Journal of Statistical Software

September 2010, Volume 36, Issue 10. *http://www.jstatsoft.org/*

spam: A Sparse Matrix R Package with Emphasis on MCMC Methods for Gaussian Markov Random Fields

Reinhard Furrer
University of Zurich

Stephan R. Sain
National Center for Atmospheric Research

Abstract

spam is an R package for sparse matrix algebra with emphasis on a Cholesky factorization of sparse positive definite matrices. The implemantation of **spam** is based on the competing philosophical maxims to be competitively fast compared to existing tools and to be easy to use, modify and extend. The first is addressed by using fast Fortran routines and the second by assuring S3 and S4 compatibility. One of the features of **spam** is to exploit the algorithmic steps of the Cholesky factorization and hence to perform only a fraction of the workload when factorizing matrices with the same sparsity structure. Simulations show that exploiting this break-down of the factorization results in a speed-up of about a factor 5 and memory savings of about a factor 10 for large matrices and slightly smaller factors for huge matrices. The article is motivated with Markov chain Monte Carlo methods for Gaussian Markov random fields, but many other statistical applications are mentioned that profit from an efficient Cholesky factorization as well.

Keywords: Cholesky factorization, compactly supported covariance function, compressed sparse row format, symmetric positive-definite matrix, stochastic modeling, S3/S4.

1. Introduction

In many areas of scientific study, there is great interest in the analysis of datasets of ever increasing size and complexity. In the geosciences, for example, weather prediction and climate model experiments utilize datasets that are measured on the scale of terabytes. New statistical modeling and computational approaches are necessary to analyze such data, and approaches that can incorporate efficient storage and manipulation of both data and model constructs can greatly aid even the most simple of statistical computations. The focus of this work is on statistical models for spatial data that can utilize regression and correlation matrices that are *sparse*, i.e., matrices that have many zeros.

Sparse matrix algebra has seen a resurrection since much of the development in the late 1970s and 1980s. To exploit sparse structure, a matrix is not stored as a two-dimensional array. Rather it is stored using only its non-zero values and an index scheme linking those values to their location in the matrix (see Section 3). This storage scheme is memory efficient but implies that for all operations involving the scheme, such as matrix multiplication and addition, new functions need to be implemented. **spam** is a software package based on and inspired by existing and publicly available Fortran routines for handling sparse matrices and Cholesky factorization, and provides a large functionality for sparse matrix algebra.

More specifically, the **spam** package implements and overloads sparse matrix algebra methods that are based on Fortran routines. Typically, a user creates sparse precision matrices (ideally directly using provided routines or by transforming regular R matrices into sparse matrices) and proceeds as if handling regular matrices. Naturally, additional visible functionality for sparse matrices is implemented, e.g., visualizing the sparsity structure. Some important features of **spam** are: (1) it supports (essentially) a single sparse matrix format; (2) it is based on transparent and simple structure(s); (3) it is tailored for Markov chain Monte Carlo (MCMC) calculations within Gaussian Markov random fields (GMRFs); (4) it is **methods**-based while providing functions using S3 syntax. These aspects imply a very steep learning curve and make **spam** very user friendly. The functionality of **spam** can be extended and modified in a very straightforward manner. While this paper focuses on sparse precision matrices of GMRFs, sparse covariance matrices in a Gaussian random fields setting are another natural field of application.

1.1. Motivation

A class of spatial models in which a sparse matrix structure arises naturally involves data that is laid out on some sort of spatial lattice. These lattices may be regular, such as the grids associated with images, remote sensing data, climate models, etc., or irregular, such as US census divisions (counties, tracts, or block-groups) or other administrative units. A powerful modeling tool for this type of data is the framework of GMRFs, introduced by the pioneering work of Besag (1974), see Rue and Held (2005) for an excellent exposition of the theory and application of GMRFs. In short, a GMRF can be specified by a multivariate Gaussian distribution with mean $\boldsymbol{\mu}$ and a precision matrix $\mathbf{Q}$, where the (i, j)th element of $\mathbf{Q}$ is zero if the process at location i is conditionally independent of the process at j given the process at all locations except $\{i, j\}$. The pattern of zero and non-zero elements in such matrices is typically due to the assumption of some sort of Markov property in space and/or time and is called the sparsity structure. Whereas the total number of non-zero elements divided by the total number of elements is called the density of the matrix; $\mathbf{Q}$, for example, usually has a very low density. Commonly, the conditional dependence structure in a GMRF is modeled using a parameter $\boldsymbol{\theta}$ and MCMC methods can be used to probe the posterior distribution of $\boldsymbol{\theta}$ (and potentially other parameters of interest) as well as the predictive distribution. In each MCMC iteration the Cholesky factor of the precision matrix $\mathbf{Q}$ needs to be calculated and it is indispensable to exploit its sparsity to be able to analyze the large datasets arising from the applications mentioned above.

1.2. The spam R package

Although used here as motivation and illustration, obtaining posterior distributions of pa-

rameters in the context of a GMRF is not the only application where efficient Cholesky factorizations are needed. To mention just a few: drawing multivariate random variables, calculating log-likelihoods, maximizing log-likelihoods, calculating determinants of covariance matrices, linear discriminant analysis, etc. Statistical calculations, which require solving a linear system or calculating determinants, usually also require pre- and post-processing of the data, visualization, etc. A successful integration of an efficient factorization algorithm not only calls for subroutines for the factorization and back- and forwardsolvers, but also is user friendly and easy to work with. As we show below, it is also important to provide access to the computational steps involved in the sparse factorization, and which are compatible with the sparse matrix storage scheme. R, often called GNU S, is the perfect environment for implementing such algorithms and functionalities in view of statistical applications, see Ihaka and Gentleman (1996); R Development Core Team (2010a). Therefore, **spam** has been conceived as a publicly available R package available from the Comprehensive R Archive Network at `http://CRAN.R-project.org/package=spam`. For reasons of efficiency many functions of **spam** are programmed in Fortran with the additional advantage of abundantly available good code. On the other hand, Fortran does not feature dynamic memory allocation. While there are several remedies, these could lead to a minor decrease in memory efficiency.

To be more specific about one of **spam**'s main features, assume we need to calculate $\mathbf{A}^{-1}\mathbf{b}$ with $\mathbf{A}$ a symmetric positive-definite matrix featuring some sparsity structure, which is usually accomplished by solving $\mathbf{Ax} = \mathbf{b}$. 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 $\mathbf{R}^\top\mathbf{y} = \mathbf{b}$ and $\mathbf{Rx} = \mathbf{y}$, called forwardsolve and backsolve, respectively. To reduce the 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., $\mathbf{U}^\top\mathbf{U} = \mathbf{P}^\top\mathbf{AP}$, with $\mathbf{U}$ an upper triangular matrix. There exist many different algorithms to find permutations which are optimal for specific matrices or at least close to optimal with respect to different criteria. Note that $\mathbf{R}$ and $\mathbf{U}$ cannot be linked through $\mathbf{P}$ alone. Figure 1 illustrates the factorization with and without permutation. For solving a linear system the two triangular solves are performed after the factorization. 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 very important to have a very efficient integration (with respect to calculation time and storage capacity) of the Cholesky factorization. In the case of GMRF, the off-diagonal non-zero elements correspond to the conditional dependence structure. However, for the calculation of the Cholesky factor, the values themselves are less important than the sparsity structure, which is often represented using a graph with edges representing the non-zero elements, see Figure 1.

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

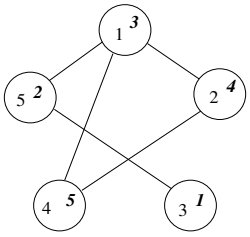
[1] Determine permutation and permute the input matrix $\mathbf{A}$ to obtain $\mathbf{P}^\top\mathbf{AP}$

[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

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



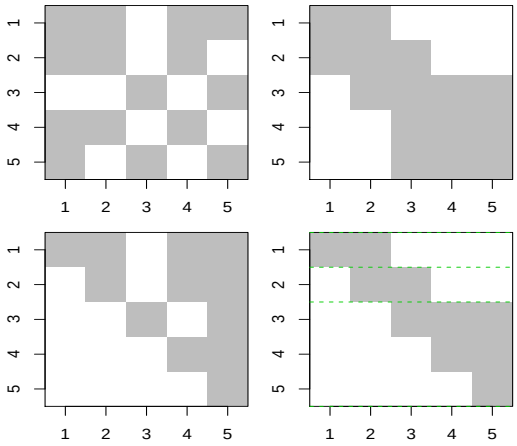


Figure 1: The symmetric positive-definite $n = 5$ matrix $\mathbf{A}$ and the sparsity structure of $\mathbf{A}$ and $\mathbf{P}^\top\mathbf{AP}$ (top row). The graph associated to the matrix $\mathbf{A}$ and the Cholesky factors $\mathbf{R}$ and $\mathbf{U}$ of $\mathbf{A}$ and $\mathbf{P}^\top\mathbf{AP}$ 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). The dashed lines in $\mathbf{U}$ indicate the supernode partition, see Section 2 and 3.2.

to very considerable gains in computational efficiency (also noticed by Rue and Held 2005, page 51). However, none of the existing sparse matrix packages in R (**SparseM**, Koenker and Ng 2010, and **Matrix**, Bates and Maechler 2010) provide the possibility to carry out Step 3 separately and **spam** fills this gap.

1.3. Outline

This article is structured as follows. The next section outlines in more detail the integration of the Cholesky factorization. Section 3 discusses the sparse matrix implementation in **spam**. In Section 4 we illustrate the performance of **spam** with simulation results for GMRFs. Section 5 illustrates **spam** two specific real data examples. Discussion and the positioning of **spam** and the Cholesky factorization in a larger framework are given in Section 6.

