Chapter 5

Lattice Data: Simulation and Estimation

Based on existing spatial R packages, we assess spatial dependency and fit simple GMRF models to data. Emphasis on computational aspects is given as well.

R-Code for this chapter: www.math.uzh.ch/furrer/download/sta330/chapter05.R.

5.1 Spatial Objects in R

In the last chapter, we have (statistically) introduced GMRF for regular and irregular lattices. The latter are more present in real-world applications. Unfortunately, much of the time in any data analysis is spent visualizing the data and gathering and assembling each area's boundary information. Before the actual modeling, we look at some indispensable software components when visualizing areal data.

For spatial data analysis, we need at least a list of (named) polygons from a database and a list of neighbors for each of these. Ideally, we have direct access to plotting methods for the polygons. The neighborhood structure is often given as a list or an adjacency matrix and may often be constructed based the polygons themselves.

The packages *sf*, *spdep*, and *spatialreg* provide a framework for handling, analyzing, and plotting spatial data. Often it is intimidating to get acquainted with the various R functions and classes. The maintainers of the above packages tightly collaborate and are aware of the technical overhead that might distract from a statistical analysis. For more details, a good start is Chapter 2 of Bivand *et al.* (2013). The basic idea of the packages is to store all objects within a family of classes, define many methods for these classes, and provide a useful number of helping functions. Thus many situations, we do not need to worry about the underlying technical details. For didactic purposes, we often illustrate the formal and the "manual" handling of the data.

In the R-Code below, we illustrate two approaches, a manual and a "formal" one. The former one is based on simple lists containing the x and y coordinates of the polygons, bounding box,

and polygon names. Thus the plotting can be done manually. The latter one relies on the formal *sf* class. Thus the objects are more complicated but plotting is easier.

R-Code 5.1: Handling spatial objects in R.

```
### First example (simple version):
library(maps)          # simple database
ncMaps <- map("county", region="North Carolina", fill=TRUE, plot=FALSE)
###   "fill=TRUE" is very important! We need regular polygons.
str(ncMaps, strict.width="cut") # no formal class, just a particular list.
## List of 4
##  $ x    : num [1:3771] -79.5 -79.5 -79.5 -79.5 -79.5 ...
##  $ y    : num [1:3771] 35.8 35.9 35.9 36 36.2 ...
##  $ range: num [1:4] -84.3 -75.5 33.9 36.6
##  $ names: chr [1:102] "north carolina,alamance" "north carolina,alexand"..
##  - attr(*, "class")= chr "map"
### 100 Counties, "currituck" consisting of 3 polygons. Polygons are separated
###   with separated NAs.
sum(is.na(ncMaps$x))
## [1] 101
### Plotting is done with "plot=TRUE" (default) in the function `map()` and
###   color specification. Alternatively:
plot(ncMaps$x, ncMaps$y, type="n")
polygon(ncMaps$x, ncMaps$y, col=sample(1:16))
### We import shape-files from different database:
nc <- st_read(system.file("shapes/sids.shp", package="spData")[1], quiet=TRUE)
nc$rates <- nc$SID74 / nc$BIR74
class(nc)           # classes "sf" and "data.frame"
## [1] "sf"          "data.frame"
dim(nc)             # for each county a lot of information
## [1] 100  24
str(nc$geometry)    # spatial info in one list element
## sfc_MULTIPOLYGON of length 100; first list element: List of 1
##  $ :List of 1
##   ..$ : num [1:27, 1:2] -81.5 -81.5 -81.6 -81.6 -81.7 ...
##  - attr(*, "class")= chr [1:3] "XY" "MULTIPOLYGON" "sfg"
### see methods(class="sf")
### The following for an even more formal analysis
# st_crs(nc) <- "+proj=longlat +datum=NAD27"
# row.names(nc) <- as.character(nc$FIPSNO)
```

5.2 Assessing Spatial Dependency

Correlation in time series is based on the evaluation of lagged values, e.g., analyzing $Y_t - Y_{t-1}$, where Y_{t-1} is the lagged value of Y_t. With lattice data, we can construct a spatial lag by considering all first-order neighbors. Depending on the application, the neighbors are weighted:

$$\sum_{j=1}^{n} w_{ij} Y_j. \tag{5.1}$$

Weights such that rows sum to one are natural and are often considered.

In the case of (arbitrary) lattice data, *Moran's I* and *Geary's C* are often used metrics to assess spatial dependencies. Let $\mathbf{Y}$ be a multivariate random n-vector and $\mathbf{W} = (w_{ij})$ a matrix of spatial weights, often encoding a first-order neighborhood structure. Moran's I is defined as

$$I = \frac{n}{\sum_{i,j} w_{ij}} \frac{\sum_{i,j} w_{ij}(Y_i - \overline{Y})(Y_j - \overline{Y})}{\sum_i (Y_i - \overline{Y})^2} \quad \in [-1, 1]. \tag{5.2}$$

Positive (negative) values of the observed statistic indicate positive (negative) spatial autocorrelation. Note that under no spatial dependency $\mathrm{E}(I) = -1/(n-1)$. Closed-form expressions for the variance exist.

Geary's C is defined as

$$C = \frac{n-1}{2\sum_{i,j} w_{ij}} \frac{\sum_{i,j} w_{ij}(Y_i - Y_j)^2}{\sum_i (Y_i - \overline{Y})^2} \quad \in [0, 2]. \tag{5.3}$$

Smaller values indicate stronger positive spatial dependency. Hence, for interpretability and comparability with Moran's I, it would make more sense to consider $1 - C$ instead of C. Moran's I and Geary's C are measures of global spatial autocorrelation. However, Geary's C is more sensitive to local spatial autocorrelation.

Example 5.1. R-Code 5.2 illustrates the Moran's I and Geary's C for the *SIDS* data. The construction of the spatial weight matrix $\mathbf{W} = (w_{ij})$ is based on neighbors, i.e., if two counties i and j are neighbors, $w_{ij} = 1$ and zero otherwise. Here we start with *nc* from Example 5.1, a *sf* object, and construct the matrix using first the function *poly2nb()*. The argument *queen=FALSE* implies that more than one shared boundary point is necessary. Typically, this means one shared boundary segment. The neighbor structure is transformed to a spatial weight matrix with *nb2listw()*. The argument *style="B"* enforces the binary coding of w_{ij}. Note that we have 100 counties. Using the polygon definition from *ncM* results in 102 polygons, and we would not have the appropriate number of neighbors. ♣

R-Code 5.2: Tests for spatial autocorrelation for the SIDS dataset.

```
(ncnb <- poly2nb(nc, queen=FALSE))     # polygons to neighbors
```

```
## Neighbour list object:
## Number of regions: 100
## Number of nonzero links: 462
## Percentage nonzero weights: 4.62
## Average number of links: 4.62
ncW <- nb2listw(ncnb,  style="B")      # neighbors to weightmatrix
nc$rates <- nc$SID74 / nc$BIR74        # as in last chapter!
moran.test(nc$rates, ncW)              # testing spatial dependencies 1
##
##  Moran I test under randomisation
##
## data:  nc$rates
## weights: ncW
##
## Moran I statistic standard deviate = 3.9, p-value = 4.8e-05
## alternative hypothesis: greater
## sample estimates:
## Moran I statistic       Expectation          Variance
##         0.2336975        -0.0101010         0.0039055
t2 <- geary.test(nc$rates, ncW)        # testing spatial dependencies 2
1 - t2$estimate[1]                     # to better compare both
## Geary C statistic
##           0.32647
```

Moran's I and Geary's C measure the dependency "globally", i.e., one single value. It is possible to assess the dependency locally. For each individual area i the local version of Moran's I is given by

$$I_i = \frac{(Y_i - \bar{Y})}{\sum_{k=1}^{n}(Y_k - \bar{Y})^2/(n-1)} \sum_{j=1}^{n} w_{ij}(Y_j - \bar{Y}), \tag{5.4}$$

where again, different books or software implementations use some variations. The local version is linked to the global one, up to a slight difference in the normalization.

The local spatial autocorrelation can further be exploited with functions `moran.plot()` and `localmoran()` (Bivand *et al.*, 2013, Section 9.3.2). The output of these functions are not straightforward to interpret. Additionally, one needs to be careful not to over-interpret single p-values from the many involved tests.

