sparseMVN: An R Package for Multivariate Normal Functions with Sparse Covariance and Precision Matrices

Michael Braun
Edwin L. Cox School of Business
Southern Methodist University

Abstract

The sparseMVN package exploits sparsity in covariance and precision matrices to speed up multivariate normal simpulation and density computation.

Keywords: multivariate normal, sparse matrices.

Simulation from, and calculating the density of, a multivariate normal (MVN) distribution linear algebra operations, such as multiplying a matrix with a vector or another matrix, decomposing a matrix into factors, and solving a system of linear equations. Without futher constraints on the covariance or precision matrix, the memory requirements and computational complexity of these component steps grow faster than the dimension of the MVN-distributed random variable. Because existing methods of working with the MVN distribution in R treat the covariance matrix as dense, these methods are not practical tools for working with high-dimensional problems.

However, in many cases the covariance or precision matrix is sparse, meaning that the proportion of non-zero elements is small, relative to the total size of the matrix. The **sparseMVN** package exploits that sparsity when computing the MVN density, and sampling MVN random variates. Matrices stored in a compressed sparse format consume less memory than dense matrices of the same dimension (the zeros are not stored explicitly), and operations like matrix-vector multiplication, solving linear systems, and computing Cholesky factors, can be performed more efficiently. Many linear algebra libraries, such as those called by the **Matrix** package, include routines that are optimized for sparse matrices. These routines are much faster than their dense matrix counterparts because they effectively skip over atomic operations that involve the zeros in the matrices.

1. Background

Let $x \in \mathbb{R}^k$ be a realization of random variable $X \sim \mathbf{MVN}(\mu, \Sigma)$, where $\mu \in \mathbb{R}^k$ is a vector, $\Sigma \in \mathbb{R}^{k \times k}$ is a positive-definite covariance matrix, and $\Sigma^{-1} \in \mathbb{R}^{k \times k}$ is a positive-definite precision matrix.

The log probability density of x is

$$\log f(x) = -\frac{k}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma| - \frac{1}{2}z^\top z, \quad \text{where } z^\top z = (x-\mu)^\top \Sigma^{-1}(x-\mu) \tag{1}$$

1.1. MVN density computation and random number generation

The two computationally intensive steps in evaluating $\log f(x)$ are computing $\log|\Sigma|$, and $z^\top z$, *without* explicitly inverting Σ or repeating mathematical operations. How one performs these steps *efficiently* in practice depends on whether the covariance matrix Σ, or the precision matrix Σ^{-1} is available. In both cases, we start by finding a lower triangular matrix root: $\Sigma = LL^\top$ or $\Sigma^{-1} = \Lambda\Lambda^\top$. Since Σ and Σ^{-1} are positive definite, we will use the Cholesky decomposition, which is the unique matrix root with all positive elements on the diagonal.

With the Cholesky decomposition in hand, we can compute the log determinant of Σ by adding the logs of the diagonal elements of the factors.

$$\log|\Sigma| = \begin{cases} 2\sum_{k=1}^K \log L_{kk} & \text{when } \Sigma \text{ is given} \\ -2\sum_{k=1}^K \log \Lambda_{kk} & \text{when } \Sigma^{-1} \text{ is given} \end{cases} \tag{2}$$

Having already computed the triangular matrix roots also speeds up the computation of $z^\top z$. If Σ^{-1} is given, $z = \Lambda^\top(x-\mu)$ can be computed efficiently as the product of an upper triangular matrix and a vector. When Σ is given, we find z by solving the lower triangular system $Lz = x - \mu$. The subsequent $z^\top z$ computation is trivially fast.

