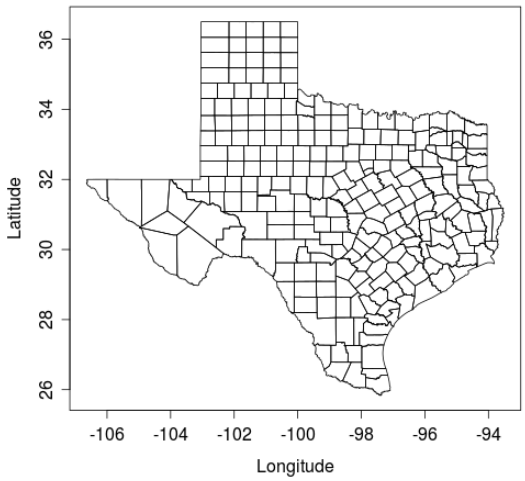

Figure 2.5: plotDataBD - plots the boundaries for Texas counties.

2.4 Integrated Data Sets

In the package, a few data sets are integrated:

`Rongelap` : introduced in Chapter 1, see Figure 1.1.

`Weed` : collected at the Bjertorp farm in the south-west of Sweden, see Figure 2.6. Weed counts of non-crop plants were observed at different locations, and camera recorded images were used to estimate the counts with the help of certain image analysis algorithm. Guillot, Loren, and Rudemo (2009).

`Earthquakes` : this data set contains information of the earthquakes with magnitude 1.0 or greater that happened during 05/20/2011 - 05/27/2011 worldwide (source: United States Geological Survey).

`TexasCounty.center` : this data set contains the central longitude and latitude coordinates for all the counties in Texas.

`TexasCounty.boundary` : this data set contains the boundary longitude and latitude coordinates for all the counties in Texas.

`TexasCounty.population` : this data set contains the information of poverty and total population in 2009 for all the counties in Texas (source: U.S. Census Bureau).

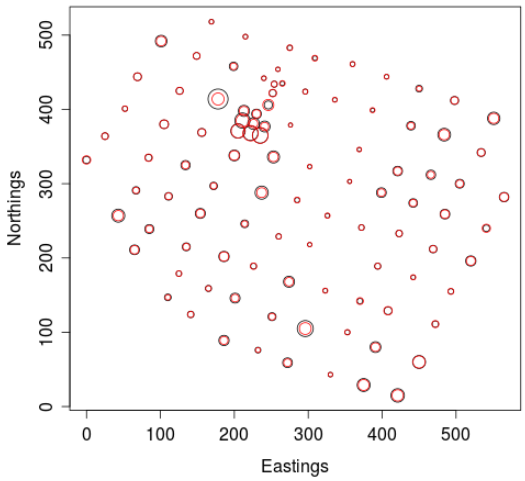

Figure 2.6: Weed - weed counts at the Bjertorp farm in the south-west of Sweden.

3

Estimation and Prediction

Robust Markov Chain Monte Carlo (MCMC) algorithms with the help of advanced techniques, such as group updating, Langevin-Hastings algorithms, and data-based transformations, are implemented in the package for posterior sampling and estimation. See my dissertation for details of the algorithms and techniques.

3.1 Posterior Sampling

3.1.1 Environment Setting

`MCMCinput` function can be used to set up the assumed model and environmental parameters for MCMC algorithms, and the setting can be saved for future use so you don't need to set it up every time you run MCMC algorithms.

```
input <- MCMCinput( run = 10000, run.S = 10,
        rho.family = "rhoPowerExp",
        Y.family = "Poisson", ifkappa=0,
        scales=c(0.5, 1.5, 0.9, 0.6, 0.5),
        phi.bound=c(0.005, 1),
        initials=list(c(-1, 2, 1), 1, 0.1, 1) )
```

The environmental parameters include:

- `run`: the number of iterations.
- `run.S`: the number of internal iterations for latent variables.

- `rho.family:` take the value of `"rhoPowerExp"` or `"rhoMatern"` which indicates the powered exponential or Matern correlation function is used.
- `Y.family:` take the value of `"Poisson"` or `"Binomial"` which indicates Poisson or Binomial distribution for response variables.
- `ifkappa:` take zero or non-zero value which indicates whether κ should be sampled.
- `scales:` a vector which indicates the tuning parameters for $(S, \beta, \sigma, \phi, \kappa)$ respectively.
- `phi.bound:` the upper and lower bound for ϕ.
- `initials:` a list which indicates the initial values for $(\beta, \sigma, \phi, \kappa)$ respectively.

During each iteration of Gibbs sampling process, the group of latent variables is updated `run.S` times to improve accuracy and reduce autocorrelations. Increasing the number of `run.S` usually does not increase the running time of the algorithms dramatically.

The setting of `scales` is very important which is directly related to the efficiency of the algorithms. Generally, you need to adjust the values of `scales` to achieve the acceptance rate of 0.574 for (S, β) and 0.25 for (σ, ϕ, κ), because the former is updated with Langevin-Hastings algorithms and the latter is updated with random-walk algorithms. See my dissertation for details.

`phi.bound` needs to be set in an appropriate range. Otherwise, it damages both the accuracy and efficiency of the algorithms. You can use `plotData` function to visualize your data set and calculate the empirical variogram to choose the range.

`initials` is important in some cases. Generally, the algorithms is able to converge to the correct region very fast. Try to use different `initials` if you have trouble in convergence, and deriving a reasonable `initials` from exploratory data analysis is always a good idea.

