Multivariate Normal Log-likelihoods in the **mvtnorm** Package

Torsten Hothorn
Universität Zürich

April 19, 2023

Contents

Licence

Chapter 1

Introduction

This document describes an implementation of Genz (1992) and, partially, of Genz and Bretz (2002), for the evaluation of N multivariate J-dimensional normal probabilities

$$p_i(\mathbf{C}_i \mid \mathbf{a}_i, \mathbf{b}_i) = \mathbb{P}(\mathbf{a}_i < \mathbf{Y}_i \leq \mathbf{b}_i \mid \mathbf{C}_i) = (2\pi)^{-\frac{J}{2}} \det(\mathbf{C}_i)^{-\frac{1}{2}} \int_{\mathbf{a}_i}^{\mathbf{b}_i} \exp\left(-\frac{1}{2}\mathbf{y}^\top \mathbf{C}_i^{-\top} \mathbf{C}_i^{-1} \mathbf{y}\right) d\mathbf{y} \tag{1.1}$$

where $\mathbf{a}_i = (a_1^{(i)}, \ldots, a_J^{(i)})^\top \in \mathbb{R}^J$ and $\mathbf{b}_i = (b_1^{(i)}, \ldots, b_J^{(i)})^\top \in \mathbb{R}^J$ are integration limits, $\mathbf{C}_i = (c_{jj}^{(i)}) \in \mathbb{R}^{J \times J}$ is a lower triangular matrix with $c_{jj}^{(i)} = 0$ for $1 \leq j < \jmath < J$, and thus $\mathbf{Y}_i \sim \mathbb{N}_J(\mathbf{0}_J, \mathbf{C}_i \mathbf{C}_i^\top)$ for $i = 1, \ldots, N$.

One application of these integrals is the estimation of the Cholesky factor $\mathbf{C}$ of a J-dimensional normal distribution based on N interval-censored observations $\mathbf{Y}_1, \ldots, \mathbf{Y}_J$ (encoded by $\mathbf{a}$ and $\mathbf{b}$) via maximum-likelihood

$$\hat{\mathbf{C}} = \operatorname*{argmax}_{\mathbf{C}} \sum_{i=1}^{N} \log(p_i(\mathbf{C} \mid \mathbf{a}_i, \mathbf{b}_i)).$$

In other applications, the Cholesky factor might also depend on i in some structured way.

Function `pmvnorm` in package `mvtnorm` computes p_i based on the covariance matrix $\mathbf{C}_i \mathbf{C}_i^\top$. However, the Cholesky $\mathbf{C}_i$ is computed in FORTRAN. Function `pmvnorm` is not vectorised over $i = 1, \ldots, N$ and thus separate calls to this function are necessary in order to compute likelihood contributions.

The implementation described here is a re-implementation (in R and C) of Alan Genz' original FORTRAN code, focusing on efficient computation of the log-likelihood $\sum_{i=1}^{N} \log(p_i)$ and the corresponding score function.

The document first describes a class and some useful methods for dealing with multiple lower triangular matrices $\mathbf{C}_i, i = 1, \ldots, N$ in Chapter 2. The multivariate normal log-likelihood, and the corresponding score function, is implemented as outlined in Chapter 3. An example demonstrating maximum-likelihood estimation of Cholesky factors in the presence of interval-censored observations is discussed last in Chapter 4.

Chapter 2

Lower Triangular Matrices

"ltMatrices.R" 2≡

⟨ *R Header* 83 ⟩
⟨ *ltMatrices* 5a ⟩
⟨ *dim ltMatrices* 5b ⟩
⟨ *dimnames ltMatrices* 5c ⟩
⟨ *names ltMatrices* 6 ⟩
⟨ *print ltMatrices* 9 ⟩
⟨ *reorder ltMatrices* 10 ⟩
⟨ *subset ltMatrices* 12 ⟩
⟨ *lower triangular elements* 14 ⟩
⟨ *diagonals ltMatrices* 16 ⟩
⟨ *diagonal matrix* 19 ⟩
⟨ *mult ltMatrices* 20b ⟩
⟨ *solve ltMatrices* 27 ⟩
⟨ *tcrossprod ltMatrices* 32 ⟩
⟨ *crossprod ltMatrices* 33 ⟩
⟨ *chol syMatrices* 34 ⟩
⟨ *add diagonal elements* 17 ⟩
⟨ *assign diagonal elements* 18 ⟩
⟨ *kronecker vec trick* 39b ⟩
⟨ *convenience functions* 42 ⟩
⟨ *aperm* 44 ⟩
⟨ *marginal* 45b ⟩
⟨ *conditional* 47b ⟩
◇

"ltMatrices.c" 3a≡

```
⟨ C Header 84 ⟩
#include <R.h>
#include <Rmath.h>
#include <Rinternals.h>
#include <Rdefines.h>
#include <Rconfig.h>
#include <R_ext/Lapack.h> /* for dtptri */
⟨ solve 26 ⟩
⟨ tcrossprod 31 ⟩
⟨ mult 22 ⟩
⟨ chol 35 ⟩
⟨ vec trick 37a ⟩
◇
```

We first need infrastructure for dealing with multiple lower triangular matrices $\mathbf{C}_i \in \mathbb{R}^{J \times J}$ for $i = 1, \ldots, N$. We note that each such matrix $\mathbf{C}$ can be stored in a vector of length $J(J+1)/2$. If all diagonal elements are one (that is, $c_{jj}^{(i)} \equiv 1, j = 1, \ldots, J$), the length of this vector is $J(J-1)/2$.

2.1 Multiple Lower Triangular Matrices

We can store N such matrices in an $J(J+1)/2 \times N$ matrix (`diag = TRUE`) or, for `diag = FALSE`, the $J(J-1)/2 \times N$ matrix.

Each vector might define the corresponding lower triangular matrix either in row or column-major order:

$$\mathbf{C} = \begin{pmatrix} c_{11} & & & 0 \\ c_{21} & c_{22} & & \\ c_{31} & c_{32} & c_{33} & \\ \vdots & \vdots & & \ddots \\ c_{J1} & c_{J2} & \cdots & c_{JJ} \end{pmatrix} \text{ matrix indexing}$$

$$= \begin{pmatrix} c_1 & & & 0 \\ c_2 & c_{J+1} & & \\ c_3 & c_{J+2} & c_{2J} & \\ \vdots & \vdots & & \ddots \\ c_J & c_{2J-1} & \cdots & c_{J(J+1)/2} \end{pmatrix} \text{ column-major, } \texttt{byrow = FALSE}$$

$$= \begin{pmatrix} & c_1 & & & & 0 \\ & c_2 & & c_3 & & \\ & c_4 & & c_5 & c_6 & \\ & \vdots & & \vdots & & \ddots \\ c_{J((J+1)/2-1)+1} & & c_{J((J+1)/2-1)+2} & \cdots & & c_{J(J+1)/2} \end{pmatrix} \text{ row-major, } \texttt{byrow = TRUE}$$

Based on some matrix `object`, the dimension J is computed and checked as

⟨ *ltMatrices dim* 3b ⟩ ≡

```
J <- floor((1 + sqrt(1 + 4 * 2 * nrow(object))) / 2 - diag)
stopifnot(nrow(object) == J * (J - 1) / 2 + diag * J)
◇
```

Fragment referenced in 5a.

Typically the J dimensions are associated with names, and we therefore compute identifiers for the vector elements in either column- or row-major order on request (for later printing)

⟨ *ltMatrices names* 4a ⟩ ≡

```
nonames <- FALSE
if (!isTRUE(names)) {
    if (is.character(names))
        stopifnot(is.character(names) &&
                  length(unique(names)) == J)
    else
        nonames <- TRUE
} else {
    names <- as.character(1:J)
}

if (!nonames) {
    L1 <- matrix(names, nrow = J, ncol = J)
    L2 <- matrix(names, nrow = J, ncol = J, byrow = TRUE)
    L <- matrix(paste(L1, L2, sep = "."), nrow = J, ncol = J)
    if (byrow)
        rownames(object) <- t(L)[upper.tri(L, diag = diag)]
    else
        rownames(object) <- L[lower.tri(L, diag = diag)]
}
```

◇

Fragment referenced in 5a.

If `object` is already a classed object representing lower triangular matrices (we will use the class name `ltMatrices`), we might want to change the storage form from row- to column-major or the other way round.

⟨ *ltMatrices input* 4b ⟩ ≡

```
if (inherits(object, "ltMatrices")) {
    ret <- .reorder(object, byrow = byrow)
    return(ret)
}
```

◇

Fragment referenced in 5a.

The constructor essentially attaches attributes to a matrix `object`, possibly after some reordering / transposing

⟨ *ltMatrices* 5a ⟩ ≡

```
ltMatrices <- function(object, diag = FALSE, byrow = FALSE, names = TRUE) {

    if (!is.matrix(object))
        object <- matrix(object, ncol = 1L)

    ⟨ ltMatrices input 4b ⟩

    ⟨ ltMatrices dim 3b ⟩

    ⟨ ltMatrices names 4a ⟩

    attr(object, "J")       <- J
    attr(object, "diag")    <- diag
    attr(object, "byrow")   <- byrow
    attr(object, "rcnames") <- names

    class(object) <- c("ltMatrices", class(object))
    object
}
```

◇

Fragment referenced in 2.

Symmetric matrices are represented by lower triangular matrix objects, but we change the class from `ltMatrices` to `syMatrices` (which disables all functionality except printing and coersion to arrays).

The dimensions of such an object are always $N \times J \times J$ and are given by

⟨ *dim ltMatrices* 5b ⟩ ≡

```
dim.ltMatrices <- function(x) {
    J <- attr(x, "J")
    class(x) <- class(x)[-1L]
    return(c(ncol(x), J, J))
}
dim.syMatrices <- dim.ltMatrices
```

◇

Fragment referenced in 2.

The corresponding dimnames can be extracted as

⟨ *dimnames ltMatrices* 5c ⟩ ≡

```
dimnames.ltMatrices <- function(x)
    return(list(colnames(unclass(x)), attr(x, "rcnames"), attr(x, "rcnames")))
dimnames.syMatrices <- dimnames.ltMatrices
```

◇

Fragment referenced in 2.

The names identifying rows and columns in each $\mathbf{C}_i$ are

⟨ *names ltMatrices* 6 ⟩ ≡

```
names.ltMatrices <- function(x) {
    return(rownames(unclass(x)))
}
names.syMatrices <- names.ltMatrices
◇
```

Fragment referenced in 2.

Let's set-up an example for illustration:

```
> library("mvtnorm")
> chk <- function(...) stopifnot(isTRUE(all.equal(...)))
> set.seed(290875)
> N <- 4
> J <- 5
> rn <- paste0("C_", 1:N)
> nm <- LETTERS[1:J]
> Jn <- J * (J - 1) / 2
> ## data
> xn <- t(matrix(runif(N * Jn), nrow = N, byrow = TRUE))
> colnames(xn) <- rn
> xd <- t(matrix(runif(N * (Jn + J)), nrow = N, byrow = TRUE))
> colnames(xd) <- rn
> (lxn <- ltMatrices(xn, byrow = TRUE, names = nm))

, , C_1

           A         B          C         D E
A 1.00000000 0.0000000 0.00000000 0.0000000 0
B 0.51236601 1.0000000 0.00000000 0.0000000 0
C 0.05847253 0.9095137 1.00000000 0.0000000 0
D 0.39448719 0.6612143 0.23352591 1.0000000 0
E 0.51647518 0.2979867 0.07517749 0.8182123 1

, , C_2

          A         B         C        D E
A 1.0000000 0.0000000 0.0000000 0.000000 0
B 0.8590665 1.0000000 0.0000000 0.000000 0
C 0.3744315 0.1022684 1.0000000 0.000000 0
D 0.1165248 0.7956529 0.8930589 1.000000 0
E 0.1948049 0.4730419 0.2377852 0.214606 1

, , C_3

          A         B         C         D E
A 1.0000000 0.0000000 0.0000000 0.0000000 0
B 0.4530153 1.0000000 0.0000000 0.0000000 0
C 0.9045608 0.9269936 1.0000000 0.0000000 0
D 0.4490011 0.1326375 0.4153967 1.0000000 0
E 0.9574833 0.4917481 0.7160702 0.2938002 1
```

```
, , C_4

             A         B           C         D E
A 1.0000000000 0.0000000 0.000000000 0.0000000 0
B 0.4877241328 1.0000000 0.000000000 0.0000000 0
C 0.0593045885 0.7625270 1.000000000 0.0000000 0
D 0.0005227393 0.1995700 0.470508903 1.0000000 0
E 0.4913541358 0.2849431 0.005961103 0.8901458 1

> dim(lxn)

[1] 4 5 5

> dimnames(lxn)

[[1]]
[1] "C_1" "C_2" "C_3" "C_4"

[[2]]
[1] "A" "B" "C" "D" "E"

[[3]]
[1] "A" "B" "C" "D" "E"

> lxd <- ltMatrices(xd, byrow = TRUE, diag = TRUE, names = nm)
> dim(lxd)

[1] 4 5 5

> dimnames(lxd)

[[1]]
[1] "C_1" "C_2" "C_3" "C_4"

[[2]]
[1] "A" "B" "C" "D" "E"

[[3]]
[1] "A" "B" "C" "D" "E"

> class(lxn) <- "syMatrices"
> lxn

, , C_1

           A         B          C         D          E
A 1.00000000 0.5123660 0.05847253 0.3944872 0.51647518
B 0.51236601 1.0000000 0.90951367 0.6612143 0.29798667
C 0.05847253 0.9095137 1.00000000 0.2335259 0.07517749
D 0.39448719 0.6612143 0.23352591 1.0000000 0.81821229
E 0.51647518 0.2979867 0.07517749 0.8182123 1.00000000

, , C_2

          A         B         C         D         E
A 1.0000000 0.8590665 0.3744315 0.1165248 0.1948049
```

```
B 0.8590665 1.0000000 0.1022684 0.7956529 0.4730419
C 0.3744315 0.1022684 1.0000000 0.8930589 0.2377852
D 0.1165248 0.7956529 0.8930589 1.0000000 0.2146060
E 0.1948049 0.4730419 0.2377852 0.2146060 1.0000000

, , C_3

          A         B         C         D         E
A 1.0000000 0.4530153 0.9045608 0.4490011 0.9574833
B 0.4530153 1.0000000 0.9269936 0.1326375 0.4917481
C 0.9045608 0.9269936 1.0000000 0.4153967 0.7160702
D 0.4490011 0.1326375 0.4153967 1.0000000 0.2938002
E 0.9574833 0.4917481 0.7160702 0.2938002 1.0000000

, , C_4

             A         B           C            D           E
A 1.0000000000 0.4877241 0.059304588 0.0005227393 0.491354136
B 0.4877241328 1.0000000 0.762527028 0.1995699527 0.284943077
C 0.0593045885 0.7625270 1.000000000 0.4705089033 0.005961103
D 0.0005227393 0.1995700 0.470508903 1.0000000000 0.890145786
E 0.4913541358 0.2849431 0.005961103 0.8901457863 1.000000000
```

2.2 Printing

For pretty printing, we coerse object of class `ltMatrices` to `array`. The method has an `symmetric` argument forcing the lower triangular matrix to by interpreted as a symmetric matrix.