2. The integration of the Cholesky factorization

In this section we discuss the individual steps and the actual integration of the Cholesky factorization in more details. The scope of this article prohibits a very detailed discussion, and we refer to George and Liu (1981) or Duff, Erisman, and Reid (1986) as general texts and to the more specific references cited below. **spam** uses a Fortran supernodal left-looking (constructing the lower triangular factor $\mathbf{R}^\top$ column-wise) Cholesky factorization originally developed by E. Ng and B. Peyton at Oak Ridge National Laboratory in the early 1990s, see Ng and Peyton (1993a). The algorithm groups columns (via elimination trees, see Liu 1992, for a definition) that share the same sparsity structure into supernodes, see Figure 1 and, e.g., Liu, Ng, and Peyton (1993). The factorization cycles over the supernodes, performing block factorization within each supernode with appropriate updates derived from previous supernodes. The algorithm has been enhanced since its first implementation by exploiting

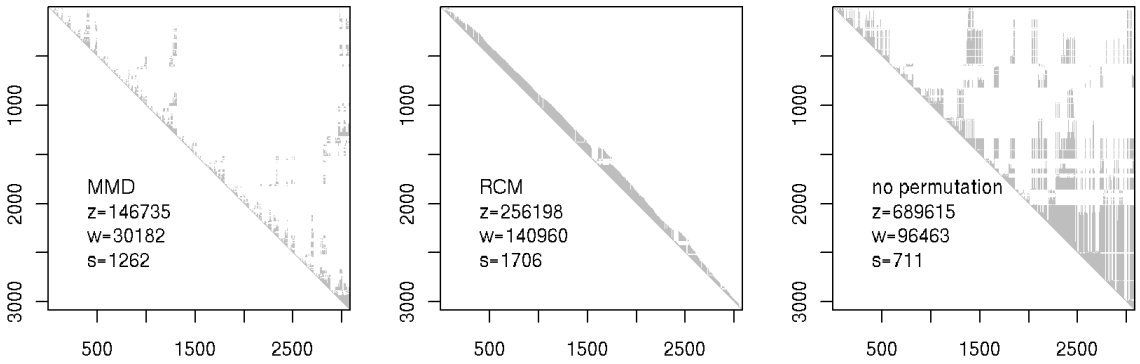


Figure 2: Sparsity structure of the Cholesky factor with MMD, RCM and no permutation of a precision matrix induced by a second order neighbor structure of the US counties. The values z, w are the sizes of the sparsity structure and of the vector containing the column indices of the sparsity structure and s is the number of supernodes.

the memory hierarchy: it splits supernodes into sub-blocks that fit into the available cache; and it unrolls the outer loop of matrix-vector products in order to reduce overhead processor instructions. Within **spam** the algorithm is used to construct the upper triangular factur **R** and, strictly speaking, becomes a supernodal "top-down" algorithm.

A more detailed pseudo algorithm of the Cholesky factorization of a symmetric positive-definite matrix and explanations of some of the steps are given below.

[0] Create the adjacency matrix data structure

[1] Determine permutation and permute the matrix

[2] Symbolic factorization

  [2a] Construct a supernodal elimination tree

  [2b] Reorder according the supernodal elimination tree

  [2c] Perform supernodal symbolic factorization

[3] Numeric factorization

  [3a] Initialization

  [3b] Perform numeric factorization

As for Step 1, there are many different algorithms to find a permutation, two are implemented in **spam**, namely, the multiple minimum degree (MMD) algorithm, (Liu 1985), and the reverse Cuthill-McKee (RCM) algorithm, (George 1971). Additionally, the user has the possibility to manually specify a permutation to be used for the Cholesky factorization. The resulting sparsity structure in the permuted matrix determines the sparsity structure of the Cholesky factor. As an illustration, Figure 2 shows 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. The values z, w are the sizes of the sparsity structure

and of the vector containing the column indices of the sparsity structure and s is the number of supernodes. Note that the actual number of non-zero elements of the Cholesky factor may be smaller than what the constructed sparsity structure indicates. How much fill-in with zeros is present depends on the permutation algorithm, in the example of Figure 2 there are 14111, 97565 and 398353 zero elements in the Cholesky factors resulting from the MMD, RCM, and no permutation, respectively.

Step 2a constructs the elimination tree and supernode elimination tree. From this tree a maximal supernode partition (i.e., the one with the fewest possible supernodes) is calculated. In Step 2b, the children of each parent in the supernodal elimination tree is reordered to minimize the storage requirement (i.e., the last child has the maximum number of non-zeros in its column of the factor). Hence, the matrix is ordered a second time, and if passing the identity permutation to Step 1, the matrix may nevertheless be reordered in Step 2b. Step 2c constructs the sparsity structure of the factor using the results of Gilbert, Ng, and Peyton (1994), which allow storage requirements to be determined in advance, regardless of the ordering strategy used. Note that the symbolic factorization subroutines are independent of any ordering algorithms.

The integration of the Cholesky factorization in **spam** preserves the computational order of the permutation and of the factorization of the underlying Fortran code. Further, the resulting precision in R is equivalent to the precision of the Fortran code. We refer to George and Liu (1981); Liu (1992), Ng and Peyton (1993a) and to Gould, Hu, and Scott (2005b,a) for a detailed discussion about the precision and efficiency of the algorithms by themselves and within the framework of a comparison of different solvers.

3. The sparse matrix implementation of spam

The implementation of **spam** is designed as a trade-off between the following competing philosophical maxims. It should be competitively fast compared to existing tools or approaches in R and it should be easy to use, modify and extend. The former is imposed to assure that the package will be useful and used in practice. The latter is necessary since statistical methods and approaches are often very specific and no single package could cover all potential tools. Hence, the user needs to understand quickly the underlying structure of the implementation of **spam** and to be able to extend it without getting desperate. (When faced with huge amounts of data, sub-sampling is one possibility; using **spam** is another.) This philosophical approach also suggests trying to assure S3 and S4 compatibility, (Chambers 1998; see also Lumley 2004). S4 has higher priority but there are only a handful of cases of S3 discrepancies, which do however not affect normal usage.

To store the non-zero elements, **spam** uses the "old Yale sparse format". In this format, a (sparse) matrix is stored with four elements (vectors), which are (1) the nonzero values row by row, (2) the ordered column indices of nonzero values, (3) the position in the previous two vectors corresponding to new rows, given as pointers, and (4) the column dimension of the matrix. We refer to this format as compressed sparse row (CSR) format. Hence, to store a matrix with z nonzero elements we thus need z reals and $z+n+2$ integers compared to $n \times n$ reals. Section 3.2 describes the format in more details.

Much of the algebraic calculations in **spam** are programmed in Fortran. Some of the Fortran code is based directly on **SPARSKIT**, a basic tool-kit for sparse matrix computations (Saad

1994). Some subroutines are optimized and tailored functions from **SPARSKIT** and a last, large set consists of new functions.

The package **spam** provides two classes, first, `spam` representing sparse matrices and, second, `spam.chol.NgPeyton` representing Cholesky factors. A class definition specifies the objects belonging to the class, these objects are called slots in R and accessed with the `@` operator, see Chambers (1998) for a more thorough discussion. The four vectors of the CSR representation are implemented as slots. In **spam**, all operations can be performed without a detailed knowledge about the slots. However, advanced users may want to work on the slots of the class `spam` directly because of computational savings (e.g., changing only the contents of a matrix while maintaining its sparsity structure, see Section 6.2). The Cholesky factor requires additional information (e.g., the used permutation) hence the class `spam.chol.NgPeyton` contains more slots, which are less intuitive. There are only very few, specific cases, where the user has to access these slots directly. Therefore, user-visibility has been disregarded for the sake of speed. The two classes are discussed in the more technical Section 3.2.

3.1. Methods for the sparse classes of spam

For both sparse classes of **spam**, standard methods like `plot`, `dim`, `backsolve`/`forwardsolve`, `determinant` (based on a Cholesky factor) are implemented and behave as in the case of full matrices. Print methods display the sparse matrix as a full matrix for small matrices and display only the non-zero values otherwise. The corresponding cutoff value, as well as other parameters, can be set and read via `spam.options`.

For the **spam** class additional methods are defined, such as `rbind`/`cbind`, `dim<-`, etc. The group generic functions from `Math`, `Math2` and `Summary` are treated particularly since they operate only on the nonzero entries of the **spam** class. For example, for the matrix `A` presented in the introduction, `range(A)` is the vector `c(0.5, 1)`; that is, the zeros are omitted from the calculation. The help files list further available methods and highlight the (dis-)similarities compared to regular matrices or arrays.

Besides the two sparse classes mentioned above, **spam** does not maintain different classes for different types of sparse matrices, such as symmetric or diagonal matrices. Doing so would result in some storage and computational gain for some matrix operations, at the cost of user visibility. Instead of creating more classes we consider additional specific operators. As an illustration, consider multiplying a diagonal matrix with a sparse matrix. The operator `%d*%` uses standard matrix multiplication if both sides are matrices or multiplies each column according the diagonal entry if the left hand side is a diagonal matrix represented by vector.

3.2. Slots of the sparse classes

This section describes the slots of the sparse classes in **spam** in more detail. The slots of the class `spam` consist of one z-vector of reals, and three vectors of integers of length z, $n+1$ and 2, that correspond to the four elements of the CSR format. These are named:

```
R> slotNames(A)

[1] "entries"     "colindices"  "rowpointers" "dimension"
```

Notice that the row-dimension of `A`, i.e., `A@dimension[1]`, is also determined by the length of `A@rowpointers`, i.e., `length(A@rowpointers) - 1`.

The slots of the Cholesky factor `spam.chol.NgPeyton` can be separated into different groups. The first is linked to storing the factor (i.e., entries and indices), the second group contains the permutation and its inverse, and the third and forth group contain relevant information relating to the factorization algorithm and auxiliary information:

```
R> slotNames(U)

 [1] "entries"     "colindices"  "colpointers" "rowpointers" "dimension"
 [6] "pivot"       "invpivot"    "supernodes"  "snmember"    "memory"
[11] "nnzA"
```

The slot `U@dimension` is again redundant. Similarly, only `U@pivot` or `U@invpivot` would be required. `U@memory` allows speed-up in the update process and `U@nnzA` contains the number of non-zero elements of the original matrix, which is used for calculating fill-in statistics of the factor.