3.1.2 Run MCMC Algorithms

`runMCMC` is the function that performs robust MCMC algorithms for generalized linear spatial models and generates posterior samples for latent variables and hyper-parameters.

```
runMCMC(Y, L = 0, loc, X = NULL, run = 200, run.S = 1,
```

```
rho.family = "rhoPowerExp", Y.family = "Poisson", ifkappa = 0,
scales = c(0.5, 1.65^2 + 0.8, 0.8, 0.7, 0.15),
phi.bound = c(0.005, 1),
initials = list(c(1), 1.5, 0.2, 1),
MCMCinput = NULL, partial = FALSE, famT = 1)
```

where

- `Y:` a vector of length n which indicates the response variables.
- `L:` a vector of length n; it indicates the time duration during which the Poisson counts are accumulated, or the total number of trials for Binomial response; if 0 is found in the vector, 1 will be used to replace all the values in the vector.
- `loc:` a $n \times 2$ matrix which indicates the coordinates of the locations.
- `X:` a $n \times p$ covariate matrix (the default value `NULL` indicates no covariate).
- `...` (same as in `MCMCinput` function)
- `MCMCinput:` a list of alternative settings; usually the result from `MCMCinput` function.
- `partial:` a logical input which indicates whether partial posterior sampling should be used; only works for `Y.family = "Poisson"`.
- `famT:` take the value of 1, 2, or 3 which indicates the type of partial posterior sampling: 1 means "mean" diagnostic statistic is used, 2 means "maximum", and 3 means "minimum"; ignored if `partial=FALSE`.

If given, `MCMCinput` will override other environmental parameters.

For example of analyzing `Weed` data,

```
data(datWeed)
input.Weed <- MCMCinput( run=2000, run.S=1, rho.family="rhoPowerExp",
        Y.family = "Poisson", ifkappa=0,
        scales=c(0.2, 3.5, 0.9, 0.6, 0.5),
        phi.bound=c(0.5, 300),
        initials=list(c(1), 1, 0.1, 1) )
res <- runMCMC(Y=Weed[,3], L=0, loc=Weed[,1:2], X=NULL,
                MCMCinput=input.Weed )
```

Be careful that if you want to include the covariates in the model, make sure the number of covariate variables (`X`) and the number of coefficients (β in `initials`) are compatible. The latter is one larger than the former because of the intercept term. For

example, we include location coordinates as the covariates to model the linear spatial trends for `Weed` data,

```
input2.Weed <- MCMCinput( run=1000, run.S=1, rho.family="rhoPowerExp",
        Y.family = "Poisson", ifkappa=0,
        scales=c(0.5, 0.00005, 0.9, 0.9, 0.5),
        phi.bound=c(0.5, 300),
        initials=list(c(4, 0, 0), 1, 0.1, 1) )
res2 <- runMCMC(Y=Weed[,3], L=0, loc=Weed[,1:2], X=Weed[,1:2],
                MCMCinput=input2.Weed )
```

Sometimes the coordinates of the locations have a large range (in `Weed` data the range is 0 to 565 meters) which may lead to very small values of coefficients, so it is more convenient to scale the coordinates with `unifLoc` function before using them as covariates, especially for preliminary analysis.

The output of `runMCMC` function is a list with elements:

- `S.posterior`: a $n \times run$ matrix containing the posterior samples for latent variables.
- `m.posterior`: a $(p+1) \times run$ matrix (in case of p covariate variables) or a vector with length `run` (no covariate case), containing the posterior samples for β.
- `s.posterior`: a vector with length `run` containing the posterior samples for σ.
- `a.posterior`: a vector with length `run` containing the posterior samples for ϕ.
- `k.posterior`: a vector with length `run` containing the posterior samples for κ in the case that `ifkappa` is set to non-zero value.
- `AccRate`: a vector which indicates the acceptance rates.

On the "screen" (or in your terminal), the running time of the algorithm and the acceptance rates will be displayed, as well as the warning messages.

```
### MCMC Starts!

### MCMC Done!

### MCMC Running Time:
   user  system elapsed
 89.080   0.010  89.286
### MCMC Acceptance Rate:
```

```
accS1  accm  accs  acca
0.208 0.570 0.299 0.322
Warning message:

L contains zero!
L is set to 1 for all locations
```

3.1.3 Generate Parallel Chains

Alternatively, when your computer have more than one CPU available or you are using high performance computing (HPC) cluster, `runMCMC.multiChain` and `runMCMC.sf` can be used to perform robust MCMC algorithms and generate posterior samples in a parallel way. Essentially, these two functions enable different CPUs to run `runMCMC` function simultaneously with different initial values. The difference is `runMCMC.multiChain` performs parallel computing with the help of {`multicore`} package while `runMCMC.sf` uses the mechanism in {`snow`} and {`snowfall`} packages for parallel computing.

```
runMCMC.multiChain(Y, L = 0, loc, X = NULL, run = 200, run.S = 1,
        rho.family = "rhoPowerExp", Y.family = "Poisson", ifkappa = 0,
        scales = c(0.5, 1.65^2 + 0.8, 0.8, 0.7, 0.15),
        phi.bound = c(0.005, 1), initials = list(c(1), 1.5, 0.2, 1),
        MCMCinput = NULL, partial = FALSE, famT = 1,
        n.chn = 2, n.cores = getOption("cores"))
runMCMC.sf(Y, L = 0, loc, X = NULL, run = 200, run.S = 1,
        rho.family = "rhoPowerExp", Y.family = "Poisson", ifkappa = 0,
        scales = c(0.5, 1.65^2 + 0.8, 0.8, 0.7, 0.15),
        phi.bound = c(0.005, 1), initials = list(c(1), 1.5, 0.2, 1),
        MCMCinput = NULL, partial = FALSE, famT = 1,
        n.chn = 2, n.cores = getOption("cores"), cluster.type="SOCK")
```

where the input arguments are similar to the arguments in `runMCMC` except extra arguments for parallel computing,

- `n.chn`: the number of Markov chain sets that will be generated in parallel.
- `n.cores`: the number of CPUs that will be used to generate parallel Markov chains.