The algorithm for simulating from an $\mathbf{MVN}(\mu, \Sigma)$ distribution depends on whether Σ or Σ^{-1} is given. As above, we start by computing the Cholesky decomposition of the given matrix. Then, let $z \sim \mathbf{MVN}(0, I_k)$ be a vector of k samples from a standard normal distribution. If Σ is given, then $x = Lz + \mu$. If Σ^{-1} is given, then solve for x in the triangular linear system $\Lambda^\top(x-\mu) = z$. Because $\mathbf{E}(z) = 0$ and $\mathbf{cov}(z) = \mathbf{E}\left(zz^\top\right) = I_k$, x is a sample from $\mathbf{MVN}(\mu, \Sigma)$.

$$\mathbf{E}(x) = \mathbf{E}(Lz + \mu) = \mathbf{E}\left(\Lambda^\top z + \mu\right) = \mu \tag{3}$$

$$\mathbf{cov}(x) = \mathbf{cov}(Lz + \mu) = \mathbf{E}\left(Lzz^\top L^\top\right) = LL^\top = \Sigma \tag{4}$$

$$\mathbf{cov}(x) = \mathbf{cov}\left(\Lambda^{\top^{-1}} z + \mu\right) = \mathbf{E}\left(\Lambda^{\top^{-1}} zz^\top \Lambda^{-1}\right) = \Lambda^{\top^{-1}}\Lambda^{-1} = (\Lambda\Lambda^\top)^{-1} = \Sigma \tag{5}$$

These algorithms apply when the covariance/precision matrix is either sparse or dense. When the matrix is dense, the computational complexity is $\mathcal{O}\left(k^3\right)$ for a Cholesky decomposition, and $\mathcal{O}\left(k^2\right)$ for either solving the triangular linear system or multiplying a triangular matrix by another matrix (Golub and Van Loan 1996). Thus, the computational cost grows cubically with k before the decomposition step, and quadratically if the decomposition has already been completed. Additionally, the storage requirement for Σ grows quadratically with k.

1.2. Existing implementations

Most, if not all, computer implementations of MVN functions follow this approach, at least in part. We have found three R packages that provide MVN functions in some form; their

Package	Citation	dens	rand	sparse	prec	Chol	prob
mvtnorm	(Genz *et al.* 2012)	x	x				x
LaplacesDemon	(Statisticat and LLC. 2016)	x	x		x	x	
MASS	(Venables and Ripley 2002)		x				
sparseMVN	this package	x	x	x	x	x	

Table 1: Comparison of capabilities of R packages that implement MVN functions. dens: computes density. rand: generates random simulates. sparse: accepts matrices in sparse format. prec: option to provide precision matrix. Chol: option to provide a precomputed Cholesky decomposition. Prob: computes cumulative probabilities.

features are summarized in Table 1. The **mvtnorm** (Genz, Bretz, Miwa, Mi, Leisch, Scheipl, and Hothorn 2012) is likely the most popular, and it is the only one of the three that is "purpose built" for multivariate distributions. Its functionality is broader than **sparseMVN**, in that it computes cumulative probabilities and quantiles in additon to densities and random simulates, and supports the mulivariate t distribution as well as the MVN. Its MVN implementation is reasonably mature, so we will use **mvtnorm** as the baseline against which we will compare **sparseMVN** later in this paper. **LaplacesDemon** Statisticat and LLC. (2016) is a more general Bayesian estimation package that includes MVN functions. Unlike **mvtnorm**, but like **sparseMVN**, it has options for the user to provide a precision matrix instead of a covariance, and to provide a Cholesky decomposition instead of the full matrix. The **MASS** package (Venables and Ripley 2002) includes a simple function for MVN random number generation, but not a function to compute the MVN density. However, none of these package support covariance or precision matrices that are stored in a sparse compressed format, which we will describe next.