For the Cholesky factor we use a slightly more complicated storage system which is a modification of the CSR format and is due to Sherman (1975). The rows of a supernode have a dense diagonal block and have identical remaining row structure, i.e., for each row of a supernode the column indices are obtained by leaving out the leftmost column index of the preceding row. This is not only exploited computationally (Ng and Peyton 1993b) but also by storing only the column indices of the first row of a supernode. For our example presented in the introduction, we have three supernodes (indicated by the horizontal lines in Figure 1) and the indices are coded as follows:

```
R> U@colindices

[1] 1 2 2 3 3 4 5

R> U@colpointers

[1]  1 3 5 8

R> U@rowpointers

[1]  1  3  5  8 10 11
```

George and Liu (1981, Section 5.4.2) discuss the gain of this storage system for large matrices. With w and s from Figure 2, the difference between z and $w + s + 1$ is the gain when using the modified scheme. However, a more important gain is a much faster access to individual elements of the matrix, because `U@rowpointers` allows a very efficient line access compared to a triplet based (i, j, u_{ij}) format.

Notice that the class `spam.chol.NgPeyton` does not extent the class `spam.chol`. However, by considering only supernodes of size one, `U@colpointers` and `U@rowpointers` are identical, and `U@colindices` corresponds to the format of the `spam` class. In view of this, it would be straightforward to implement other factorization routines (not considering supernodes) leading to different classes for the Cholesky factor. Another possibility would be to define a

virtual class `spam.chol` (also called superclass) and extending classes `spam.chol.NgPeyton` and `spam.chol.someothermethod`.

4. Simulation results for GMRF

In this simulation study, we illustrate Cholesky factorizations in the framework of GMRF. We use a lattice on a regular grid of different sizes and different neighbor structures as well as an irregular lattice, namely the counties of the contiguous USA. The county boundaries we use are from the **maps** package (Becker, Wilks, Brownrigg, and Minka 2010) providing 3082 counties. We consider that two counties are neighbors if they share at least one edge of their polygon description in **maps**. In **spam** adjacency matrices can be constructed using the function `nearest.dist` for regular grids or the function `spam` if the neighbors are available as indices pairs $\{i, j\}$.

For timing and memory usage, we use the R functions `system.time` and `Rprof` as in the following construct:

```
R> Rprof(memory.profiling = TRUE, interval = 0.0001)
R> ressystime <- system.time(expression)
R> Rprof(NULL)
R> resRprof <- summaryRprof(memory = "both")$by.total
```

where `expression` is the R expression under investigation (e.g., to construct Figure 3 we use the expression `{ for(i in 1:100) ch1 <- chol(Qspam) }` for different precision matrices `Qspam`). From `ressystime`, we retain the component `user.self` and, from `resRprof`, we use `mem.total` of `"system.time"`. The small time interval argument of `Rprof` (here set to `0.0001`) helps (at least partially) to circumvent the issues in precisely measuring the memory amount with `Rprof`; see also R Development Core Team (2010b). However, our simulations show that the measurement of timing and memory usage varies and repeating the same simulation indicates a coefficient of variation of about 2% and 0.8%, respectively.

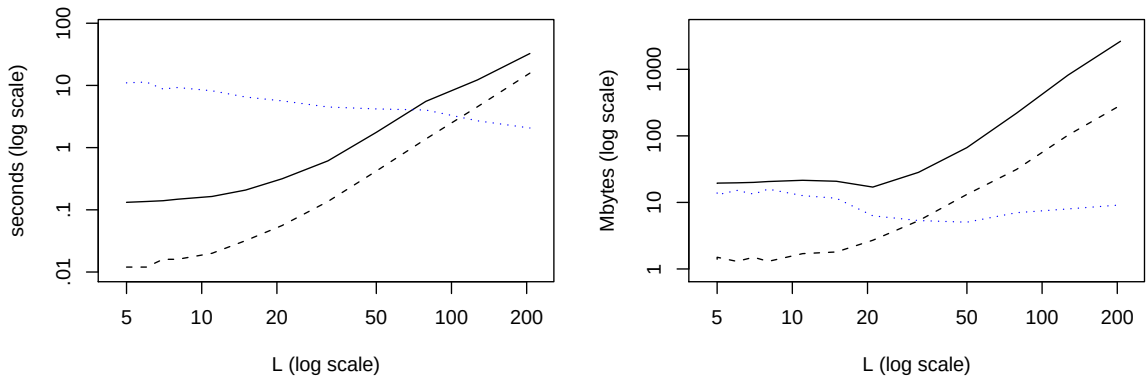


Figure 3: Total time (left) and memory usage (right) for 101 Cholesky factorizations (solid) and one factorization and 100 updates (dashed) of a precision matrix from different sizes L of regular $L \times L$ grids with a second order neighbor structure. The dotted line is the ratio between both curves. The precision matrix from $L = 200$ has $L^4 = 1.6 \cdot 10^9$ elements.

	Regular grid		US counties	
Options or arguments	time	memory	time	memory
Using the specific call `chol.spam`	0.971	0.942	1.001	1.047
Option `safemode = c(FALSE, FALSE, FALSE)`	0.958	0.959	1.008	1.009
Option `cholsymmetrycheck = FALSE`	0.759	0.760	0.883	0.811
Passing `memory = list(nnzR = ..., nnzcolindices = ...)`	0.938	1.030	0.963	1.178
All of the above	0.748	0.700	0.863	0.905
All of the above and passing `pivot = ...` to `chol.spam`	0.769	0.682	0.772	0.917
All of the above and option `cholpivotcheck = FALSE`	0.701	0.672	0.766	0.911
Numeric update only using `update`	0.177	0.196	0.200	0.142

Table 1: Relative (to a generic `chol` call) gain of time and memory usage with different options and arguments in the case of a second order neighbor structure of a regular 50×50 grid and of the US counties. The time and memory usage for the generic call `chol` are 2.1 seconds, 53.7 Megabytes and 5.2 seconds, 145.4 Megabytes, respectively.

The simulations are done with **spam** 0.22-0 and R 2.9.2 on an i686-pc-linux-gnu computer with a 2.66 GHz Intel Core2 Duo processor and 2 Gigabyte of RAM.

We first compare the total time and the memory required for Cholesky factorizations for different sizes of regular grids. In our MCMC framework, the sparsity structure of the precision matrix does not change and we can compare the time and memory requirements with one Cholesky factorization followed by numerical updates of the factor (Step 3). Figure 3 shows the total time (left) and memory usage (right) for 101 Cholesky factorization (solid) and one factorizations and 100 updates (dashed) of a precision matrix from different sizes L of regular $L \times L$ grids with a second order neighbor structure. We have chosen fixed but arbitrary values for the conditional dependence of the first and second order neighbors. The precision matrix from $L = 200$ has $L^4 = 1.6 \cdot 10^9$ elements. The update is performed with the function `update` that takes as arguments a Cholesky factor and a symmetric positive-definite matrix with the same sparsity structure. The gain in using the update only decreases slightly as the size of the matrices increases. For matrices up to 50000 elements the update is about 10 times faster and uses less than 15 times the memory.

The package **spam** offers several options that can be used to increase speed and decrease memory allocation compared to the default values. Most of the options are linked to reduced input testing and validation, which can often be eliminated after preliminary testing or within an MCMC framework. Table 1 gives the relative speed-up of different options in the case of the two neighbor structure of a regular 50×50 grid and of the US counties. If the user knows that the matrix is symmetric, a test can be avoided with the flag `cholsymmetrycheck = FALSE`. Minor additional improvements consist in setting `safemode = c(FALSE, FALSE, FALSE)`, specifying, for example, if elements of a sparse matrix should be tested for storage mode double or for the presence of `NA`s. The size of the Cholesky factor is determined during the symbolic factorization (Step 2c) but we need to allocate vectors in R of appropriate sizes for the Fortran call. There is a trade-off in reserving enough space to hold the factor and its structure versus computational efficiency. **spam** addresses this issue as follows. We have simple formulas that try to estimate the necessary sizes. If the estimated size is too small the Fortran routine returns an error to R, which allocates more space and calls the Fortran routine again. However, to save time and memory the user can also pass better estimates of the allocation sizes to `chol` with the argument `memory = list(nnzR = ..., nnzcolindices =`

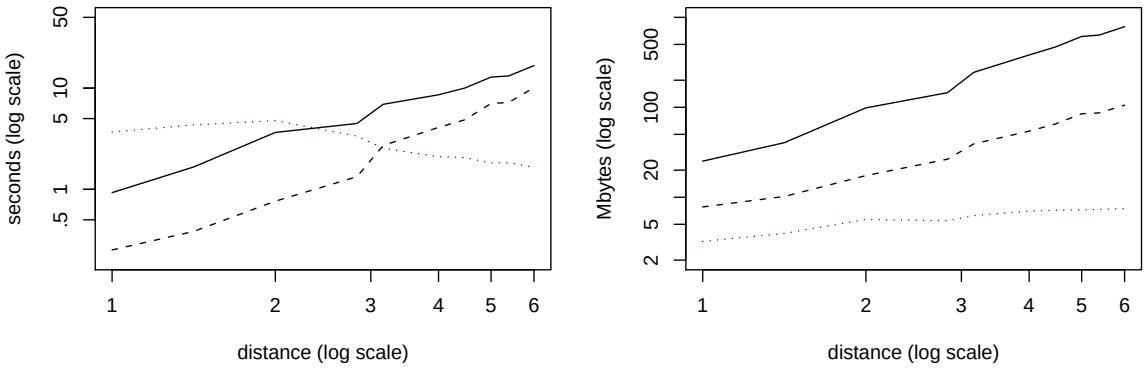


Figure 4: Total time (left) and memory usage (right) for 101 Cholesky factorizations (solid) and one factorization and 100 updates (dashed) of a precision matrix resulting from a regular 50×50 grid as a function of the distance for which grid points are considered as neighbors. The dotted line is the ratio between both curves. For distance 6 each grid point has up to 112 neighbors and the dependence structure requires at least 18 parameters.

