Illustrations and Examples

Reinhard Furrer, Roman Flury

2026-05-28

Contents

1 Rational for spam

At the core of drawing multivariate normal random variables, calculating or maximizing multivariate normal log-likelihoods, calculating determinants, etc., we need to solve a large (huge) linear system involving a variance matrix, i.e., a symmetric, positive definite matrix.

Assume that we have such a symmetric, positive definite matrix $\boldsymbol{Q}$ of size $n \times n$ that contains many zeros (through tapering, Furrer et al. [2006], Furrer and Bengtsson [2007], or through a Markovian conditional construction). Typically, $\boldsymbol{Q}$ contains only $\mathcal{O}(n)$ non-zero elements compared to $\mathcal{O}(n^2)$ for a regular, **full** matrix. To take advantage of the few non-zero elements, special structures to represent the matrix are required, i.e., only the positions of the non-zeros and their values are kept in memory. Further, new algorithms work with these structures are required. The package `spam` provides this functionality, see Furrer and Sain [2010] for a detailed exposition.

2 A Simple Example

This first section illustrates with a simple example how to work with `spam`. Within a running R we install and load the current `spam` version from CRAN.

```
install.packages("spam")


library("spam")
```

We create a trivial matrix and "coerce" it to a sparse matrix.

```
Fmat <- matrix(c(3, 0, 1, 0, 2, 0, 1, 0, 3), nrow = 3, ncol = 3)
Smat <- as.spam(Fmat)
```

spam is conceptualized such that for many operations, the user proceeds as with ordinary full matrices. For example:

```
Fmat
#>      [,1] [,2] [,3]
#> [1,]    3    0    1
#> [2,]    0    2    0
#> [3,]    1    0    3
Smat
#>      [,1] [,2] [,3]
#> [1,]    3    0    1
#> [2,]    0    2    0
#> [3,]    1    0    3
#> Class 'spam' (32-bit)
Smat %*% t(Smat)
#>      [,1] [,2] [,3]
#> [1,]   10    0    6
#> [2,]    0    4    0
#> [3,]    6    0   10
#> Class 'spam' (32-bit)
Fmat %*% t(Smat)
#>      [,1] [,2] [,3]
#> [1,]   10    0    6
#> [2,]    0    4    0
#> [3,]    6    0   10
#> Class 'spam' (32-bit)
```

Hence, the user should not be worried which objects are sparse matrices and which are not. Of course not all operations result in sparse objects again,

```
rep(1, 3) %*% Smat
#>      [,1] [,2] [,3]
#> [1,]    4    2    4
```

However, other operations yield to different results when applied to full or sparse matrices

```
range(Fmat)
#> [1] 0 3
```

```
range(Smat)
#> [1] 1 3
```

3 Creating Sparse Matrices

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 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 Representation describes the format in more details.

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, for example, changing only the contents of a matrix while maintaining its sparsity structure, see Section Tips. 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 Representation.

4 Displaying

As seen in Section A Simple Example, printing small matrices result in an expected output, i.e., the content of the matrix plus a line indicating the class of the object:

```
Smat
#>      [,1] [,2] [,3]
#> [1,]    3    0    1
#> [2,]    0    2    0
#> [3,]    1    0    3
```

```
#> Class 'spam' (32-bit)
```

For larger objects, not all the elements are printed. For example:

```
diag.spam(100)
#>   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
#>  [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
#>  [75] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
#> Class 'spam' (32-bit)
```

The size of the matrix when switching from the first printing format to the second is a `spam` option, see Section Options. Naturally, we can also use the `str` command which gives us further insight into the individual slots of the `spam` object:

```
str(Smat)
#> Formal class 'spam' [package "spam"] with 4 slots
#>   ..@ entries    : num [1:5] 3 1 2 1 3
#>   ..@ colindices : int [1:5] 1 3 2 1 3
#>   ..@ rowpointers: int [1:4] 1 3 4 6
#>   ..@ dimension  : int [1:2] 3 3
```

Alternatively, calling `summary` gives additional information of the matrix.

```
summary(Smat)
#> Matrix object of class 'spam' of dimension 3x3,
#>     with 5 (row-wise) nonzero elements.
#>     Density of the matrix is 55.6%.
#> Class 'spam' (32-bit)
```

`summary` itself prints on the standard output but also returns a list containing the number of non-zeros (`nnz`) and the density (`density`) (percentage of `nnz` over the total number of elements).

Quite often, it is interesting to look at the sparsity structure of a sparse matrix. This is implemented with the command `display`. Again, depending on the size, the structure is shown as proper rectangles or as points. Figure 1 is the result of the following code.

Additionally, compare Figures of this document. Depending on the `cex` value, the `image` may issue a warning, meaning that the dot-size is probably not optimal. In fact, visually the density of the matrix `Smat1` seems to be around some percentage whereas the actual density is 0.024.