Example 5.2. R-Code 5.3 illustrates local Moran's I for the `SIDS` data. The visualization is based on a function adapted from https://github.com/gisUTM/spatialplots and included in the chapter's R script. The interpretation is not straightforward, as there seems to be a missing symmetry. The lower panel of Figure 5.2 plots the lagged rates versus the rates. Alignment along the diagonal line indicates spatial dependency. The plot also marks outlying values from the regression `lm(y~Wy)` where `Wy=W %*% y` are the lagged variables. ♣

R-Code 5.3: Tests for spatial autocorrelation for the SIDS dataset.

```
ncW <- nb2listw(ncnb, style="W") # neighbors to weightmatrix, normalized!
local.moran <- localmoran(nc$rates, ncW)
plot.localmoran(nc, "rates", local.moran=local.moran, weights=ncW)

moran.plot(nc$rates, ncW, xlab="SIDS rates", ylab="Lagged variable",
    labels=FALSE, ylim=c(0, .01), xlim=c(0, 0.01))
abline(c(0,1), col="gray")
```

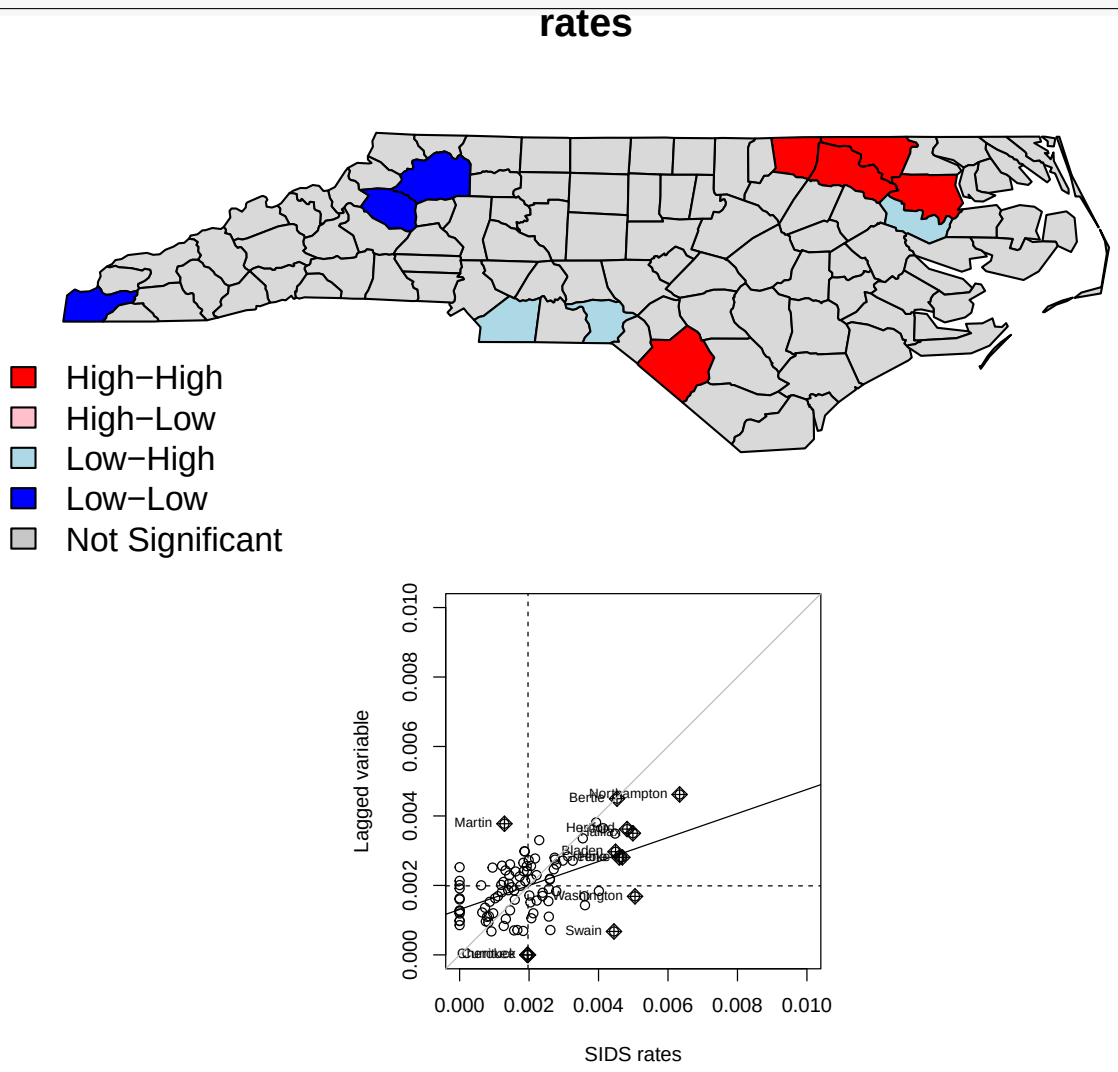


Figure 5.1: Visualization of local Moran's I. (See R-Code 5.3.)

In certain situations, some areas/polygons do not have neighbors. In such situations, most approaches may handle such polygons differently (e.g., the resulting value is zero or *NA*). The argument *zero.policy* specifies the choices.

If data is available on a regular grid, autocovariances along the dimensions can be calculated similarly to time series. Under the assumption of stationarity, it is possible to estimate an autocovariance function based on the distance between the grid points. This approach is much more natural in the framework of geostatistics, and we discuss it extensively in Chapter 10.

5.3 Specific Models for GMRF

We now introduce a low-dimensional parametrization of a GMRFs by reducing the number of parameters $\{b_{ij}\}$ and τ_i^2 under the symmetry constraint and positive definiteness of the precision matrix.

We often assume that $\tau_i = \tau$, for all i. That means that the conditional precision is constant. Another simplification is that the coefficients b_{ij} do not depend on j, i.e., $b_{ij} = b_i$, no preference is given to "neighboring" information. We might even further simplify to $b_{ij} = b = \theta$ for all $j \sim i$, leading to

$$Y_i \mid \boldsymbol{y}_{-i} \sim \mathcal{N}\Big(\sum_{j,j\sim i} \theta y_j, \tau^2\Big), \quad \tau > 0, \qquad i = 1, \ldots, n, \tag{5.5}$$

where $j \sim i$ indicates a first-order neighbor.

The neighbor structure can be encoded in a matrix, often denoted with $\mathbf{A} = (a_{ij})$ (adjacency matrix) or $\mathbf{W}$ (spatial weight matrix). In the former case, we have a $a_{ij} = 1$ if $i \sim j$ and zero otherwise. In the latter case, we may have $\mathbf{W} = \mathbf{A}$, or w_{ij} is proportional to the number of its neighbors or similar. To link with the last chapter and literature elsewhere, we often write $\mathbf{B} = \lambda\mathbf{W}$, for some λ.

Example 5.3. R-Code 5.4 illustrates the construction of the weight matrix $\mathbf{W}$ using the five counties from the US state Rhode Island. The upper bound of λ is numerically determined. The lower bound can be determined similarly. Note that there might be several zeros, and finding a lower bound for the interval with *uniroot()* needs some care.

Constant conditional precision does not imply constant (marginal) variances, as illustrated, i.e., we have a non-stationary model. ♣

R-Code 5.4: Construction of the weight matrices, valid parameter space, and resulting covariance matrix.

```
ri <- map("county","Rhode Island", fill=TRUE, plot=FALSE)
str(ri, strict.width="cut")
## List of 4
##  $ x    : num [1:144] -71.3 -71.3 -71.2 -71.2 -71.2 ...
##  $ y    : num [1:144] 41.8 41.8 41.8 41.7 41.7 ...
##  $ range: num [1:4] -71.9 -71.1 41.3 42
##  $ names: chr [1:5] "rhode island,bristol" "rhode island,kent" "rhode i"..
##  - attr(*, "class")= chr "map"
id <- sapply(strsplit(ri$names, ","), function(x) x[2])
ri.poly <- st_as_sf(ri, IDs=id)            # Convert to sf-class
ri.nb <- poly2nb(ri.poly)                  # Convert sf to nb object

(ri.matB <- nb2mat(ri.nb, style="B"))      # only 0-1 entries
```

```
##   [,1] [,2] [,3] [,4] [,5]
## 1    0    1    1    1    0
## 2    1    0    1    1    1
## 3    1    1    0    0    1
## 4    1    1    0    0    0
## 5    0    1    1    0    0
## attr(,"call")
## nb2mat(neighbours = ri.nb, style = "B")
(ri.matW <- nb2mat(ri.nb, style="W"))      # row sums to one
##      [,1]    [,2]    [,3]    [,4]    [,5]
## 1 0.00000 0.33333 0.33333 0.33333 0.00000
## 2 0.25000 0.00000 0.25000 0.25000 0.25000
## 3 0.33333 0.33333 0.00000 0.00000 0.33333
## 4 0.50000 0.50000 0.00000 0.00000 0.00000
## 5 0.00000 0.50000 0.50000 0.00000 0.00000
## attr(,"call")
## nb2mat(neighbours = ri.nb, style = "W")
### NOT symmetric (there are also styles "C" or "U")
### Valid parameter range for largest lambda:
f <- function(x, mat) det(diag(5)-x*mat)   # needs to be positive
c(uniroot(f, c(0, 1), mat=ri.matB)$root,   # gets upper bound for both cases
  uniroot(f, c(0, 1.5), mat=ri.matW)$root)
## [1] 0.34066 1.00000
ll <- seq(-2, to=1.5, l=500)               # plot is not shown in the script
plot(ll, sapply(ll, f, mat=ri.matB), type="l")
abline(h=0)
lines(ll, sapply(ll, f, mat=ri.matW), col=4)
### Example of resulting SAR-type covariance matrix. We assume iid structure
###   for the errors (see Table 4.1). We fix lambda at 1/2 of possible range.
### We display only diagonal terms:
diag(solve(diag(5) - 0.17 * ri.matB) %*% solve(diag(5) - 0.17 * ri.matB))
## [1] 1.5108 1.7056 1.5108 1.3099 1.3099
diag(solve(diag(5) - 0.5 * ri.matW) %*% solve(diag(5) - 0.5 * ri.matW))
## [1] 1.5121 1.6454 1.5121 1.3769 1.3769
```