⟨ *extract slots* 8 ⟩ ≡

```
diag <- attr(x, "diag")
byrow <- attr(x, "byrow")
d <- dim(x)
J <- d[2L]
dn <- dimnames(x)
```

◇

Fragment referenced in 9, 10, 11, 14, 16, 18, 20b.

⟨ *print ltMatrices* 9 ⟩ ≡

```
as.array.ltMatrices <- function(x, symmetric = FALSE, ...) {

    ⟨ extract slots 8 ⟩

    class(x) <- class(x)[-1L]
    x <- t(x)

    L <- matrix(1L, nrow = J, ncol = J)
    diag(L) <- 2L
    if (byrow) {
        L[upper.tri(L, diag = diag)] <- floor(2L + 1:(J * (J - 1) / 2L + diag * J))
        L <- t(L)
    } else {
        L[lower.tri(L, diag = diag)] <- floor(2L + 1:(J * (J - 1) / 2L + diag * J))
    }
    if (symmetric) {
        L[upper.tri(L)] <- 0L
        dg <- diag(L)
        L <- L + t(L)
        diag(L) <- dg
    }
    ret <- t(cbind(0, 1, x)[, c(L), drop = FALSE])
    class(ret) <- "array"
    dim(ret) <- d[3:1]
    dimnames(ret) <- dn[3:1]
    return(ret)
}

as.array.syMatrices <- function(x, ...)
    return(as.array.ltMatrices(x, symmetric = TRUE))

print.ltMatrices <- function(x, ...)
    print(as.array(x))

print.syMatrices <- function(x, ...)
    print(as.array(x))
◇
```

Fragment referenced in 2.

2.3 Reordering

It is sometimes convenient to have access to lower triangular matrices in either column- or row-major order and this little helper function switches between the two forms

⟨ *reorder ltMatrices* 10 ⟩ ≡

```
.reorder <- function(x, byrow = FALSE) {

    stopifnot(inherits(x, "ltMatrices"))
    if (attr(x, "byrow") == byrow) return(x)

    ⟨ extract slots 8 ⟩

    class(x) <- class(x)[-1L]

    rL <- cL <- diag(0, nrow = J)
    rL[lower.tri(rL, diag = diag)] <- cL[upper.tri(cL, diag = diag)] <- 1:nrow(x)
    cL <- t(cL)
    if (byrow) ### row -> col order
        return(ltMatrices(x[cL[lower.tri(cL, diag = diag)], , drop = FALSE],
                          diag = diag, byrow = FALSE, names = dn[[2L]]))
    ### col -> row order
    return(ltMatrices(x[t(rL)[upper.tri(rL, diag = diag)], , drop = FALSE],
                      diag = diag, byrow = TRUE, names = dn[[2L]]))
}
```

◇

Fragment referenced in 2.

We can check if this works by switching back and forth between column-major and row-major order

```
> ## constructor + .reorder + as.array
> a <- as.array(ltMatrices(xn, byrow = TRUE))
> b <- as.array(ltMatrices(ltMatrices(xn, byrow = TRUE), byrow = FALSE))
> chk(a, b)
> a <- as.array(ltMatrices(xn, byrow = FALSE))
> b <- as.array(ltMatrices(ltMatrices(xn, byrow = FALSE), byrow = TRUE))
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = TRUE, diag = TRUE))
> b <- as.array(ltMatrices(ltMatrices(xd, byrow = TRUE, diag = TRUE), byrow = FALSE))
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = FALSE, diag = TRUE))
> b <- as.array(ltMatrices(ltMatrices(xd, byrow = FALSE, diag = TRUE), byrow = TRUE))
> chk(a, b)
```

2.4 Subsetting

We might want to select subsets of observations $i \in \{1, \ldots, N\}$ or rows/columns $j \in \{1, \ldots, J\}$ of the corresponding matrices $\mathbf{C}_i$.

⟨ *.subset ltMatrices* 11 ⟩ ≡

```
.subset_ltMatrices <- function(x, i, j, ..., drop = FALSE) {

    if (drop) warning("argument drop is ignored")
    if (missing(i) && missing(j)) return(x)

    ⟨ extract slots 8 ⟩

    class(x) <- class(x)[-1L]

    if (!missing(j)) {

        j <- (1:J)[j] ### get rid of negative indices

        if (length(j) == 1L && !diag) {
            return(ltMatrices(matrix(1, ncol = ncol(x), nrow = 1), diag = TRUE,
                              byrow = byrow, names = dn[[2L]][j]))
        }
        L <- diag(0L, nrow = J)
        Jp <- sum(upper.tri(L, diag = diag))
        if (byrow) {
            L[upper.tri(L, diag = diag)] <- 1:Jp
            L <- L + t(L)
            diag(L) <- diag(L) / 2
            L <- L[j, j, drop = FALSE]
            L <- L[upper.tri(L, diag = diag)]
        } else {
            L[lower.tri(L, diag = diag)] <- 1:Jp
            L <- L + t(L)
            diag(L) <- diag(L) / 2
            L <- L[j, j, drop = FALSE]
            L <- L[lower.tri(L, diag = diag)]
        }
        if (missing(i)) {
            return(ltMatrices(x[c(L), , drop = FALSE], diag = diag,
                              byrow = byrow, names = dn[[2L]][j]))
        }
        return(ltMatrices(x[c(L), i, drop = FALSE], diag = diag,
                          byrow = byrow, names = dn[[2L]][j]))
    }
    return(ltMatrices(x[, i, drop = FALSE], diag = diag,
                      byrow = byrow, names = dn[[2L]]))
}
```

◇

Fragment referenced in 12.

⟨ *subset ltMatrices* 12 ⟩ ≡

```
⟨ .subset ltMatrices 11 ⟩
### if j is not ordered, result is not a lower triangular matrix
"[.ltMatrices" <- function(x, i, j, ..., drop = FALSE) {
    if (!missing(j)) {
        if (all(j > 0)) {
            if (any(diff(j) < 0)) stop("invalid subset argument j")
        }
    }

    return(.subset_ltMatrices(x = x, i = i, j = j, ..., drop = drop))
}

"[.syMatrices" <- function(x, i, j, ..., drop = FALSE) {
    class(x)[1L] <- "ltMatrices"
    ret <- .subset_ltMatrices(x = x, i = i, j = j, ..., drop = drop)
    class(ret)[1L] <- "syMatrices"
    ret
}
◇
```

Fragment referenced in 2.

We check if this works by first subsetting the `ltMatrices` object. Second, we coerse the object to an array and do the subset for the latter object. Both results must agree.

```
> ## subset
> a <- as.array(ltMatrices(xn, byrow = FALSE)[1:2, 2:4])
> b <- as.array(ltMatrices(xn, byrow = FALSE))[2:4, 2:4, 1:2]
> chk(a, b)
> a <- as.array(ltMatrices(xn, byrow = TRUE)[1:2, 2:4])
> b <- as.array(ltMatrices(xn, byrow = TRUE))[2:4, 2:4, 1:2]
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = FALSE, diag = TRUE)[1:2, 2:4])
> b <- as.array(ltMatrices(xd, byrow = FALSE, diag = TRUE))[2:4, 2:4, 1:2]
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = TRUE, diag = TRUE)[1:2, 2:4])
> b <- as.array(ltMatrices(xd, byrow = TRUE, diag = TRUE))[2:4, 2:4, 1:2]
> chk(a, b)
```

With a different subset

```
> ## subset
> j <- c(1, 3, 5)
> a <- as.array(ltMatrices(xn, byrow = FALSE)[1:2, j])
> b <- as.array(ltMatrices(xn, byrow = FALSE))[j, j, 1:2]
> chk(a, b)
> a <- as.array(ltMatrices(xn, byrow = TRUE)[1:2, j])
> b <- as.array(ltMatrices(xn, byrow = TRUE))[j, j, 1:2]
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = FALSE, diag = TRUE)[1:2, j])
> b <- as.array(ltMatrices(xd, byrow = FALSE, diag = TRUE))[j, j, 1:2]
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = TRUE, diag = TRUE)[1:2, j])
```

```
> b <- as.array(ltMatrices(xd, byrow = TRUE, diag = TRUE))[j, j, 1:2]
> chk(a, b)
```

with negative subsets

```
> ## subset
> j <- -c(1, 3, 5)
> a <- as.array(ltMatrices(xn, byrow = FALSE)[1:2, j])
> b <- as.array(ltMatrices(xn, byrow = FALSE))[j, j, 1:2]
> chk(a, b)
> a <- as.array(ltMatrices(xn, byrow = TRUE)[1:2, j])
> b <- as.array(ltMatrices(xn, byrow = TRUE))[j, j, 1:2]
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = FALSE, diag = TRUE)[1:2, j])
> b <- as.array(ltMatrices(xd, byrow = FALSE, diag = TRUE))[j, j, 1:2]
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = TRUE, diag = TRUE)[1:2, j])
> b <- as.array(ltMatrices(xd, byrow = TRUE, diag = TRUE))[j, j, 1:2]
> chk(a, b)
```

and with non-increasing argument `j` (this won't work for lower triangular matrices, only for symmetric matrices)

```
> ## subset
> j <- sample(1:J)
> ltM <- ltMatrices(xn, byrow = FALSE)
> try(ltM[1:2, j])
> class(ltM) <- "syMatrices"
> a <- as.array(ltM[1:2, j])
> b <- as.array(ltM)[j, j, 1:2]
> chk(a, b)
```

Extracting the lower triangular elements from an `ltMatrices` object (or from an object of class `syMatrices`) returns a matrix with N columns

⟨ *lower triangular elements* 14 ⟩ ≡

```
Lower_tri <- function(x, diag = FALSE, byrow = attr(x, "byrow")) {

    if (inherits(x, "syMatrices"))
        class(x)[1L] <- "ltMatrices"
    stopifnot(inherits(x, "ltMatrices"))
    adiag <- diag
    x <- ltMatrices(x, byrow = byrow)

    ⟨ extract slots 8 ⟩

    if (diag == adiag)
        return(unclass(x))

    if (!diag && adiag) {
        diagonals(x) <- 1
        return(unclass(x))
    }

    x <- unclass(x)
    if (J == 1) {
        idx <- 1L
    } else {
      if (byrow)
          idx <- cumsum(c(1, 2:J))
      else
          idx <- cumsum(c(1, J:2))
    }
    return(x[-idx,,drop = FALSE])
}
```

◇

Fragment referenced in 2.

```
> ## J <- 4
> M <- ltMatrices(matrix(1:10, nrow = 10, ncol = 2), diag = TRUE)
> Lower_tri(M, diag = FALSE)

    [,1] [,2]
2.1    2    2
3.1    3    3
4.1    4    4
3.2    6    6
4.2    7    7
4.3    9    9

> Lower_tri(M, diag = TRUE)

    [,1] [,2]
1.1    1    1
2.1    2    2
3.1    3    3
4.1    4    4
2.2    5    5
3.2    6    6
4.2    7    7
```

```
3.3    8    8
4.3    9    9
4.4   10   10
attr(,"J")
[1] 4
attr(,"diag")
[1] TRUE
attr(,"byrow")
[1] FALSE
attr(,"rcnames")
[1] "1" "2" "3" "4"
```

```
> M <- ltMatrices(matrix(1:6, nrow = 6, ncol = 2), diag = FALSE)
> Lower_tri(M, diag = FALSE)
```

```
    [,1] [,2]
2.1    1    1
3.1    2    2
4.1    3    3
3.2    4    4
4.2    5    5
4.3    6    6
attr(,"J")
[1] 4
attr(,"diag")
[1] FALSE
attr(,"byrow")
[1] FALSE
attr(,"rcnames")
[1] "1" "2" "3" "4"
```

```
> Lower_tri(M, diag = TRUE)
```

```
    [,1] [,2]
1.1    1    1
2.1    1    1
3.1    2    2
4.1    3    3
2.2    1    1
3.2    4    4
4.2    5    5
3.3    1    1
4.3    6    6
4.4    1    1
attr(,"J")
[1] 4
attr(,"diag")
[1] TRUE
attr(,"byrow")
[1] FALSE
attr(,"rcnames")
[1] "1" "2" "3" "4"
```

```
> ## multiple symmetric matrices
> Lower_tri(invchol2cor(M))
```

```
          [,1]       [,2]
2.1 -0.7071068 -0.7071068
3.1  0.4364358  0.4364358
4.1 -0.4481107 -0.4481107
3.2 -0.9258201 -0.9258201
4.2  0.9189002  0.9189002
4.3 -0.9974149 -0.9974149
attr(,"J")
[1] 4
attr(,"diag")
[1] FALSE
attr(,"byrow")
[1] FALSE
attr(,"rcnames")
[1] "1" "2" "3" "4"
```

2.5 Diagonal Elements

The diagonal elements of each matrix $\mathbf{C}_i$ can be extracted and are always returned as an $J \times N$ matrix.

