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 number of elements in a multivariate normal (MVN) covariance or precision matrix grows quadratically in the number of variables. Thus, sampling from, and computing densities of, an MVN random variable is not scalable when all matrix elements are stored as a dense R matrix, but may be when the matrix is sparse, and stored in a suitable compressed format. This package provides standard MVN sampling and density algorithms that are optimized for sparse covariance and precision matrices.

Keywords: multivariate normal, sparse matrices, covariance, simulation.

The **mvtnorm** package (**?**) provides the `dmvnorm` function to compute the density of a multivariate normal (MVN) distributon, and the `rmvnorm` function to simulate MVN random variables. These functions require the user to supply a full, "dense" covariance matrix; if starting with a precision matrix, the user must first invert it explicitly. This covariance matrix is dense in the sense that, for an M-dimensional MVN random variable, all M^2 elements are stored, so memory requirements grow quadratically with the size of the problem. Internally, both functions factor the covariance matrix using a Cholesky decomposition, whose complexity is $\mathcal{O}(M^3)$(**?**).[1] This factorization is performed every time the function is called, even if the covariance matrix does not change from call to call. Also, `rmvnorm` involves multiplication of a triangular matrix, and `dmvnorm` involves solving a triangular linear system. Both of these operations are $\mathcal{O}(M^2)$ (**?**) on dense matrices. MVN functions in other packages, such as **MASS** (**?**) and **LaplacesDemon** (**?**), face similar limitations.[2] Thus, existing tools for working with the MVN distribution in R are not practical for high-dimensional MVN random variables.

However, for many applications the covariance or precision matrix is sparse, meaning that the proportion of nonzero elements is small, relative to the total size of the matrix. The functions in the **sparseMVN** package exploit that sparsity to reduce memory requirements, and to gain computational efficiencies. The `dmvn.sparse` function computes the MVN density, and the `rmvn.sparse` function samples from an MVN random variable. Instead of requiring the user to supply a dense covariance matrix, `dmvn.sparse` and `rmvn.sparse` accept a pre-computed Cholesky decomposition of either the covariance or precision matrix in a compressed sparse

[1] `dmvnorm` has options for eigen and singular value decompositions. These are both $\mathcal{O}(M^3)$ as well.

[2] **LaplacesDemon** does offer options for the user to supply pre-factored covariance and precision matrices. This avoids repeated calls to the $\mathcal{O}(M^3)$ factorization step, but not the $\mathcal{O}(M^2)$ matrix multiplication and linear system solution steps.

format. This approach has several advantages:

1. Memory requirements are smaller because only the nonzero elements of the matrix are stored in a compressed sparse format.
2. Linear algebra algorithms that are optimzed for sparse matrices are more efficient because they avoid operations on matrix elements that are known to be zero.
3. When the precision matrix is initially available, there is no need to invert it into a covariance matrix explicitly. This feature of **sparseMVN** preserves sparsity, because the inverse of a sparse matrix is not necessarily sparse.
4. The Cholesky factor of the matrix is computed once, before the first **sparseMVN** function call, and is not repeated with subsequent calls (as long as the matrix does not change).

The functions in **sparseMVN** rely on sparse matrix classes and functions defined in the **Matrix** package (**?**). The user creates the covariance or precision matrix as a sparse, symmetric *dsCMatrix* matrix, and computes the sparse Cholesky factor using the `Cholesky` function. Other than ensuring that the factor for the covariance or precision matrix is in the correct format, the **sparseMVN** functions behave in much the same way as the corresponding **mvtnorm** functions. Internally, **sparseMVN** uses standard methods of computing the MVN density and simulating MVN random variables (see Section 1.1). 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 nonzero 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 (**???**).

1. Background

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

The log probability density of x is

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

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. For 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 compute the log determinant of Σ by adding the logs of the diagonal elements of the factors.

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

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 $X \sim \mathbf{MVN}(\mu, \Sigma)$ also depends on whether Σ or Σ^{-1} is given. As above, we start by computing the Cholesky decomposition of the given covariance or precision matrix. Define a random variable $Z \sim \mathbf{MVN}(0, I_M)$, and generate a realization z as a vector of M samples from a standard normal distribution. If Σ is given, then evaluate $x = Lz + \mu$. If Σ^{-1} is given, then solve for x in the triangular linear system $\Lambda^\top(x-\mu) = z$. The resulting x is a sample from $\mathbf{MVN}(\mu, \Sigma)$. We confirm the mean and covariance of X as follows:

$$\mathbf{E}(X) = \mathbf{E}(LZ+\mu) = \mathbf{E}\Big(\Lambda^\top Z + \mu\Big) = \mu \tag{3}$$

$$\mathbf{cov}(X) = \mathbf{cov}(LZ+\mu) = \mathbf{E}\Big(LZZ^\top L^\top\Big) = LL^\top = \Sigma \tag{4}$$

$$\mathbf{cov}(X) = \mathbf{cov}\Big(\Lambda^{\top^{-1}} Z + \mu\Big) = \mathbf{E}\Big(\Lambda^{\top^{-1}} ZZ^\top \Lambda^{-1}\Big) = \Lambda^{\top^{-1}}\Lambda^{-1} = (\Lambda\Lambda^\top)^{-1} = \Sigma \tag{5}$$

These algorithms apply when the covariance/precision matrix is either sparse or dense. When the matrix is dense, the computational complexity is $\mathcal{O}\big(M^3\big)$ for a Cholesky decomposition, and $\mathcal{O}\big(M^2\big)$ for either solving the triangular linear system or multiplying a triangular matrix by another matrix (**?**). Thus, the computational cost grows cubically with M before the decomposition step, and quadratically if the decomposition has already been completed. Additionally, the storage requirement for Σ (or Σ^{-1}) grows quadratically with M.

1.2. Sparse matrices in R

The **Matrix** package (**?**) defines various classes for storing sparse matrices in compressed formats. The most important class for our purposes is *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. The storage requirements for a sparse $M \times M$ symmetric matrix with V unique nonzero elements in one triangle are for V double precision numbers, $V + M + 1$ integers, and some metadata. In contrast, a dense representation of the same matrix stores M^2 double precision values, regardless of symmetry and the number of zeros. If V grows more slowly than M^2, the matrix becomes increasingly sparse (a smaller percentage of elements are nonzero), with greater efficiency gains from storing the matrix in a compressed sparse format.

An example

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

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, $w_i \in \mathbb{R}^k$ 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 w_i, but does not depend directly on β_j and w_j for another household $j \neq i$.

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

The coefficient vectors β_i are distributed across the population of households following a MVN distribution with mean $\mu \in \mathbb{R}^k$ and covariance $\mathbf{A} \in \mathbb{R}^{k \times k}$. Assume that we know $\mathbf{A}$, 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

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

Because one element of β_i can be correlated with another element of β_i (for the same unit), we allow for 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 can be nonzero. However, since the β_i and β_j are independent samples, 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 nonzero, 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.

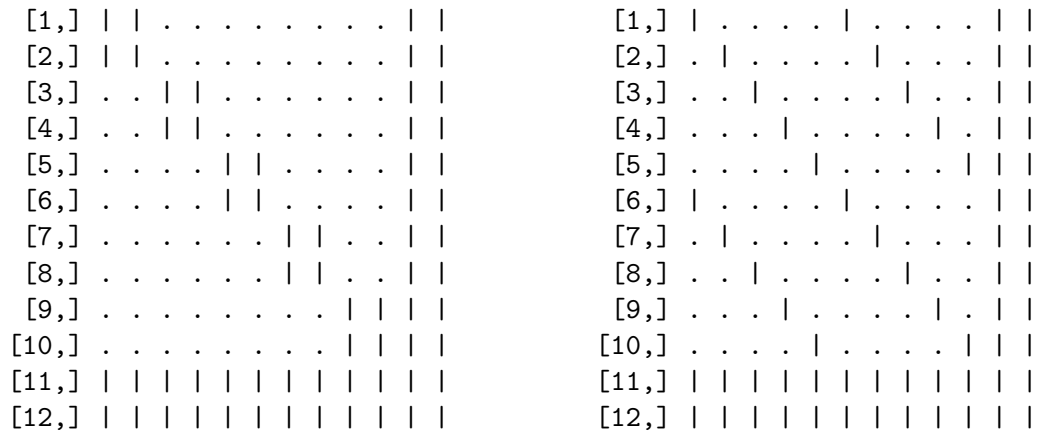
$$\beta_{11}, \ldots, \beta_{1k}, \beta_{21}, \ldots, \beta_{2k}, \;\; \ldots \;\; , \beta_{N1}, \ldots, \beta_{Nk}, \mu_1, \ldots, \mu_k \tag{8}$$

In this case, the Hessian has a "block-arrow" pattern. Figure 1a illustrates this pattern for $N = 5$ and $k = 2$ (12 total variables).