We refer to Example 5.8 and corresponding R-Code 5.11 for the discussion of a more complex parameterization, where the coefficients b_{ij} depend on the degree of neighbor, $b_{ij} = b_1$ for adjacent cells (first order neighbors) and $b_{ij} = b_2$ for adjacent cells of adjacent cells (second-order neighbors).

The vignette cran.r-project.org/web/packages/spdep/vignettes/nb.pdf gives further insights in building neighborhood structures (*vignette("nb", package="spdep")*) .

Example 5.4. R-Code 5.5 fits a simple CAR model using the function *spautolm()*. For illustration, we use a small setting based on artificial data. The function's output is similar to a classical *lm()* output. We revisit the output of the function in Example 5.7. ♣

R-Code 5.5: Using the *spautolm* function to fit a simple model.

```
library("spatialreg")
y <- c(-0.58, -1.22, 1.68, 0.98, 0.44)     # artificial data for RI!
ri.B <- nb2listw(ri.nb, style="B")         # neighbors to weight matrix
carfit <- spautolm(y ~ 1, listw=ri.B, family="CAR") # binary weight matix
summary(carfit, adj.se=FALSE)
##
## Call: spautolm(formula = y ~ 1, listw = ri.B, family = "CAR")
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.41115 -0.25837 -0.16130  0.37442  0.45640
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.15434    0.11912  1.2956   0.1951
##
## Lambda: -0.58653 LR test value: 4.8371 p-value: 0.027854
## Numerical Hessian standard error of lambda: 0.050678
##
## Log likelihood: -4.8976
## ML residual variance (sigma squared): 0.18747, (sigma: 0.43298)
## Number of observations: 5
## Number of parameters estimated: 3
## AIC: 15.795
rbind(yhat=fitted(carfit), resid=resid(carfit))
##             1        2      3        4        5
## yhat  -0.4187 -0.96163 1.2236  1.39115 0.065581
## resid -0.1613 -0.25837 0.4564 -0.41115 0.374419
```

Remark 5.1. The function *spautolm()* with argument *family="SAR"* from package *spatialreg* and the function *errorsarlm()* from the same package are essentially identical and thus deliver the same results. ♡

Example 5.5. As a more realistic example, we investigate the spatial dependency for the SIDS dataset. R-Code 5.6 fits a CAR model (5.5) using (i) the neighbor structure used in the literature and (ii) a first-order neighbor structure. As seen in Figure 4.2, the data contains a possible outlier. The code illustrates the influence on the estimates and fit of this single value. For further details, see cran.r-project.org/web/packages/spdep/vignettes/sids.pdf.

Depending on the data source, a slightly different neighborhood structure of the dataset is used, resulting in minor differences. ♣

R-Code 5.6: SIDS again. See text for further explanations. (See Figure 5.3.)

```
ncW <- nb2listw(poly2nb(nc), style="B")
# summary(ncW)
library(spatialreg)    # `spautolm()` was formerly part of package spdep
carfit1 <- spautolm(rates ~ 1, data=nc, listw=ncW, family="CAR")
# summary(carfit1)     #  we should look at
index <- which.max(nc$rates)  # "remove" outlier:
as.character(nc$NAME[index])
## [1] "Anson"
nc$ratesNO <- nc$rates
nc$ratesNO[index] <- nc$rates[index]/10
carfit2 <- spautolm(ratesNO ~ 1, data=nc, listw=ncW, family="CAR")
# summary(carfit2)
nc$fit1 <- fitted(carfit1)    # with outlier
nc$fit2 <- fitted(carfit2)    # without outlier
nc$resid1 <- resid(carfit1)
nc$resid2 <- resid(carfit2)
nc$diff2to1 <- fitted(carfit2) - fitted(carfit1)
rbind(M1=c(coef(carfit1), s2=carfit1$fit$s2, resid=range(nc$resid1)),
      M2=c(coef(carfit2), s2=carfit2$fit$s2,   range(nc$resid2)))
##    (Intercept)  lambda         s2     resid1    resid2
## M1   0.0020020 0.13062 2.1205e-06 -0.0021387 0.0076864
## M2   0.0018462 0.15057 1.4774e-06 -0.0022982 0.0033259
both <- c(nc$fit1, nc$fit2)
fitbreaks <- seq(min(both), max(both), by = diff(range(both))/10)
plot(nc[ c("fit1", "fit2")], reset=FALSE, breaks=fitbreaks)
both <- c(nc$resid1, nc$resid2)
residbreaks <- seq(min(both), max(both), by = diff(range(both))/10)
plot(nc[ c("resid1","resid2")], breaks=residbreaks)
plot(nc[ c("diff2to1")], reset=FALSE)
```

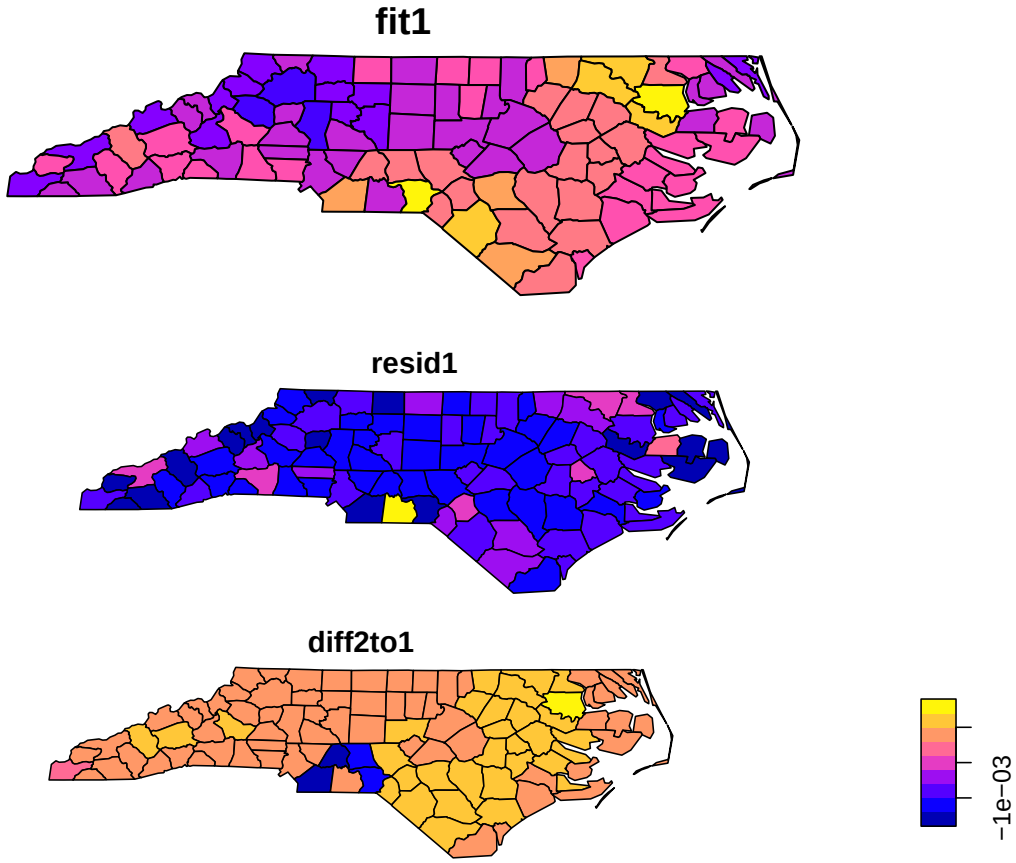


Figure 5.2: Model fits (top row) residuals (middle row), and differences (bottom) in the fits for SIDS rates. For the first fit the original data is used, for the second fit the value of Anson county is divided by 10. (See R-Code 5.6.)