1.3. Sparse matrices in R

By "sparse matrix," we mean a matrix for which relatively few elements are non-zero. The **Matrix** package (Bates and Maechler 2015) defines various classes for storing sparse matrices in a compressed format, and the distinctions among them are beyond the scope of this paper. The most important one for our purposes is a *dsCMatrix*, which defines a symmetric matrix, with numeric (double precision) elements, in a column-compressed format. Three vectors define the underlying matrix: the unique nonzero values (just one triangle is needed), the indices in the value vector for the first value in each column, and the indices of the rows in which each value is located. Roughly speaking, the storage requirement for a sparse $k \times k$ symmetric matrix with z unique non-zero elements in one triangle is z double precision numbers, $z+k+1$ integers, and some metadata. In contrast, a dense representation of the same matrix would contain k^2 double precision values, regardless of symmetry and the number of zeros. The more sparse the matrix, the greater the memory consumption gains from a sparse matrix format.

An example

To illustrate how sparse matrices require less memory resources when compressed than when stored densely, consider the following example, which is borrowed heavily from the vignette of the **sparseHessianFD** package (Braun 2016).

Suppose we have a dataset of N households, each with T opportunities to purchase a particular product. Let y_i be the number of times household i purchases the product, out of the T purchase opportunities, and let p_i be the probability of purchase. The heterogeneous parameter p_i is the same for all T opportunities, so y_i is a binomial random variable.

Let $\beta_i \in \mathbb{R}^k$ be a heterogeneous coefficient vector that is specific to household i, such that $\beta_i = (\beta_{i1}, \ldots, \beta_{ik})$. Similarly, $z_i \in \mathbb{R}^k$ is a vector of household-specific covariates. Define each p_i such that the log odds of p_i is a linear function of β_i and z_i, but does not depend directly on β_j and z_j for another household $j \neq i$.

$$p_i = \frac{\exp(z_i'\beta_i)}{1 + \exp(z_i'\beta_i)},\ \ i = 1...N \tag{6}$$

The coefficient vectors β_i are distributed across the population of households following a multivariate normal distribution with mean $\mu \in \mathbb{R}^k$ and covariance $\mathbf{\Sigma} \in \mathbb{R}^{k\times k}$. Assume that we know $\mathbf{\Sigma}$, but not μ, so we place a multivariate normal prior on μ, with mean 0 and covariance $\mathbf{\Omega} \in \mathbb{R}^{k\times k}$. Thus, the parameter vector $x \in \mathbb{R}^{(N+1)k}$ consists of the Nk elements in the N β_i vectors, and the k elements in μ.

The log posterior density, ignoring any normalization constants, is

$$\log \pi(\beta_{1:N}, \mu | \mathbf{Y}, \mathbf{Z}, \mathbf{\Sigma}, \mathbf{\Omega}) = \sum_{i=1}^{N} \left(p_i^{y_i} (1-p_i)^{T-y_i} - \frac{1}{2} (\beta_i - \mu)^\top \mathbf{\Sigma}^{-1} (\beta_i - \mu) \right) - \frac{1}{2} \mu^\top \mathbf{\Omega}^{-1} \mu \tag{7}$$

Because one element of β_i can be correlated with another element of β_i (for the same unit), we allow for all of the cross-partials between elements of β_i for any i to be nonzero. Also, because the mean of each β_i depends on μ, the cross-partials between μ and any β_i could be nonzero. However, since the β_i and β_j are independently distributed, and the y_i are conditionally independent, the cross-partial derivatives between an element of β_i and any element of any β_j for $j \neq i$, must be zero. When N is much greater than k, there will be many more zero cross-partial derivatives than non-zero, and the Hessian of the log posterior density will be sparse.

The sparsity pattern depends on how the variables are ordered. One such ordering is to group all of the coefficients in the β_i for each unit together, and place μ at the end.

$$\beta_{11}, \ldots, \beta_{1k}, \beta_{21}, \ldots, \beta_{2k},\ \ldots\ , \beta_{N1}, \ldots, \beta_{Nk}, \mu_1, \ldots, \mu_k \tag{8}$$

In this case, the Hessian has a "block-arrow" structure. For example, if $N = 5$ and $k = 2$, then there are 12 total variables, and the Hessian will have the pattern in Figure 1.

