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

Reinhard Furrer
Mathematical and Computer Sciences
Colorado School of Mines
Golden, Colorado

Stephan R. Sain
Geophysical Statistics Project
National Center for Atmospheric Research
Boulder, Colorado

MCS-08-05

June 2008

Department of Mathematical and Computer Sciences
Colorado School of Mines
Golden, CO 80401-1887, USA
Phone: (303) 273-3860
Fax: (303) 273-3875
Email: rfurrer@mines.edu

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

Reinhard Furrer
Mathematical and Computer Sciences
Colorado School of Mines
Golden, Colorado
rfurrer@mines.edu

Stephan R. Sain
Geophysical Statistics Project
National Center for Atmospheric Research
Boulder, Colorado
ssain@ucar.edu

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 S4 and S3 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 sparseness structure. Simulations show that exploiting this break-down of the factorization results in a speed-up of about a factor 10 and memory savings of about a factor 15 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 models 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.

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 U.S. census divisions (counties, tracts, or

block-groups) or other administrative units. A powerful modeling tool for this type of data is the framework of Gaussian Markov random fields (GMRF), 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,jth element of $\mathbf{Q}$ is zero if the distribution at location i is conditionally independent of j given all location 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. The matrix $\mathbf{Q}$ usually features many zero elements and we call the pattern of zero and non-zero elements the sparseness structure. We also refer to the density of the matrix as the number of non-zeros over the total number of elements. Commonly, the conditional dependence structure in a GMRF is modeled using a parameter $\boldsymbol{\theta}$ and Markov chain Monte Carlo (MCMC) methods can be used to probe the posterior distribution of the parameters 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 sparseness 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 parameters 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 implementation 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 (2007), therefore spam has been conceived as a publicly available R package. 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. There are several remedies, however 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 sparseness structure, which is usually accomplished by solving $\mathbf{Ax} = \mathbf{b}$. We proceed by factorizing $\mathbf{A}$ into $\mathbf{R}^\mathsf{T}\mathbf{R}$, where $\mathbf{R}$ is an upper triangular matrix, called the Cholesky factor or Cholesky triangle of $\mathbf{A}$, followed by solving $\mathbf{R}^\mathsf{T}\mathbf{y} = \mathbf{b}$ 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., $\mathbf{U}^\mathsf{T}\mathbf{U} = \mathbf{P}^\mathsf{T}\mathbf{AP}$, with $\mathbf{U}$ an upper triangular matrix. There exist many different algorithms to find permutations which are optimal 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 bottle-neck in the computation of the log-likelihood of a Gaussian model), illustrating that it is very important to have a very efficient implementation (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 sparseness 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 $\mathbf{P}^\mathsf{T}\mathbf{AP}$
[2] Symbolic factorization, where the sparseness structure of $\mathbf{U}$ is constructed
[3] Numeric factorization, where the elements of $\mathbf{U}$ are computed

When factorizing matrices with the same sparseness 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 (also noticed by Rue and Held, 2005, page 51). However, none of the existing sparse matrix packages in R (SparseM, Matrix) provide the possibility to carry out Step 3 separately and spam fills this gap.

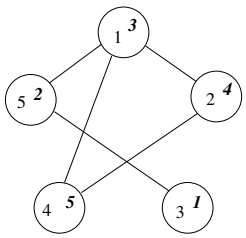
1.3 Outline

This article is structured as follows. The next section outlines in more detail the implementation 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 GMRF. Discussion and the positioning of spam and the Cholesky factorization in a larger framework are given in Section 5.