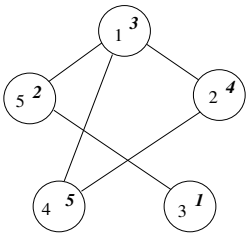
5.4 Exploiting the Sparsity Structure

In the case of GMRF, the off-diagonal non-zero elements of the precision matrix $\mathbf{Q}$ are associated with the conditional dependence structure. As by the Markovian property, the number of neighbors is small, the precision matrix $\mathbf{Q}$ is *sparse*, i.e., contains only $\mathcal{O}(n)$ non-zero elements compared to $\mathcal{O}(n^2)$ for a regular, *full* matrix. To take advantage of the few non-zero elements, special structures to represent the matrix are required, i.e., only the positions of the non-zeros and their values are kept in memory. Because of these special structures, tailored algorithms are required to fully exploit the sparsity structure. The package `spam` provides this functionality; see Furrer and Sain (2009) for a detailed exposition.

The sparsity structure is very important for the calculation based on the precision matrix $\mathbf{Q}$. It determines the (conditional) dependency structure and drives the computational cost. Hence, the sparsity structure is often represented using a graph with edges representing the non-zero elements or a "pixel" image of the zero/non-zero structure. Figure 5.4 gives such an illustration for an "arbitrary" 5×5 matrix $\mathbf{A}$.

It is important to note that the labeling (order of the variables) influences the structures that result from relevant computations. Reordering is performed through a so-called permutation. In

$$A = \begin{pmatrix} 1 & 0.5 & 0 & 0.5 & 0.5 \\ 0.5 & 1 & 0 & 0.5 & 0 \\ 0 & 0 & 1 & 0 & 0.5 \\ 0.5 & 0.5 & 0 & 1 & 0 \\ 0.5 & 0 & 0.5 & 0 & 1 \end{pmatrix}$$



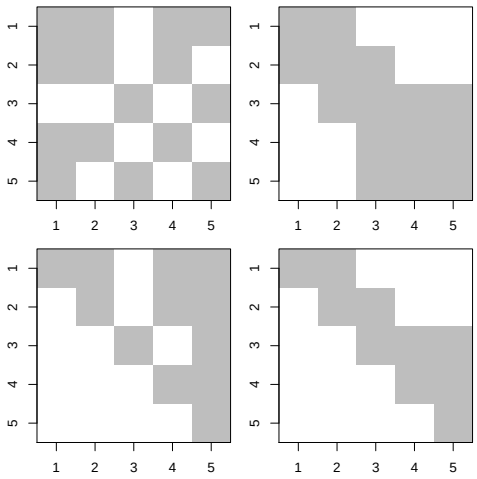


Figure 5.3: The symmetric positive-definite $n = 5$ matrix $\mathbf{A}$ and the sparsity structure of $\mathbf{A}$ and $\mathbf{P}^\top\mathbf{AP}$ (top row). The graph associated with the matrix $\mathbf{A}$ and the Cholesky factors $\mathbf{R}$ and $\mathbf{U}$ of $\mathbf{A}$ and $\mathbf{P}^\top\mathbf{AP}$ respectively are given in the bottom row. The nodes of the graph are labeled according to $\mathbf{A}$ (upright) and $\mathbf{P}^\top\mathbf{AP}$ (italics).

matrix notation, a permutation is a matrix having exactly one element 1 per row and column. All other elements are zero.

5.4.1 The *spam* Package

There are several R packages available to handle sparse matrices: *Matrix*, *SparseM*, *spam*. We use the last one, which provides an extensive set of functions for sparse matrix algebra. Major differences with *Matrix* are: (1) *spam* only supports (essentially) one sparse matrix format, (2) it is based on transparent and simple structure(s), (3) it is tailored for MCMC calculations within GMRF and (4) S3 and S4 like-"compatible" ... and it is fast. R-Code 5.8 gives a quick overview and shows that the handling of sparse matrices is straightforward.

R-Code 5.7: The shortest possible illustration of the package *spam*. The second part contains code for Figure 5.4.

```
library(spam)
mat <- spam(sample(c(0,1), size=18, replace=T, prob=c(.8,.2)), 2, 9)
mat
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    0    0    0    0    0    0    0    0    0
## [2,]    1    0    0    1    0    0    0    0    1
## Class 'spam' (32-bit)
class(mat)
## [1] "spam"
## attr(,"package")
## [1] "spam"
```

```
str(mat)
## Formal class 'spam' [package "spam"] with 4 slots
##   ..@ entries    : num [1:3] 1 1 1
##   ..@ colindices : int [1:3] 1 4 9
##   ..@ rowpointers: int [1:3] 1 1 4
##   ..@ dimension  : int [1:2] 2 9
diag(mat) <- 4
solve(mat %*% t(mat), c(1:2))
## [1] 0.038194 0.097222
### Lets construct a second sparse matrix `A`
A <- 0.5 * diag.spam(5)
i <- c(2, 4, 4, 5, 5)
j <- c(1, 1, 2, 1, 3)
A[cbind(i, j)] <- rep(.5, length(i))
A <- t(A) + A       # this is the matrix as in Figure 5.1
summary(A)
## Matrix object of class 'spam' of dimension 5x5,
##     with 15 (row-wise) nonzero elements.
##     Density of the matrix is 60%.
## Class 'spam' (32-bit)
U <- chol(A)
class(U)            # Special class, you do not really need to work with.
## [1] "spam.chol.NgPeyton"
## attr(,"package")
## [1] "spam"
(pivot <- U@pivot)  # The permutation is found automatically!
## [1] 3 5 1 2 4
U@invpivot          # Inverse of the permutation,  see also `?permutation`
## [1] 3 4 1 5 2
P <- diag.spam(5)[U@invpivot,]
norm(A[ pivot, pivot] - t(P) %*% A %*% P ) # sum of squared differences.
## [1] 0
spam::display(A)
spam::display(U)
```

5.4.2 Never Calculate the Actual Inverse of a SPD Matrix

Covariance matrices (and thus precision matrices) are symmetric and positive definite matrices. When calculating or maximizing multivariate normal log-likelihoods, we need to calculate

determinants ($\det(\boldsymbol{\Sigma})$) and quadratic forms ($\boldsymbol{y}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{y}$).

To be more specific, assume we need to calculate $\mathbf{A}^{-1}\boldsymbol{b}$ with $\mathbf{A}$ a symmetric positive-definite matrix featuring some sparsity structure, which is usually accomplished by solving $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. We proceed by factorizing $\mathbf{A}$ into $\mathbf{R}^\top\mathbf{R}$, where $\mathbf{R}$ is an upper triangular matrix, called the Cholesky factor or Cholesky triangle of $\mathbf{A}$, followed by solving $\mathbf{R}^\top \boldsymbol{y} = \boldsymbol{b}$ and $\mathbf{R}\boldsymbol{x} = \boldsymbol{y}$, called forwardsolve and backsolve, respectively. Note that the exposition could be done with the lower triangular matrix $\mathbf{L} = \mathbf{R}^\top$.

From a computational point of view, there is a huge difference between `solve(A)%*% b` and `solve(A, b)`!

Calculating the determinant can be done through various paths. For symmetric positive definite matrices, the best approach is to perform a Cholesky factorization and use the property

$$\det(\mathbf{A}) = \det(\mathbf{R}^\top\mathbf{R}) = \det(\mathbf{R})^2 = \prod_i r_{ii}^2, \tag{5.6}$$

where r_{ii} are the diagonal entries of $\mathbf{R}$.

Notice that the Cholesky factor can be seen as a *matrix square root* and is thus used when drawing multivariate standard random variables as well; see Section 1.2.2.