When N is large, the proportion of non-zero elements in the Hessian is small, and thus the Hessian is sparse. For example, if $N = 5$, $k = 2$, and $p = 2$, then there are 12 total variables, and the Hessian will have the following "block-arrow" pattern.

There are 144 elements in this symmetric matrix. If the matrix is stored in the standard **base** R dense format, memory is reserved for all 144 values, even though only 64 values are non-zero, and only 38 values are unique. For larger matrices, the reduction in memory requirements

```
#
#  [1,] | | . . . . . . . . | |
#  [2,] | | . . . . . . . . | |
#  [3,] . . | | . . . . . . | |
#  [4,] . . | | . . . . . . | |
#  [5,] . . . . | | . . . . | |
#  [6,] . . . . | | . . . . | |
#  [7,] . . . . . . | | . . | |
#  [8,] . . . . . . | | . . | |
#  [9,] . . . . . . . . | | | |
# [10,] . . . . . . . . | | | |
# [11,] | | | | | | | | | | | |
# [12,] | | | | | | | | | | | |
```

Figure 1: A "block-arrow" sparsity pattern.

by storing the matrix in a sparse format can be substantial.[1]. If $N = 1000$, there are 2,002 variables in the problem, and more than 4 million elements in the Hessian. However, only $12,004$ of those elements are non-zero. If we work with only the lower triangle of the Hessian we only need to work with only 7,003 unique values. The dense matrix requires 30.6 Mb of RAM, while a sparse symmetric matrix of the *dsCMatrix* class requires only 91.5 Kb.

This example is relevant because, when evaluated at the posterior mode, the Hessian is the precision matrix of a MVN approximatation to the posterior distribution of $(\beta_{1:N}, \mu)$. If we were to simulate from this MVN using the **mvtnorm** package, we would first need to invert the dense Σ^{-1} matrix to get the covariance matrix. Each call to `rmvnorm` would involve a new decomposition of Σ, which is wasteful if Σ has not changed from the previous call. Using **sparseMVN**, Σ^{-1} is stored in a sparse compressed format, does not need to be inverted explicitly, and can be factorized only once in advance of repeated calls to the relevant functions.

2. Using the sparseMVN package

The `rmvn.sparse` generates random simulates for an MVN distribution, and `dmvn.sparse` computes the MVN log density. The signatures are

```r
rmvn.sparse(ndraws, mu, CH, prec=TRUE, log=TRUE)
dmvn.sparse(x, mu, CH, prec=TRUE, log=TRUE)
```

The argument `x` is a matrix with each MVN sample in a row. `rmvn.sparse` returns a matrix x with `ndraws` rows and `length(mu)` columns. `dmvn.sparse` returns a vector of length `ndraws`: densities if `log=FALSE`, and log densities if `log=TRUE`. `mu` is a numeric vector.

`CH` is either a *dCHMsimpl* or *dCHMsuper* object, and is computed using the funcCholesky function in the **Matrix** package. The `prec` argument identifies if `CH` is the decomposition of

[1]Because sparse matrix structures store row and column indices of the non-zero values, they may use more memory than dense storage if the total number of elements is small

either the covariance (Σ, `prec=FALSE`) or precision (Σ^{-1}, `prec=TRUE`) matrix. These functions do require the user to compute the Cholesky decomposition beforehand, but this needs to be done only once (as long as Σ or Σ^{-1} does not change).

More details about the `Cholesky` function are available in the documentation to the **Matrix** package, but it is a simple function to use. The first argument is a sparse symmetric Matrix stored as a *dsCMatrix* object. As far as we know, there is no particular need to deviate from the defaults of the remaining arguments. If `Cholesky` uses a fill-reducing permutation to compute `CH`, the functions in **sparseMVN** will handle that directly, with no additional user intervention required.

Do not use the `chol` function in **base** R. `Cholesky` is designed for decomposing *sparse* matrices, which involves permuting rows and columns to maintain sparsity of the factor. The *dCHM-simple* and *dCHMsuper* objects store this permutation, which `rmvn.sparse` and `dmvn.sparse` need.