2 The Implementation of the Cholesky Factorization

In this section we discuss the individual steps and the actual implementation 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 *et al.* (1986) as general texts and to the more specific references cited below. spam uses a Fortran supernodal left-looking (constructing the factor row-wise) Cholesky factorization originally developed by E. Ng and B. Peyton at Oak Ridge National Laboratory in the early 1990s, see Ng and Peyton (1993b). The algorithm groups rows (via elimination trees, see Liu, 1992, for a definition) that share the same sparseness structure into supernodes, see Figure 1 and, e.g., Liu *et al.* (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 in SPARSPAK (George and Ng, 1981, 1984) by exploiting 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

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



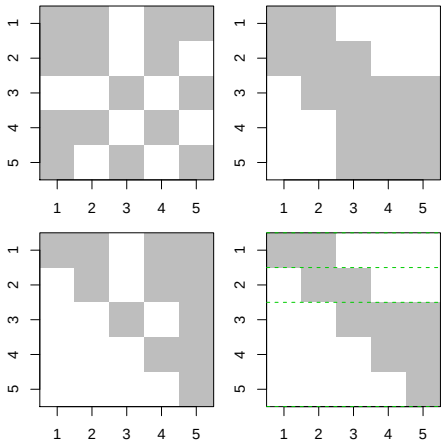


Figure 1: The symmetric positive definite $n = 5$ matrix $\mathbf{A}$ and the sparseness structure of $\mathbf{A}$ and $\mathbf{P}^\mathsf{T}\mathbf{AP}$ (top row). The graph associated to the matrix $\mathbf{A}$ and the Cholesky factors $\mathbf{R}$ and $\mathbf{U}$ of $\mathbf{A}$ and $\mathbf{P}^\mathsf{T}\mathbf{AP}$ respectively are given in the bottom row. The nodes of the graph are labeled according to $\mathbf{A}$ (upright) and $\mathbf{P}^\mathsf{T}\mathbf{AP}$ (italics). The dashed lines in $\mathbf{U}$ indicate the supernode partition, see Section 2 and 3.2.

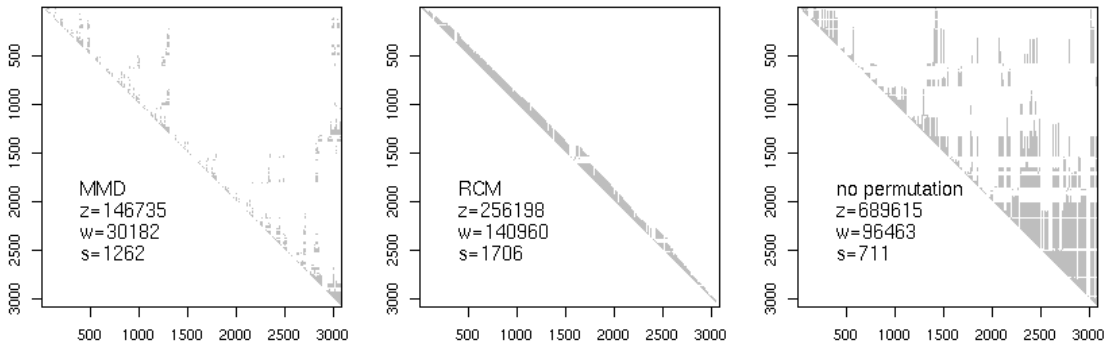


Figure 2: Sparseness structure of the Cholesky factor with MMD, RCM and no permutation of a precision matrix induced by a second order neighbor structure of the US counties. The values z, w are the sizes of the sparseness structure and of the vector containing the column indices of the sparseness structure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

processor instructions.

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] Initialization of the adjacency matrix data structure
[1] Determine permutation and permute the matrix
[2] Symbolic factorization
  [2a] Initialize and 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 sparseness structure in the permuted matrix determines the sparseness structure of the Cholesky factor. As an illustration, Figure 2 shows the sparseness 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 sparseness structure and of the vector containing the column indices of the sparseness 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 sparseness 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 for the stack (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 sparseness structure of the factor using the results of Gilbert *et al.* (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 implementation 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 (1993b) and to Gould *et al.* (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 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 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 SparseKit, a basic tool-kit for sparse matrix computations (Saad, 1994), some subroutines are optimized and tailored functions from SparseKit and a last set consists of new functions.