5.4.3 Solving Linear Systems

The Cholesky factor of a banded matrix is again a banded matrix. However, arbitrary sparse matrices may produce full Cholesky factors. To reduce this so-called *fill-in* of the Cholesky factor $\mathbf{R}$, we permute the columns and rows of $\mathbf{A}$ according to a (cleverly chosen) permutation $\mathbf{P}$, i.e., $\mathbf{U}^\top\mathbf{U} = \mathbf{P}^\top\mathbf{A}\mathbf{P}$, with $\mathbf{U}$ an upper triangular matrix. Many different algorithms exist to find permutations that are optimal for specific matrices or at least close to optimal with respect to different criteria. The cost of finding a good permutation matrix $\mathbf{P}$ is at least of order $\mathcal{O}(n^{3/2})$ (for lattices in two dimensions).

Note that $\mathbf{R}$ and $\mathbf{U}$ cannot be linked through $\mathbf{P}$ alone. Figure 5.4 illustrates the factorization with and without permutation. Two triangular solves are performed after the factorization for solving a linear system. The determinant of $\mathbf{A}$ is the squared product of the diagonal elements of its Cholesky factor $\mathbf{R}$. Hence the same factorization can be used to calculate determinants (a necessary and computational bottleneck in the computation of the log-likelihood of a Gaussian model), illustrating that it is crucial to have a very efficient integration (with respect to calculation time and storage capacity) of the Cholesky factorization.

A typical Cholesky factorization of a sparse matrix consists of the steps illustrated in the following pseudo-code algorithm.

---

[1] Determine permutation and permute the input matrix $\mathbf{A}$ to obtain $\mathbf{P}^\top\mathbf{A}\mathbf{P}$

[2] Symbolic factorization, where the sparsity structure of $\mathbf{U}$ is constructed

[3] Numeric factorization, where the elements of $\mathbf{U}$ are computed

---

When factorizing matrices with the same sparsity structure, Steps 1 and 2 do not need to be repeated. In MCMC algorithms, this is commonly the case, and exploiting this shortcut leads to very considerable gains in computational efficiency (we revisit this in the coming chapters).

As for Step 1, there are many different algorithms to find a permutation, for example, the multiple minimum degree (MMD) algorithm, (Liu, 1985), and the reverse Cuthill-McKee (RCM) algorithm, (George, 1971). The resulting sparsity structure in the permuted matrix determines the sparsity structure of the Cholesky factor. As an illustration, R-Code 5.9 and Figure 5.5 illustrate the sparsity structure of the Cholesky factor resulting from an MMD, an RCM, and no permutation of a precision matrix induced by a second-order neighbor structure of the US counties.

R-Code 5.8: Illustrating the sparsity structure of the Cholesky factor using different permutation schemes implemented in *spam*. (See Figure 5.5.)

```
In <- diag.spam(nrow(UScounties.storder))
Q <- In + .1 * UScounties.storder + .1 * UScounties.ndorder
summary(Q)
## Matrix object of class 'spam' of dimension 3082x3082,
##     with 59978 (row-wise) nonzero elements.
##     Density of the matrix is 0.631%.
## Class 'spam' (32-bit)
struct <- chol(Q)
spam::display(Q, ylab="", xlab="", cex=1)  # Without cex, a warning is issued
spam::display (struct, ylab="", xlab="", cex=1)
summary(struct)
## (Upper) Cholesky factor of class 'spam.chol.NgPeyton' of dimension 3082x
## 3082 with 146735 (row-wise) nonzero elements.
##     Density of the factor is 1.54%.
##     Fill-in ratio is 4.65
##     (Optimal argument for 'chol' is 'memory=list(nnzR=146735)'.)
## Class 'spam.chol.NgPeyton'
(nnzMMD <- sum(struct@entries > .Machine$double.eps ))
## [1] 81345
struct <- chol(Q, pivot="RCM")
spam::display(struct, ylab="", xlab="", cex=1)
summary(struct)
## (Upper) Cholesky factor of class 'spam.chol.NgPeyton' of dimension 3082x
## 3082 with 256198 (row-wise) nonzero elements.
##     Density of the factor is 2.7%.
##     Fill-in ratio is 8.13
##     (Optimal argument for 'chol' is 'memory=list(nnzR=256198)'.)
## Class 'spam.chol.NgPeyton'
(nnzRCM <- sum(struct@entries > .Machine$double.eps ))
## [1] 135457
```

```
struct <- chol(Q, pivot=FALSE)
spam::display (struct, ylab="", xlab="", cex=1)
summary(struct)
## (Upper) Cholesky factor of class 'spam.chol.NgPeyton' of dimension 3082x
## 3082 with 689615 (row-wise) nonzero elements.
##     Density of the factor is 7.26%.
##     Fill-in ratio is 21.9
##     (Optimal argument for 'chol' is 'memory=list(nnzR=689615)'.)
## Class 'spam.chol.NgPeyton'
(nnzNONE <- sum(struct@entries > .Machine$double.eps ))
## [1] 270140
```

How much fill-in with zeros is present depends on the permutation algorithm. In the example of Figure 5.5 there are 146 735, 256 198 and 689 615 non-zero elements in the Cholesky factors with MMD, RCM, and no permutation, respectively. Note that the actual number of non-zero elements of the Cholesky factor may be smaller than what the constructed sparsity structure indicates, Here, there are 81345, 135457 and 270140 zero elements (up to machine precision) that are not exploited.

We finish this section with examples illustrating further the functionality of *spam* in the context of GMRFs.

Example 5.6. We manually implemented the *spautolm* functionality. Once a model has been specified for specific data (here function *mle.CAR()*), parameter estimation can be carried out, here with the *optim* function. R Code 5.10 shows how to fit a CAR model based on (5.5) (and data as in Example 5.4).

The function *mle.CAR()* is very similar to the function *spam::mle.nomean()*, both represent a rudimentary approach to estimate parameters in a multivariate normal setting. ♣

R-Code 5.9: Defining a likelihood function of a CAR model, "manually" optimizing it and comparing it with the output of the *spautolm* function.

```
options(spam.cholupdatesingular="null")
mle.CAR <- function (y, W, theta)  {
  n <- length(y)
  In <- diag.spam(n)
```