The function `image` goes beyond the structure of the matrix by using a specified color scheme for the values. Labels can be added manually or with `image.plot` from the package `fields`.

5 Solving Linear Systems

To be more specific about one of `spam`'s main features, assume we need to calculate $\boldsymbol{A}^{-1}\boldsymbol{b}$ with $\boldsymbol{A}$ a symmetric positive definite matrix featuring some sparsity structure, which is usually accomplished by solving $\boldsymbol{Ax} = \boldsymbol{b}$. We proceed by factorizing $\boldsymbol{A}$ into $\boldsymbol{R}^T\boldsymbol{R}$, where $\boldsymbol{R}$ is an upper triangular matrix, called the Cholesky factor

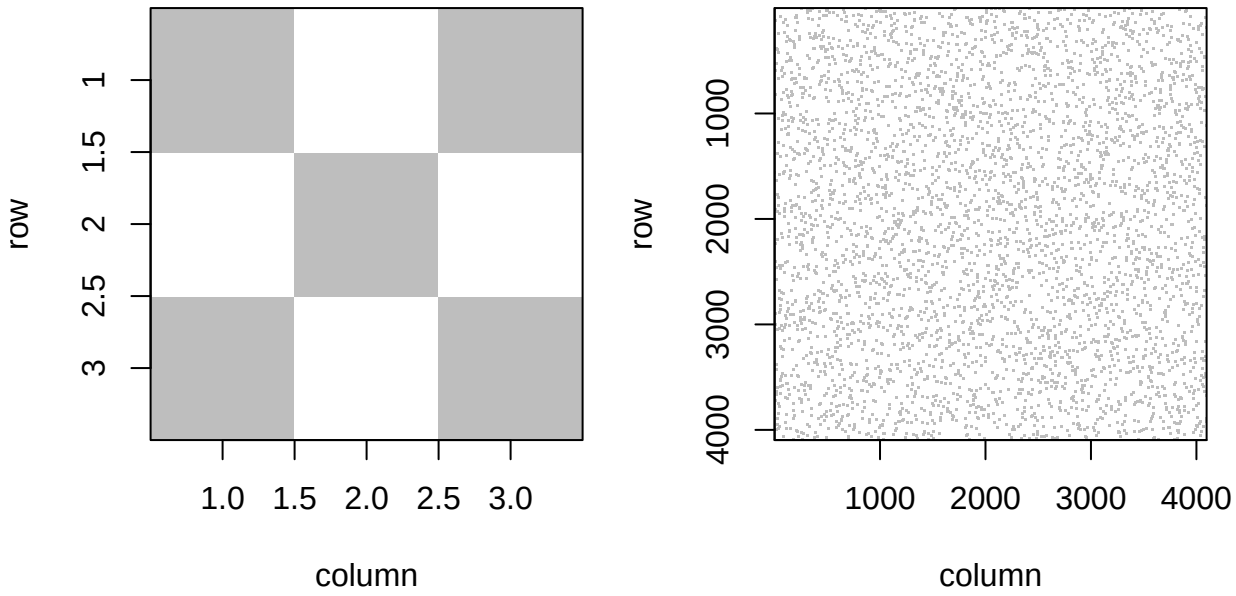

Figure 1: Sparsity structure of sparse matrices.

or Cholesky triangle of $\boldsymbol{A}$, followed by solving $\boldsymbol{R}^T\boldsymbol{y} = \boldsymbol{b}$ and $\boldsymbol{R}\boldsymbol{x} = \boldsymbol{y}$, called forwardsolve and backsolve, respectively. To reduce the fill-in of the Cholesky factor $\boldsymbol{R}$, we permute the columns and rows of $\boldsymbol{A}$ according to a (cleverly chosen) permutation $\boldsymbol{P}$, i.e., $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}$, with $\boldsymbol{U}$ an upper triangular matrix. There exist many different algorithms to find permutations which are optimal for specific matrices %tridiagonal matrices finite element/difference matrices defined on square grids or at least close to optimal with respect to different criteria. Note that $\boldsymbol{R}$ and $\boldsymbol{U}$ cannot be linked through $\boldsymbol{P}$ alone. The right panel of Figure 2 illustrates the factorization with and without permutation. For solving a linear system the two triangular solves are performed after the factorization. The determinant of $\boldsymbol{A}$ is the squared product of the diagonal elements of its Cholesky factor $\boldsymbol{R}$. 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 or using a "pixel" image of the zero/non-zero structure, see Figure 2.

```
i <- c(2, 4, 4, 5, 5)
j <- c(1, 1, 2, 1, 3)

A <- spam(0, nrow = 5, ncol = 5)
A[cbind(i, j)] <- rep(0.5, length(i))
A <- t(A) + A + diag.spam(5)
A
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]  1.0  0.5  0.0  0.5  0.5
#> [2,]  0.5  1.0  0.0  0.5  0.0
```

```
#> [3,]  0.0  0.0  1.0  0.0  0.5
#> [4,]  0.5  0.5  0.0  1.0  0.0
#> [5,]  0.5  0.0  0.5  0.0  1.0
#> Class 'spam' (32-bit)

U <- chol(x = A)
```