- `cluster.type`: type of cluster to be used for parallel computing; can be `"SOCK"`, `"MPI"`, `"PVM"`, or `"NWS"`.

In the case the number of available CPUs is less than `n.chn`, Markov chains will be put in a queue.

For example, the parallel version of analyzing `Weed` data is

```
require(multicore)
res.prl <- runMCMC.multiChain(Y=Weed[,3], L=0, loc=Weed[,1:2],
                   X=NULL, MCMCinput=input.Weed, n.chn=4, n.cores=4 )
```

or

```
require(snowfall)
res.prl <- runMCMC.sf(Y=Weed[,3], L=0, loc=Weed[,1:2],
                   X=NULL, MCMCinput=input.Weed, n.chn=4, n.cores=4,
                   cluster.type="SOCK" )
```

The output of `runMCMC.multiChain` and `runMCMC.sf` is a list with length equal to `n.chn`. Each element in this list is a list similar to the output from `runMCMC` function which contains the result of one set of posterior samples.

3.2 Posterior Sample Handling

3.2.1 Burn-in, Thinning, and Mixing

Once posterior samples are generated, `cutChain` function can be used to modify them for "burn-in" and "thinning". It takes a list with elements containing the posterior samples (usually the output from `runMCMC`) as input.

```
# For the output of runMCMC()
res <- runMCMC(Y, L, loc, X=NULL, MCMCinput=input )
res.m <- cutChain(res, chain.ind=1:4, burnin=1000, thinning=10)
```

For the output from parallel MCMC functions (`runMCMC.multiChain` and `runMCMC.sf`), `lapply` function is needed to apply `cutChain` on each element. Then use `mixChain` function to mix the corresponding parallel chains into one.

```
# For the output of runMCMC.multiChain() and runMCMC.sf()
res.prl <- runMCMC.multiChain(Y, L, loc, X=NULL, MCMCinput=input,
           n.chn=4, n.cores=4)
res.m.prl <- lapply(res.prl, cutChain, chain.ind=1:4, burnin=1000,
                    thinning=10)
res.m <- mixChain(res.m.prl)
```

***Caution: before mixing parallel chains**

It is necessary to examine at least one of the parallel chains to make sure they are already converged and well-mixed. See the next section for details about how to examine the chain of posterior samples.

3.2.2 Examine Posterior Samples

{coda} package is a well-known package for output analysis and diagnostics for MCMC simulations. It provides a variety of functions for visualizing the posterior samples and performing convergence tests. See the documentation of {coda} package for details. Here are a few examples,

```
# convert to mcmc class
chn1.mcmc <- mcmc(cbind(sigma=res.m$s, phi=res.m$a))
# basic information and summarized statistics
summary(chn1.mcmc)
# trace and density plots
plot(chn1.mcmc, auto.layout = TRUE)
# cross-correlation plot
crosscorr.plot(chn1.mcmc)
# auto-correlation plot
autocorr.plot(chn1.mcmc)
# effective sample size adjusted for autocorrelation
effectiveSize(chn1.mcmc)
# Geweke's convergence diagnostic
geweke.diag(chn1.mcmc, frac1=0.1, frac2=0.5)
# Geweke-Brooks plot
geweke.plot(chn1.mcmc, frac1=0.1, frac2=0.5)
# Heidelberger and Welch's convergence diagnostic
```

```
heidel.diag(chn1.mcmc, eps=0.1, pvalue=0.05)
```

Besides the functions in {coda} package, `plotACF` is provided specifically for plotting auto-correlation curves for latent variables, shown in Figure 3.1, and `findMode` is provided to estimates the mode of empirical density function for posterior samples.

```
plotACF(res.m$S.posterior)
phi.est <- findMode(res.m$a.posterior)
```

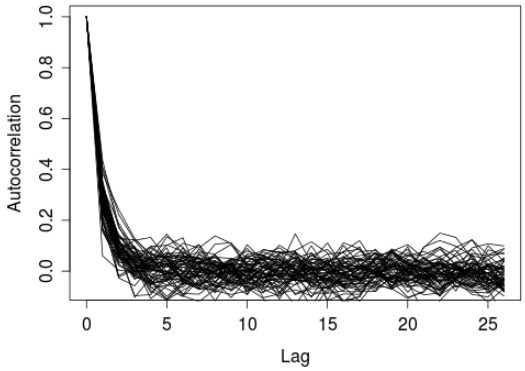

Figure 3.1: plotACF - plots auto-correlation curves for latent variables.