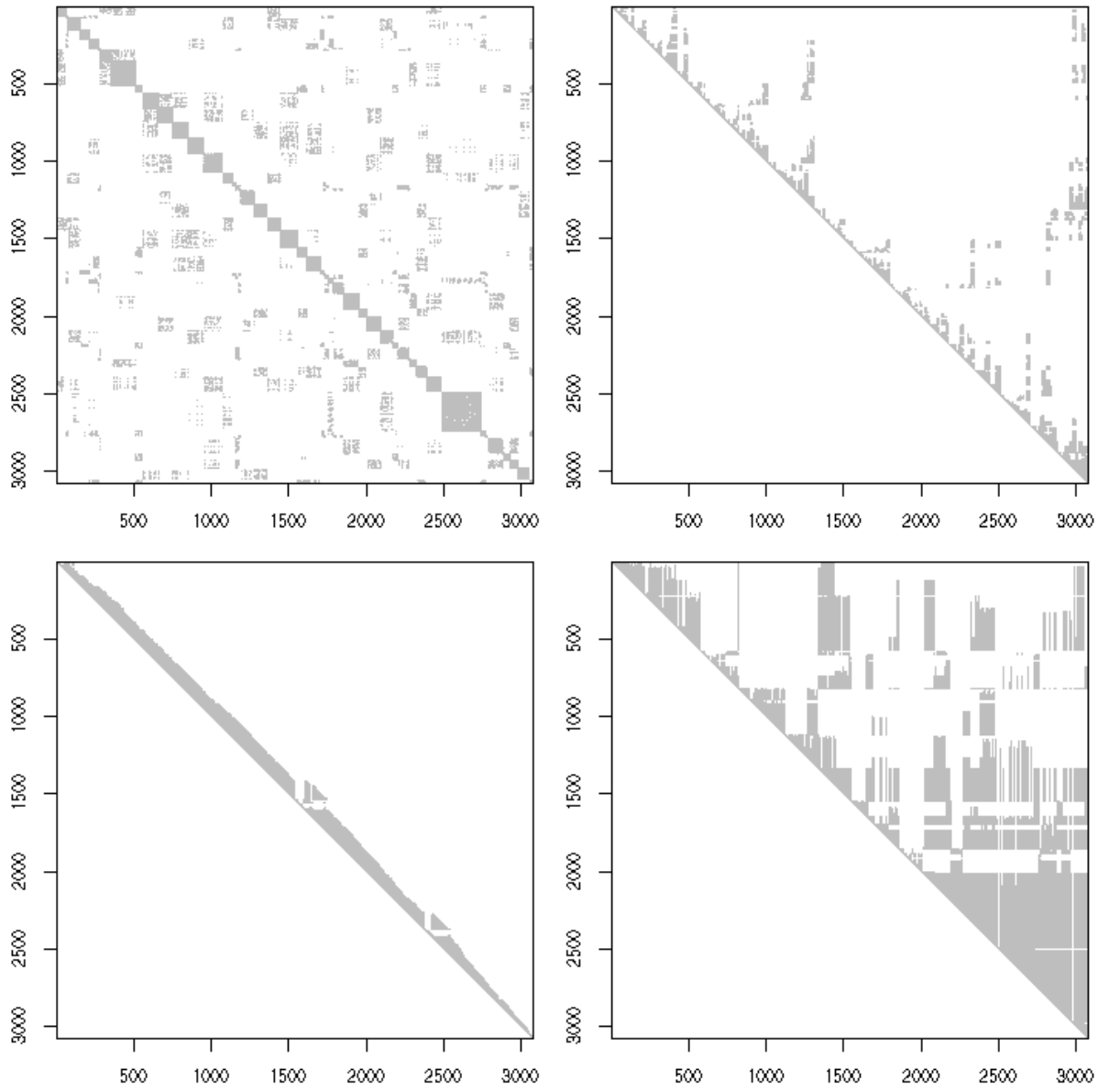


Figure 5.4: Top left represents the sparsity structure of a precision matrix induced by a second-order neighbor structure of the US counties. Sparsity structure of the Cholesky factor with MMD (top right), RCM (bottom left), and no permutation of the precision matrix. (See R-Code 5.9.)

```
Qstruct <-  chol(In - 0.0001 * W)
neg2loglikelihood <- function(theta) {
  Q <-  (In - theta[2] * W)
  cholQ <- update(Qstruct, Q)
  if (is.null(cholQ)) return(1e6)
  resid <- y - theta[1]
  return(n * log(2*pi * theta[3]) - 2*c(determinant(cholQ)$modulus) +
```

```
        sum(resid * (Q %*% resid))/theta[3] )
  }
  out <- optim(theta, neg2loglikelihood, method="L-BFGS-B",
               lower=c(-Inf, -Inf, 1e-5), hessian=TRUE)
  if (out$convergence !=0) cat("Convergence issues, please inspect\n")
  return(out)
}

listw <- nb2listw(ri.nb, style="B")
W <- as.spam.listw(listw)
print(tm1 <- mle.CAR(y , W, c(0,-.1,1)))
## $par
## [1]  0.15433 -0.58652  0.18748
##
## $value
## [1] 9.7953
##
## $counts
## function gradient
##       50       50
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
##
## $hessian
##             [,1]     [,2]        [,3]
## [1,] 140.9364476   -9.608  4.8488e-03
## [2,]  -9.6080461 1127.902 -2.2225e+02
## [3,]   0.0048488 -222.249  1.4229e+02
rbind(spamautolm=c(coef(carfit), sigma2=carfit$fit$s2, logLik=carfit$LL),
    manual=c(tm1[[1]],tm1[[2]]/-2))
##            (Intercept)   lambda  sigma2  logLik
## spamautolm     0.15434 -0.58653 0.18747 -4.8976
## manual         0.15433 -0.58652 0.18748 -4.8976
### Differences in the uncertainty estimates, should not be compared 1-2-1
rbind(spamautolm=c(se.Inter=summary(carfit)$Coef[,2], se.lambda=
    carfit$lambda.se), manual=c(sqrt(solve(tm1$hessian)[c(1,5)])))
##            se.Inter se.lambda
```

```
## spamautolm 0.119121  0.050678
## manual     0.084269  0.035803
```

Example 5.7. Illustration of numerically determine the valid parameter space. Consider a CAR model with a first- and a second-order neighbor structure. More specifically, we consider

$$Y_i \mid \boldsymbol{y}_{-i} \sim \mathcal{N}\Big(\sum_{j,j\sim i} \theta_1 y_j + \sum_{j,j\approx i} \theta_2 y_j, \tau^2\Big), \quad \tau > 0, \qquad i = 1, \dots, n, \tag{5.7}$$

where $j \sim i$ and $j \approx i$ indicate first- and second-order neighbors, respectively. That means, the resulting precision matrix $\mathbf{Q}$ has the structure $\mathbf{Q} = \tau^{-2}(\mathbf{I} - \theta_1\mathbf{W}_1 - \theta_2\mathbf{W}_2)$ where $\mathbf{W}_i$ are (binary) spatial weight matrices. We have to impose constraints on (θ_1, θ_2), such that $\mathbf{Q}$ is positive definite, i.e., we need to determine the valid parameter space $\boldsymbol{\Theta} \subset \mathbb{R}^2$. Here, we do not have a closed form description of the parameter space (recall Example 5.3).

We first construct an (arbitrary but valid) precision matrix based on the first- and second-order structure to exploit the `spam` options. To ensure validity, we choose very small values of (θ_1, θ_2), as we know that $(0, 0) \in \boldsymbol{\Theta}$. Then, we cycle over a specified fine grid of `theta1` and `theta2` and verify if the precision matrix is positive definite. If the matrix passed to `update` is not symmetric positive definite, which means that the value of `tmp` is `NULL`, the pair (`theta1[i]`,`theta2[2]`) lies not within $\boldsymbol{\Theta}$. Figure 5.6 shows $\boldsymbol{\Theta}$. The valid range is color-coded according to the value of $\log(\det \mathbf{Q})$. Notice the domain's asymmetry and the determinant's very small values. The maximum (being zero) is at the origin, see also Remark 5.2.

Based on the fine grid used, the loop takes several minutes. Naturally, the convexity of the domain could easily be exploited. Of course, in practice, the precise space $\boldsymbol{\Theta}$ is not necessary for optimization in a maximum likelihood estimation context. Similarly, in Bayesian settings, it would be possible to place a prior over $\boldsymbol{\Theta}$ without knowing it explicitly. ♣

Remark 5.2. In the case of independence, the precision/covariance matrix is the identity. For simplicity, assume $\sigma^2 = 1$, then the determinant is equal to one, as all eigenvalues are 1. Adding spatial structure will decrease the determinant, as some of the eigenvalues will be larger than one, others smaller. However, since the sum of the eigenvalues remains constant (here n), the product of these decreases. ♡

R-Code 5.10 Determining the valid parameter space $\boldsymbol{\Theta}$ through a numerical assessment of the precision matrix. (See Figure 5.6.)

```
In <- diag.spam(nrow(UScounties.storder))
struct <- chol(In - .1 * UScounties.storder - .01 * UScounties.ndorder)
###    which is a valid, but (arbitrary) precision matrix.
options(spam.cholupdatesingular="null")       # We want to avoid errors.
len1 <- 300      # even
len2 <- 150
theta1 <- seq(-.515, .225, len=len1)
theta2 <- seq(-.19, .095, len=len2)
grid <- array(NA, c(len1, len2))
for (i in 1:len1) {
  for(j in 1:len2) {
    tmp <- update(struct, In - theta1[i]*UScounties.storder
                              - theta2[j]* UScounties.ndorder)
    if(!is.null(tmp))   grid[i,j] <- determinant(tmp)$modulus
  }
}
image.plot(theta1, theta2, grid, xlab=expression(theta[1]),
           ylab=expression(theta[2]), xlim=c(-.45, .22), ylim=c(-.2, .1))
abline(v=0, h=0, lty=2)
```

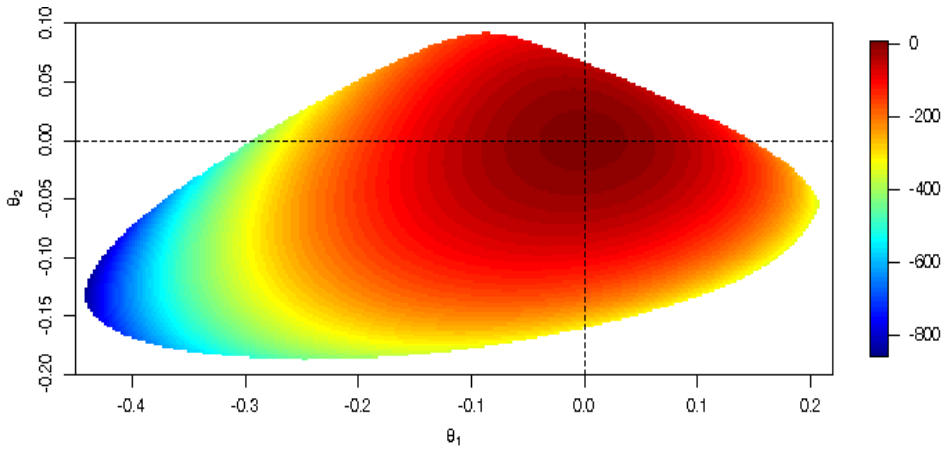


Figure 5.5: The domain $\boldsymbol{\Theta}$ for the US counties with a second-order neighbor structure. The values represent $\log(\det \mathbf{Q})$ and the white area represents $\boldsymbol{\Theta}^c$. (See R-Code 5.11.)