⟨ *diagonals ltMatrices* 16 ⟩ ≡

```
diagonals <- function(x, ...)
    UseMethod("diagonals")

diagonals.ltMatrices <- function(x, ...) {

    ⟨ extract slots 8 ⟩

    class(x) <- class(x)[-1L]

    if (!diag) {
        ret <- matrix(1, nrow = J, ncol = ncol(x))
        colnames(ret) <- dn[[1L]]
        rownames(ret) <- dn[[2L]]
        return(ret)
    } else {
        if (J == 1L) return(x)
        if (byrow)
            idx <- cumsum(c(1, 2:J))
        else
            idx <- cumsum(c(1, J:2))
        ret <- x[idx, , drop = FALSE]
        rownames(ret) <- dn[[2L]]
        return(ret)
    }
}

diagonals.syMatrices <- diagonals.ltMatrices

diagonals.matrix <- function(x, ...) diag(x)
◇
```

Fragment referenced in 2.

```
> all(diagonals(ltMatrices(xn, byrow = TRUE)) == 1L)
```

```
[1] TRUE
```

Sometimes we need to add diagonal elements to an `ltMatrices` object defined without diagonal elements.

⟨ *add diagonal elements* 17 ⟩ ≡

```
.adddiag <- function(x) {

    stopifnot(inherits(x, "ltMatrices"))

    if (attr(x, "diag")) return(x)

    byrow_orig <- attr(x, "byrow")

    x <- ltMatrices(x, byrow = FALSE)

    N <- dim(x)[1L]
    J <- dim(x)[2L]
    nm <- dimnames(x)[[2L]]

    L <- diag(J)
    L[lower.tri(L, diag = TRUE)] <- 1:(J * (J + 1) / 2)

    D <- diag(J)
    ret <- matrix(D[lower.tri(D, diag = TRUE)],
                  nrow = J * (J + 1) / 2, ncol = N)
    colnames(ret) <- colnames(unclass(x))
    ret[L[lower.tri(L, diag = FALSE)],] <- unclass(x)

    ret <- ltMatrices(ret, diag = TRUE, byrow = FALSE, names = nm)
    ret <- ltMatrices(ret, byrow = byrow_orig)

    ret
}
```

◇

Fragment referenced in 2.

⟨ *assign diagonal elements* 18 ⟩ ≡

```
"diagonals<-" <- function(x, value)
    UseMethod("diagonals<-")

"diagonals<-.ltMatrices" <- function(x, value) {

    ⟨ extract slots 8 ⟩

    if (byrow)
        idx <- cumsum(c(1, 2:J))
    else
        idx <- cumsum(c(1, J:2))

    ### diagonals(x) <- NULL returns ltMatrices(..., diag = FALSE)
    if (is.null(value)) {
        if (!attr(x, "diag")) return(x)
        if (J == 1L) {
            x[] <- 1
            return(x)
        }
        return(ltMatrices(unclass(x)[-idx,,drop = FALSE], diag = FALSE,
                          byrow = byrow, names = dn[[2L]]))
    }

    x <- .adddiag(x)

    if (!is.matrix(value))
        value <- matrix(value, nrow = J, ncol = d[1L])

    stopifnot(is.matrix(value) && nrow(value) == J
                               && ncol(value) == d[1L])

    if (J == 1L) {
        x[] <- value
        return(x)
    }

    x[idx, ] <- value

    return(x)
}

"diagonals<-.syMatrices" <- function(x, value) {

    class(x)[1L] <- "ltMatrices"
    diagonals(x) <- value
    class(x)[1L] <- "syMatrices"

    return(x)
}
```

◇

Fragment referenced in 2.

```
> lxd2 <- lxn
> diagonals(lxd2) <- 1
> chk(as.array(lxd2), as.array(lxn))
```

A unit diagonal matrix is not treated as a special case but as an `ltMatrices` object with all lower triangular elements being zero

⟨ *diagonal matrix* 19 ⟩ ≡

```
diagonals.integer <- function(x, ...)
    ltMatrices(rep(0, x * (x - 1) / 2), diag = FALSE, ...)
```

◇

Fragment referenced in 2.

```
> (I5 <- diagonals(5L))

, , 1

  1 2 3 4 5
1 1 0 0 0 0
2 0 1 0 0 0
3 0 0 1 0 0
4 0 0 0 1 0
5 0 0 0 0 1

> diagonals(I5) <- 1:5
> I5

, , 1

  1 2 3 4 5
1 1 0 0 0 0
2 0 2 0 0 0
3 0 0 3 0 0
4 0 0 0 4 0
5 0 0 0 0 5
```

2.6 Multiplication

Multiplications $\mathbf{C}_i\mathbf{y}_i$ or $\mathbf{C}_i^\top\mathbf{y}_i$ with $\mathbf{y}_i \in \mathbb{R}^J$ for $i = 1, \ldots, N$ can be computed with `y` being an $J \times N$ matrix of columns-wise stacked vectors $(\mathbf{y}_1 \mid \mathbf{y}_2 \mid \cdots \mid \mathbf{y}_N)$. If `y` is a single vector, it is recycled N times.

If the number of columns of a matrix `y` is neither one nor N, we compute $\mathbf{C}_i\mathbf{y}_j$ for all $i = 1, \ldots, N$ and j. This is dangerous but needed in `cond_mvnorm` alter on.

We start with $\mathbf{C}_i^\top\mathbf{y}_i$ (`transpose = TRUE`), which can conveniently be computed in R (although no attention is paid to the lower triangular structure of `x`)

⟨ *mult ltMatrices transpose* 20a ⟩ ≡

```
if (transpose) {
    J <- dim(x)[2L]
    if (dim(x)[1L] == 1L) x <- x[rep(1, N),]
    ax <- as.array(x)
    ay <- array(y[rep(1:J, J),,drop = FALSE], dim = dim(ax),
                dimnames = dimnames(ax))
    ret <- ay * ax
    ### was: return(margin.table(ret, 2:3))
    ret <- matrix(colSums(matrix(ret, nrow = dim(ret)[1L])),
                  nrow = dim(ret)[2L], ncol = dim(ret)[3L],
                  dimnames = dimnames(ret)[-1L])
    return(ret)
}
```

◇

Fragment referenced in 20b.

For $\mathbf{C}_i\mathbf{y}_i$, we call C code computing the product efficiently without copying data by using the lower triangular structure of `x`

⟨ *mult ltMatrices* 20b ⟩ ≡

```
### C %*% y
Mult <- function(x, y, transpose = FALSE) {

    if (!inherits(x, "ltMatrices")) {
        if (!transpose) return(x %*% y)
        return(crossprod(x, y))
    }

    ⟨ extract slots 8 ⟩

    if (!is.matrix(y)) y <- matrix(y, nrow = d[2L], ncol = d[1L])
    N <- ifelse(d[1L] == 1, ncol(y), d[1L])
    stopifnot(nrow(y) == d[2L])
    if (ncol(y) != N)
        return(sapply(1:ncol(y), function(i) Mult(x, y[,i], transpose = transpose)))

    ⟨ mult ltMatrices transpose 20a ⟩

    x <- ltMatrices(x, byrow = TRUE)

    class(x) <- class(x)[-1L]
    storage.mode(x) <- "double"
    storage.mode(y) <- "double"

    ret <- .Call(mvtnorm_R_ltMatrices_Mult, x, y, as.integer(N),
                 as.integer(d[2L]), as.logical(diag))

    rownames(ret) <- dn[[2L]]
    if (length(dn[[1L]]) == N)
        colnames(ret) <- dn[[1L]]
    return(ret)
}
```

◇

Fragment referenced in 2.

The underlying C code assumes $\mathbf{C}_i$ (here called `C`) to be in row-major order.

⟨ *RC input* 21a ⟩ ≡

```
/* pointer to C matrices */
double *dC = REAL(C);
/* number of matrices */
int iN = INTEGER(N)[0];
/* dimension of matrices */
int iJ = INTEGER(J)[0];
/* C contains diagonal elements */
Rboolean Rdiag = asLogical(diag);
/* p = J * (J - 1) / 2 + diag * J */
int len = iJ * (iJ - 1) / 2 + Rdiag * iJ;
```

◇

Fragment referenced in 22, 26, 31, 37a.

We also allow $\mathbf{C}_i$ to be constant (N is then determined from `ncol(y)`). The following fragment ensures that we only loop over $\mathbf{C}_i$ if `dim(x)[1L] > 1`

⟨ *C length* 21b ⟩ ≡

```
int p;
if (LENGTH(C) == len)
    /* C is constant for i = 1, ..., N */
    p = 0;
else
    /* C contains C_1, ...., C_N */
    p = len;
```

◇

Fragment referenced in 22, 26, 37a.

The C workhorse is now

⟨ *mult* 22 ⟩ ≡

```
SEXP R_ltMatrices_Mult (SEXP C, SEXP y, SEXP N, SEXP J, SEXP diag) {

    SEXP ans;
    double *dans, *dy = REAL(y);
    int i, j, k, start;

    ⟨ RC input 21a ⟩

    ⟨ C length 21b ⟩

    PROTECT(ans = allocMatrix(REALSXP, iJ, iN));
    dans = REAL(ans);

    for (i = 0; i < iN; i++) {
        start = 0;
        for (j = 0; j < iJ; j++) {
            dans[j] = 0.0;
            for (k = 0; k < j; k++)
                dans[j] += dC[start + k] * dy[k];
            if (Rdiag) {
                dans[j] += dC[start + j] * dy[j];
                start += j + 1;
            } else {
                dans[j] += dy[j];
                start += j;
            }
        }
        dC += p;
        dy += iJ;
        dans += iJ;
    }
    UNPROTECT(1);
    return(ans);
}
```

◇

Fragment referenced in 3a.

Some checks for $\mathbf{C}_i\mathbf{y}_i$

```
> lxn <- ltMatrices(xn, byrow = TRUE)
> lxd <- ltMatrices(xd, byrow = TRUE, diag = TRUE)
> y <- matrix(runif(N * J), nrow = J)
> a <- Mult(lxn, y)
> A <- as.array(lxn)
> b <- do.call("rbind", lapply(1:ncol(y), function(i) t(A[,,i] %*% y[,i,drop = FALSE])))
> chk(a, t(b), check.attributes = FALSE)
> a <- Mult(lxd, y)
> A <- as.array(lxd)
> b <- do.call("rbind", lapply(1:ncol(y), function(i) t(A[,,i] %*% y[,i,drop = FALSE])))
> chk(a, t(b), check.attributes = FALSE)
> ### recycle C
> chk(Mult(lxn[rep(1, N),], y), Mult(lxn[1,], y), check.attributes = FALSE)
> ### recycle y
> chk(Mult(lxn, y[,1]), Mult(lxn, y[,rep(1, N)]))
```

```
> ### tcrossprod as multiplication
> i <- sample(1:N)[1]
> M <- t(as.array(lxn)[,,i])
> a <- sapply(1:J, function(j) Mult(lxn[i,], M[,j,drop = FALSE]))
> rownames(a) <- colnames(a) <- dimnames(lxn)[[2L]]
> b <- as.array(Tcrossprod(lxn[i,]))[,,1]
> chk(a, b, check.attributes = FALSE)
```

and for $\mathbf{C}_i^\top \mathbf{y}_i$

```
> a <- Mult(lxn, y, transpose = TRUE)
> A <- as.array(lxn)
> b <- do.call("rbind", lapply(1:ncol(y), function(i) t(t(A[,,i]) %*% y[,i,drop = FALSE])))
> chk(a, t(b), check.attributes = FALSE)
> a <- Mult(lxd, y, transpose = TRUE)
> A <- as.array(lxd)
> b <- do.call("rbind", lapply(1:ncol(y), function(i) t(t(A[,,i]) %*% y[,i,drop = FALSE])))
> chk(a, t(b), check.attributes = FALSE)
> ### recycle C
> chk(Mult(lxn[rep(1, N),], y, transpose = TRUE),
+     Mult(lxn[1,], y, transpose = TRUE), check.attributes = FALSE)
> ### recycle y
> chk(Mult(lxn, y[,1], transpose = TRUE),
+     Mult(lxn, y[,rep(1, N)], transpose = TRUE))
```

2.7 Solving Linear Systems

Compute $\mathbf{C}_i^{-1}$ or solve $\mathbf{C}_i\mathbf{x}_i = \mathbf{y}_i$ for $\mathbf{x}_i$ for all $i = 1, \ldots, N$. We sometimes also need $\mathbf{C}_i^\top \mathbf{x}_i = \mathbf{y}_i$ triggered by `transpose = TRUE`.