3.3 Prediction

Based on the posterior samples from sampled locations, `preY` function can generate posterior predictive samples of latent and response variables for unsampled locations.

```
predY(res.m, loc, locp, X = NULL, Xp = NULL, Lp = 0, k = 1,
      rho.family = "rhoPowerExp", Y.family = "Poisson",
      parallel = NULL, n.cores = getOption("cores"),
      cluster.type = "SOCK")
Ypred.avg <- rowMeans(Ypred$Y)
```

The coordinates for both sampled and unsampled locations (`loc` and `locp`) are needed as well as the covariate matrix (`X` and `Xp`). Also, this function can perform in parallel way with the help of {`multicore`} (or {`snowfall`}) package if letting

`parallel="multicore"` (or `"snowfall"`) and using `n.cores` to specify the number of CPUs that will be used for parallel computing.

Continue with the example for `Weed` data. For 50 unsampled locations, their predictions (the average of posterior predictive samples for response variables) are shown in Figure 3.2.

```
locp <- cbind(Weed[1:50,1]+30, Weed[1:50,2])
Ypred <- predY(res.m, loc=Weed[,1:2], locp, X=NULL, Xp=NULL,
               Lp=rep(1, nrow(locp)), k=1,
               rho.family="rhoPowerExp", Y.family="Poisson")
plotData(Weed[,3], Weed[,1:2], Ypred.avg, locp,
         xlab="Eastings", ylab="Northings")
```

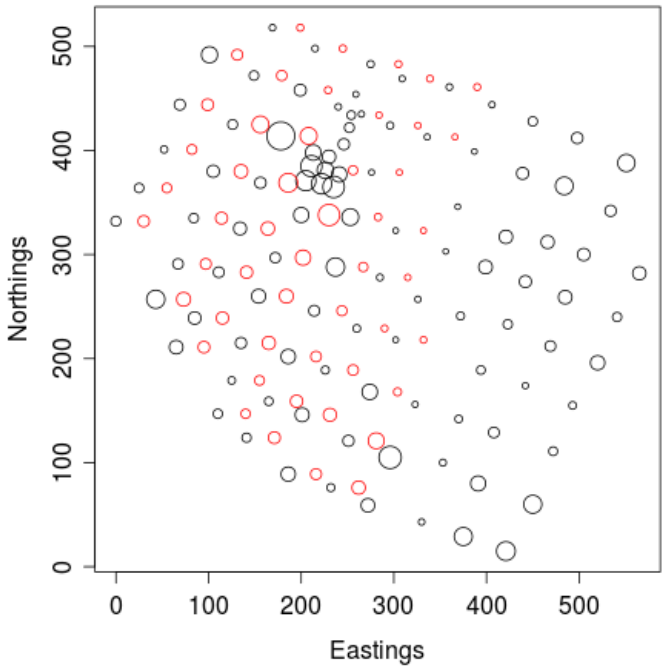

Figure 3.2: predY - Prediction for `Weed` data: black circles indicate sampled locations and red circles indicate unsampled locations (the size of bubble indicates the amount of weed counts).

4

Model Checking

4.1 Bayesian Model Checking

To perform Bayesian model checking procedure, we need

1. define a diagnostic statistic which summarizes the data set and represents certain feature;
2. simulate a (large) number of reference data sets;
3. compare the diagnostic statistics derived from the observed and reference data sets and the discrepancy will reveal the information of goodness of fitting.

4.1.1 Define Diagnostic Statistic

We need to define a function which take the vector of response variables as input and output the diagnostic statistic, for example,

```
# the average as diagnostic statistic
funcT <- function(Y){ mean(Y) }

# the Pearson residual type of diagnostic statistic
funcT <- function(Y){ sum((Y - mean(Y))^2/var(Y)) }
```

4.1.2 Simulate Reference Data Sets

There are two functions in the package for simulating reference (replicated) data sets: `repYeb` and `repYpost`. `repYeb` simulates replicated data sets from the model with

known parameters which are either pre-determined or estimated from posterior samples, while `repYpost` simulates them based on posterior samples of latent variables.

```
repYeb(N.sim, loc, L, X = NULL, rho.family = "rhoPowerExp",
       Y.family="Poisson", res.m = NULL, est = "mode",
       beta = NULL, sigma = NULL, phi = NULL, k = 1)
}
repYpost(res.m, L, Y.family="Poisson")
```

where

- `N.sim`: the number of replicated data sets to be simulated.
- `est`: take the value of `"mode"` which indicates the mode of posterior samples will be used as the parameter estimate; otherwise, the mean will be used.

The output of both functions is a matrix ($n \times N.sim$ for `repYeb` and $n \times N$ for `repYpost` where N is the length of posterior samples in `res.m`) containing $N.sim$ or N replicated data sets.

There are two ways to use `repYeb`: (1) input the result containing posterior samples (`res.m`) and set up `est` to use either posterior mode or mean as the estimates of model parameters; or (2) do not input `res.m` but input pre-determined model parameters (maybe obtained from another estimating process). Be aware that the second way can be used as an alternative way to simulate massive data sets with given parameters.

```
# Estimate parameters from posterior samples
Yrep <- repYeb(N.sim=2000, loc, L, res.m = res.m, est = "mode")
# Pre-determined parameters
Yrep <- repYeb(N.sim=2000, loc, L, beta = 5, sigma = 1,
                  phi = 0.1, k = 1)
```

The use of `repYpost` is fairly easy since you only need to input the result containing posterior samples for latent variables (`res.m`) and specify the distribution for response variable. Notice that the posterior samples for parameters in `res.m` are not used in this function.