5.5 Further Example

We consider here an additional example based on the oral cavity cancer data. We fit a simple CAR model with a binary weight matrix to the standardized mortality rates. The weight matrix is constructed in two different ways for illustrative purposes: once with the build-in function *adjacency.landkreis()* from the package *spam* and second manually based on a file defining the neighbor list. The same package also provides the latter. The range of possible values of λ is approximately from -0.296 to 0.159. The estimated value $\widehat{\lambda} = 0.152$ is quite close to the boundary of the valid range. This often indicates that the proposed CAR model is not sufficiently flexible. This might also be seen as the drawn realizations seem more speckled than the observed data (see Figure 5.7). For a better comparison, we use the same seed, such that the first $n = 544$ random numbers from *rnorm()* are equivalent (for the GMRF realization, they are further transformed).

Notice that only the marginal conditional precision is constant, not the marginal variances or marginal standard deviation.

R-Code 5.11: Oral cavity cancer example. (See Figure 5.7.)

```
library(spam)
data(Oral, package="spam")
hist(Oral$SMR, main="", xlab="SMR")
abline(v=mean(Oral$SMR), col=4, lwd=2)
filename <- system.file("demodata/germany.adjacency", package="spam")
# system(paste("head ", filename), intern=F)  # show content of the file
W <- adjacency.landkreis(filename)
barplot(table(diff(W@rowpointers)), xlab="# of neighbors")
### The following uses as input a classical ASCII format of nb files.
n <- as.numeric(readLines(filename, n=1))
nnodes <- nodes <- numeric(n)
adj <- list()
for (i in 1:n) {
  tmp <- as.numeric(scan(filename, skip=i, nlines=1, quiet=T,
                         what=list(rep("", 13)))[[1]])
  nodes[i] <- tmp[1]
  nnodes[i] <- tmp[2]
  adj[[i]] <- as.integer(tmp[-c(1:2)]+1)
}
adj <- adj[order(nodes)]
attr(adj, "region.id") <- germany.info$id
attr(adj, "sym") <- TRUE
class(adj) <- "nb"
Dlistw <- nb2listw(adj, style="B")
W1 <- as.spam.listw(Dlistw)
all.equal.spam(W, W1)
```

```
## [1] TRUE
# summary(Dlistw)
# table(unlist(lapply(adj, length)))
### We can now start spatial modeling. For example a simple CAR:
carfit <- spautolm(SMR ~ 1, data=Oral, listw=Dlistw, family="CAR")
###   strictly speaking, we should include an offset of 1 here:
#  lm(SMR ~ offset(rep(1,n))+ 1, data=Oral)
### aparently spautolm() does not incorporate this.
options(spam.cholupdatesingular="null")
tm1 <- mle.CAR(Oral$SMR, W, c(0,.1,2))
### again, surprising similar results:
coefs <- c(coef(carfit), sigma2=carfit$fit$s2)
rbind(spamautolm= c(coefs, logLik=carfit$LL),
    manual=c(tm1[[1]], tm1[[2]]/-2))
##             (Intercept)  lambda   sigma2  logLik
## spamautolm        1.001 0.15219 0.088880 -141.58
## manual            1.001 0.15216 0.088894 -141.58
### Valid parameter range for lambda:
lambda <- c(seq(-.2959, to=-.2958, by=0.00005),
            seq(.159, to=.1591, by=0.00005))
for (i in 1:length(lambda))
  if (class(try(chol(diag.spam(n)-lambda[i]*W),silent=TRUE))!=
                        "spam.chol.NgPeyton") lambda[i] <- NA
lambda   # estimates are quite on the boundaries!!!
## [1]       NA -0.29585 -0.29580  0.15900  0.15905       NA

zl <- c(-0.6, 2.4)
germany.plot(Oral$SMR, main="SMR",border=NA, zlim=zl)
set.seed(14)
germany.plot(rnorm(n, mean=coefs[1], sd=sqrt(coefs[3])),
             main="White noise", border=NA, zlim=zl)
N <- 1000     # We simulate many realizations with similar parameters
Q <- (diag.spam(n)- coefs[2]*W)/coefs[3]
set.seed(14)
ex <- rmvnorm.prec(N, mu=coefs[1], Q=Q)
lambdahat <- numeric(4)  # We only look at four
for (i in 1:4) {
  germany.plot(ex[i,], main=paste("Sample",i),border=NA, zlim=zl)
#   mle.CAR(ex[i,], W, c(1,.12,09))  # leads often to convergence issues!!
  lambdahat[i] <- spautolm(ex[i,] ~ 1, listw=Dlistw, family="CAR")$lambda
}
```

```
lambdahat
## [1] 0.15437 0.15669 0.14492 0.15754
germany.plot(colMeans(ex), main="Means", border=NA)
germany.plot(apply(ex, 2, sd), main="SD", border=NA)
germany.plot(diag(solve(cov(ex))), main="Precision", border=NA)
```

We now extend our model to two parameters, much in the spirit of Equation (5.7). R-Code 5.13 illustrates such an approach by defining a likelihood function. The estimates are pretty close to zero. To evaluate if they are close to the boundary, we need a similar illustration as in R-Code 5.11. Overall, the model fit is definitely better, and we reduce the negative log-likelihood from 183.2 to 274.5 for one single additional parameter. Samples drawn from the estimated precision matrix are somewhat smoother than with one parameter only, as shown by Figure 5.8 for one specific example.

The estimation does not take a long time (less than a couple of seconds, although there are 363 likelihood evaluations (see *tm2$counts[1]*).

R-Code 5.12: Oral cavity cancer example. (See Figure 5.8.)

```
options(spam.cholupdatesingular="null")
mle.CAR2 <- function (y, W1, W2, theta)  {
  n <- length(y)
  In <- diag.spam(n)
  Qstruct <-  chol(In - 0.0001 * W1 - 0.0001 * W2)
  neg2loglikelihood <- function(theta) {
    Q <- (In - theta[2] * W1 - theta[3] * W2)
    cholQ <- update(Qstruct, Q)
    if (is.null(cholQ)) return(1e6)
    resid <- y - theta[1]
    return(n * log(2*pi * theta[4]) - 2*c(determinant(cholQ)$modulus) +
               sum(resid * (Q %*% resid))/theta[4] )
  }
  out <- optim(theta, neg2loglikelihood, method="L-BFGS-B",
                 lower=c(-Inf, -Inf, -Inf, 1e-5), hessian=TRUE)
  if (out$convergence !=0) cat("Convergence issues, please inspect\n")
  return(out)
}

listw2 <- nblag(adj,2)[[2]]   # constructs higher order neighbors
W2 <- as.spam.listw(nb2listw(listw2, style="B"))
tm2 <- mle.CAR2(Oral$SMR, W1, W2, c(1,-0.0,-0.0,.1)) # W1 from above
```

```
## Convergence issues, please inspect
print(unlist(tm2[c(2,1)]))    # Hessian and other stuff not relevant here
##      value       par1       par2       par3       par4
## 274.491868   0.994615   0.033386   0.056566   0.105442
print(unlist(tm1[c(2,1)]))    # quite a few function calls
##      value       par1       par2       par3
## 283.165444   1.001000   0.152156   0.088894
Q2 <- (diag.spam(n)- tm2$par[2]*W1 - tm2$par[3]*W2)/tm2$par[4]
set.seed(14)
ex2 <- rmvnorm.prec(1, mu=tm2$par[1], Q=Q2)

germany.plot(ex2[1,], main="Sample 1, 2 pars",border=NA, zlim=zl)
germany.plot(ex[1,], main="Sample 1", border=NA, zlim=zl)
germany.plot(ex2[1,]-ex[1,], main="Difference",border=NA)
```

5.6 *Details of Spatial Classes

All spatial objects (data, locations, etc.) are linked to a bounding box (defining the spatial domain, slot *bbox*) and a coordinate reference system (defining map projections and transformations to references projections, slot *proj4string*, itself of class *CRS*) (Bivand *et al.*, 2013, Section 4.1). For many years the package *sp* provided many different classes for spatial objects. The core class for the spatial objects is *Spatial* with many subclasses, e.g., *SpatialPoints* (extending with a *coords* slot), *SpatialLines* (extending with a *lines* slot), *SpatialPolygons*, etc.