`C` is $\mathbf{C}_i, i = 1, \ldots, N$ in column-major order (matrix of dimension $J(J-1)/2 + J\text{diag} \times N$), and `y` is the $J \times N$ matrix $(\mathbf{y}_1 \mid \mathbf{y}_2 \mid \cdots \mid \mathbf{y}_N)$. This function returns the $J \times N$ matrix $(\mathbf{x}_1 \mid \mathbf{x}_2 \mid \cdots \mid \mathbf{x}_N)$ of solutions.

If `y` is not given, $\mathbf{C}_i^{-1}$ is returned in column-major order (matrix of dimension $J(J \pm 1)/2 \times N$). If all $\mathbf{C}_i$ have unit diagonals, so will $\mathbf{C}_i^{-1}$.

⟨ *setup memory* 23 ⟩ ≡

```
/* return object: include unit diagonal elements if Rdiag == 0 */

/* add diagonal elements (expected by Lapack) */
nrow = (Rdiag ? len : len + iJ);
ncol = (p > 0 ? iN : 1);
PROTECT(ans = allocMatrix(REALSXP, nrow, ncol));
dans = REAL(ans);

ansx = ans;
dansx = dans;
dy = dans;
if (y != R_NilValue) {
    dy = REAL(y);
    PROTECT(ansx = allocMatrix(REALSXP, iJ, iN));
    dansx = REAL(ansx);
}
◇
```

Fragment referenced in 26.

The LAPACK functions `dtptri` and `dtpsv` assume that diagonal elements are present, even for unit diagonal matrices.

⟨ *copy elements* 24a ⟩ ≡

```
/* copy data and insert unit diagonal elements when necessary */
if (p > 0 || i == 0) {
    jj = 0;
    k = 0;
    idx = 0;
    j = 0;
    while(j < len) {
        if (!Rdiag && (jj == idx)) {
            dans[jj] = 1.0;
            idx = idx + (iJ - k);
            k++;
        } else {
            dans[jj] = dC[j];
            j++;
        }
        jj++;
    }
    if (!Rdiag) dans[idx] = 1.0;
}

if (y != R_NilValue) {
    for (j = 0; j < iJ; j++)
        dansx[j] = dy[j];
}
```

◇

Fragment referenced in 26.

The LAPACK workhorses are called here

⟨ *call Lapack* 24b ⟩ ≡

```
if (y == R_NilValue) {
    /* compute inverse */
    F77_CALL(dtptri)(&lo, &di, &iJ, dans, &info FCONE FCONE);
    if (info != 0)
        error("Cannot solve ltmatices");
} else {
    /* solve linear system */
    F77_CALL(dtpsv)(&lo, &tr, &di, &iJ, dans, dansx, &ONE FCONE FCONE FCONE);
    dansx += iJ;
    dy += iJ;
}
```

◇

Fragment referenced in 26.

⟨ *return objects* 25 ⟩ ≡

```
if (y == R_NilValue) {
    UNPROTECT(1);
    /* note: ans always includes diagonal elements */
    return(ans);
} else {
    UNPROTECT(2);
    return(ansx);
}
```

◇

Fragment referenced in 26.

We finally put everything together in a dedicated C function

⟨ *solve* 26 ⟩ ≡

```
SEXP R_ltMatrices_solve (SEXP C, SEXP y, SEXP N, SEXP J, SEXP diag, SEXP transpose)
{

    SEXP ans, ansx;
    double *dans, *dansx, *dy;
    int i, j, k, info, nrow, ncol, jj, idx, ONE = 1;

    ⟨ RC input 21a ⟩

    ⟨ C length 21b ⟩

    char di, lo = 'L', tr = 'N';
    if (Rdiag) {
        /* non-unit diagonal elements */
        di = 'N';
    } else {
        /* unit diagonal elements */
        di = 'U';
    }

    /* t(C) instead of C */
    Rboolean Rtranspose = asLogical(transpose);
    if (Rtranspose) {
        /* t(C) */
        tr = 'T';
    } else {
        /* C */
        tr = 'N';
    }

    ⟨ setup memory 23 ⟩

    /* loop over matrices, ie columns of C  / y */
    for (i = 0; i < iN; i++) {

        ⟨ copy elements 24a ⟩

        ⟨ call Lapack 24b ⟩

        /* next matrix */
        if (p > 0) {
            dans += nrow;
            dC += p;
        }
    }

    ⟨ return objects 25 ⟩
}
```

◇

Fragment referenced in 3a.

with R interface

⟨ *solve ltMatrices* 27 ⟩ ≡

```
solve.ltMatrices <- function(a, b, transpose = FALSE, ...) {

    byrow_orig <- attr(a, "byrow")

    x <- ltMatrices(a, byrow = FALSE)
    diag <- attr(x, "diag")
    d <- dim(x)
    J <- d[2L]
    dn <- dimnames(x)
    class(x) <- class(x)[-1L]
    storage.mode(x) <- "double"

    if (!missing(b)) {
        if (!is.matrix(b)) b <- matrix(b, nrow = J, ncol = ncol(x))
        stopifnot(nrow(b) == J)
        N <- ifelse(d[1L] == 1, ncol(b), d[1L])
        stopifnot(ncol(b) == N)
        storage.mode(b) <- "double"
        ret <- .Call(mvtnorm_R_ltMatrices_solve, x, b,
                     as.integer(N), as.integer(J), as.logical(diag),
                     as.logical(transpose))
        if (d[1L] == N) {
            colnames(ret) <- dn[[1L]]
        } else {
            colnames(ret) <- colnames(b)
        }
        rownames(ret) <- dn[[2L]]
        return(ret)
    }

    if (transpose) stop("cannot compute inverse of t(a)")
    ret <- try(.Call(mvtnorm_R_ltMatrices_solve, x, NULL,
                     as.integer(ncol(x)), as.integer(J), as.logical(diag),
                     as.logical(FALSE)))
    colnames(ret) <- dn[[1L]]

    if (!diag)
        ### ret always includes diagonal elements, remove here
        ret <- ret[- cumsum(c(1, J:2)), , drop = FALSE]

    ret <- ltMatrices(ret, diag = diag, byrow = FALSE, names = dn[[2L]])
    ret <- ltMatrices(ret, byrow = byrow_orig)
    return(ret)
}
```

◇

Fragment referenced in 2.

and some checks

```
> ## solve
> A <- as.array(lxn)
> a <- solve(lxn)
> a <- as.array(a)
> b <- array(apply(A, 3L, function(x) solve(x), simplify = TRUE), dim = rev(dim(lxn)))
```

```
> chk(a, b, check.attributes = FALSE)
> A <- as.array(lxd)
> a <- as.array(solve(lxd))
> b <- array(apply(A, 3L, function(x) solve(x), simplify = TRUE), dim = rev(dim(lxd)))
> chk(a, b, check.attributes = FALSE)
> chk(solve(lxn, y), Mult(solve(lxn), y))
> chk(solve(lxd, y), Mult(solve(lxd), y))
> ### recycle C
> chk(solve(lxn[1,], y), as.array(solve(lxn[1,]))[,,1] %*% y)
> chk(solve(lxn[rep(1, N),], y), solve(lxn[1,], y), check.attributes = FALSE)
> ### recycle y
> chk(solve(lxn, y[,1]), solve(lxn, y[,rep(1, N)]))
```

also for $\mathbf{C}_i^\top \mathbf{x}_i = \mathbf{y}_i$

```
> chk(solve(lxn[1,], y, transpose = TRUE),
+     t(as.array(solve(lxn[1,]))[,,1]) %*% y)
```

2.8 Crossproducts

Compute $\mathbf{C}_i\mathbf{C}_i^\top$ or $\mathrm{diag}(\mathbf{C}_i\mathbf{C}_i^\top)$ (`diag_only = TRUE`) for $i = 1, \ldots, N$. These are symmetric matrices, so we store them as a lower triangular matrix using a different class name `syMatrices`. We write one `C` function for computing $\mathbf{C}_i\mathbf{C}_i^\top$ or $\mathbf{C}_i^\top\mathbf{C}_i$ (`Rtranspose` being `TRUE`).

We differentiate between computation of the diagonal elements of the crossproduct

⟨ *first element* 28 ⟩ ≡

```
dans[0] = 1.0;
if (Rdiag)
    dans[0] = pow(dC[0], 2);
if (Rtranspose) { // crossprod
    for (k = 1; k < iJ; k++)
        dans[0] += pow(dC[IDX(k + 1, 1, iJ, Rdiag)], 2);
}
```

◇

Fragment referenced in 29, 30a.

⟨ *tcrossprod diagonal only* 29 ⟩ ≡

```
PROTECT(ans = allocMatrix(REALSXP, iJ, iN));
dans = REAL(ans);
for (n = 0; n < iN; n++) {
    ⟨ first element 28 ⟩
    for (i = 1; i < iJ; i++) {
        dans[i] = 0.0;
        if (Rtranspose) { // crossprod
            for (k = i + 1; k < iJ; k++)
                dans[i] += pow(dC[IDX(k + 1, i + 1, iJ, Rdiag)], 2);
        } else {         // tcrossprod
            for (k = 0; k < i; k++)
                dans[i] += pow(dC[IDX(i + 1, k + 1, iJ, Rdiag)], 2);
        }
        if (Rdiag) {
            dans[i] += pow(dC[IDX(i + 1, i + 1, iJ, Rdiag)], 2);
        } else {
            dans[i] += 1.0;
        }
    }
    dans += iJ;
    dC += len;
}
```

◇

Fragment referenced in 31.

and computation of the full $J \times J$ crossproduct matrix

⟨ *tcrossprod full* 30a ⟩ ≡

```
nrow = iJ * (iJ + 1) / 2;
PROTECT(ans = allocMatrix(REALSXP, nrow, iN));
dans = REAL(ans);
for (n = 0; n < INTEGER(N)[0]; n++) {
    ⟨ first element 28 ⟩
    for (i = 1; i < iJ; i++) {
        for (j = 0; j <= i; j++) {
            ix = IDX(i + 1, j + 1, iJ, 1);
            dans[ix] = 0.0;
            if (Rtranspose) { // crossprod
                for (k = i + 1; k < iJ; k++)
                    dans[ix] +=
                        dC[IDX(k + 1, i + 1, iJ, Rdiag)] *
                        dC[IDX(k + 1, j + 1, iJ, Rdiag)];
            } else {          // tcrossprod
                for (k = 0; k < j; k++)
                    dans[ix] +=
                        dC[IDX(i + 1, k + 1, iJ, Rdiag)] *
                        dC[IDX(j + 1, k + 1, iJ, Rdiag)];
            }
            if (Rdiag) {
                if (Rtranspose) {
                    dans[ix] +=
                        dC[IDX(i + 1, i + 1, iJ, Rdiag)] *
                        dC[IDX(i + 1, j + 1, iJ, Rdiag)];
                } else {
                    dans[ix] +=
                        dC[IDX(i + 1, j + 1, iJ, Rdiag)] *
                        dC[IDX(j + 1, j + 1, iJ, Rdiag)];
                }
            } else {
                if (j < i)
                    dans[ix] += dC[IDX(i + 1, j + 1, iJ, Rdiag)];
                else
                    dans[ix] += 1.0;
            }
        }
    }
    dans += nrow;
    dC += len;
}
◇
```

Fragment referenced in 31.

and put both cases together

⟨ *IDX* 30b ⟩ ≡

```
#define IDX(i, j, n, d) ((i) >= (j) ? (n) * ((j) - 1) - ((j) - 2) * ((j) - 1)/2 + (i) - (j) - (!d) * (j) :
◇
```

Fragment referenced in 31, 37a.

⟨ *tcrossprod* 31 ⟩ ≡

```
⟨ IDX 30b ⟩

SEXP R_ltMatrices_tcrossprod (SEXP C, SEXP N, SEXP J, SEXP diag, SEXP diag_only, SEXP transpose) {

    SEXP ans;
    double *dans;
    int i, j, n, k, ix, nrow;

    ⟨ RC input 21a ⟩

    Rboolean Rdiag_only = asLogical(diag_only);
    Rboolean Rtranspose = asLogical(transpose);

    if (Rdiag_only) {

        ⟨ tcrossprod diagonal only 29 ⟩

    } else {

        ⟨ tcrossprod full 30a ⟩

    }
    UNPROTECT(1);
    return(ans);
}
◇
```

Fragment referenced in 3a.

with R interface

⟨ *tcrossprod ltMatrices* 32 ⟩ ≡

```
### C %*% t(C) => returns object of class syMatrices
### diag(C %*% t(C)) => returns matrix of diagonal elements
.Tcrossprod <- function(x, diag_only = FALSE, transpose = FALSE) {

    if (!inherits(x, "ltMatrices")) {
        ret <- tcrossprod(x)
        if (diag_only) ret <- diag(ret)
        return(ret)
    }

    byrow_orig <- attr(x, "byrow")
    diag <- attr(x, "diag")
    d <- dim(x)
    J <- d[2L]
    dn <- dimnames(x)

    x <- ltMatrices(x, byrow = FALSE)
    class(x) <- class(x)[-1L]
    N <- d[1L]
    storage.mode(x) <- "double"

    ret <- .Call(mvtnorm_R_ltMatrices_tcrossprod, x, as.integer(N), as.integer(J),
                 as.logical(diag), as.logical(diag_only), as.logical(transpose))
    colnames(ret) <- dn[[1L]]
    if (diag_only) {
        rownames(ret) <- dn[[2L]]
    } else {
        ret <- ltMatrices(ret, diag = TRUE, byrow = FALSE, names = dn[[2L]])
        ret <- ltMatrices(ret, byrow = byrow_orig)
        class(ret)[1L] <- "syMatrices"
    }
    return(ret)
}
Tcrossprod <- function(x, diag_only = FALSE)
    .Tcrossprod(x = x, diag_only = diag_only, transpose = FALSE)
◇
```

Fragment referenced in 2.