4.1.3 Compare Diagnostic Statistics

`pRPS` calculates the *p-value* and *relative predictive surprise* (RPS) by comparing diagnostic statistics from the observed and reference data sets and `plot_pRPS` provides the visualization. For example, if we use 2 as the diagnostic statistic from the observed data set and use some normal random samples as the diagnostic statistics from the reference data set, the result is shown in Figure 4.1.

```
pRPS(T.obs = 2, T.rep = rnorm(1000))
plot_pRPS(2, rnorm(1000), nm="t")
```

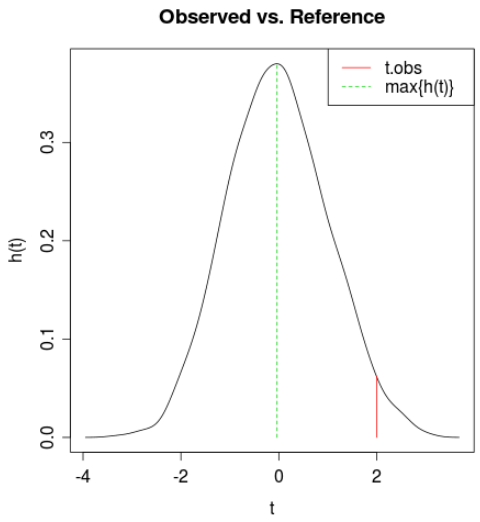

Figure 4.1: plot_pRPS - compare diagnostic statistics from the observed and reference data sets.

Overall, `BMCT` function conducts Bayesian model checking by comparing observed and reference data sets with respect to defined diagnostic statistic and produces the result of "p-value" and "RPS" (as well as the plot if `ifplot = TRUE`).

```
BMCT(Y.obs, Y.rep, funcT, ifplot = FALSE)
```

4.2 Transformed Residual Checking

To perform transformed residual checking procedure, we need

1. simulate a (large) number of reference data sets from the fitted model to approximate the transformed residuals for observed data;
2. examine the distribution of transformed residuals and compare it with standard normal distribution by using graphical tools;
3. calculate the Hellinger distance for transformed residuals with standard normal distribution served as reference distribution;
4. build the "baseline distribution" for standard normal distribution;
5. compare the Hellinger distance calculated from observed data with the baseline distribution and calculate the p-value;
6. determine the goodness of model fitting based on the information of graphs and p-value.

4.2.1 Approximate Transformed Residuals

Since the reference data sets come from the fitted model, we can easily use `repYeb` function to simulate them.

> **Caution:** reference data sets simulated by `repYpost` function are not appropriate for transformed residual checking.

With the reference data sets, `tranR` function can approximate transformed residuals for the observed data by using reference data.

```
Y.rep <- repYeb(N.sim = 5000, loc, L, res.m = res.m)
etran <- tranR(Y.obs, Y.rep, discrete = FALSE)
```

where `discrete` indicates if the distribution of response variable is discrete (the approximating method is different for discrete and continuous distribution). However, we notice that in practice setting `discrete = FALSE` helps increase the accuracy of approximation even the distribution of response variable is actually discrete (unless the number of reference data sets is large enough).

4.2.2 Plot Transformed Residuals

`plot_etran` function can plot transformed residuals in different types of plot: scatter plot, QQ-plot, density plot, and relative density plot (with standard normal distribution served as the base). Inside of this function, `reldist` function in {`reldist`} package is used to compute the relative density, so it is required.

```
require(reldist)
plot_etran(etran, fig = 1:4)
```

where `fig` indicates which types will be plotted. An example of the plots is shown in Figure 4.2.

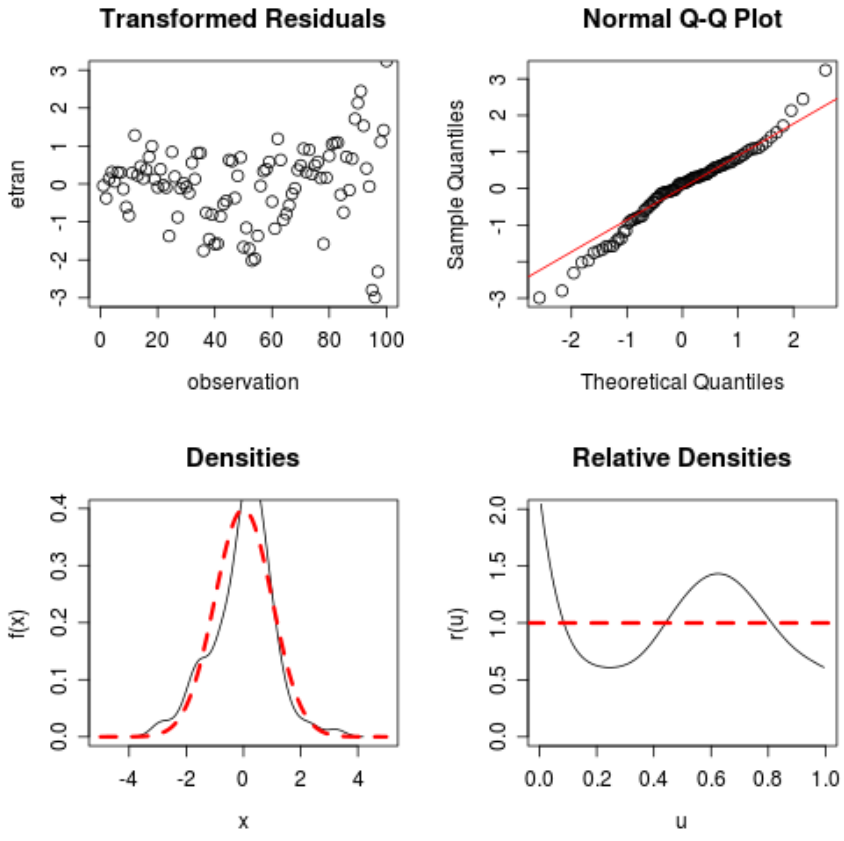

Figure 4.2: plot_etran - plots transformed residuals.