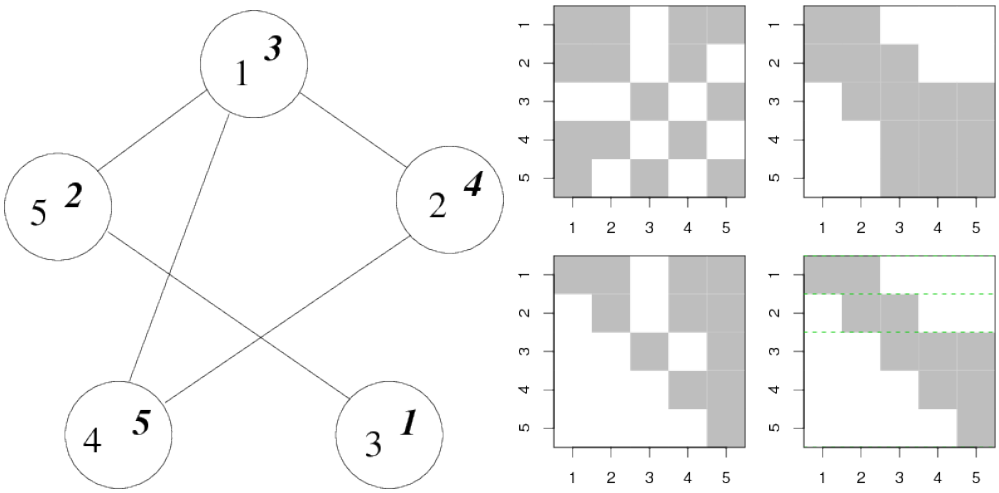

Figure 2: On the left side the associated graph to the matrix $\boldsymbol{A}$ is visualized. The nodes of the graph are labeled according to $\boldsymbol{A}$ (upright) and $\boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}$ (italics). On the right side the sparsity structure of $\boldsymbol{A}$ and $\boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}$ (top row) and the Cholesky factors $\boldsymbol{R}$ and $\boldsymbol{U}$ of $\boldsymbol{A}$ and $\boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}$ respectively are given in the bottom row. The dashed lines in $\boldsymbol{U}$ indicate the supernode partition.

The Cholesky factor of a banded matrix is again a banded matrix. But arbitrary matrices may produce full Cholesky factors. To reduce this so-called *fill-in* of the Cholesky factor $\boldsymbol{R}$, we permute the columns and rows of $\boldsymbol{A}$ according to a (cleverly chosen) permutation $\boldsymbol{P}$, i.e., $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}$, with $\boldsymbol{U}$ 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. The cost of finding a good permutation matrix $\boldsymbol{P}$ is at least of order $\mathcal{O}(n^{3/2})$. However, there exist good, but suboptimal, approaches for $\mathcal{O}(n\log(n))$.

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

Step	Description
[1]	Determine permutation and permute the input matrix $\boldsymbol{A}$ to obtain $\boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}$
[2]	Symbolic factorization, where the sparsity structure of $\boldsymbol{U}$ is constructed
[3]	Numeric factorization, where the elements of $\boldsymbol{U}$ are computed

When factorizing matrices with the same sparsity structure Step 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.

As for Step 1, there are many different algorithms to find a permutation, for example, the multiple minimum degree (MMD) algorithm, Liu [1985], and the reverse Cuthill-McKee (RCM) algorithm, George [1971]. The resulting sparsity structure in the permuted matrix determines the sparsity structure of the Cholesky factor. As an illustration, Figure 3 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.

How much fill-in with zeros is present depends on the permutation algorithm, in the example of Figure 3 there are 146735, 256198 and 689615 non-zero elements in the Cholesky factors resulting from the MMD, RCM, and no permutation, respectively. Note that the actual number of non-zero elements of the Cholesky factor may be smaller than what the constructed sparsity structure indicates. Here, there are 14111, 97565 and 398353 zero elements (up to machine precision) that are not exploited.