2.1. An example

Suppose we want to generate samples from an MVN approximation to the posterior distribution of our example model. The package includes functions to simulate data for the example, and return the log posterior density, gradient and Hessian. The `trust.optim` function in the **trustOptim** package (Braun 2014) is a nonlinear optimizer that accepts a sparse Hessian to accelerate convergence.

```
D <- binary.sim(N=50, k=2, T=50)
priors <- list(inv.Sigma=diag(2), inv.Omega=diag(2))
start <- rep(c(-1,1),51)
opt <- trust.optim(start,
                   fn=sparseMVN::binary.f,
                   gr=sparseMVN::binary.grad,
                   hs=sparseMVN::binary.hess,
                   data=D, priors=list(inv.Sigma=diag(2),
                                       inv.Omega=diag(2)),
                   method="Sparse",
                   control=list(function.scale.factor=-1))
```

The posterior mode, and the Hessian evaluated at that point, are returned by `trust.optim`. They serve as the mean and the negative precision of the MVN approximation to the posterior distribution of the model.

```
R <- 100
pm <- opt[["solution"]]
H <- -opt[["hessian"]]
CH <- Cholesky(H)
samples <- rmvn.sparse(R, pm, CH, prec=TRUE)
```

We can then compute the MVN log density for each sample.

```
logf <- dmvn.sparse(samples, pm, CH, prec=TRUE)
```

The ability to accept a precision matrix, rather than having to invert it to a covariance matrix, is a valuable feature of **sparseMVN**. This is because the inverse of a sparse matrix is not necessarily sparse. In the following chunk, we invert the Hessian, and drop zero values to maintain any remaining sparseness. Note that there are 10,404 total elements in the Hessian.

```
Matrix::nnzero(H)

# [1] 402

Hinv <- drop0(solve(H))
Matrix::nnzero(Hinv)

# [1] 10404
```

Nevertheless, we should check that the log densities from `dmvn.sparse` correspond to those that we would get from `dmvnorm`.

```
logf_dense <- dmvnorm(samples, pm, as.matrix(Hinv), log=TRUE)
all.equal(logf, logf_dense)

# [1] TRUE
```

3. Timing

In this section we compare the run times between `rmvn.sparse` and `rmvnorm`, and between `dmvn.sparse` and `dmvnorm`. For each case, we construct a matrix with a block-arrow structure as in Figure 1. Cases vary by the number of blocks in the diagonal (analogous to the number of heterogeneous units in the binary choice example), the number of random samples being selected, and whether the underlying matrix is a covariance or precision.

```
load("runtimes.Rdata")
runtimes <- ungroup(runtimes)
TM <- filter(runtimes, stringr::str_detect(step, "[rd]_")) %>%
    select(-p,-nels,-nnz) %>%
    tidyr::gather(stat,value,c(mean,sd)) %>%
    reshape2::dcast(s+N+k+prec+nvars+nnzLT+pct.nnz~step+stat)
```

When a covariance matrix

When a precision matrix

The times were generated from the code in jssvignettes/replication.R.

						random				density			
					pct	sparse		dense		sparse		dense	
R	N	k	vars	nnz	nnz	mean	sd	mean	sd	mean	sd	mean	sd
20	5	3	18	81	0.44	1.38	0.29	1.64	0.15	0.32	0.22	0.18	0.04
20	10	3	33	156	0.26	1.44	0.42	1.86	0.24	0.32	0.07	0.25	0.08
20	15	3	48	231	0.18	1.38	0.14	1.90	0.16	0.47	0.12	0.28	0.06
100	5	3	18	81	0.44	1.29	0.11	1.85	0.41	0.37	0.03	0.20	0.04
100	10	3	33	156	0.26	1.63	0.17	1.82	0.14	0.64	0.07	0.28	0.03
100	15	3	48	231	0.18	1.91	0.59	1.90	0.17	0.96	0.16	0.42	0.07