4.2.3 Calculate Hellinger Distance

To compare the distribution of transformed residuals with standard normal distribution, `e2dist` function can be used to calculate the distance between them. Inside of this function, `HellingerDist` and `KolmogorovDist` functions in {`distrEx`} package are used to compute two types of *Hellinger distance* ("Discrete Hellinger" and "Smooth Hellinger") and *Kolmogorov distance*. The difference between the two types of Hellinger distance is that

- for "Discrete Hellinger", a discretization of standard normal distribution is conducted, and the distance is computed between empirical distribution of transformed residuals and the discretized distribution;
- for "Smooth Hellinger", the transformed residuals are convoluted with the normal distribution `Norm(mean = 0, sd = h.smooth)` which leads to an absolutely continuous distribution, and afterwards the distance between the smoothed empirical distribution and standard normal is computed.

```
d.obs <- e2dist(etran)
```

The output is a vector with length 3, for example

```
> d.obs
Hellinger.discre Hellinger.smooth       Kolmogorov
       0.3442068        0.2200154        0.096800
```

4.2.4 Build Baseline Distribution

`baseline.dist` function generates the samples of distances to build the baseline distribution for standard normal distribution.

```
d.base <- baseline.dist(n, iter)
```

where `n` is the number of transformed residuals which should be equal to the number of locations in the observed data and `iter` is the number of distance samples to generate.

Alternatively, `baseline.parallel` function is a parallel version of `baseline.dist`. It generates samples in parallel way.

```
baseline.parallel(n, iter, n.cores = getOption("cores"))
```

Be aware that the samples for baseline distribution are only needed to be generated once and then can be saved for future use for any other observed data with the same size. A integrated data set `Dbase_n100N5000` contains the baseline samples for 100 residuals with 5000 iterations.

```
data(Dbase_n100N5000)
str(d.base)
```

The output of `baseline.dist` and `baseline.parallel` is a $iter \times 3$ matrix for three types of distance: "Discrete Hellinger" and "Smooth Hellinger", and "Kolmogorov".

With these samples, `plot_baseline` can be used to visualize the baseline distribution of distance. For example, the baseline distribution of "Discrete Hellinger" for 100 residuals is shown in Figure 4.3.

```
plot_baseline(d.samples = d.base[,1], dist.name = colnames(d.base)[1])
```

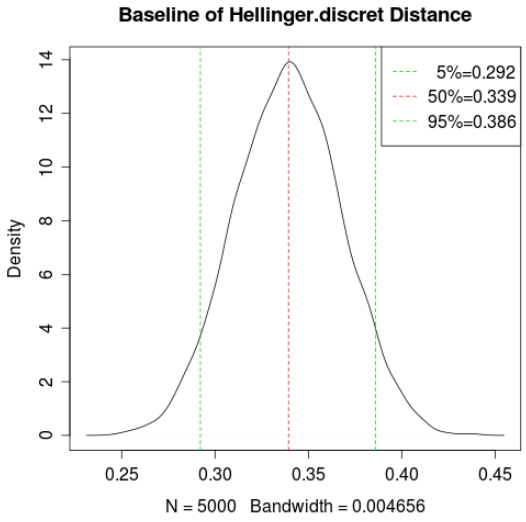

Figure 4.3: plot_baseline - the baseline distribution of "Discrete Hellinger" for 100 residuals.

4.2.5 Determine the Goodness of Fitting

By comparing the distance calculated from the observed data and the corresponding baseline distribution of distance, we can calculates one-side p-value with `pOne` function,

```
pOne(d.obs, d.base)
```

where `d.obs` is a value (or a vector) containing the distance for observed data and `d.base` is a vector (or a matrix) containing the samples for the baseline. The output is a p-value (or a vector of p-values). If p-value is small, it suggests that the transformed residuals unlikely follow standard normal distribution which means our assumptions about the fitted model is wrong. These assumptions include

1. the assumed model is the true model;
2. the estimation of model parameters is accurate.

See my dissertation for details.

5

Installation and Running

5.1 Dependent Tools and Libraries

The dependent tools and libraries (packages) that are required for installing {`geoCount`} include,

- R (>= 2.12.0)
- C++ compiler: for example GNU Compiler Collection (GCC)
- GNU Scientific Library (GSL): a numerical library for C and C++
- LAPACK and BLAS (or ATLAS): libraries for numerical linear algebra operations
- Standard development tools such as make etc.
- R packages
  - {`Rcpp`} (>= 0.9.4): provides a C++ API as an extension to the R system
  - {`RcppArmadillo`} (>= 0.2.19): integration for "Armadillo" which is a templated C++ linear algebra library
  - {`coda`}: for Markov chain diagnostics
  - {`distrEx`}: for calculating Hellinger and Kolmogorov distances between two distributions
  - {`reldist`}: for calculating the relative density
  - {`multicore, snow, snowfall`}: for parallel computing

Caution: {`Rcpp`} and {`RcppArmadillo`} are required to install {`geoCount`}. Other R packages are only required when you use the corresponding functions in the package.