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, for example, changing only the contents of a matrix while maintaining its sparseness structure, see Section 5.2. 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`, `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 in case of 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`.

The group generic functions from `Math`, `Math2` and `Summary` are treated particularly in spam since they operate only on the nonzero entries. For example, for the matrix `A` presented in the introduction `range(A)` is the vector `c(0.5, 1)`, i.e. the zeros are omitted from the calculation.

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 details. 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, they correspond to the four elements of the CSR format, and are named

```
> 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`.

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, the third and forth group contain relevant information relating to the factorization algorithm and auxiliary information:

```
> 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 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, 1993a) 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:

```
> U@colindices
[1] 1 2 2 3 3 4 5
> U@colpointers
[1]  1  3  5  8
> 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.

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 and an irregular lattice, namely the counties of the contiguous USA. The county boundaries we use are from the `maps` package (Minka, 2006) providing

3082 counties. We consider that two counties are neighbors if they share at least one edge of their polygon description in maps.

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

```
> Rprof( memory.profiling=TRUE, interval = 0.0001)
> ressystime <- system.time(    expression    )
> Rprof( NULL)
> 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` helps (at least partially) to circumvent the issues in precisely measuring the memory amount with `Rprof`, see http://cran.r-project.org/doc/manuals/R-exts.html#Profiling-R-code-for-memory-use. 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.

The simulations are done with spam-0.14-0 and R-2.6.0 on an i486-pc-linux-gnu computer with a 1.73 GHz Centrino processor and 1 Gbyte of RAM.

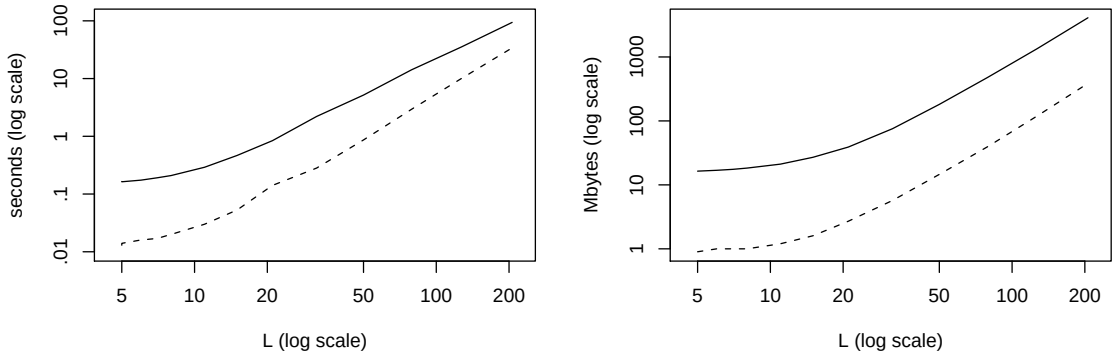


Figure 3: Total time (left) and memory usage (right) for 101 Cholesky factorizations (solid) and one factorization and 100 updates (dashed) of a precision matrix from different sizes L of regular $L \times L$ grids with a second order neighbor structure. The precision matrix from $L = 200$ has $L^4 =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

We first compare the total time and the memory required for Cholesky factorizations for different sizes of regular grids. In our MCMC framework, the sparseness 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 sparseness 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.

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.

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 6.2 seconds, 174.5 Mbytes and 15.1 seconds, 416.6 Mbytes, respectively.