`...)`. The minimal sizes for a fixed sparsity structure can be obtained from a `summary` call. If the user specifies the permutation to be used in `chol` with `pivot = ...` the argument `memory = list(nnzR = ..., nnzcolindices = ...)` should be given to fully exploit the time gain of doing so. Further, the flag `cholpivotcheck = FALSE` improves the computational savings of manually specifying the permutation additionally.

As an illustration for the last two rows of Table 1, consider a precision matrix `Qspam` of class `spam` and perform a first decomposition `Qfact <- chol(Qspam)`. Successive factorizations of a new precision matrix `Qspamnew` can be performed as follows.

```
R> tmp <- summary(Qfact)
R> pivot <- ordering(ch1)
R> spam.options(cholsymmetrycheck = FALSE, safemode = c(FALSE, FALSE, FALSE),
+    cholpivotcheck = FALSE)
R> Qfactnew <- chol.spam(Qspamnew, pivot = pivot,
+    memory = list(nnzR = tmp$nnzR, nnzcolindices = tmp$nnzc))
```

Of course, all of the above could be also be done by the following single command.

```
R> Qfactnew <- update(Qfact, Qspamnew)
```

When approximating isotropic second order stationary Gaussian fields by GMRF (cf, Rue and Held 2005, Section 5.1), many neighbors need to be considered in the dependence structure. Figure 4 shows the total time and memory for 101 Cholesky factorizations and one factorization and 100 updates for a precision matrix resulting from a regular 50×50 grid as a function of the distance for which grid points are considered as neighbors. For distance 6 each grid point has up to 112 neighbors and the dependence structure requires at least 18 parameters. We refer to Rue and Held (2005) for a detailed discussion and issues arising from the approximation.

The results of this section are based on 101 Cholesky factorizations and computation time scales virtually linearly for multiples thereof. However, in a practical MCMC setting the

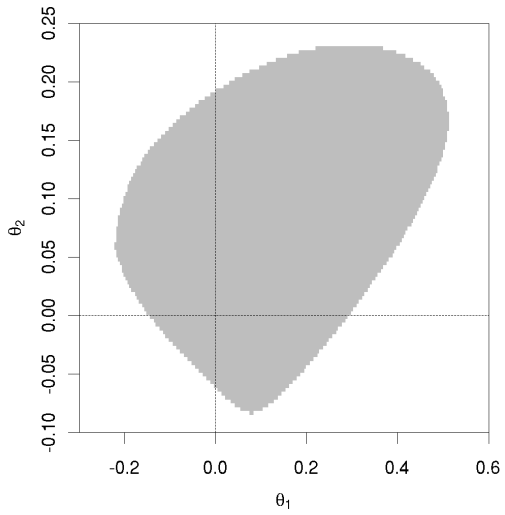


Figure 5: Valid parameter space for the second order neighbor model of the US counties.

factorization is only one part of each iteration and, additionally, the set of the valid parameters is often unknown. The first issue is addressed with competitive algorithms in **spam** but also needs to be considered when writing R code, see Section 6.2. A typical procedure for the second issue is to sample from a hypothetical parameter space and to use a trial-and-error approach by calling the `update` function and verifying if the resulting matrix is positive definite. (For simple examples, it may be possible to give bounds on the parameter space that can be used when sampling, see also Rue and Held (2005), Section 2.7.) In the cases of a non-admissible value, the functions hand back an error, a warning or the value `NULL`, depending on the value of a specific flag. Figure 5 illustrates the valid parameter space for the second order neighbor model of the US counties. The 'brute force' code used for Figure 5 is as follows.

```
R> spam.options("cholupdatesingular" = "null")
R> In <- diag.spam(nrow(UScounties.storder))
R> struct <- chol(In + 0.2 * UScounties.storder + 0.1 * UScounties.ndorder)
R> len.1 <- 180
R> len.2 <- 100
R> theta.1 <- seq(-.225, to = 0.515, len = len.1)
R> theta.2 <- seq(-.09, to = 0.235, len = len.2)
R> grid <- array(NA, c(len.1, len.2))
R> for(i in 1:len.1)
+    for(j in 1:len.2)
+      grid[i, j] <- is.null(update(struct, In + theta.1[i] *
+        UScounties.storder + theta.2[j] * UScounties.ndorder))
```

On the aforementioned computer, about 50 tests are evaluated per second. Hence, it takes about 6 minutes to execute the above code. The bounds for `theta.1` and `theta.2` were empirically determined.

5. Data examples

In this section we illustrate the **spam** package by analyzing two datasets which are modeled using latent GMRF. Both examples are also discussed (without documenting code) in Rue and Held (2005), Sections 4.2.1 and 4.4.2, to which we refer for technical details. Rue and Held (2005) use in both cases a slightly different approach for the MCMC steps, here we illustrate **spam** with a conceptually simpler but computationally tougher version of the Gibbs sampler.

We assume that the observations $\mathbf{y}$ are conditionally independent given latent parameters $\boldsymbol{\eta}$ and additional parameters $\boldsymbol{\theta}_\mathbf{y}$

$$\pi(\mathbf{y} \mid \boldsymbol{\eta}, \boldsymbol{\theta}_\mathbf{y}) = \prod_{i=1}^{n} \pi(y_i \mid \eta_i, \boldsymbol{\theta}_\mathbf{y}),$$

where $\pi(\cdot \mid \cdot)$ denotes the conditional density of the first argument given the second argument. The latent parameters $\boldsymbol{\eta}$ are part of a larger latent random field $\mathbf{x}$, which is modeled as a GMRF with mean $\boldsymbol{\mu}$ and precision matrix $\mathbf{Q}$, both depending on parameters $\boldsymbol{\theta}_\mathbf{x}$; that is,

$$\pi(\mathbf{x} \mid \boldsymbol{\theta}_\mathbf{x}) \propto \exp\Big(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \mathbf{Q}(\mathbf{x}-\boldsymbol{\mu})\Big).$$

5.1. Normal response model

Consider the R dataset `"UKDriverDeaths"`, a time series giving the monthly totals of car drivers in Great Britain killed or seriously injured from January 1969 to December 1984 ($n = 192$). The series y_i exhibits a strong seasonal component (denoted by s_i) and a (possibly) smooth trend (denoted by t_i). Here, we want to predict the pattern $\eta_i = s_i + t_i$ for the next $m = 12$ months. We assume that the square root responses are normal and conditionally independent:

$$\pi(\mathbf{y} \mid \boldsymbol{\eta}, \theta_\mathbf{y}) = \pi(\mathbf{y} \mid \mathbf{s}, \mathbf{t}, \kappa_\mathbf{y}) \propto \kappa_\mathbf{y}^{\frac{n}{2}} \exp\Big(-\frac{\kappa_\mathbf{y}}{2} \sum_{i=1}^{n} (y_i - t_i - s_i)^2\Big).$$

We assume further that $\sum_{j=0}^{11} s_{i+j}$, $i = 1, \ldots, n+1$, are independent normals with mean zero and precision $\kappa_\mathbf{s}$ (an intrinsic GMRF model for seasonal variation, e.g., Rue and Held 2005, page 122) and $t_i - 2t_{i+1} - t_{i+2}$, $i = 1, \ldots, n+m-2$, are independent normals with mean zero and precision $\kappa_\mathbf{t}$ (an intrinsic second order random walk model). Hence,

$$\pi(\mathbf{s} \mid \kappa_\mathbf{s}) \propto \kappa_\mathbf{s}^{\frac{n+1}{2}} \exp\Big(-\frac{1}{2}\mathbf{s}^\top \mathbf{Q}_\mathbf{s}\mathbf{s}\Big),$$
$$\pi(\mathbf{t} \mid \kappa_\mathbf{t}) \propto \kappa_\mathbf{t}^{\frac{n+m-2}{2}} \exp\Big(-\frac{1}{2}\mathbf{t}^\top \mathbf{Q}_\mathbf{t}\mathbf{t}\Big),$$

where $\mathbf{Q}_\mathbf{s}$ and $\mathbf{Q}_\mathbf{t}$ are given by analogues of equations (3.59) and (3.40) of Rue and Held (2005). Using independent Gamma priors for the three precisions, e.g., $\pi(\kappa_\mathbf{s}) \propto \kappa_\mathbf{s}^{\alpha_\mathbf{s}-1} \exp\big(-\kappa_\mathbf{s}\beta_\mathbf{s}\big)$,

the full joint density is

$$\begin{aligned}\pi(\mathbf{y},\mathbf{s},\mathbf{t},\boldsymbol{\kappa}) &= \pi(\mathbf{y}\mid\mathbf{s},\mathbf{t},\kappa_{\mathbf{y}})\;\pi(\mathbf{s}\mid\kappa_{\mathbf{s}})\;\pi(\mathbf{t}\mid\kappa_{\mathbf{t}})\;\pi(\boldsymbol{\kappa})\\ &\propto \kappa_{\mathbf{s}}^{\alpha_{\mathbf{s}}+\frac{n+1}{2}-1}\kappa_{\mathbf{t}}^{\alpha_{\mathbf{t}}+\frac{n+m-2}{2}-1}\kappa_{\mathbf{y}}^{\alpha_{\mathbf{y}}+\frac{n}{2}-1}\exp\big(-\kappa_{\mathbf{s}}\beta_{\mathbf{s}}-\kappa_{\mathbf{y}}\beta_{\mathbf{t}}-\kappa_{\mathbf{y}}\beta_{\mathbf{y}}\big)\\ &\quad\times\exp\left(-\frac{1}{2}(\mathbf{s}^\top,\mathbf{t}^\top,\mathbf{y}^\top)\begin{pmatrix}\mathbf{Q}_{\mathbf{ss}} & \mathbf{Q}_{\mathbf{st}} & \mathbf{Q}_{\mathbf{sy}}\\ \mathbf{Q}_{\mathbf{ts}} & \mathbf{Q}_{\mathbf{tt}} & \mathbf{Q}_{\mathbf{ty}}\\ \mathbf{Q}_{\mathbf{ys}} & \mathbf{Q}_{\mathbf{yt}} & \mathbf{Q}_{\mathbf{yy}}\end{pmatrix}\begin{pmatrix}\mathbf{s}\\ \mathbf{t}\\ \mathbf{y}\end{pmatrix}\right).\end{aligned}$$