We could have created yet another generic `tcrossprod`, but `base::tcrossprod` is more general and, because speed is an issue, we don't want to waste time on methods dispatch.

```
> ## Tcrossprod
> a <- as.array(Tcrossprod(lxn))
> b <- array(apply(as.array(lxn), 3L, function(x) tcrossprod(x), simplify = TRUE),
+            dim = rev(dim(lxn)))
> chk(a, b, check.attributes = FALSE)
> # diagonal elements only
> d <- Tcrossprod(lxn, diag_only = TRUE)
> chk(d, apply(a, 3, diag))
> chk(d, diagonals(Tcrossprod(lxn)))
> a <- as.array(Tcrossprod(lxd))
> b <- array(apply(as.array(lxd), 3L, function(x) tcrossprod(x), simplify = TRUE),
+            dim = rev(dim(lxd)))
> chk(a, b, check.attributes = FALSE)
```

```
> # diagonal elements only
> d <- Tcrossprod(lxd, diag_only = TRUE)
> chk(d, apply(a, 3, diag))
> chk(d, diagonals(Tcrossprod(lxd)))
```

We also add `Crossprod`, which is a call to `Tcrossprod` with the `transpose` switch turned on

⟨ *crossprod ltMatrices* 33 ⟩ ≡

```
Crossprod <- function(x, diag_only = FALSE)
    .Tcrossprod(x, diag_only = diag_only, transpose = TRUE)
```

◇

Fragment referenced in 2.

and run some checks

```
> ## Crossprod
> a <- as.array(Crossprod(lxn))
> b <- array(apply(as.array(lxn), 3L, function(x) crossprod(x), simplify = TRUE),
+            dim = rev(dim(lxn)))
> chk(a, b, check.attributes = FALSE)
> # diagonal elements only
> d <- Crossprod(lxn, diag_only = TRUE)
> chk(d, apply(a, 3, diag))
> chk(d, diagonals(Crossprod(lxn)))
> a <- as.array(Crossprod(lxd))
> b <- array(apply(as.array(lxd), 3L, function(x) crossprod(x), simplify = TRUE),
+            dim = rev(dim(lxd)))
> chk(a, b, check.attributes = FALSE)
> # diagonal elements only
> d <- Crossprod(lxd, diag_only = TRUE)
> chk(d, apply(a, 3, diag))
> chk(d, diagonals(Crossprod(lxd)))
```

2.9 Cholesky Factorisation

There might arise needs to compute the Cholesky factorisation $\mathbf{\Sigma}_i = \mathbf{C}_i\mathbf{C}_i^\top$ for multiple symmetric matrices $\mathbf{\Sigma}_i$, stored as a matrix in class `syMatrices`.

⟨ *chol syMatrices* 34 ⟩ ≡

```
chol.syMatrices <- function(x, ...) {

    byrow_orig <- attr(x, "byrow")
    dnm <- dimnames(x)
    stopifnot(attr(x, "diag"))
    d <- dim(x)

    ### x is of class syMatrices, coerse to ltMatrices first and re-arrange
    ### second
    x <- ltMatrices(unclass(x), diag = TRUE,
                    byrow = byrow_orig, names = dnm[[2L]])
    x <- ltMatrices(x, byrow = FALSE)
    class(x) <- class(x)[-1]
    storage.mode(x) <- "double"

    ret <- .Call(mvtnorm_R_syMatrices_chol, x,
                 as.integer(d[1L]), as.integer(d[2L]))
    colnames(ret) <- dnm[[1L]]

    ret <- ltMatrices(ret, diag = TRUE,
                      byrow = FALSE, names = dnm[[2L]])
    ret <- ltMatrices(ret, byrow = byrow_orig)

    return(ret)
}
```

◇

Fragment referenced in 2.

Luckily, we already have the data in the correct packed colum-major storage, so we swiftly loop over $i = 1, \ldots, N$ in C and hand over to LAPACK

⟨ *chol* 35 ⟩ ≡

```
SEXP R_syMatrices_chol (SEXP Sigma, SEXP N, SEXP J) {

    SEXP ans;
    double *dans, *dSigma;
    int iJ = INTEGER(J)[0];
    int pJ = iJ * (iJ + 1) / 2;
    int iN = INTEGER(N)[0];
    int i, j, info = 0;
    char lo = 'L';

    PROTECT(ans = allocMatrix(REALSXP, pJ, iN));
    dans = REAL(ans);
    dSigma = REAL(Sigma);

    for (i = 0; i < iN; i++) {

        /* copy data */
        for (j = 0; j < pJ; j++)
            dans[j] = dSigma[j];

        F77_CALL(dpptrf)(&lo, &iJ, dans, &info FCONE);

        if (info != 0) {
            if (info > 0)
                error("the leading minor of order %d is not positive definite",
                      info);
            error("argument %d of Lapack routine %s had invalid value",
                  -info, "dpptrf");
            }

        dSigma += pJ;
        dans += pJ;
    }
    UNPROTECT(1);
    return(ans);
}
```

◇

Fragment referenced in 3a.

This new `chol` method can be used to revert `Tcrossprod` for `ltMatrices` with and without unit diagonals:

```
> Sigma <- Tcrossprod(lxd)
> chk(chol(Sigma), lxd)
> Sigma <- Tcrossprod(lxn)
> ## Sigma and chol(Sigma) always have diagonal, lxn doesn't
> chk(as.array(chol(Sigma)), as.array(lxn))
```

2.10 Kronecker Products

We sometimes need to compute $\text{vec}(\mathbf{S})^\top(\mathbf{A}^\top \otimes \mathbf{C})$, where $\mathbf{S}$ is a lower triangular or other $J \times J$ matrix and $\mathbf{A}$ and $\mathbf{C}$ are lower triangular $J \times J$ matrices. With the "vec trick", we have $\text{vec}(\mathbf{S})^\top(\mathbf{A}^\top \otimes \mathbf{C}) = \text{vec}(\mathbf{C}^\top \mathbf{S}\mathbf{A}^\top)^\top$. The LAPACK function `dtrmm` computes products of lower triangular matrices with other matrices, so we simply call this function looping over $i = 1, \ldots, N$.

⟨ *t(C) S t(A)* 36 ⟩ ≡

```
char siR = 'R', siL = 'L', lo = 'L', tr = 'N', trT = 'T', di = 'N', trs;
double ONE = 1.0;
int iJ2 = iJ * iJ;

double tmp[iJ2];
for (j = 0; j < iJ2; j++) tmp[j] = 0.0;

ans = PROTECT(allocMatrix(REALSXP, iJ2, iN));
dans = REAL(ans);

for (i = 0; i < LENGTH(ans); i++) dans[i] = 0.0;

for (i = 0; i < iN; i++) {

    /* A := C */
    for (j = 0; j < iJ; j++) {
        for (k = 0; k <= j; k++)
            tmp[k * iJ + j] = dC[IDX(j + 1, k + 1, iJ, 1L)];
    }

    /* S was already expanded in R code; B = S */
    for (j = 0; j < iJ2; j++) dans[j] = dS[j];

    /* B := t(A) %*% B */
    trs = (RtC ? trT : tr);
    F77_CALL(dtrmm)(&siL, &lo, &trs, &di, &iJ, &iJ, &ONE, tmp, &iJ,
                    dans, &iJ FCONE FCONE FCONE FCONE);

    /* A */
    for (j = 0; j < iJ; j++) {
        for (k = 0; k <= j; k++)
            tmp[k * iJ + j] = dA[IDX(j + 1, k + 1, iJ, 1L)];
    }

    /* B := B %*% t(A) */
    trs = (RtA ? trT : tr);
    F77_CALL(dtrmm)(&siR, &lo, &trs, &di, &iJ, &iJ, &ONE, tmp, &iJ,
                    dans, &iJ FCONE FCONE FCONE FCONE);

    dans += iJ2;
    dC += p;
    dS += iJ2;
    dA += p;
}
```

◇

Fragment referenced in 37a.

⟨ *vec trick* 37a ⟩ ≡

```
⟨ IDX 30b ⟩

SEXP R_vectrick(SEXP C, SEXP N, SEXP J, SEXP S, SEXP A, SEXP diag, SEXP trans) {

    int i, j, k;
    SEXP ans;
    double *dS, *dans, *dA;

    /* note: diag is needed by this chunk but has no consequences */
    ⟨ RC input 21a ⟩
    ⟨ C length 21b ⟩
    dS = REAL(S);
    dA = REAL(A);

    Rboolean RtC = LOGICAL(trans)[0];
    Rboolean RtA = LOGICAL(trans)[1];

    ⟨ t(C) S t(A) 36 ⟩

    UNPROTECT(1);
    return(ans);
}
◇
```

Fragment referenced in 3a.

In R, we compute $\mathbf{C}^\top\mathbf{S}\mathbf{A}^\top$ by default or $\mathbf{C}\mathbf{S}\mathbf{A}^\top$ or $\mathbf{C}^\top\mathbf{S}\mathbf{A}$ or $\mathbf{C}^\top\mathbf{S}\mathbf{A}^\top$ by using the `trans` argument in `vectrick`. Argument `C` is an `ltMatrices` object

⟨ *check C argument* 37b ⟩ ≡

```
stopifnot(inherits(C, "ltMatrices"))
if (!attr(C, "diag")) diagonals(C) <- 1
C_byrow_orig <- attr(C, "byrow")
C <- ltMatrices(C, byrow = FALSE)
dC <- dim(C)
nm <- attr(C, "rcnames")
N <- dC[1L]
J <- dC[2L]
class(C) <- class(C)[-1L]
storage.mode(C) <- "double"
◇
```

Fragment referenced in 39b.

`S` can be an `ltMatrices` object or a $J^2 \times N$ matrix whose columns of vectorised $J \times J$ matrices

⟨ *check S argument* 38 ⟩ ≡

```
SltM <- inherits(S, "ltMatrices")
if (SltM) {
    if (!attr(S, "diag")) diagonals(S) <- 1
    S_byrow_orig <- attr(S, "byrow")
    stopifnot(S_byrow_orig == C_byrow_orig)
    S <- ltMatrices(S, byrow = FALSE)
    dS <- dim(S)
    stopifnot(dC[2L] == dS[2L])
    if (dC[1] != 1L) {
        stopifnot(dC[1L] == dS[1L])
    } else {
        N <- dS[1L]
    }
    ## argument A in dtrmm is not in packed form, so expand in J x J
    ## matrix
    S <- t(matrix(as.array(S), byrow = TRUE, nrow = dS[1L]))
    class(S) <- class(S)[-1L]
} else {
    stopifnot(is.matrix(S))
    stopifnot(nrow(S) == J^2)
    if (dC[1] != 1L) {
        stopifnot(dC[1L] == ncol(S))
    } else {
        N <- ncol(S)
    }
}
storage.mode(S) <- "double"
```

◇

Fragment referenced in 39b.

`A` is an `ltMatrices` object

⟨ *check A argument* 39a ⟩ ≡

```
if (missing(A)) {
    A <- C
} else {
    stopifnot(inherits(A, "ltMatrices"))
    if (!attr(A, "diag")) diagonals(A) <- 1
    A_byrow_orig <- attr(A, "byrow")
    stopifnot(C_byrow_orig == A_byrow_orig)
    A <- ltMatrices(A, byrow = FALSE)
    dA <- dim(A)
    stopifnot(dC[2L] == dA[2L])
    class(A) <- class(A)[-1L]
    storage.mode(A) <- "double"
    if (dC[1L] != dA[1L]) {
        if (dC[1L] == 1L)
            C <- C[, rep(1, N), drop = FALSE]
        if (dA[1L] == 1L)
            A <- A[, rep(1, N), drop = FALSE]
        stopifnot(ncol(A) == ncol(C))
    }
}
```

◇

Fragment referenced in 39b.

We put everything together in function `vectrick`

⟨ *kronecker vec trick* 39b ⟩ ≡

```
vectrick <- function(C, S, A, transpose = c(TRUE, TRUE)) {

    stopifnot(all(is.logical(transpose)))
    stopifnot(length(transpose) == 2L)

    ⟨ check C argument 37b ⟩
    ⟨ check S argument 38 ⟩
    ⟨ check A argument 39a ⟩

    ret <- .Call(mvtnorm_R_vectrick, C, as.integer(N), as.integer(J), S, A,
                 as.logical(TRUE), as.logical(transpose))

    if (!SltM) return(matrix(c(ret), ncol = N))

    L <- matrix(1:(J^2), nrow = J)
    ret <- ltMatrices(ret[L[lower.tri(L, diag = TRUE)],,drop = FALSE],
                      diag = TRUE, byrow = FALSE, names = nm)
    ret <- ltMatrices(ret, byrow = C_byrow_orig)
    return(ret)
}
```

◇

Fragment referenced in 2.