5.2 Install in Linux/Unix

5.2.1 Install Dependent Tools

Usually, the compliers, libraries, and standard development tools are already installed in your system since they have very common use for many other packages and software.

If that is not the case, to install them in **Ubuntu**, the most convenient way is open "**Synaptic Package Manager**" and search the keywords ("gcc", "gsl", "blas", "lapack", etc.). From the searching result, choose the packages to install. Or, you can directly install them in terminal by using the command,

```
sudo apt-get install thepackage
```

In **Unix** system, you may need to contact the system administrator for installing.

The dependent R packages can be installed in R from the GUI menu or by using the command,

```
install.packages("packagename")
```

5.2.2 Install {`geoCount`}

Once the dependent tools and packages are installed, in R you can install {`geoCount`} from the source code by using the command,

```
install.packages("geoCount_1.1.tar.gz")
```

or in terminal by using the command,

```
R CMD INSTALL geoCount_1.1.tar.gz
```

See here[1] for details if you have trouble.

[1] http://cran.r-project.org/doc/manuals/R-admin.html#Installing-packages

5.3 Install in Windows

5.3.1 The Simple Way

I already compiled {`geoCount`} in Windows for 32-bit R-2.14.1. So from the compiled binary "zip" file, the installation of {`geoCount`} can be easily done in R from the GUI menu (choose "install from local zip file") or by using the command,

```
install.packages("geoCount_1.1.zip")
```

5.3.2 The Hard Way

- First, you need to build `GSL` (to make life easier, you can find a binary installer here[1] or use `gnuwin32-gsl`[2]).
- Second, add `LIB_GSL` variable into your environment by selecting

  ```
  Control Panel -> System -> Advanced -> Environment Variables
  -> Add
  ```

  Add a variable with name `LIB_GSL` and the variable value will be the path to `GSL` (where you installed it).

  > **Caution:** if you installed `GSL` at `C:\GSL` for example, the value you need to input for `LIB_GSL` variable is `C:/GSL`, NOT `C:\GSL`! You need to use "/" to replace "\"!

  If you don't want to change your environment or have trouble to add `LIB_GSL` variable, you can do the following instead.

  Alternatively, you can unzip the source code from {`geoCount_1.1.tar.gz`} and find `Makevars.win` under the directory of

  ```
  /PathTo/geoCount/src/
  ```

  Open it with an editor and you will see

[1] http://ascend4.org/Binary_installer_for_GSL-1.13_on_MinGW
[2] http://gnuwin32.sourceforge.net/packages/gsl.htm

```
## This assumes that the LIB_GSL variable points to .......
PKG_CPPFLAGS=-I$(LIB_GSL)/include -I../inst/include

## This assume that we can call Rscript to ask Rcpp about .......
## Use the R_HOME indirection to support installations of .......
PKG_LIBS=-L$(LIB_GSL)/lib -lgsl -lgslcblas $(shell $(R_HOME)/.......
```

You can replace `$(LIB_GSL)` (there are two) with the correct path of `GSL` (for example `C:/GSL`, NOT `C:\GSL`!).

- Third, install the Windows toolset for R (essential:"Rtools"; optional: MikTeX) from here[1].

- Four, set `PATH` variable by selecting

  ```
  Control Panel -> System -> Advanced -> Environment Variables
  -> PATH (edit it!)
  ```

  This could be done when you installing "Rtools". If not, make sure you have the following paths added in the `PATH` variable,

  ```
  C:\Rtools\bin;C:\Rtools\MinGW\bin;C:\Program files\R\R-2.14.1\bin\i386;
  ```

- Five, open "Dos Command" window and go to the the directory containing the source package. Install {`geoCount`} by the command,

  ```
  R CMD INSTALL geoCount_1.1.tar.gz
  ```

  or in the case you have unzipped the file and modified the `Makevars.win` file, use the command,

  ```
  R CMD INSTALL geoCount
  ```

Done!

[1] http://cran.r-project.org/doc/manuals/R-admin.html#The-Windows-toolset

5.4 Conflict with Optimized BLAS

Basic Linear Algebra Subprograms (BLAS[1]) is a *de facto* application programming interface standard for publishing libraries to perform basic linear algebra operations such as vector and matrix multiplication. Most linear algebra operations done in R are actually passed to BLAS to finish. R, by default, uses an implementation of BLAS which is not optimized. By using an optimized BLAS which is able to enhance multi-threaded execution of BLAS routines, you can make your linear algebra operations in R run faster. Besides vendor-provided optimized BLAS libraries, GotoBLAS[2] is a very good free version available for a given set of machines, and J. P. Olmsted[3] provides a tutorial on how to use an optimized BLAS with R.

Simply speaking, without optimized BLAS R performs the linear algebra operations with single thread (CPU), and with optimized BLAS R is able to performs the linear algebra operations with multiple threads (CPUs).

However, multi-threaded execution of linear algebra operations with optimized BLAS libraries is conflicted with the mechanisms of parallel computing provided in {`multicore`}, {`snow`}, and {`snowfall`} packages, because they are both trying to perform multi-threaded computing and "fighting" against each other. So when using parallel functions in {`geoCount`} (or other parallel functions in {`multicore`}, {`snow`}, and {`snowfall`} packages), the optimized BLAS libraries should not be used which will reduce the performance (or even terminate the program).