The individual block precisions are, for example, $\mathbf{Q}_{\mathbf{ss}}=\mathbf{Q}_{\mathbf{s}}+\kappa_{\mathbf{y}}\mathbf{D}^\top\mathbf{D}$, $\mathbf{Q}_{\mathbf{ss}}=\mathbf{Q}_{\mathbf{s}}+\kappa_{\mathbf{y}}\mathbf{D}^\top\mathbf{D}$, $\mathbf{Q}_{\mathbf{yy}}=\mathbf{I}_n$, $\mathbf{Q}_{\mathbf{st}}=\mathbf{I}_{n+m}$, $\mathbf{Q}_{\mathbf{sy}}=\mathbf{Q}_{\mathbf{ty}}=\mathbf{D}^\top$ with $\mathbf{D}=(\mathbf{I}_n,\mathbf{0})$. It is now straightforward to implement a Gibbs sampler based on the full conditionals $\pi(\mathbf{s},\mathbf{t}\mid\boldsymbol{\kappa},\mathbf{y})$ and $\pi(\boldsymbol{\kappa}\mid\mathbf{s},\mathbf{t},\mathbf{y})$. The R code to implement is as follows. We first load the data, calculate the square root counts and specify the hyperparameters of the prior for $\boldsymbol{\kappa}=(\kappa_{\mathbf{y}},\kappa_{\mathbf{s}},\kappa_{\mathbf{t}})^\top$ as in Rue and Held (2005).

```
R> data("UKDriverDeaths")
R> y <- sqrt(c(UKDriverDeaths))
R> n <- length(y)
R> m <- 12
R> nm <- n + m
R> priorshape <-  c(4, 1, 1)
R> priorinvscale <- c(4, 0.1, 0.0005)
```

Note that `m` denotes the length of one season, the duration of our prediction. The individual block precisions are now constructed (based on unit precisions).

```
R> Qsy <- diag.spam(n)
R> dim(Qsy) <- c(n + m, n)
R> Qty <- Qsy
R> Qst <- spam(0, nm, nm)
R> Qst[cbind(1:n, 1:n)] <- rep(1, n)
R> Qss <- spam(0, nm, nm)
R> for(i in 0:(nm - m))
+    Qss[i + 1:m, i + 1:m] <- Qss[i + 1:m,i + 1:m] + 1
R> Qtt <- spam(0, nm, nm)
R> Qtt[cbind(1:(nm - 1), 2:nm)] <- -c(2, rep(4, nm - 3), 2)
R> Qtt[cbind(1:(nm - 2), 3:nm)] <- rep(1, nm - 2)
R> Qtt <- Qtt + t(Qtt)
R> diag(Qtt) <- c(1, 5, rep(6, nm - 4), 5, 1)
```

We construct now a "template" precision matrix of the GMRF characterized by $\pi(\mathbf{s},\mathbf{t}\mid\boldsymbol{\kappa},\mathbf{y})$ to obtain the structure of the Cholesky factor. The sparsity structure of the precision matrix and of its Cholesky factor are shown in Figure 6.

```
R> Qst_yk <- rbind(cbind(Qss + diag.spam(nm), Qst),
+    cbind(Qst, Qtt + diag.spam(nm)))
R> struct <- chol(Qst_yk)
```

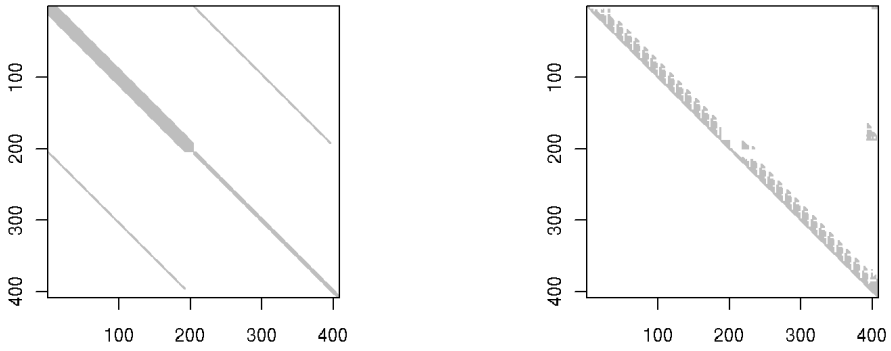


Figure 6: The sparsity structure of the precision matrix of $\pi(\mathbf{s}, \mathbf{t} \mid \boldsymbol{\kappa}, \mathbf{y})$ and of its Cholesky factor.

The code from now on does not differ for sparse and non-sparse input matrices. We need to specify some parameters for the Gibbs sampler, initialize the arrays containing the posterior samples and starting values for $\boldsymbol{\kappa}$.

```
R> burnin <- 10
R> ngibbs <- 500
R> totalg <- ngibbs + burnin
R> set.seed(14)
R> spost <- tpost <- array(0, c(totalg, nm))
R> kpost <- array(0, c(totalg, 3))
R> kpost[1,] <- c(0.5, 28, 500)
R> postshape <- priorshape + c(n / 2, (n + 1) / 2, (n + m - 2) / 2)
```

The Gibbs loop is now as follows:

```
R> for(i in 2:totalg) {
+    Q <- rbind(cbind(kpost[i - 1, 2] * Qss + kpost[i - 1, 1] * Qst,
+      kpost[i - 1, 1] * Qst), cbind(kpost[i - 1, 1] * Qst,
+      kpost[i - 1, 3] * Qtt + kpost[i - 1, 1] * Qst))
+    b <- c(kpost[i - 1, 1] * Qsy %*% y, kpost[i - 1, 1] * Qsy %*% y)
+
+    tmp <- rmvnorm.canonical(1, b, Q, Lstruct = struct)
+    spost[i,] <- tmp[1:nm]
+    tpost[i,] <- tmp[1:nm + nm]
+
+    tmp <- y - spost[i, 1:n]-tpost[i, 1:n]
+    postinvscale <- priorinvscale +
+      c( sum( tmp^2)/2, t(spost[i,]) %*% (Qss %*% spost[i,]) / 2,
+      t(tpost[i,]) %*% (Qtt %*% tpost[i,]) / 2)
+    kpost[i,] <- rgamma(3, postshape, postinvscale)
+  }
```

The loop takes a few seconds to run. After eliminating the burn-in, summary statistics can be calculated. For example, for the precisions we have:

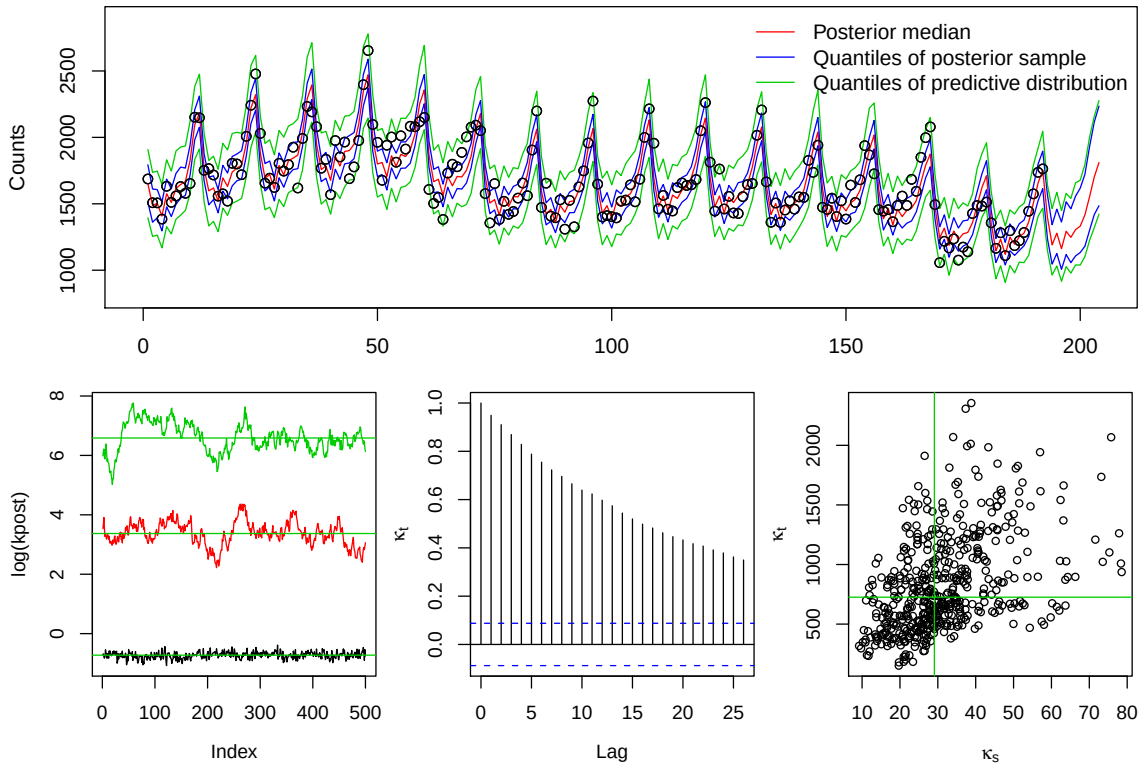


Figure 7: Observed counts, the posterior median, its quantiles and the quantiles of the predictive distribution (top panel). Trace plots of the log precisions, autocorrelation plot of $\kappa_\mathbf{t}$ and scatter plot of $\kappa_\mathbf{t}$ against $\kappa_\mathbf{s}$ (lower row). Solid lines indicate posterior medians.

```
R> summary(kpost)
```

```
       V1              V2              V3
 Min.   :0.3406   Min.   : 9.21   Min.   : 151.8
 1st Qu.:0.4509   1st Qu.:22.55   1st Qu.: 527.7
 Median :0.4843   Median :29.10   Median : 726.0
 Mean   :0.4859   Mean   :31.44   Mean   : 827.5
 3rd Qu.:0.5204   3rd Qu.:37.31   3rd Qu.:1057.4
 Max.   :0.6800   Max.   :78.54   Max.   :2355.1
```

The predictive distribution of $\mathbf{y}$ is obtained by adding a mean zero normal random variable with precision `kpost[i, 1]` to `spost[i,] + tpost[i,]`. Figure 7 gives the posterior median, the posterior quantiles and the quantiles of the predictive distribution. The trace plots of the log precisions do not indicate evidence against bad mixing.