Another possibility is to group coefficients for each covariate together.

$$\beta_{11}, \ldots, \beta_{N1}, \beta_{12}, \ldots, \beta_{N2}, \;\; \ldots \;\; , \beta_{1k}, \ldots, \beta_{Nk}, \mu_1, \ldots, \mu_k \tag{9}$$

Now the Hessian has an "banded" sparsity pattern, as in Figure 1b.



(a) A "block-arrow" sparsity pattern.

(b) A "banded" sparsity pattern.

Figure 1: Two examples of sparsity patterns for a hierarchical model.

In both cases, the number of nonzeros is the same. 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 nonzero, and only 38 values are unique. For larger matrices, the reduction in memory requirements by storing the matrix in a sparse format can be substantial.[3]. If N =1,000, then M =2,002, with more than 4 million elements in the Hessian. However, only 12,004 of those elements are nonzero, with 7,003 unique values in the lower triangle. The dense matrix requires 30.6 Mb of RAM, while a sparse symmetric matrix of the *dsCMatrix* class requires only 91.6 Kb.

This example is relevant because, when evaluated at the posterior mode, the Hessian matrix of the log posterior is the MVN precision matrix Σ^{-1} of a MVN approximatation to the posterior distribution of $(\beta_{1:N}, \mu)$. If we were to simulate from this MVN using `rmvnorm`, or evaluate MVN densities using `dmvnorm`, we would need to invert Σ^{-1} to Σ first, and store it as a dense matrix. Internally, `rmvnorm` and `dmvnorm` invoke dense linear algebra routines, including matrix factorization.

2. Using the sparseMVN package

The signatures of the **sparseMVN** key sparse matrix functions are

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

The `rmvn.sparse` function returns a matrix x with `n` rows and `length(mu)` columns. `dmvn.sparse` returns a vector of length `n`: densities if `log=FALSE`, and log densities if `log=TRUE`.

Table 1 describes the function arguments. These functions do require the user to compute the Cholesky decomposition `CH` beforehand, but this needs to be done only once (as long

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

x	A numeric matrix. Each row is an MVN sample.
mu	A numeric vector. The mean of the MVN random variable.
CH	Either a *dCHMsimpl* or *dCHMsuper* object representing the Cholesky decomposition of the covariance/precision matrix.
prec	Logical value that identifies CH as the Cholesky decomposition of either a covariance (Σ, `prec=TRUE`) or precision(Σ^{-1}, `prec=FALSE`) matrix.
n	Number of random samples to be generated.
log	If `log=TRUE`, the log density is returned.

Table 1: Arguments to the `rmvn.sparse` and `dmvn.sparse` functions.

as Σ or Σ^{-1} does not change). `CH` should be computed using the `Cholesky` function from the **Matrix** package. More details about the `Cholesky` function are available in the **Matrix** documentation, 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 **sparseMVN** functions will handle that directly, with no additional user intervention required. The `chol` function in **base** R should not be used.

2.1. An example

Suppose we want to generate samples from an MVN approximation to the posterior distribution of our example model from Section 1.2. **sparseMVN** includes functions to simulate data for the example (`binary.sim`), and to compute the log posterior density (`binary.f`), gradient (`binary.grad`), and Hessian (`binary.hess`). The `trust.optim` function in the **trustOptim** package (**?**) is a nonlinear optimizer that estimates the curvature of the objective function using a sparse Hessian.

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

The call to `trust.optim` returns the posterior mode, and the Hessian evaluated at the mode. These results serve as the mean and the negative precision of the MVN approximation to the posterior distribution of the model.

```
R> R <- 100
R> pm <- opt[["solution"]]
```

```
R> H <- -opt[["hessian"]]
R> CH <- Cholesky(H)
```

We can then sample from the posterior using an MVN approximation, and compute the MVN log density for each sample.

```
R> samples <- rmvn.sparse(R, pm, CH, prec=TRUE)
R> 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.

```
R> Matrix::nnzero(H)

# [1] 402

R> Hinv <- drop0(solve(H))
R> 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`.