	Regular grid		US counties	
Options or arguments	time	memory	time	memory
Using the specific call `chol.spam`	1.001	0.992	0.954	1.004
Option `safemode=c(FALSE,FALSE,FALSE)`	0.961	1.002	0.988	0.997
Option `cholsymmetrycheck=FALSE`	0.568	0.524	0.646	0.493
Passing `memory=list(nnzR=..., nnzcolindices=...)` to `chol`	0.969	0.979	0.928	0.972
All of the above	0.561	0.508	0.618	0.490
All of the above and passing `pivot=...` to `chol.spam`	0.542	0.528	0.572	0.496
All of the above and option `cholpivotcheck=FALSE`	0.510	0.511	0.557	0.489
Numeric update only using `update`	0.132	0.070	0.170	0.063

The size of the Cholesky factor is determined during the symbolic factorization (Step 2c) but we need to allocate vectors 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=...)`. The minimal sizes for a fixed sparseness 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 can be performed as follows.

```
> tmp <- summary(Qfact)    # get the sizes for the vector allocation
> pivot <- ordering(ch1)    # equivalent to ch1@pivot
> spam.options(cholsymmetrycheck=FALSE, safemode=c(FALSE,FALSE,FALSE),
+                 cholpivotcheck=FALSE)
> Qfactnew <- chol.spam(Qspamnew,  pivot=pivot,
+                         memory=list(nnzR=tmp$nnzR, nnzcolindices=tmp$nnzc))
+         # Qspamnew has the same sparseness structure as Qspam
```

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

```
> 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 factorization is only one part of each iteration and, additionally, the set of the valid parameters is often unknown. The first issue is addressed

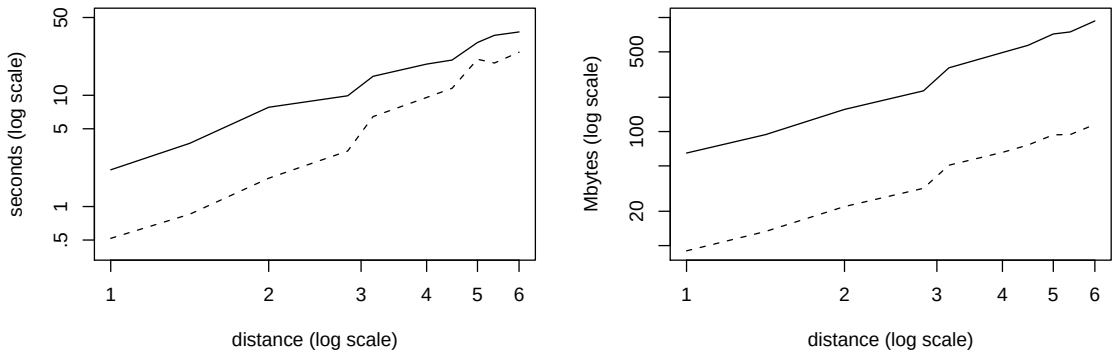


Figure 4: Total time (left) and memory usage (right) for 101 Cholesky factorizations (solid) and one factorization and 100 updates (dashed) of a precision matrix resulting from a regular 50×50 grid as a function of the distance for which grid points are considered as neighbors. For distance 6 each grid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

with competitive algorithms in spam but also needs to be considered when writing R code, see Section 5.2. 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. 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. 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.

5 Discussion

This paper is not a manual or tutorial for spam, since it only highlights some of its functionalities and details can be found in 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 *et al.* (2006). The R package fields (Nychka, 2007), 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 sparseness 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 sparseness 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-0.14-0 available from `http://www.mines.edu/~rfurrer/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. Once installed, the illustrations of this article can be reproduced using `demo("article-csda")`. Sources, binaries and documentation of spam are also available for download from the Comprehensive R Archive Network `http://cran.r-project.org/`.

5.1 spam and other Sparse Matrix R Packages

spam is not the only R package for sparse matrix algebra. The packages SparseM (Koenker and Ng, 2003) and Matrix (Bates and Maechler, 2006) 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 (1993b) using the MMD permutation and almost exclusively on SparseKit. 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 (see Figure 6 of Furrer *et al.*, 2006 for a comparison between SparseM and MATLAB).

5.2 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 $\mathbf{x}^\mathsf{T}(\mathbf{A}\mathbf{x}^\mathsf{T})$ instead of $(\mathbf{x}^\mathsf{T}\mathbf{A})\mathbf{x}^\mathsf{T}$, 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.