The source code of this example is given in `demo("article-jss-example1")`. For this example Knorr-Held and Rue (2002) suggest to use a Metropolis–Hastings step and to update the precisions with a scaling factor δ having density $\pi(\delta) \propto 1 + 1/\delta$, for $\delta \in [1/D, D]$, where $D > 1$ is a tuning parameter, see also Rue and Held (2005).

5.2. Besag-York-Mollié model

In this second example, we consider the number of cases of oral cavity cancer for a 5 year period (1986–1990) in the $n = 544$ districts (Landkreise) of Germany (Knorr-Held and Raßer 2000; Held, Natario, Fenton, Rue, and Becker 2005) and explore the spatial distribution of the relative risk. The common approach is to assume that the data are conditionally independent Poisson counts

$$\pi(y_i \mid \eta_i) \propto \exp\big(y_i\eta_i - e_i \exp(\eta_i)\big), \qquad i = 1, \ldots, n,$$

where e_i is the expected number of cases in region i. The raw counts y_i and the standardized mortality ratios (SMRs) y_i/e_i are displayed in Figure 8, left and middle panel. For the log-relative risk, we use $\boldsymbol{\eta} = \mathbf{u} + \mathbf{v}$, where $\mathbf{v}$ is a zero mean white noise with precision $\kappa_\mathbf{v}$ and $\mathbf{u}$ is a spatially structured component (Besag, York, and Mollié 1991; Mollié 1996). More precisely, $\mathbf{u}$ is a first order intrinsic GMRF with density (Rue and Held 2005, Section 3.3.2)

$$\pi(\mathbf{u} \mid \kappa_\mathbf{u}) \propto \kappa_\mathbf{u}^{\frac{n-1}{2}} \exp\Big(-\frac{\kappa_\mathbf{u}}{2}\sum_{i\sim j}(u_i - u_j)^2\Big), \tag{1}$$

where $i \sim j$ denotes the set of all unordered pairs of neighbors, i.e., regions sharing a common border. As suggested by Rue and Held (2005), we reparameterize by setting

$$\pi(\boldsymbol{\eta} \mid \mathbf{u}, \kappa_\mathbf{v}) \propto \kappa_\mathbf{v}^{\frac{n}{2}} \exp\Big(-\frac{\kappa_\mathbf{v}}{2}(\boldsymbol{\eta} - \mathbf{u})^\top(\boldsymbol{\eta} - \mathbf{u})\Big) \quad \text{and} \quad \mathbf{x} = \begin{pmatrix} \mathbf{u} \\ \boldsymbol{\eta} \end{pmatrix}.$$

With Gamma priors for the precision parameters, the posterior density is

$$\begin{aligned}\pi(\mathbf{x}, \boldsymbol{\kappa} \mid \mathbf{y}) \propto{}& \kappa_\mathbf{v}^{\alpha_\mathbf{v}+\frac{n}{2}-1}\kappa_\mathbf{u}^{\alpha_\mathbf{u}+\frac{n-1}{2}-1} \\ &\times \exp\left(-\kappa_\mathbf{v}\beta_\mathbf{v} - \kappa_\mathbf{u}\beta_\mathbf{u} + \sum_{i=1}^n\big(y_i\eta_i - e_i\exp(\eta_i)\big) - \frac{1}{2}\mathbf{x}^\top\begin{pmatrix} \kappa_\mathbf{u}\mathbf{R} + \kappa_\mathbf{v}\mathbf{I} & -\kappa_\mathbf{v}\mathbf{I} \\ -\kappa_\mathbf{v}\mathbf{I} & \kappa_\mathbf{v}\mathbf{I}\end{pmatrix}\mathbf{x}\right)\end{aligned}$$

where $\mathbf{R}$ is the "structure" matrix imposed by (1). While $\pi(\mathbf{x} \mid \boldsymbol{\kappa})$ is a GMRF, $\pi(\mathbf{x} \mid \boldsymbol{\kappa}, \mathbf{y})$ is not. We use a second order Taylor approximation of $\sum_{i=1}^n y_i\eta_i - e_i\exp(\eta_i)$ (as a function of $\boldsymbol{\eta}$) around $\boldsymbol{\eta}_0 = (\eta_{01}, \ldots, \eta_{0n})^\top$ to construct an appropriate GMRF that we use as a proposal in a Metropolis–Hastings step. More specifically, we use the proposal $q(\mathbf{x} \mid \mathbf{x}^{(i)}, \boldsymbol{\kappa})$ with density proportional to

$$\exp\left(-\frac{1}{2}\mathbf{x}^\top\begin{pmatrix} \kappa_\mathbf{u}\mathbf{R} + \kappa_\mathbf{v}\mathbf{I} & -\kappa_\mathbf{v}\mathbf{I} \\ -\kappa_\mathbf{v}\mathbf{I} & \kappa_\mathbf{v}\mathbf{I}\end{pmatrix}\mathbf{x} - \frac{1}{2}\boldsymbol{\eta}^\top\mathrm{diag}(\mathbf{c})\boldsymbol{\eta} + \mathbf{b}^\top\boldsymbol{\eta}\right), \tag{2}$$

where $\mathbf{c}_i = e_i\exp(\eta_{0i})$ and $\mathbf{b}_i = y_i + (\eta_{0i} - 1)\mathbf{c}_i$. Hence, one possible choice of $\boldsymbol{\eta}_0$ is the current state of $\boldsymbol{\eta}$; for other choices see, e.g., Rue and Held (2005).

We use a block update (see, e.g., Knorr-Held and Rue 2002) by sampling first $\boldsymbol{\kappa}^\star$ from $\pi(\boldsymbol{\kappa}^\star \mid \mathbf{x}, \mathbf{y})$ and then sampling $\mathbf{x}^\star$ from $q(\mathbf{x}^\star \mid \boldsymbol{\kappa}^\star, \mathbf{x}, \mathbf{y})$. The joint proposal $(\boldsymbol{\kappa}^\star, \mathbf{x}^\star)$ is then accepted/rejected jointly with probability

$$\alpha = \min\left\{1, \frac{\pi(\boldsymbol{\kappa}^\star, \mathbf{x}^\star \mid \mathbf{y})}{\pi(\boldsymbol{\kappa}, \mathbf{x} \mid \mathbf{y})}\,\frac{q(\boldsymbol{\kappa}, \mathbf{x} \mid \boldsymbol{\kappa}^\star, \mathbf{x}^\star, \mathbf{y})}{q(\boldsymbol{\kappa}^\star, \mathbf{x}^\star \mid \boldsymbol{\kappa}, \mathbf{x}, \mathbf{y})}\right\}, \tag{3}$$

where $q(\boldsymbol{\kappa}^\star, \mathbf{x}^\star \mid \boldsymbol{\kappa}, \mathbf{x}, \mathbf{y}) = q(\mathbf{x}^\star \mid \boldsymbol{\kappa}^\star, \mathbf{x}, \mathbf{y})\, \pi(\boldsymbol{\kappa}^\star \mid \mathbf{x}, \mathbf{y})$.

We guide the reader through the R code of the Gibbs sampler, also given in `demo("article-jss-example2")`. First we need to setup data and adjacency structure, provided in the **spam** package for convenience and also available from http://www.r-inla.org/, or from http://www.math.ntnu.no/~hrue/GMRF-book/germany.graph and http://www.math.ntnu.no/~hrue/GMRF-book/oral.txt.

```
R> data("oral")
R> attach(oral)
R> A <- adjacency.landkreis("../germany.graph")
R> n <- dim(A)[1]
```

Next, we set the hyperparameters, define the parameters for the Gibbs sampler and allocate variables for the posterior, containing the starting values.

```
R> ahyper <- c(1, 1)
R> bhyper <- c(0.5, 0.01)
R> burnin <- 500
R> ngibbs <- 1500
R> totalg <- burnin + ngibbs
R> set.seed(14)
R> upost <- npost <- array(0, c(totalg, n))
R> kpost <- array(0, c(totalg, 2))
R> kpost[1,] <- c(40, 500)
R> upost[1,] <- npost[1,] <- rep(0, n)
R> accept <- numeric(totalg)
```

The next few commands construct templates of the individual block precision matrices as given in (2), and pre-calculate quantities, notably of (3) for $i = 1$.

```
R> Q1 <- R <- diag.spam(diff(A@rowpointers)) - A
R> dim(Q1) <- c(2 * n, 2 * n)
R> Q2 <-  rbind(cbind(diag.spam(n), -diag.spam(n)),
+    cbind(-diag.spam(n), diag.spam(n)))
R> diagC <- as.spam(diag.spam(c(rep(0, n), rep(1, n))))
R> struct <- chol(Q1 + Q2 + diag.spam(2 * n),
+    memory = list(nnzcolindices = 5500))
R> u <- upost[1,]
R> eta <- npost[1,]
R> uRu <- t(u) %*% (R %*% u) / 2
R> etauetau <- t(eta - u) %*% (eta - u) / 2
R> postshape <- ahyper + c(n - 1, n) / 2
```