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

# [1] TRUE

# `summarise()` has grouped output by 'N', 'k', 'stat', 'pattern'. You can override using the
# `summarise()` has grouped output by 'N', 'k', 'stat', 'pattern'. You can override using the
```

3. Timing

In this section we show the efficiency gains from **sparseMVN** by comparing the run times between `rmvn.sparse` and `rmvnorm`, and between `dmvn.sparse` and `dmvnorm`. In these tests, we construct covariance/precision matrices with the same structure as the Hessian of the log posterior density of the example model in Section 2.1. Parameters are ordered such that the matrix has a block-arrow pattern, as in Figure 1a. The size and sparsity of the test matrices vary through manipulation of the number of blocks (N), the size of each block (k), and the number of rows/columns in the margin (also k). Each test matrix has $(N + 1)k$ rows and columns. Table 2 summarizes the case conditions.

				nonzeros		
	N	variables	elements	full	lower tri	sparsity
k=2	10	22	484	124	73	0.256
	20	42	1,764	244	143	0.138
	50	102	10,404	604	353	0.058
	100	202	40,804	1,204	703	0.030
	200	402	161,604	2,404	1,403	0.015
	300	602	362,404	3,604	2,103	0.010
	400	802	643,204	4,804	2,803	0.007
	500	1,002	1,004,004	6,004	3,503	0.006
k=4	10	44	1,936	496	270	0.256
	20	84	7,056	976	530	0.138
	50	204	41,616	2,416	1,310	0.058
	100	404	163,216	4,816	2,610	0.030
	200	804	646,416	9,616	5,210	0.015
	300	1,204	1,449,616	14,416	7,810	0.010
	400	1,604	2,572,816	19,216	10,410	0.007
	500	2,004	4,016,016	24,016	13,010	0.006

Table 2: Cases for timing comparision. N and k refer, respectively, to the number of blocks in the block-arrow structure (analogous to heterogeneous units in the binary choice example), and the size of each block. The total number of variables is $M = (N + 1)k$, and the total number of elements in the matrix is M^2. The three rightmost columns present the number of nonzeros in the full matrix and lower triangle, and the sparsity (proportion of matrix elements that are nonzero).

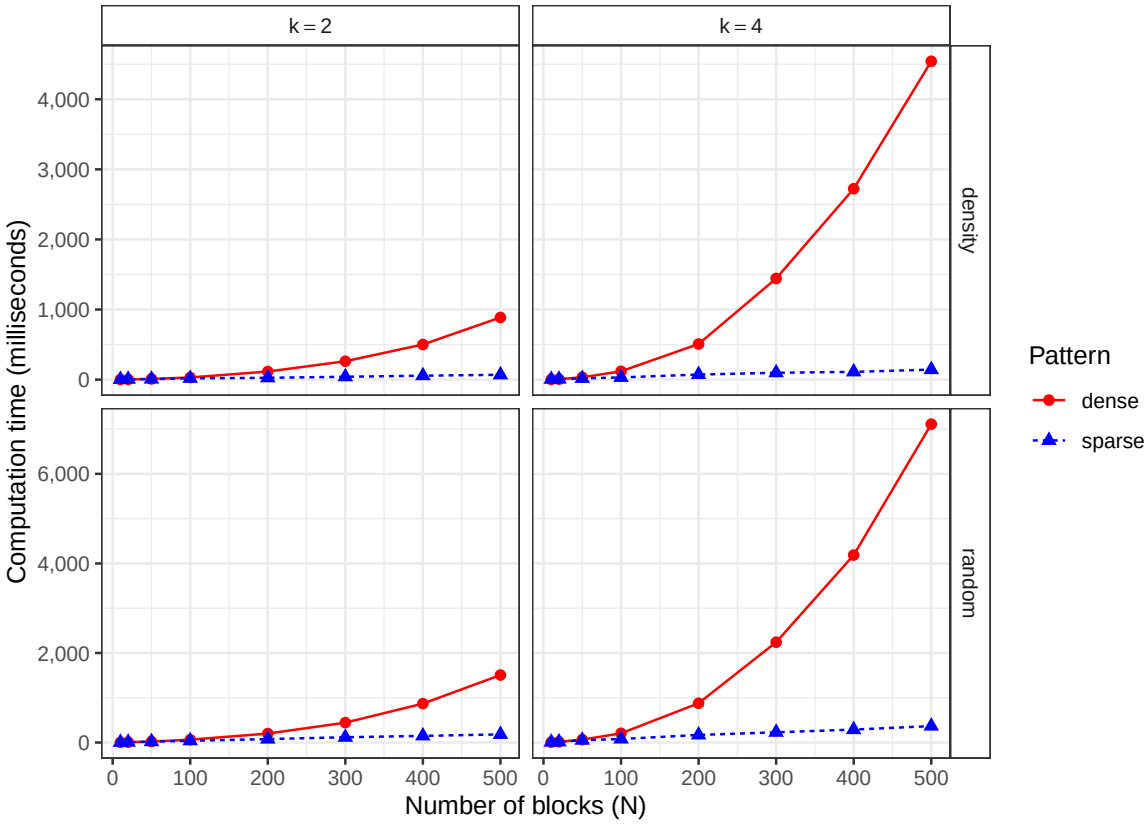


Figure 2: Mean computation time for simulating 1,000 MVN samples, and computing 1,000 MVN densities, averaged over 200 replications. Densities were computed using `dmvnorm` and `dmvn.sparse`, while random samples were generated using `rmvnorm` and `rmvn.sparse`.