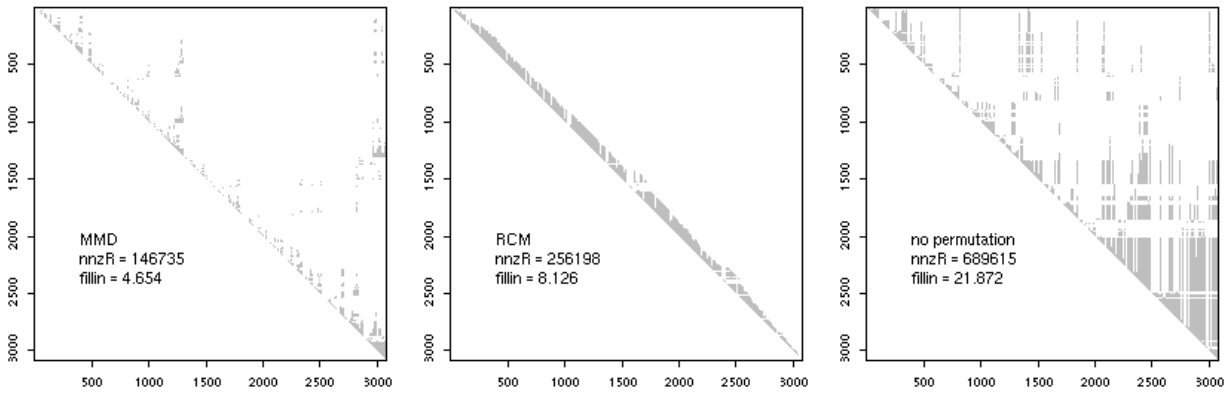

Figure 3: Sparsity structure of the Cholesky factor with MMD, RCM and no permutation of a precision matrix induced by a second-order neighbor structure of the US counties. The values *nnzR* and *fillin* are the number of non-zero elements in the sparsity structure of the factor and the fill-in, respectively.

6 More about Methods

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 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`.

6.1 Methods with Particular Behavior

For the `spam` class additional methods are defined, for examples `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)`, i.e. the zeros are omitted from the calculation. The help lists further available methods and highlights the (dis-)similarities compared to when applied to regular matrices or arrays.

6.2 Particular Methods with Ordinary Behavior

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.

6.3 Importing Foreign Formats

`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, both packages do not provide the possibility to split up the Cholesky factorization as discussed previously. 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 [1993] 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.

`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].

`spam` has a few functions that allow to transform matrix formats of the different packages.

`spam` also contains functions that download matrices from MatrixMarket, a web side that stores many different sparse matrices. The function `read.MM(file)`, very similar to the function `readMM` from `Matrix`, opens a connection, specified by the argument, and reads a matrix market file. However, as entries of `spam` matrices are of mode `double`, integers matrices are coerced to doubles, patterns lead to matrices containing ones and complex are coerced to the real part thereof. In these aforementioned cases, a warning is issued.

MatrixMarket also defines an array format, in which case a (possibly) dense `spam` object is return (retaining only elements which are larger than `getOption('spam.eps')`), a warning is issued.

Similarly to `read.MM(file)`, the function `read.HB(file)` reads matrices in the Harwell-Boeing format. Currently, only real assembled Harwell-Boeing can be read with `read.HB`. Reading MatrixMarket formats is more flexible.

The functions are based on `readHB` and `readMM` from the library `Matrix` to build the connection and read the raw data. At present, `read.MM(file)` is more flexible than `readMM`.

For many operations, `spam` is faster than `Matrix` and `SparseM`. 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).

Bibliography

Douglas Bates and Martin Maechler. *Matrix: A Matrix package for R*, 2006. R package version 0.995-12.

John M. Chambers. *Programming with Data: A Guide to the S Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998. ISBN 0387985034.

Timothy A. Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, 30(2):196–199, 2004. ISSN 0098-3500. doi: 10.1145/992200.992206.

R. Furrer and T. Bengtsson. Estimation of high-dimensional prior and posteriori covariance matrices in kalman filter variants. *J. Multivariate Anal.*, 98(2):227–255, 2007. doi: 10.1016/j.jmva.2006.08.003.

R. Furrer and S. R. Sain. spam: A sparse matrix R package with emphasis on MCMC methods for Gaussian Markov random fields. *Journal of Statistical Software*, 36(10):1–25, 2010. URL https://www.jstatsoft.org/v36/i10/.

R. Furrer, M. G. Genton, and D. Nychka. Covariance tapering for interpolation of large spatial datasets. *J. Comput. Graph. Statist.*, 15(3):502–523, 2006.

John Alan George. *Computer implementation of the finite element method*. PhD thesis, Stanford University, Stanford, CA, USA, 1971.

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

Joseph W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software (TOMS)*, 11(2):141–153, 1985. ISSN 0098-3500. doi: 10.1145/214392.214398.

Thomas Lumley. Programmers' niche: A simple class, in S3 and S4. *R News*, 4(1)(1):33–36, 2004.

Esmond G. Ng and Barry W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, 14(5):1034–1056, 1993. ISSN 1064-8275. doi: 10.1137/0914063.

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