The Gibbs sampler proceeds now with sampling $\boldsymbol{\kappa}^\star$ and $\mathbf{x}^\star$ and then calculating the acceptance probability (3) on a log scale. Note that some quantities only need to be recalculated if we accept the proposal, i.e., if `(logU < logalpha)` is true.

```
R> for (i in 2:totalg) {
+    kstar <- rgamma(2, postshape, bhyper + c(uRu, etauetau))
+    expeta0E <- exp(eta) * E
+    expeta0Eeta01 <- expeta0E * (eta - 1)
+    diagC@entries <- expeta0E
+    Q <- kstar[1] * Q1 + kstar[2] * Q2 + diagC
+    b <- c(rep(0, n), Y + expeta0Eeta01)
+
+    xstar <- rmvnorm.canonical(1, b, Q, Lstruct = struct)
+    ustar <- xstar[1:n]
+    nstar <- xstar[1:n + n]
+
+    uRustar <- t(ustar) %*% (R %*% ustar) / 2
+    etauetaustar <- t(nstar - ustar) %*% (nstar - ustar) / 2
+
+    exptmp <- sum(expeta0Eeta01 * (eta - nstar) -
+      E * (exp(eta) - exp(nstar))) -
+      sum( nstar^2 * expeta0E) / 2 + sum(eta^2 * expeta0E) / 2 -
+      kstar[1] * uRu + kpost[i - 1, 1] * uRustar -
+      kstar[2] * etauetau + kpost[i - 1, 2] * etauetaustar
+    factmp <- (postshape - 1) * (log(kstar) - log(kpost[i - 1, 1]))
+
+    logalpha <- min(0, exptmp + sum(factmp))
+    logU <- log(runif(1))
+
+    if (logU < logalpha) {
+      upost[i,] <- u <- ustar
+      npost[i,] <- eta <- nstar
+      kpost[i,] <- kstar
+      uRu <- uRustar
+      etauetau <- etauetaustar
+      accept[i] <- 1
+    } else {
+      upost[i,] <- upost[i - 1,]
+      npost[i,] <- npost[i - 1,]
+      kpost[i,] <- kpost[i - 1,]
+    }
+  }
```

After the loop, we eliminate the burn-in from the samples and proceed with the usual evaluation of the posterior sample. The right panel of Figure 8 shows the posterior median of the estimated relative risks, i.e., $\exp(\mathbf{u})$. Figure 9 gives several diagnostics plots for the samples of the posterior precisions $\kappa_{\mathbf{u}}$ and $\kappa_{\mathbf{v}}$. Note that $\kappa_{\mathbf{v}}$ exhibits a somewhat slow mixing. The proposal $q(\boldsymbol{\kappa}^\star, \mathbf{x}^\star \mid \boldsymbol{\kappa}, \mathbf{x}, \mathbf{y})$ depends on the precision priors and when choosing substantially different priors, the acceptance rate may be much lower.

The Gibbs sampler as illustrated above takes about 5.9 seconds per 1000 iterations. Not passing the `struct` object to `rmvnorm.canonical` increases the total computation time by

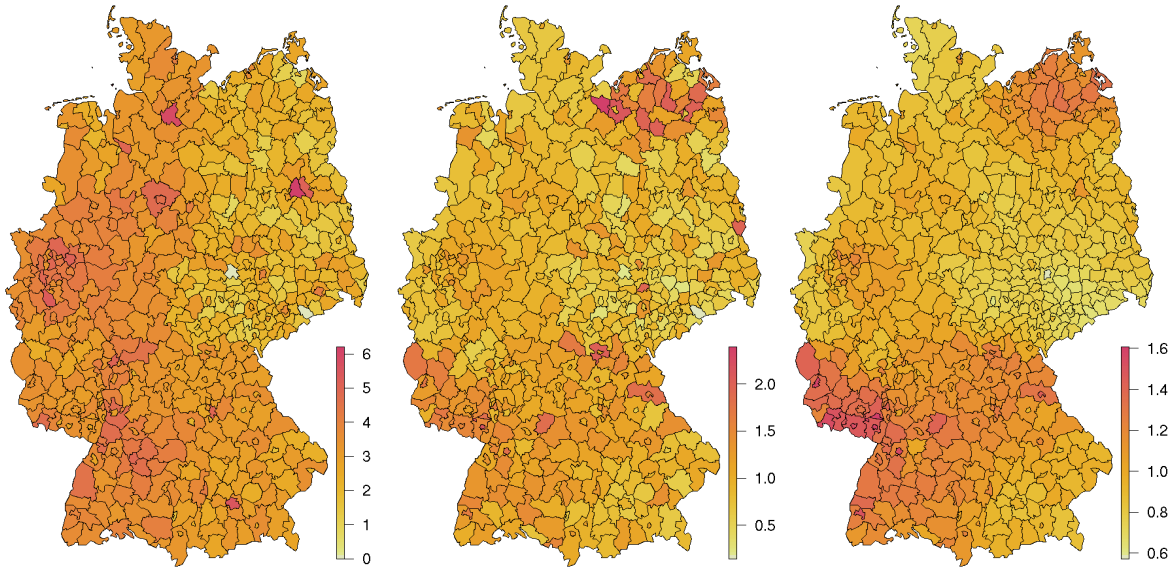


Figure 8: Observed counts (log scale, left), standardized mortality rates (middle) and posterior median of the estimated relative risk ($\exp(\mathbf{u})$, right) of the oral cavity cancer data. Note the different scales.

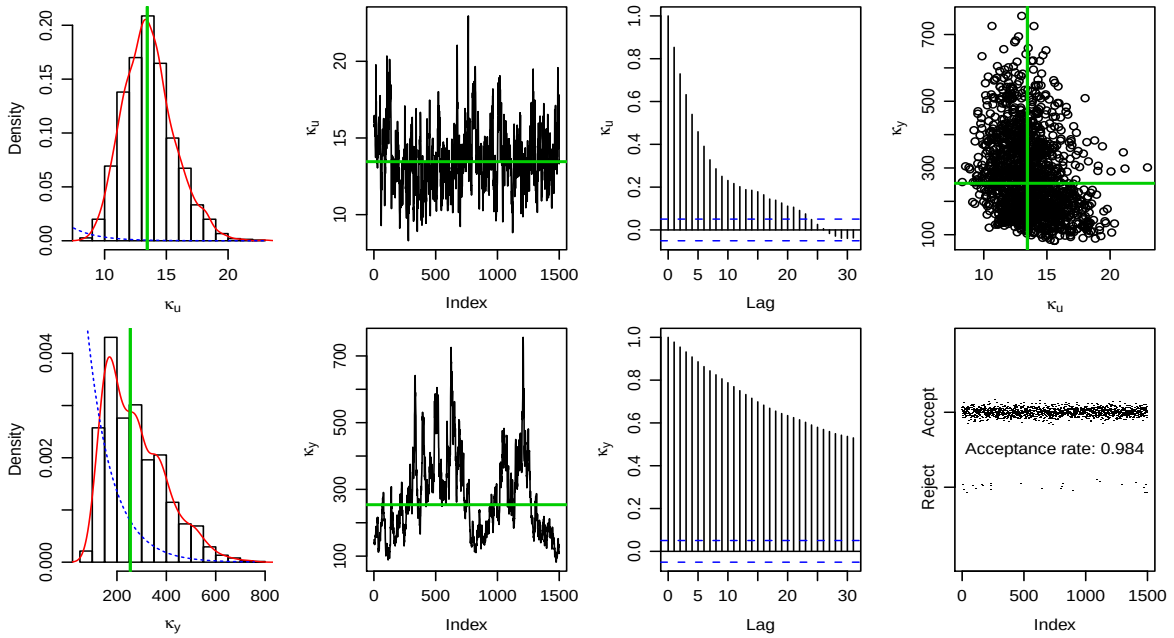


Figure 9: Posterior density (`density` smooth in solid red, prior in dotted blue), trace plots, autocorrelation function (no thinning) and scatter plots of the precision parameters for $\kappa_{\mathbf{u}}$ (top) and $\kappa_{\mathbf{y}}$ (bottom). The thick green lines represent the posterior median. Lower right most panel: history of accepting the joint proposal (jittered).

roughly a factor of 1.7 and when working with full matrices, by a factor 40. The code can be improved for slight gains in time but loosing somewhat its readability.

6. Discussion

This paper highlights some of the functionalities of the R package **spam**. However, for details we refer to the enclosed help pages. The package is based on stable and well tested code but unlikely to be entirely free of minor bugs. Also, as time evolves, we intend to enhance the package with more functionalities and more efficient algorithms or more efficient implementations thereof. The function `todo()` of **spam** sheds some insights into intended future directions.

We have motivated the need for **spam** and illustrated this paper with MCMC methods for GMRF. However, there are many other statistical tools that profit from the functionalities of **spam**, as outlined in the motivation, and many of them involve covariance matrices. Naturally, any sparse covariance matrix calls for the use of **spam**. Sparse covariance matrices arise from compactly supported covariance functions or from tapering (direct multiplication of a covariance function with a compactly supported one), cf. Furrer, Genton, and Nychka (2006). The R package **fields** (Furrer, Nychka, and Sain 2009), providing tools for spatial data, uses **spam** as a required package.

In contrast to the precision matrix of GMRF, the range parameter of the covariance function, which is directly related to the support, is often of interest and within an MCMC framework would be sampled as well. Changing the range changes the sparsity structure of the corresponding matrix and reusing the first steps in the factorization is not possible. However, often an upper bound of the range is known and a sparsity structure using this upper bound can be constructed. During individual factorizations, the covariance matrix is filled according to this structure and not according to the actual support of the covariance matrix.

The illustration of this paper have been done with **spam** 0.22-0 available from `http://www.math.uzh.ch/furrer/software/spam/`, where the R code is distributed under the GNU Public License and the file `LICENCE` contains the details of the license agreement for the Fortran code. Sources, binaries and documentation of **spam** are also available for download from the Comprehensive R Archive Network `http://CRAN.R-project.org/package=spam`. Once installed, the figures and tables of this article can be reproduced using `demo("article-jss")`, `demo("article-jss-example1")` and `demo("article-jss-example2")`.