Figure 2 compares mean run times to compute 1,000 MVN densities, and generate 1,000 MVN samples, using functions in **sparseMVN** (`rmvn.sparse`, `dmvn.sparse`) and **mvtnorm** (`rmvnorm`, `dmvnorm`). Times were collected over 200 replications on a 2013-vintage Mac Pro with a 12-core 2.7 GHz Intel Xeon E5 processor and 64 GB of RAM. The times for **mvtnorm** are faster than **sparseMVN** for low dimensional cases ($N \leq 50$), but grow quadratically in the number of variables.[4] This is because the number of elements stored in a dense covariance matrix grows quadratically with the number of variables. In this example, storage and computation requirements for the sparse matrix grow linearly with the number of variables, so the **sparseMVN** run times grow linearly as well (**?**, sec. 4). The comparative advantage of **sparseMVN** increases with the sparsity of the covariance matrix.[5]

The `sparseMVN` functions always require a sparse Cholesky decomposition of the covariance or precision matrix, and the **mvtnorm** functions require a dense precision matrix to be inverted into a dense covariance matrix. Figure 3 compares the computation times of these preparatory

[4] As an example, in the $N = 10$, $k = 2$ case, the mean time to compute 1,000 MVN densities is 1.1 ms using `dmvnorm`, but more than 3.7 ms using `dmvn.sparse`.

[5] Across all cases there was hardly any difference in the **sparseMVN** run times when providing the precision matrix instead of the covariance.

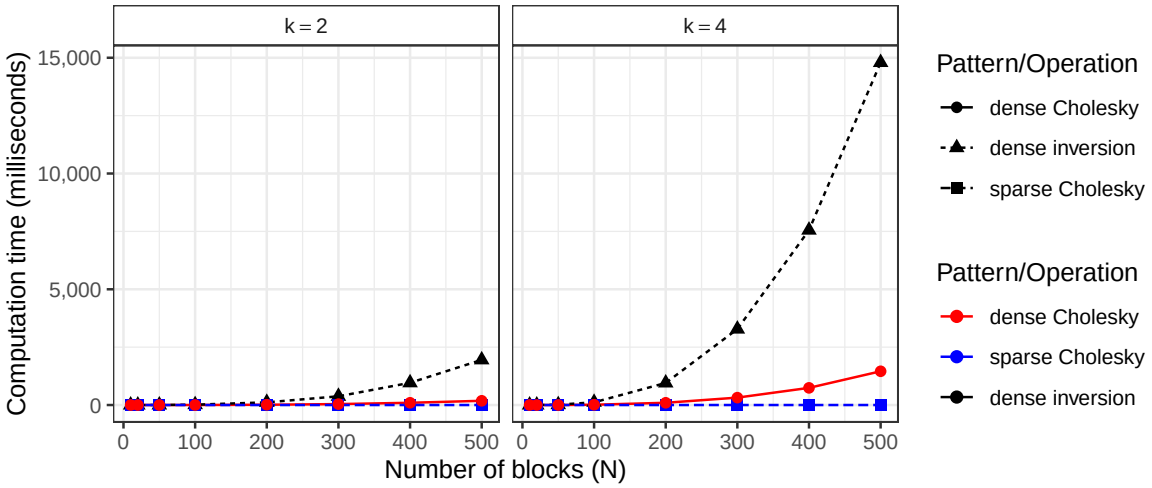


Figure 3: Computation time for Cholesky decomposition of sparse and dense matrices, and inversion of dense matrices.

steps. There are three cases to consider: inverting a dense matrix using the `solve` function, decomposing a sparse matrix using `Cholesky`, and decomposing a dense matrix using `chol`. Applying `chol` to a dense function is not a required operation in advance of calling `rmvnorm` or `dmvnorm`, but those functions will invoke some kind of decomposition internally. We include it in our comparison because it comprises a substantial part of the computation time. The decomposition and inversion operations on the dense matrices grow cubically as the size of the matrix increases. The sparse Cholesky decomposition time is negligible. For example, the mean run time for the $N = 500$, $k = 4$ case is about 0.39 ms.

Code to replicate the data used in Figures 2 and 3 is available as an online supplement to this paper, and in the `doc/` directory of the installed package.

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