Here is a small example

```
> J <- 10
> d <- TRUE
```

```
> L <- diag(J)
> L[lower.tri(L, diag = d)] <- prm <- runif(J * (J + c(-1, 1)[d + 1]) / 2)
> C <- solve(L)
> D <- -kronecker(t(C), C)
> S <- diag(J)
> S[lower.tri(S, diag = TRUE)] <- x <- runif(J * (J + 1) / 2)
> SD0 <- matrix(c(S) %*% D, ncol = J)
> SD1 <- -crossprod(C, tcrossprod(S, C))
> a <- ltMatrices(C[lower.tri(C, diag = TRUE)], diag = TRUE, byrow = FALSE)
> b <- ltMatrices(x, diag = TRUE, byrow = FALSE)
> SD2 <- -vectrick(a, b, a)
> chk(SD0[lower.tri(SD0, diag = d)],
+     SD1[lower.tri(SD1, diag = d)])
> chk(SD0[lower.tri(SD0, diag = d)],
+     c(unclass(SD2)))
> ### same; but SD2 is vec(SD0)
> S <- t(matrix(as.array(b), byrow = FALSE, nrow = 1))
> SD2 <- -vectrick(a, S, a)
> chk(c(SD0), c(SD2))
> ### N > 1
> N <- 4
> prm <- runif(J * (J - 1) / 2)
> C <- ltMatrices(prm)
> S <- matrix(runif(J^2 * N), ncol = N)
> A <- vectrick(C, S, C)
> Cx <- as.array(C)[,,1]
> B <- apply(S, 2, function(x) t(Cx) %*% matrix(x, ncol = J) %*% t(Cx))
> chk(A, B)
> A <- vectrick(C, S, C, transpose = c(FALSE, FALSE))
> Cx <- as.array(C)[,,1]
> B <- apply(S, 2, function(x) Cx %*% matrix(x, ncol = J) %*% Cx)
> chk(A, B)
```

2.11 Convenience Functions

We add a few convenience functions for computing covariance matrices $\mathbf{\Sigma}_i = \mathbf{C}_i\mathbf{C}_i^\top$, precision matrices $\mathbf{P}_i = \mathbf{L}_i^\top\mathbf{L}_i$, correlation matrices $\mathbf{R}_i = \tilde{\mathbf{C}}_i\tilde{\mathbf{C}}_i^\top$ (where $\tilde{\mathbf{C}}_i = \text{diag}(\mathbf{C}_i\mathbf{C}_i^\top)^{-\frac{1}{2}}\mathbf{C}_i$), or matrices of partial correlations $\mathbf{A}_i = -\tilde{\mathbf{L}}_i^\top\tilde{\mathbf{L}}_i$ with $\tilde{\mathbf{L}}_i = \mathbf{L}_i\text{diag}(\mathbf{L}_i^\top\mathbf{L}_i)^{-\frac{1}{2}}$ from $\mathbf{L}_i$ (`invchol`) or $\mathbf{C}_i = \mathbf{L}_i^{-1}$ (`chol`) for $i = 1, \ldots, N$.

First, we set-up functions for computing $\tilde{\mathbf{C}}_i$

⟨ *D times C* 41a ⟩ ≡

```
Dchol <- function(x, D = 1 / sqrt(Tcrossprod(x, diag_only = TRUE))) {

    x <- .adddiag(x)

    byrow_orig <- attr(x, "byrow")

    x <- ltMatrices(x, byrow = TRUE)

    N <- dim(x)[1L]
    J <- dim(x)[2L]
    nm <- dimnames(x)[[2L]]

    x <- unclass(x) * D[rep(1:J, 1:J),,drop = FALSE]

    ret <- ltMatrices(x, diag = TRUE, byrow = TRUE, names = nm)
    ret <- ltMatrices(ret, byrow = byrow_orig)
    return(ret)
}
```

◇

Fragment referenced in 42.

and $\tilde{\mathbf{C}}_i^{-1} = \mathbf{L}_i \text{diag}(\mathbf{L}_i^{-1}\mathbf{L}_i^{-\top})^{\frac{1}{2}}$

⟨ *L times D* 41b ⟩ ≡

```
### invcholD = solve(Dchol)
invcholD <- function(x, D = sqrt(Tcrossprod(solve(x), diag_only = TRUE))) {

    x <- .adddiag(x)

    byrow_orig <- attr(x, "byrow")

    x <- ltMatrices(x, byrow = FALSE)

    N <- dim(x)[1L]
    J <- dim(x)[2L]
    nm <- dimnames(x)[[2L]]

    x <- unclass(x) * D[rep(1:J, J:1),,drop = FALSE]

    ret <- ltMatrices(x, diag = TRUE, byrow = FALSE, names = nm)
    ret <- ltMatrices(ret, byrow = byrow_orig)
    return(ret)
}
```

◇

Fragment referenced in 42.

and now the convenience functions are one-liners:

⟨ *convenience functions* 42 ⟩ ≡

⟨ *D times C* 41a ⟩
⟨ *L times D* 41b ⟩

```
### C -> Sigma
chol2cov <- function(x)
    Tcrossprod(x)

### L -> C
invchol2chol <- function(x)
    solve(x)

### C -> L
chol2invchol <- function(x)
    solve(x)

### L -> Sigma
invchol2cov <- function(x)
    chol2cov(invchol2chol(x))

### L -> Precision
invchol2pre <- function(x)
    Crossprod(x)

### C -> Precision
chol2pre <- function(x)
    Crossprod(chol2invchol(x))

### C -> R
chol2cor <- function(x) {
    ret <- Tcrossprod(Dchol(x))
    diagonals(ret) <- NULL
    return(ret)
}

### L -> R
invchol2cor <- function(x) {
    ret <- chol2cor(invchol2chol(x))
    diagonals(ret) <- NULL
    return(ret)
}

### L -> A
invchol2pc <- function(x) {
    ret <- -Crossprod(invcholD(x, D = 1 / sqrt(Crossprod(x, diag_only = TRUE))))
    diagonals(ret) <- 0
    ret
}

### C -> A
chol2pc <- function(x)
    invchol2pc(solve(x))
```

◇

Fragment referenced in 2.

Here are some tests

```
> prec2pc <- function(x) {
+     ret <- -cov2cor(x)
+     diag(ret) <- 0
+     ret
+ }
> L <- lxn
> Sigma <- apply(as.array(L), 3,
+                function(x) tcrossprod(solve(x)), simplify = FALSE)
> Prec <- lapply(Sigma, solve)
> Corr <- lapply(Sigma, cov2cor)
> CP <- lapply(Corr, solve)
> PC <- lapply(Prec, function(x) prec2pc(x))
> chk(unlist(Sigma), c(as.array(invchol2cov(L))),
+     check.attributes = FALSE)
> chk(unlist(Prec), c(as.array(invchol2pre(L))),
+     check.attributes = FALSE)
> chk(unlist(Corr), c(as.array(invchol2cor(L))),
+     check.attributes = FALSE)
> chk(unlist(CP), c(as.array(Crossprod(invcholD(L)))),
+     check.attributes = FALSE)
> chk(unlist(PC), c(as.array(invchol2pc(L))),
+     check.attributes = FALSE)

> C <- lxn
> Sigma <- apply(as.array(C), 3,
+                function(x) tcrossprod(x), simplify = FALSE)
> Prec <- lapply(Sigma, solve)
> Corr <- lapply(Sigma, cov2cor)
> CP <- lapply(Corr, solve)
> PC <- lapply(Prec, function(x) prec2pc(x))
> chk(unlist(Sigma), c(as.array(chol2cov(C))),
+     check.attributes = FALSE)
> chk(unlist(Prec), c(as.array(chol2pre(C))),
+     check.attributes = FALSE)
> chk(unlist(Corr), c(as.array(chol2cor(C))),
+     check.attributes = FALSE)
> chk(unlist(CP), c(as.array(Crossprod(solve(Dchol(C))))),
+     check.attributes = FALSE)
> chk(unlist(PC), c(as.array(chol2pc(C))),
+     check.attributes = FALSE)

> L <- lxd
> Sigma <- apply(as.array(L), 3,
+                function(x) tcrossprod(solve(x)), simplify = FALSE)
> Prec <- lapply(Sigma, solve)
> Corr <- lapply(Sigma, cov2cor)
> CP <- lapply(Corr, solve)
> PC <- lapply(Prec, function(x) prec2pc(x))
> chk(unlist(Sigma), c(as.array(invchol2cov(L))),
+     check.attributes = FALSE)
> chk(unlist(Prec), c(as.array(invchol2pre(L))),
+     check.attributes = FALSE)
> chk(unlist(Corr), c(as.array(invchol2cor(L))),
+     check.attributes = FALSE)
```

```
> chk(unlist(CP), c(as.array(Crossprod(invcholD(L)))),
+     check.attributes = FALSE)
> chk(unlist(PC), c(as.array(invchol2pc(L))),
+     check.attributes = FALSE)

> C <- lxd
> Sigma <- apply(as.array(C), 3,
+                function(x) tcrossprod(x), simplify = FALSE)
> Prec <- lapply(Sigma, solve)
> Corr <- lapply(Sigma, cov2cor)
> CP <- lapply(Corr, solve)
> PC <- lapply(Prec, function(x) prec2pc(x))
> chk(unlist(Sigma), c(as.array(chol2cov(C))),
+     check.attributes = FALSE)
> chk(unlist(Prec), c(as.array(chol2pre(C))),
+     check.attributes = FALSE)
> chk(unlist(Corr), c(as.array(chol2cor(C))),
+     check.attributes = FALSE)
> chk(unlist(CP), c(as.array(Crossprod(solve(Dchol(C))))),
+     check.attributes = FALSE)
> chk(unlist(PC), c(as.array(chol2pc(C))),
+     check.attributes = FALSE)
```

⟨ *aperm* 44 ⟩ ≡

```
aperm.ltMatrices <- function(a, perm, chol = FALSE, ...) {

    if (chol) { ### a is Cholesky of covariance
        Sperm <- chol2cov(a)[,perm]
        return(chol(Sperm))
    }

    Sperm <- invchol2cov(a)[,perm]
    chol2invchol(chol(Sperm))
}
```

◇

Fragment referenced in 2.

```
> L <- lxn
> J <- dim(L)[2L]
> Lp <- aperm(a = L, perm = p <- sample(1:J), chol = FALSE)
> chk(invchol2cov(L)[,p], invchol2cov(Lp))
> C <- lxn
> J <- dim(C)[2L]
> Cp <- aperm(a = C, perm = p <- sample(1:J), chol = TRUE)
> chk(chol2cov(C)[,p], chol2cov(Cp))
```

2.12 Marginal and Conditional Normal Distributions

Marginal and conditional distributions from distributions $\mathbf{Y}_i \sim \mathbb{N}_J(\mathbf{0}_J, \mathbf{C}_i\mathbf{C}_i^\top)$ (`chol` argument for $\mathbf{C}_i$ for $i = 1, \ldots, N$) or $\mathbf{Y}_i \sim \mathbb{N}_J(\mathbf{0}_J, \mathbf{L}_i^{-1}\mathbf{L}_i^{-\top})$ (`invchol` argument for $\mathbf{L}_i$ for $i = 1, \ldots, N$) shall be computed.

⟨ *mc input checks* 45a ⟩ ≡

```
stopifnot(xor(missing(chol), missing(invchol)))
x <- if (missing(chol)) invchol else chol

stopifnot(inherits(x, "ltMatrices"))

N <- dim(x)[1L]
J <- dim(x)[2L]
if (is.character(which)) which <- match(which, dimnames(x)[[2L]])
stopifnot(all(which %in% 1:J))
```

◇

Fragment referenced in 45b, 47b.

The first j marginal distributions can be obtained from subsetting $\mathbf{C}$ or $\mathbf{L}$ directly. Arbitrary marginal distributions are based on the corresponding subset of the covariance matrix for which we compute a corresponding Cholesky factor (such that we can use `lpmvnorm` later on).

⟨ *marginal* 45b ⟩ ≡

```
marg_mvnorm <- function(chol, invchol, which = 1L) {

    ⟨ mc input checks 45a ⟩

    if (which[1] == 1L && (length(which) == 1L ||
                           all(diff(which) == 1L))) {
        ### which is 1:j
        tmp <- x[,which]
    } else {
        if (missing(chol)) x <- solve(x)
        tmp <- base::chol(Tcrossprod(x)[,which])
        if (missing(chol)) tmp <- solve(tmp)
    }

    if (missing(chol))
        ret <- list(invchol = tmp)
    else
        ret <- list(chol = tmp)

    ret
}
```

◇

Fragment referenced in 2.

We compute conditional distributions from the precision matrices $\mathbf{\Sigma}_i^{-1} = \mathbf{P}_i = \mathbf{L}_i^\top \mathbf{L}_i$ (we omit the i index from now on). For an arbitrary subset $\mathbf{j} \subset \{1, \ldots, J\}$, the conditional distribution of $\mathbf{Y}_{-\mathbf{j}}$ given $\mathbf{Y}_\mathbf{j} = \mathbf{y}_\mathbf{j}$ is

$$\mathbf{Y}_{-\mathbf{j}} \mid \mathbf{Y}_\mathbf{j} = \mathbf{y}_\mathbf{j} \sim \mathbb{N}_{|\mathbf{j}|} \left(-\mathbf{P}^{-1}_{-\mathbf{j},-\mathbf{j}} \mathbf{P}_{-\mathbf{j},\mathbf{j}} \mathbf{y}_\mathbf{j}, \mathbf{P}^{-1}_{-\mathbf{j},-\mathbf{j}} \right)$$

and we return a Cholesky factor $\tilde{\mathbf{C}}$ such that $\mathbf{P}^{-1}_{-\mathbf{j},-\mathbf{j}} = \tilde{\mathbf{C}} \tilde{\mathbf{C}}^\top$ (if `chol` was given) or $\tilde{\mathbf{L}} = \tilde{\mathbf{C}}^{-1}$ (if `invchol` was given).