Table 2: Time (milliseconds) to simulate R MVN samples, or compute R MVN densities, using sparse (`rmvn.sparse`, `dmvn.sparse`) or dense (`rmvnorm`, `dmvnorm`) methods, for a covariance matrix with a block-arrow structure. N is number of blocks, k is the size of the block, nnz is the number of non-zeros in the lower triangle of the matrix, and pct.nnz is the percentage of elements that are nonzero (a measure of sparsity). Means and standard deviations are across XX replications.

						random				density			
					pct	sparse		dense		sparse		dense	
R	N	k	vars	nnz	nnz	mean	sd	mean	sd	mean	sd	mean	sd
20	5	3	18	81	0.44	1.17	0.11	1.62	0.05	0.24	0.02	0.15	0.02
20	10	3	33	156	0.26	1.42	0.25	1.89	0.24	0.37	0.06	0.23	0.05
20	15	3	48	231	0.18	1.46	0.21	1.93	0.17	0.52	0.11	0.40	0.09
100	5	3	18	81	0.44	1.33	0.11	1.66	0.06	0.39	0.03	0.18	0.03
100	10	3	33	156	0.26	1.67	0.38	1.75	0.28	0.70	0.14	0.31	0.06
100	15	3	48	231	0.18	1.86	0.21	1.98	0.18	1.00	0.11	0.57	0.42

Table 3: Time (milliseconds) to simulate R MVN samples, or compute R MVN densities, using sparse (`rmvn.sparse`, `dmvn.sparse`) or dense (`rmvnorm`, `dmvnorm`) methods, for a precision matrix with a block-arrow structure. See caption to Table 2 for definitions of columns.

4. Other packages for creating and using sparse matrices

4.1. sparseHessianFD

Suppose you have a objective function that has a sparse Hessian (e.g., the log posterior density for a hierarchical model). You have an R function that computes the value of the objective, and another function that computes its gradient. You may also need the Hessian, either for a nonlinear optimization routine, or as the negative precision matrix of an MVN approximation.

It's hard enough to get the gradient, but the derivation of the Hessian might be too tedious or complicated for it to be worthwhile. However, it should not be too hard to identify which elements of the Hessian are non-zero. If you have both the gradient, and the Hessian emphpattern, then you can use the **sparseHessianFD** package (Braun 2016) to estimate the Hessian itself.

The **sparseHessianFD** package estimates the Hessian numerically, but in a way that exploits the fact that the Hessian is sparse, and that the pattern is known. The package contains functions that return the Hessian as a sparse *dgCMatrix*. This object can be coerced into a *dsCMatrix*, which in turn can be used by `rmvn.sparse` and `dmvn.sparse`.

4.2. trustOptim

The **trustOptim** package provides a nonlinear optimization routine that takes the Hessian as a sparse *dgCMatrix* object. This optimizer is useful for unconstrained optimization of a high-dimensional objective function with a sparse Hessian. It uses a trust region algorithm, which may be more stable and robust than line search approaches. Also, it applies a stopping rule based on the norm of the gradient, as opposed to whether the algorithm makes "sufficient progress." (Many optimizers, especially `optim` in **base** R, stop too early, before the gradient is truly flat).

5. Other

Since a large proportion of elements in the matrix are zero, we need to store only the row and column indices, and the values, of the unique non-zero elements. The efficiency gains in **sparseMVN** come from storing the covariance or precision matrix in a compressed format without explicit zeros, and applying linear algebra routines that are optimized for those sparse matrix structures. The **Matrix** package calls sparse linear algebra routines that are implemented in the **CHOLMOD** library (Chen, Davis, Hager, and Rajamanickam 2008; Davis and Hager 1999, 2009); more information about these routines is available there.