```
> 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, DMS-0355474 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. and Maechler, M. (2006). *Matrix: A Matrix package for R.* R package version 0.995-12. 10

Besag, J. (1974). Spatial interaction and the statistical analysis of lattice systems (with discussion). *Journal of the Royal Statistical Society, Series B*, **36**, 192–225. 2

Chambers, J. M. (1998). *Programming with Data: A Guide to the S Language.* Springer-Verlag New York, Inc., Secaucus, NJ, USA. 5

Davis, T. A. (2004). Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, **30**, 196–199. 10

Duff, I. S., Erisman, A. M., and Reid, J. K. (1986). *Direct methods for sparse matrices.* Oxford University Press, Inc., New York, NY, USA. 3

Furrer, R., Genton, M. G., and Nychka, D. (2006). Covariance tapering for interpolation of large spatial datasets. *Journal of Computational and Graphical Statistics*, **15**, 502–523. 9, 10

George, A. and Liu, J. W. H. (1981). *Computer solution of large sparse positive definite systems.* Prentice-Hall Inc., Englewood Cliffs, N. J. 3, 5, 6

George, A. and Ng, E. (1981). A brief description of sparspak waterloo sparse linear equations package. *ACM SIGNUM Newsletter*, **16**, 17–20. 3

George, A. and Ng, E. (1984). A new release of sparspak: the waterloo sparse matrix package. *ACM SIGNUM Newsletter*, **19**, 9–13. 3

George, J. A. (1971). *Computer implementation of the finite element method.* PhD thesis, Stanford University, Stanford, CA, USA. 4

Gilbert, J. R., Ng, E. G., and Peyton, B. W. (1994). An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, **15**, 1075–1091. 4

Gould, N. I. M., Hu, Y., and Scott, J. A. (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. 5

Gould, N. I. M., Hu, Y., and Scott, J. A. (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. 5

Ihaka, R. and Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, **5**, 299–314. 2

Koenker, R. and Ng, P. (2003). *SparseM: Sparse Matrix Package for R.* http://www.econ.uiuc.edu/~roger/research/sparse/SparseM.pdf. 10

Liu, J. W. H. (1985). Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software (TOMS)*, **11**, 141–153. 4

Liu, J. W. H. (1992). The multifrontal method for sparse matrix solution: theory and practice. *SIAM Review*, **34**, 82–109. 3, 5

Liu, J. W. H., Ng, E. G., and Peyton, B. W. (1993). On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, **14**, 242–252. 3

Lumley, T. (2004). Programmers' niche: A simple class, in S3 and S4. *R News*, **4**(1), 33–36. 5

Minka, T. P. (2006). *maps: Draw Geographical Maps*. Original S code by Richard A. Becker and Allan R. Wilks. R version by Ray Brownrigg Enhancements by Thomas P Minka. R package version 2.0-31. 6

Ng, E. and Peyton, B. W. (1993a). A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM Journal on Scientific Computing*, **14**, 761–769. 6

Ng, E. G. and Peyton, B. W. (1993b). Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, **14**, 1034–1056. 3, 5, 10

Nychka, D. (2007). *fields: Tools for spatial data*. R package version 4.1. http://www.image.ucar.edu/GSP/Software/Fields. 9

R Development Core Team (2007). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, http://www.R-project.org. 2

Rue, H. and Held, L. (2005). *Gaussian Markov Random Fields: Theory and Applications*. Chapman & Hall, London. 2, 3, 8, 9

Saad, Y. (1994). *SPARSEKIT: A basic tool kit for sparse matrix computations*. Available at http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html. 5, 10

Sherman, A. H. (1975). *On the efficient solution of sparse systems of linear and nonlinear equations*. PhD thesis, Yale University, New Haven, CT, USA. 6