5.5 Running on High Performance Cluster

A typical HPC cluster usually consists of hundreds or thousands of processors, for example "cheetah.cbi.utsa.edu" cluster at UTSA has 392 processing cores with 2 GB RAM per core. Considering it serves for many individual clients to perform different computing jobs, a job management system (also called job scheduler) is installed on the cluster to handle the job requests from all the clients, schedule and assign available processors to perform the jobs. To run R codes in parallel way on cluster, we need

[1] http://www.netlib.org/blas/

[2] http://www.tacc.utexas.edu/tacc-projects/gotoblas2/

[3] http://www.rochester.edu/college/gradstudents/jolmsted/files/computing/BLAS.pdf

to write a job script that describes our request of computing resource (how many processors we need? for how long?) and computing environment.

For Sun Grid Engine system installed on "cheetah.cbi.utsa.edu", a sample job script is shown below.

```
#!/bin/bash
# The name of the job
#$ -N R-parallel
# Giving the name of the output log file
#$ -o R_parallel.log
# Combining output/error messages into one file
#$ -j y
# One needs to tell the queue system
# to use the current directory as the working directory
# Or else the script may fail as it will execute
# in your top level home directory /home/username
#$ -cwd
# Here we tell the queue that we want the orte parallel
# enivironment and request 5 slots
# This option take the following form: -pe nameOfEnv min-Max
#$ -pe orte 5-10

# Now come the commands to be executed

setenv PATH ${PATH}:/share/apps/gsl/bin
setenv LD_LIBRARY_PATH /share/apps/gsl/lib:${LD_LIBRARY_PATH}

/opt/openmpi/bin/mpirun -n $NSLOTS R --vanilla
< test_snow_bootstrap.R > test_snow_bootstrap.log

exit 0
```

> **Caution:** after the job parameters are set up, the path to the libraries required for the computation needs to be added into environment and then the R code can be executed in batch mode.

Another way to utilize many processors on HPC cluster is submitting a job array (containing many similar job requests). A sample job script is shown below.

```
#!/bin/bash
#$ -N R-array
#$ -o R_jobarray.log
#$ -j y
#$ -cwd
#$ -t 1-10
# -M liang.jing@utsa.edu
# -m e

setenv PATH ${PATH}:/share/apps/gsl/bin
setenv LD_LIBRARY_PATH /share/apps/gsl/lib:${LD_LIBRARY_PATH}

R --vanilla --args $SGE_TASK_ID < test_jobarray.R
```

where `-t 1-10` describes how many times the program will be repeated with the task ID `1-10` assigned to each job. By taking the parameter `$SGE_TASK_ID` as an input argument into R, the R program will be able to perform different codes for each running job. The corresponding R code should be in the following form.

```
### test_jobarray.R
myarg <- commandArgs()
id <- as.numeric(myarg[length(myarg)])

if(id == 1) {
    # use method 1
...
}
if(id == 2) {
    # use method 2
...
}
...
```

or something like

```
### test_jobarray.R
myarg <- commandArgs()
id <- as.numeric(myarg[length(myarg)])

data <- dataSets[[id]]

# analyze the data
...
```

More details about how to write job scripts can be found on Oxford e-Research Center[1].

[1] http://www.oerc.ox.ac.uk/computing-resources/osc/support/documentation-help/job-schedulers/

References

[1] M. J. Bayarri and M. E. Castellanos. Bayesian checking of the second levels of hierarchical models. *Statistical Science*, 22:322, 2007.

[2] G. E. P. Box and D. R. Cox. An analysis of transformations (with discussion). *Journal of the Royal Statistical Society, Series B*, 26:211, 1964.

[3] O. F. Christensen, G. O. Roberts, and M. Sköld. Robust markov chain monte carlo methods for spatial generalized linear mixed models. *Journal of Computational and Graphical Statistics*, 15:1, 2006.

[4] V. De Oliveira, B. Kedem, and D.A. Short. Bayesian prediction of transformed gaussian random fields. *Journal of the American Statistical Association*, 92:1422, 1997.

[5] P. J. Diggle, L. Harper, and S. L. Simon. *Statistics for the environment 3: pollution assesment and control*, chapter Geostatistical analysis of residual contamination from nuclea testing, page 89. Wiley, 1997.

[6] P. J. Diggle and P. J. Ribeiro. *Model-based Geostatistics*. Springer Series in Statistics, 2007.

[7] P. J. Diggle, J. A. Tawn, and R. A. Moyeed. Model based geostatistics (with discussion). *Applied Statistics*, 47:299, 1998.

[8] T. Gneiting. *Symmetric Positive Definite Functions with Applications in Spatial Statistics*. PhD thesis, University of Bayreuth, 1997.

[9] L. Jing. *Bayesian Model Checking for Generalized Linear Spatial Models for Count Data*. PhD thesis, University of Texas at San Antonio, 2011.

[10] B. Matern. Spatial variation. Technical report, Statens Skogsforsningsinstitut, Stockholm, 1960.

[11] S. E. Olsen. Positionsbestemt dyrkning (forsknings-projekt vedr. gradueret plant-edyrkning 1992-1996). Technical report, Danmarks JordbrugsForskning, 1997.

[12] Health Resources and Services Administration. Health professions, area resource file (arf) system. Technical report, Quality Resource Systems, Inc., Fairfax, VA, 2003.

[13] M. Schlather. Introduction to positive definite functions and to unconditional simulation of random fields. Technical report, Dept. Maths and Stats, Lancaster University, Lancaster, UK, 1999.