The exact amount of time and memory that are saved by saving the covariance/precision matrix in a sparse format depends on the sparsity pattern. But for the hierarchical model example from earlier in this section, the number of non-zero elements grows only linearly with N. The result is that all of the steps of sampling from an MVN also grow linearly with N. Section 4 of Braun and Damien (2016) explains why this is so.

1. Each call to the function involves a new matrix factorization step. This can be costly for applications in which x or μ changes from call to call, but Σ does not.

2. In some applications, the precision matrix Σ^{-1}, and not the covariance matrix Σ, is readily available (e.g., estimating the asymptotic covariance from the inverse of a Hessian of a maximum likelihood estimator). To use the **mvtnorm** functions, Σ^{-1} would first have to be inverted explicitly.

3. The **mvtnorm** functions treat Σ as if it were dense, even if there is a large proportion of structural zeros.

The **sparseMVN** package addresses these limitations.

1. The `rmvn.sparse` and `dmvn.sparse` functions take as their matrix argument a sparse Cholesky decomposition. The user does need to do this explicitly beforehand, but once it is done, it does not have to be done again.

2. Both functions include an argument to identify the sparse Cholesky decomposition as a factor of a covariance matrix (`prec=FALSE`) or precision matrix (`prec=TRUE`).

Even when either Σ or Σ^{-1} are dense, there may be advantages to using **sparseMVN** instead of **mvtnorm**. For example, if the user were starting with a large, dense precision matrix Σ^{-1}, and computing MVN densities repeatedly, it may take less time to compute the Cholesky decomposition of $\Sigma^{-1]}$ once, than to invert it and have the **mvtnorm** functions decompose Σ over and over. Nevertheless, the main purpose of **sparseMVN** is to exploit sparsity in either Σ or Σ^{-1} when it exists.

References

Bates D, Maechler M (2015). ***Matrix**: Sparse and Dense Matrix Classes and Methods.* R package version 12-4, URL `https://CRAN.R-project.org/package=Matrix`.

Braun M (2014). "trustOptim: An R Package for Trust Region Optimization with Sparse Hessians." *Journal of Statistical Software*, **60**(4), 1–16. URL `http://www.jstatsoft.org/v60/i04/`.

Braun M (2016). *sparseHessianFD: An R package for estimating sparse Hessians.* URL `https://cran.r-project.org/package=sparseHessianFD`.

Braun M, Damien P (2016). "Scalable Rejection Sampling for Bayesian Hierarchical Models." *Marketing Science*, **35**(3), 427–444. `doi:10.1287/mksc.2014.0901`.

Chen Y, Davis TA, Hager WW, Rajamanickam S (2008). "Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate." *ACM Transactions on Mathematical Software*, **35**(3), 1–14. `doi:10.1145/1391989.1391995`.

Davis TA, Hager WW (1999). "Modifying a Sparse Cholesky Factorization." *SIAM Journal on Matrix Analysis and Applications*, **20**(3), 606–627. `doi:10.1137/S0895479897321076`.

Davis TA, Hager WW (2009). "Dynamic Supernodes in Sparse Cholesky Update/Downdate and Triangular Solves." *ACM Transactions on Mathematical Software*, **35**(4), 1–23. `doi:10.1145/1462173.1462176`.

Genz A, Bretz F, Miwa T, Mi X, Leisch F, Scheipl F, Hothorn T (2012). *mvtnorm: Multivariate Normal and t Distributions.*

Golub GH, Van Loan CF (1996). *Matrix Computations.* 3rd edition. Johns Hopkins University Press.

Statisticat, LLC (2016). *LaplacesDemon: Complete Environment for Bayesian Inference.* R package version 16.0.1, URL `https://cran.r-project.org/package=LaplacesDemon`.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S.* Fourth edition. Springer-Verlag.

Affiliation:

Michael Braun
Edwin L. Cox School of Business
Southern Methodist University
6212 Bishop Blvd.
Dallas, TX 75275
E-mail: `braunm@smu.edu`
URL: `http://www.smu.edu/Cox/Departments/FacultyDirectory/BraunMichael`