We can implement this as

⟨ *cond general* 46 ⟩ ≡

```
stopifnot(!center)

if (!missing(chol)) ### chol is C = Cholesky of covariance
    P <- Crossprod(solve(chol)) ### P = t(L) %*% L with L = C^-1
else                ### invcol is L = Cholesky of precision
    P <- Crossprod(invchol)

Pw <- P[, -which]
chol <- solve(base::chol(Pw))
Pa <- as.array(P)
Sa <- as.array(S <- Crossprod(chol))
if (dim(chol)[1L] == 1L) {
   Pa <- Pa[,,1]
   Sa <- Sa[,,1]
   mean <- -Sa %*% Pa[-which, which, drop = FALSE] %*% given
} else {
   if (ncol(given) == N) {
       mean <- sapply(1:N, function(i)
           -Sa[,,i] %*% Pa[-which,which,i] %*% given[,i,drop = FALSE])
   } else {  ### compare to Mult() with ncol(y) !%in% (1, N)
       mean <- sapply(1:N, function(i)
           -Sa[,,i] %*% Pa[-which,which,i] %*% given)
   }
}
```

◇

Fragment referenced in 47b.

If $\mathbf{j} = \{1, \ldots, j < J\}$ and $\mathbf{L}$ is given, computations simplify a lot because the conditional precision matrix is

$$\mathbf{P}_{-\mathbf{j},-\mathbf{j}} = (\mathbf{L}^\top \mathbf{L})_{-\mathbf{j},-\mathbf{j}} = \mathbf{L}^\top_{-\mathbf{j},-\mathbf{j}} \mathbf{L}_{-\mathbf{j},-\mathbf{j}}$$

and thus we return $\tilde{\mathbf{L}} = \mathbf{L}_{-\mathbf{j},-\mathbf{j}}$ (if `invchol` was given) or $\tilde{\mathbf{C}} = \mathbf{L}^{-1}_{-\mathbf{j},-\mathbf{j}}$ (if `chol` was given). The conditional mean is

$$\begin{aligned} -\mathbf{P}^{-1}_{-\mathbf{j},-\mathbf{j}} \mathbf{P}_{-\mathbf{j},\mathbf{j}} \mathbf{y}_\mathbf{j} &= -\mathbf{L}^{-1}_{-\mathbf{j},-\mathbf{j}} \mathbf{L}^{-\top}_{-\mathbf{j},-\mathbf{j}} \mathbf{L}^\top_{-\mathbf{j},-\mathbf{j}} \mathbf{L}_{-\mathbf{j},\mathbf{j}} \mathbf{y}_\mathbf{j} \\ &= -\mathbf{L}^{-1}_{-\mathbf{j},-\mathbf{j}} \mathbf{L}_{-\mathbf{j},\mathbf{j}} \mathbf{y}_\mathbf{j}. \end{aligned}$$

We sometimes, for example when scores with respect to $\mathbf{L}^{-1}_{-\mathbf{j},-\mathbf{j}}$ shall be computed in `slpmvnorm`, need the negative rescaled mean $\mathbf{L}_{-\mathbf{j},\mathbf{j}} \mathbf{y}_\mathbf{j}$ and the `center = TRUE` argument triggers this values to be returned.

The implementation reads

⟨ *cond simple* 47a ⟩ ≡

```
if (which[1] == 1L && (length(which) == 1L ||
                       all(diff(which) == 1L))) {
    ### which is 1:j
    L <- if (missing(invchol)) solve(chol) else invchol
    tmp <- matrix(0, ncol = ncol(given), nrow = J - length(which))
    centerm <- Mult(L, rbind(given, tmp))[-which,,drop = FALSE]
    L <- L[,-which]
    if (missing(invchol)) {
        if (center)
            return(list(center = centerm, chol = solve(L)))
        return(list(mean = -solve(L, centerm), chol = solve(L)))
    }
    if (center)
        return(list(center = centerm, invchol = L))
    return(list(mean = -solve(L, centerm), invchol = L))
}
```

◇

Fragment referenced in 47b.

⟨ *conditional* 47b ⟩ ≡

```
cond_mvnorm <- function(chol, invchol, which_given = 1L, given, center = FALSE) {

    which <- which_given
    ⟨ mc input checks 45a ⟩

    if (N == 1) N <- NCOL(given)
    stopifnot(is.matrix(given) && nrow(given) == length(which))

    ⟨ cond simple 47a ⟩

    ⟨ cond general 46 ⟩

    chol <- base::chol(S)
    if (missing(invchol))
        return(list(mean = mean, chol = chol))

    return(list(mean = mean, invchol = solve(chol)))
}
```

◇

Fragment referenced in 2.

Let's check this against the commonly used formula based on the covariance matrix, first for the marginal distribution

```
> Sigma <- Tcrossprod(lxd)
> j <- 1:3
> chk(Sigma[,j], Tcrossprod(marg_mvnorm(chol = lxd, which = j)$chol))
> j <- 2:4
> chk(Sigma[,j], Tcrossprod(marg_mvnorm(chol = lxd, which = j)$chol))
> Sigma <- Tcrossprod(solve(lxd))
> j <- 1:3
```

```
> chk(Sigma[,j], Tcrossprod(solve(marg_mvnorm(invchol = lxd, which = j)$invchol)))
> j <- 2:4
> chk(Sigma[,j], Tcrossprod(solve(marg_mvnorm(invchol = lxd, which = j)$invchol)))
```

and then for conditional distributions. The general case is

```
> Sigma <- as.array(Tcrossprod(lxd))[,,1]
> j <- 2:4
> y <- matrix(c(-1, 2, 1), nrow = 3)
> cm <- Sigma[-j, j,drop = FALSE] %*% solve(Sigma[j,j]) %*%  y
> cS <- Sigma[-j, -j] - Sigma[-j,j,drop = FALSE] %*%
+       solve(Sigma[j,j]) %*% Sigma[j,-j,drop = FALSE]
> cmv <- cond_mvnorm(chol = lxd[1,], which = j, given = y)
> chk(cm, cmv$mean)
> chk(cS, as.array(Tcrossprod(cmv$chol))[,,1])
> Sigma <- as.array(Tcrossprod(solve(lxd)))[,,1]
> j <- 2:4
> y <- matrix(c(-1, 2, 1), nrow = 3)
> cm <- Sigma[-j, j,drop = FALSE] %*% solve(Sigma[j,j]) %*%  y
> cS <- Sigma[-j, -j] - Sigma[-j,j,drop = FALSE] %*%
+       solve(Sigma[j,j]) %*% Sigma[j,-j,drop = FALSE]
> cmv <- cond_mvnorm(invchol = lxd[1,], which = j, given = y)
> chk(cm, cmv$mean)
> chk(cS, as.array(Tcrossprod(solve(cmv$invchol)))[,,1])
```

and the simple case is

```
> Sigma <- as.array(Tcrossprod(lxd))[,,1]
> j <- 1:3
> y <- matrix(c(-1, 2, 1), nrow = 3)
> cm <- Sigma[-j, j,drop = FALSE] %*% solve(Sigma[j,j]) %*%  y
> cS <- Sigma[-j, -j] - Sigma[-j,j,drop = FALSE] %*%
+       solve(Sigma[j,j]) %*% Sigma[j,-j,drop = FALSE]
> cmv <- cond_mvnorm(chol = lxd[1,], which = j, given = y)
> chk(c(cm), c(cmv$mean))
> chk(cS, as.array(Tcrossprod(cmv$chol))[,,1])
> Sigma <- as.array(Tcrossprod(solve(lxd)))[,,1]
> j <- 1:3
> y <- matrix(c(-1, 2, 1), nrow = 3)
> cm <- Sigma[-j, j,drop = FALSE] %*% solve(Sigma[j,j]) %*%  y
> cS <- Sigma[-j, -j] - Sigma[-j,j,drop = FALSE] %*%
+       solve(Sigma[j,j]) %*% Sigma[j,-j,drop = FALSE]
> cmv <- cond_mvnorm(invchol = lxd[1,], which = j, given = y)
> chk(c(cm), c(cmv$mean))
> chk(cS, as.array(Tcrossprod(solve(cmv$invchol)))[,,1])
```

2.13 Application Example

Let's say we have $\mathbf{Y}_i \sim \mathbb{N}_J(\mathbf{0}_J, \mathbf{C}_i\mathbf{C}_i^\top)$ for $i = 1, \ldots, N$ and we know the Cholesky factors $\mathbf{L}_i = \mathbf{C}_i^{-1}$ of the N precision matrices $\Sigma^{-1} = \mathbf{L}_i\mathbf{L}_i^\top$. We generate $\mathbf{Y}_i = \mathbf{L}_i^{-1}\mathbf{Z}_i$ from $\mathbf{Z}_i \sim \mathbb{N}_J(\mathbf{0}_J, \mathbf{I}_J)$. Evaluating the corresponding log-likelihood is now straightforward and fast, compared to repeated calls to `dmvnorm`

```
> N <- 1000
> J <- 50
```

```
> lt <- ltMatrices(matrix(runif(N * J * (J + 1) / 2) + 1, ncol = N),
+                  diag = TRUE, byrow = FALSE)
> Z <- matrix(rnorm(N * J), ncol = N)
> Y <- solve(lt, Z)
> ll1 <- sum(dnorm(Mult(lt, Y), log = TRUE)) + sum(log(diagonals(lt)))
> S <- as.array(Tcrossprod(solve(lt)))
> ll2 <- sum(sapply(1:N, function(i) dmvnorm(x = Y[,i], sigma = S[,,i], log = TRUE)))
> chk(ll1, ll2)
```

The `dmvnorm` function now also has `chol` and `invchol` arguments such that we can use

```
> ll3 <- sum(dmvnorm(Y, invchol = lt, log = TRUE))
> chk(ll1, ll3)
```

Note that argument `x` in `dmvnorm` is an $N \times J$ matrix when `sigma` is given (the traditional interface) BUT expects an $J \times \mathbb{N}$ matrix when `chol` or `invchol` are specified. The reason is that `Mult` or `solve` work with $J \times \mathbb{N}$ matrices and we want to avoid matrix transposes.

Sometimes it is preferable to split the joint distribution into a marginal distribution of some elements and the conditional distribution given these elements. The joint density is, of course, the product of the marginal and conditional densities and we can check if this works for our example by

```
> ## marginal of and conditional on these
> (j <- 1:5 * 10)

[1] 10 20 30 40 50

> md <- marg_mvnorm(invchol = lt, which = j)
> cd <- cond_mvnorm(invchol = lt, which = j, given = Y[j,])
> ll3 <- sum(dnorm(Mult(md$invchol, Y[j,]), log = TRUE)) +
+        sum(log(diagonals(md$invchol))) +
+        sum(dnorm(Mult(cd$invchol, Y[-j,] - cd$mean), log = TRUE)) +
+        sum(log(diagonals(cd$invchol)))
> chk(ll1, ll3)
```

Chapter 3

Multivariate Normal Log-likelihoods

We now discuss code for evaluating the log-likelihood

$$\sum_{i=1}^{N} \log(p_i(\mathbf{C}_i \mid \mathbf{a}_i, \mathbf{b}_i))$$

This is relatively simple to achieve using the existing `pmvnorm`, so a prototype might look like

⟨ *lpmvnormR* 50 ⟩ ≡

```
library("mvtnorm")
lpmvnormR <- function(lower, upper, mean = 0, center = NULL, chol, logLik = TRUE, ...) {

    ⟨ input checks 52 ⟩

    sigma <- Tcrossprod(chol)
    S <- as.array(sigma)
    idx <- 1

    ret <- error <- numeric(N)
    for (i in 1:N) {
        if (dim(sigma)[[1L]] > 1) idx <- i
        tmp <- pmvnorm(lower = lower[,i], upper = upper[,i], sigma = S[,,idx], ...)
        ret[i] <- tmp
        error[i] <- attr(tmp, "error")
    }
    attr(ret, "error") <- error

    if (logLik)
        return(sum(log(pmax(ret, .Machine$double.eps))))

    ret
}
```

◇

Fragment never referenced.

However, the underlying FORTRAN code first computes the Cholesky factor based on the covariance matrix, which is clearly a waste of time. Repeated calls to FORTRAN also cost some time. The code (based on and evaluated in Genz and Bretz, 2002) implements a specific form of quasi-Monte-Carlo integration without allowing the user to change the scheme (or to fall-back to simple Monte-Carlo). We therefore

implement our own, and simplistic version, with the aim to speed-things up such that maximum-likelihood estimation becomes a bit faster.

Let's look at an example first. This code estimates $p_1, \ldots, p_{10}$ for a 5-dimensional normal

```
> J <- 5
> N <- 10
> x <- matrix(runif(N * J * (J + 1) / 2), ncol = N)
> lx <- ltMatrices(x, byrow = TRUE, diag = TRUE)
> a <- matrix(runif(N * J), nrow = J) - 2
> a[sample(J * N)[1:2]] <- -Inf
> b <- a + 2 + matrix(runif(N * J), nrow = J)
> b[sample(J * N)[1:2]] <- Inf
> (phat <- c(lpmvnormR(a, b, chol = lx, logLik = FALSE)))

 [1] 0.2369329 0.2337179 0.2842052 0.3915213 0.4662496 0.0000000 0.5900784
 [8] 0.4618524 0.4872819 0.0000000
```

We want to achieve the same result a bit more general and a bit faster.