R-Code 5.13: *Spatial* class of package *sp*.

```
library(sp)
# getClass("Spatial") # structure of the class and its subclasses
# vignette("intro_sp", package="sp")
getSlots("SpatialPolygons")
##    polygons   plotOrder        bbox proj4string
##      "list"   "integer"    "matrix"       "CRS"
```

For example, the manual construction of a *SpatialPolygons* is very tedious and hardly done in practice. Often the relevant objects are created by querying databases or other spatial objects such as extracting the information from the *maps* or *mapdata* packages, for example.

Moreover, the entire framework has been shifted more towards 'simple features' provided by the package *sf*. The mitigation is not entirely completed yet but it is generally recommended. See also Figure 5.9.

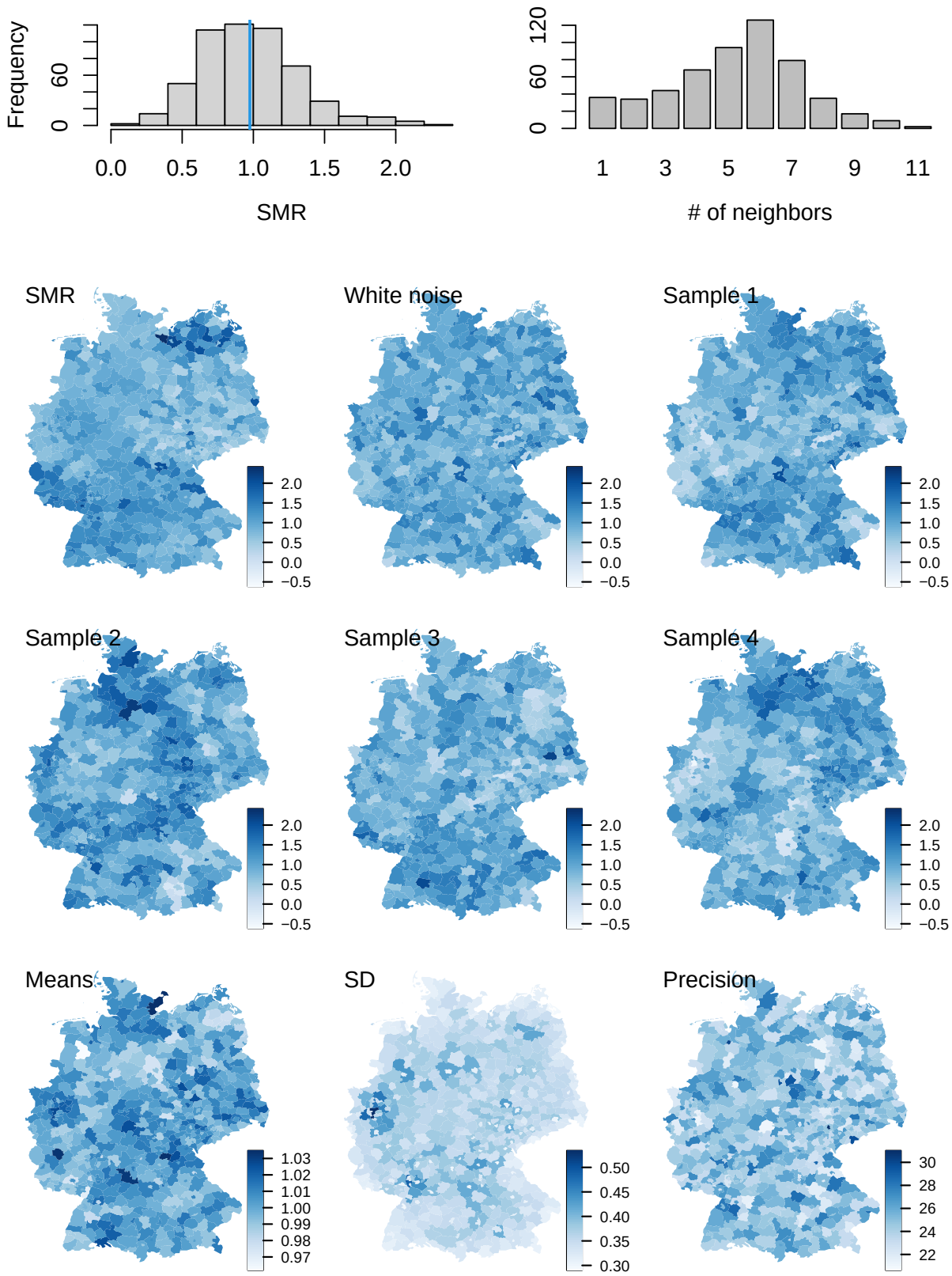


Figure 5.6: Top row: histogram of SMR of oral cavity cancer (left) and the number of neighbors of each of the districts (right). Bottom panels: SMR, white noise comparison, four samples having the same mean and precision matrix (unconditional simulation), means, marginal standard deviation, and precision of 1000 samples. (See R-Code 5.12.)

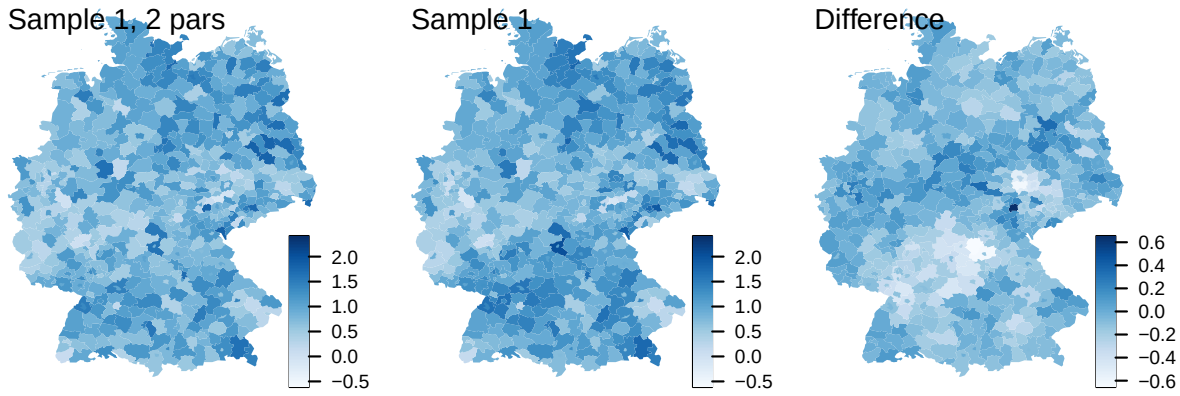


Figure 5.7: Comparison of a realization with a one-parameter and two-parameter model. Middle panel is similar as Figure 5.7. (See R-Code 5.13.)

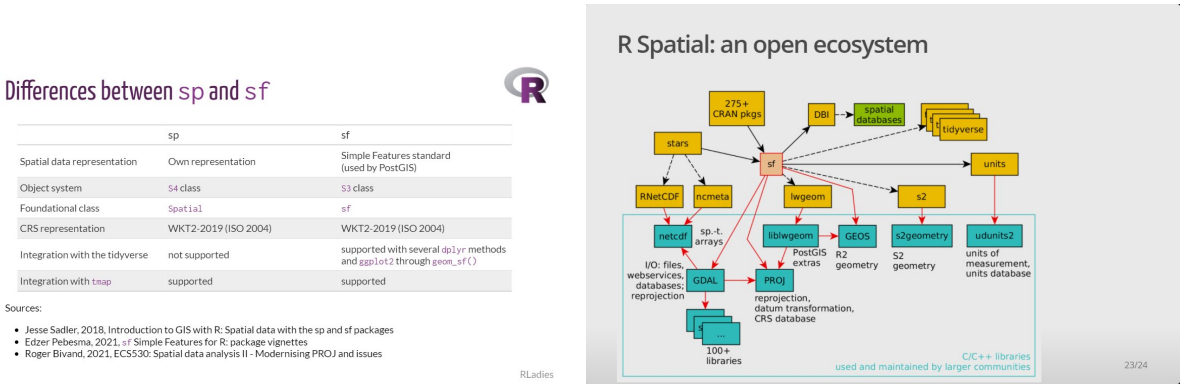


Figure 5.8: Similarities and differences between the packages *sp* and *sf* (source left) and the R spatial ecosystem (source right).