6.1. spam and other sparse matrix R packages

spam is not the only R package for sparse matrix algebra. The packages **SparseM** (Koenker and Ng 2010) and **Matrix** (Bates and Maechler 2010) contain similar functionalities for handling sparse matrices, however, recall that both packages do not provide the possibility to split up the Cholesky factorization as discussed in this paper. We briefly discuss the major differences with respect to **spam**; for a detailed description see their manual.

SparseM is also based on the Fortran Cholesky factorization of Ng and Peyton (1993a) using the MMD permutation and almost exclusively on **SPARSKIT**. It was originally designed for large least squares problems and later also ported to S4 but is in a few cases inconsistent with existing R methods. It supports different sparse storage systems. Hence, besides wrapping issues and minor Fortran optimization its computational performance is comparable to **spam**.

Matrix incorporates many classes for sparse and full matrices and is based on C. For sparse matrices, it uses different storage formats, defines classes for different types of matrices and uses a Cholesky factorization based on **UMFPACK** (Davis 2004).

It would also be interesting to compare **spam** and the sparse matrix routines of MATLAB, The MathWorks, Inc. (2007) (see Figure 6 of Furrer *et al.* 2006 for a comparison between **SparseM** and MATLAB).

6.2. More hints for efficient computation

In many settings, having a fast Cholesky factorization routine is essential but not sufficient. Compared with other sparse matrix packages, **spam** is very competitive with respect to sparse matrix operations. However, given the row-oriented storage scheme, some operations are inherently slow and should be used carefully. Of course, a storage format based on a column oriented scheme does not solve the problem and there is no clear advantage of one over the other (Saad 1994). In this section we give a few examples of slow operations and mention a few tips for more efficient computation.

The mentioned inefficiency is often a result of not being able to access individual elements of a matrix directly. For example, if $\mathbf{A}$ is a sparse matrix in **spam**, we do not have direct memory access to an arbitrary element a_{ij}, but we need to search within the individual elements of the ith line, until we have reached the jth element or the position where it should be (because of the ordered column indices).

Similarly, it is much more efficient to access entire rows instead of columns. Hence, one should never subset a column of a symmetric matrix but using rows instead. Likewise, an inner product should always be calculated with $\mathbf{x}^\top(\mathbf{A}\mathbf{x}^\top)$ instead of $(\mathbf{x}^\top\mathbf{A})\mathbf{x}^\top$, the latter being equivalent to omitting the parentheses.

Finally, if `A` is a square matrix and `D` is a diagonal matrix of the same dimension, `A <- D %*% (A %*% D)` is be optimized as follows.

```
R> A@entries <- A@entries * D@entries[A@colindices] *
+    D@entries[rep_int(1:n, diff(A@rowpointers))]
```

If all R code optimization is still insufficient to enable the envisioned statistical analysis, as a last resort, there is always the possibility to implement larger blocks in Fortran or C directly.

Acknowledgements

The idea of writing a new sparse package for R was initiated by the many discussions with Steve Sain and Doug Nychka while the first author was a Postdoctoral visitor at the National Center for Atmospheric Research. The research of the first author was supported in part by National Science Foundation grant DMS-0621118. The research of the second author was supported by National Science Foundation grants ATM-0534173 and DMS-0707069. The National Center for Atmospheric Research is managed by the University Corporation for Atmospheric Research under the sponsorship of the National Science Foundation.

References

Bates D, Maechler M (2010). ***Matrix**: Sparse and Dense Matrix Classes and Methods*. R package version 0.999375-42, URL http://CRAN.R-project.org/package=Matrix.

Becker RA, Wilks AR, Brownrigg R, Minka TP (2010). ***maps**: Draw Geographical Maps*. R package version 2.1-4, URL http://CRAN.R-project.org/package=maps.

Besag J (1974). "Spatial Interaction and the Statistical Analysis of Lattice Systems." *Journal of the Royal Statistical Society B*, **36**(2), 192–225.

Besag J, York J, Mollié A (1991). "Bayesian Image Restoration, with Two Applications in Spatial Statistics." *Annals of the Institute of Statistical Mathematics*, **43**, 1–59.

Chambers JM (1998). *Programming with Data: A Guide to the S Language*. Springer-Verlag, Secaucus, NJ, USA.

Davis TA (2004). "Algorithm 832: **UMFPACK** V4.3—An Unsymmetric-Pattern Multifrontal Method." *ACM Transactions on Mathematical Software*, **30**(2), 196–199. doi:10.1145/992200.992206.

Duff IS, Erisman AM, Reid JK (1986). *Direct Methods for Sparse Matrices*. Oxford University Press, New York, NY, USA.

Furrer R, Genton MG, Nychka D (2006). "Covariance Tapering for Interpolation of Large Spatial Datasets." *Journal of Computational and Graphical Statistics*, **15**(3), 502–523.

Furrer R, Nychka D, Sain S (2009). ***fields**: Tools for Spatial Data*. R package version 6.01, URL http://CRAN.R-project.org/package=fields.

George A, Liu JWH (1981). *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall Inc., Englewood Cliffs, N. J. Prentice-Hall Series in Computational Mathematics.

George JA (1971). *Computer Implementation of the Finite Element Method*. Ph.D. thesis, Stanford University, Stanford, CA, USA.

Gilbert JR, Ng EG, Peyton BW (1994). "An Efficient Algorithm to Compute Row and Column Counts for Sparse Cholesky Factorization." *SIAM Journal on Matrix Analysis and Applications*, **15**(4), 1075–1091. doi:10.1137/S0895479892236921.

Gould NIM, Hu Y, Scott JA (2005a). *Complete Results for a Numerical Evaluation of Sparse Direct Solvers for the Solution of Large, Sparse, Symmetric Linear Systems of Equations*. Numerical Analysis Internal Report 2005-1 (revision 2). Rutherford Appleton Laboratory. Available from http://www.numerical.rl.ac.uk/reports/reports.shtml.

Gould NIM, Hu Y, Scott JA (2005b). "A Numerical Evaluation of Sparse Direct Symmetric Solvers for the Solution of Large Sparse, Symmetric Linear Systems of Equations." *Technical report*, RAL-TR-2005-005. Rutherford Appleton Laboratory. Available from http://www.numerical.rl.ac.uk/reports/reports.shtml.

Held L, Natario I, Fenton S, Rue H, Becker N (2005). "Towards Joint Disease Mapping." *Statistical Methods in Medical Research*, **14**(1), 61–82.

Ihaka R, Gentleman R (1996). "R: A Language for Data Analysis and Graphics." *Journal of Computational and Graphical Statistics*, **5**(3), 299–314.

Knorr-Held L, Raßer G (2000). "Bayesian Detection of Clusters and Discontinuities in Disease Maps." *Biometrics*, **56**(1), 13–21.

Knorr-Held L, Rue H (2002). "On Block Updating in Markov Random Models for Disease Mapping." *Scandinavian Journal of Statistics*, **29**(4), 597–614.

Koenker R, Ng P (2010). ***SparseM**: Sparse Linear Algebra*. R package version 0.85, URL http://CRAN.R-project.org/package=SparseM.

Liu JWH (1985). "Modification of the Minimum-Degree Algorithm by Multiple Elimination." *ACM Transactions on Mathematical Software*, **11**(2), 141–153. doi:10.1145/214392.214398.

Liu JWH (1992). "The Multifrontal Method for Sparse Matrix Solution: Theory and Practice." *SIAM Review*, **34**(1), 82–109. doi:10.1137/1034004.

Liu JWH, Ng EG, Peyton BW (1993). "On Finding Supernodes for Sparse Matrix Computations." *SIAM Journal on Matrix Analysis and Applications*, **14**(1), 242–252. doi:10.1137/0614019.

Lumley T (2004). "Programmers' Niche: A Simple Class, in S3 and S4." *R News*, **4**(1), 33–36. URL http://CRAN.R-project.org/doc/Rnews/.

Mollié A (1996). "Bayesian Mapping of Disease." In WR Gilks, S Richardson, DJ Spiegelhalter (eds.), *Markov Chain Monte Carlo in Practice*, pp. 359–379. Chapman & Hall, London.

Ng EG, Peyton BW (1993a). "Block Sparse Cholesky Algorithms on Advanced Uniprocessor Computers." *SIAM Journal on Scientific Computing*, **14**(5), 1034–1056. doi:10.1137/0914063.

Ng EG, Peyton BW (1993b). "A Supernodal Cholesky Factorization Algorithm for Shared-Memory Multiprocessors." *SIAM Journal on Scientific Computing*, **14**(4), 761–769. doi:10.1137/0914048.

R Development Core Team (2010a). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

R Development Core Team (2010b). *Writing R Extensions*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-11-9, URL http://www.R-project.org/.

Rue H, Held L (2005). *Gaussian Markov Random Fields: Theory and Applications*. Chapman & Hall, London.

Saad Y (1994). ***SPARSKIT**: A Basic Tool Kit for Sparse Matrix Computations*. Available at http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html.

Sherman AH (1975). *On the Efficient Solution of Sparse Systems of Linear and Nonlinear Equations*. Ph.D. thesis, Yale University, New Haven, CT, USA.

The MathWorks, Inc (2007). *MATLAB – The Language of Technical Computing, Version 7.5*. The MathWorks, Inc., Natick, Massachusetts. URL http://www.mathworks.com/products/matlab/.

Affiliation:

Reinhard Furrer
Institute of Mathematics
University of Zurich
CH-8057 Zurich, Switzerland
Email: reinhard.furrer@math.uzh.ch
URL: http://www.math.uzh.ch/furrer/

Stephan R. Sain
Geophysical Statistics Project
National Center for Atmospheric Research
Boulder, CO 80307-3000, United States of America
Email: ssain@ucar.edu
URL: http://www.image.ucar.edu/~ssain/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 36, Issue 10 *Submitted:* 2008-07-04
September 2010 *Accepted:* 2010-05-31