3.1 Algorithm

"lpmvnorm.R" 51a≡

⟨ *R Header* 83 ⟩
⟨ *lpmvnorm* 61 ⟩
⟨ *slpmvnorm* 72 ⟩
◇

"lpmvnorm.c" 51b≡

⟨ *C Header* 84 ⟩
```
#include <R.h>
#include <Rmath.h>
#include <Rinternals.h>
#include <Rdefines.h>
#include <Rconfig.h>
#include <R_ext/BLAS.h> /* for dtrmm */
```
⟨ *pnorm fast* 56b ⟩
⟨ *pnorm slow* 56c ⟩
⟨ *R lpmvnorm* 59 ⟩
⟨ *R slpmvnorm* 69 ⟩
◇

We implement the algorithm described by Genz (1992). The key point here is that the original J-dimensional problem (1.1) is transformed into an integral over $[0, 1]^{J-1}$.

For each $i = 1, \ldots, N$, do

1. Input $\mathbf{C}_i$ (`chol`), $\mathbf{a}_i$ (`lower`), $\mathbf{b}_i$ (`upper`), and control parameters α, ϵ, and $M_{\max}$ (`M`).

⟨ *input checks* 52 ⟩ ≡

```
if (!is.matrix(lower)) lower <- matrix(lower, ncol = 1)
if (!is.matrix(upper)) upper <- matrix(upper, ncol = 1)
stopifnot(isTRUE(all.equal(dim(lower), dim(upper))))

stopifnot(inherits(chol, "ltMatrices"))
byrow_orig <- attr(chol, "byrow")
chol <- ltMatrices(chol, byrow = TRUE)
d <- dim(chol)
### allow single matrix C
N <- ifelse(d[1L] == 1, ncol(lower), d[1L])
J <- d[2L]

stopifnot(nrow(lower) == J && ncol(lower) == N)
stopifnot(nrow(upper) == J && ncol(upper) == N)
if (is.matrix(mean))
    stopifnot(nrow(mean) == J && ncol(mean) == N)

lower <- lower - mean
upper <- upper - mean

if (!is.null(center)) {
    if (!is.matrix(center)) center <- matrix(center, ncol = 1)
    stopifnot(nrow(center) == J && ncol(center == N))
}
```

◇

Fragment referenced in 50, 61, 72.

2. Standardise integration limits $a_j^{(i)}/c_{jj}^{(i)}$, $b_j^{(i)}/c_{jj}^{(i)}$, and rows $c_{\jmath j}^{(i)}/c_{jj}^{(i)}$ for $1 \leq \jmath < j < J$.

⟨ *standardise* 53a ⟩ ≡

```
if (attr(chol, "diag")) {
    ### diagonals returns J x N and lower/upper are J x N, so
    ### elementwise standardisation is simple
    dchol <- diagonals(chol)
    ### zero diagonals not allowed
    stopifnot(all(abs(dchol) > (.Machine$double.eps)))
    ac <- lower / c(dchol)
    bc <- upper / c(dchol)
    ### the following is equivalent to Dchol(chol, D = 1 / dchol)
    ### but returns an object without diagonal elements (expected by
    ### R_lpmvnorm)
    ### CHECK if dimensions are correct
    C <- unclass(chol) / c(dchol[rep(1:J, 1:J),])
    if (J > 1) ### else: univariate problem; C is no longer used
        C <- ltMatrices(C[-cumsum(c(1, 2:J)), , drop = FALSE],
                        byrow = TRUE, diag = FALSE)
} else {
    ac <- lower
    bc <- upper
    C <- ltMatrices(chol, byrow = TRUE)
}
uC <- unclass(C)
◇
```

Fragment referenced in 61, 72.

3. Initialise intsum = varsum = 0, $M = 0$, $d_1 = \Phi\left(a_1^{(i)}\right)$, $e_1 = \Phi\left(b_1^{(i)}\right)$ and $f_1 = e_1 - d_1$.

⟨ *initialisation* 53b ⟩ ≡

```
x0 = 0.0;
if (LENGTH(center))
    x0 = -dcenter[0];
d0 = pnorm_ptr(da[0], x0);
e0 = pnorm_ptr(db[0], x0);
emd0 = e0 - d0;
f0 = emd0;
intsum = (iJ > 1 ? 0.0 : f0);
◇
```

Fragment referenced in 59, 69.

4. Repeat

⟨ *init logLik loop* 53c ⟩ ≡

```
d = d0;
f = f0;
emd = emd0;
start = 0;
◇
```

Fragment referenced in 59, 64c.

(a) Generate uniform $w_1, \ldots, w_{J-1} \in [0, 1]$.

(b) For $j = 2, \ldots, J$ set

$$y_{j-1} \quad = \quad \Phi^{-1}\left(d_{j-1} + w_{j-1}(e_{j-1} - d_{j-1})\right)$$

We either generate w_{j-1} on the fly or use pre-computed weights (`w`).

⟨ *compute* y 54a ⟩ ≡

```
Wtmp = (W == R_NilValue ? unif_rand() : dW[j - 1]);
tmp = d + Wtmp * emd;
if (tmp < dtol) {
    y[j - 1] = q0;
} else {
    if (tmp > mdtol)
        y[j - 1] = -q0;
    else
        y[j - 1] = qnorm(tmp, 0.0, 1.0, 1L, 0L);
}
```

◇

Fragment referenced in 55b, 68a.

$$x_{j-1} \quad = \quad \sum_{j=1}^{j-1} c_{jj}^{(i)} y_j$$

⟨ *compute* x 54b ⟩ ≡

```
x = 0.0;
if (LENGTH(center)) {
    for (k = 0; k < j; k++)
        x += dC[start + k] * (y[k] - dcenter[k]);
    x -= dcenter[j];
} else {
    for (k = 0; k < j; k++)
        x += dC[start + k] * y[k];
}
```

◇

Fragment referenced in 55b, 68a.

$$\begin{aligned} d_j &= \Phi\left(a_j^{(i)} - x_{j-1}\right) \\ e_j &= \Phi\left(b_j^{(i)} - x_{j-1}\right) \end{aligned}$$

⟨ *update* d, e 54c ⟩ ≡

```
d = pnorm_ptr(da[j], x);
e = pnorm_ptr(db[j], x);
emd = e - d;
```

◇

Fragment referenced in 55b, 68a.

$$f_j \quad = \quad (e_j - d_j) f_{j-1}.$$

⟨ *update* f 55a ⟩ ≡

```
start += j;
f *= emd;
```
◇

Fragment referenced in 55b, 68a.

We put everything together in a loop starting with the second dimension

⟨ *inner logLik loop* 55b ⟩ ≡

```
for (j = 1; j < iJ; j++) {
    ⟨ compute y 54a ⟩
    ⟨ compute x 54b ⟩
    ⟨ update d, e 54c ⟩
    ⟨ update f 55a ⟩
}
```
◇

Fragment referenced in 59.

(c) Set intsum = intsum+f_J, varsum = varsum+f_J^2, $M = M+1$, and error = $\sqrt{(\text{varsum}/M - (\text{intsum}/M)^2)/M}$.

⟨ *increment* 55c ⟩ ≡

```
intsum += f;
```
◇

Fragment referenced in 59.

We refrain from early stopping and error estimation.

Until error $< \epsilon$ or $M = M_{\max}$

5. Output $\hat{p}_i = \text{intsum}/M$.

We return $\log \hat{p}_i$ for each i, or we immediately sum-up over i.

⟨ *output* 55d ⟩ ≡

```
dans[0] += (intsum < dtol ? l0 : log(intsum)) - lM;
if (!RlogLik)
    dans += 1L;
```
◇

Fragment referenced in 59.

and move on to the next observation (note that `p` might be 0, in case $\mathbf{C}_i \equiv \mathbf{C}$).

⟨ *move on* 56a ⟩ ≡

```
da += iJ;
db += iJ;
dC += p;
if (LENGTH(center)) dcenter += iJ;
◇
```

Fragment referenced in 59, 69.

It turned out that calls to `pnorm` are expensive, so a slightly faster alternative (suggested by Matić et al., 2018) can be used (`fast = TRUE` in the calls to `lpmvnorm` and `slpmvnorm`):

⟨ *pnorm fast* 56b ⟩ ≡

```
/* see https://doi.org/10.2139/ssrn.2842681  */
const double g2 =  -0.0150234471495426236132;
const double g4 = 0.00066609851170101874728 9;
const double g6 = 5.0793732451898110369 4e-06;
const double g8 = -2.9234527367319462776 2e-06;
const double g10 = 1.3479773351698920436 1e-07;
const double m2dpi = -2.0 / M_PI; //3.141592653589793115998;

double C_pnorm_fast (double x, double m) {

    double tmp, ret;
    double x2, x4, x6, x8, x10;

    if (R_FINITE(x)) {
        x = x - m;
        x2 = x * x;
        x4 = x2 * x2;
        x6 = x4 * x2;
        x8 = x6 * x2;
        x10 = x8 * x2;
        tmp = 1 + g2 * x2 + g4 * x4 + g6 * x6  + g8 * x8 + g10 * x10;
        tmp = m2dpi * x2 * tmp;
        ret = .5 + ((x > 0) - (x < 0)) * sqrt(1 - exp(tmp)) / 2.0;
    } else {
        ret = (x > 0 ? 1.0 : 0.0);
    }
    return(ret);
}
◇
```

Fragment referenced in 51b.

⟨ *pnorm slow* 56c ⟩ ≡

```
double C_pnorm_slow (double x, double m) {
    return(pnorm(x, m, 1.0, 1L, 0L));
}
◇
```

Fragment referenced in 51b.

The `fast` argument can be used to switch on the faster but less accurate version of `pnorm`

⟨ *pnorm* 57a ⟩ ≡

```
Rboolean Rfast = asLogical(fast);
double (*pnorm_ptr)(double, double) = C_pnorm_slow;
if (Rfast)
    pnorm_ptr = C_pnorm_fast;
◇
```

Fragment referenced in 59, 69.

We allow a new set of weights for each observation or one set for all observations. In the former case, the number of columns is $M \times N$ and in the latter just M.

⟨ *W length* 57b ⟩ ≡

```
int pW = 0;
if (W != R_NilValue) {
    if (LENGTH(W) == (iJ - 1) * iM) {
        pW = 0;
    } else {
        if (LENGTH(W) != (iJ - 1) * iN * iM)
            error("Length of W incorrect");
        pW = 1;
    }
    dW = REAL(W);
}
◇
```

Fragment referenced in 59, 69.

⟨ *dimensions* 57c ⟩ ≡

```
int iM = INTEGER(M)[0];
int iN = INTEGER(N)[0];
int iJ = INTEGER(J)[0];

da = REAL(a);
db = REAL(b);
dC = REAL(C);
dW = REAL(C); // make -Wmaybe-uninitialized happy

if (LENGTH(C) == iJ * (iJ - 1) / 2)
    p = 0;
else
    p = LENGTH(C) / iN;
◇
```

Fragment referenced in 59, 69.

⟨ *setup return object* 58a ⟩ ≡

```
len = (RlogLik ? 1 : iN);
PROTECT(ans = allocVector(REALSXP, len));
dans = REAL(ans);
for (int i = 0; i < len; i++)
    dans[i] = 0.0;
```

◇

Fragment referenced in 59.

The case $J = 1$ does not loop over M

⟨ *univariate problem* 58b ⟩ ≡

```
if (iJ == 1) {
    iM = 0;
    lM = 0.0;
} else {
    lM = log((double) iM);
}
```

◇

Fragment referenced in 59.

⟨ *init center* 58c ⟩ ≡

```
dcenter = REAL(center);
if (LENGTH(center)) {
    if (LENGTH(center) != iN * iJ)
        error("incorrect dimensions of center");
}
```

◇

Fragment referenced in 59, 69.

We put the code together in a dedicated C function

⟨ *R lpmvnorm* 59 ⟩ ≡

```
SEXP R_lpmvnorm(SEXP a, SEXP b, SEXP C, SEXP center, SEXP N, SEXP J,
                SEXP W, SEXP M, SEXP tol, SEXP logLik, SEXP fast) {

    SEXP ans;
    double *da, *db, *dC, *dW, *dans, dtol = REAL(tol)[0];
    double *dcenter;
    double mdtol = 1.0 - dtol;
    double d0, e0, emd0, f0, q0, l0, lM, x0, intsum;
    int p, len;

    Rboolean RlogLik = asLogical(logLik);

    ⟨ pnorm 57a ⟩

    ⟨ dimensions 57c ⟩

    ⟨ W length 57b ⟩

    ⟨ init center 58c ⟩

    int start, j, k;
    double tmp, Wtmp, e, d, f, emd, x, y[iJ - 1];

    ⟨ setup return object 58a ⟩

    q0 = qnorm(dtol, 0.0, 1.0, 1L, 0L);
    l0 = log(dtol);

    ⟨ univariate problem 58b ⟩

    if (W == R_NilValue)
        GetRNGstate();

    for (int i = 0; i < iN; i++) {

        x0 = 0;
        ⟨ initialisation 53b ⟩

        if (W != R_NilValue && pW == 0)
            dW = REAL(W);

        for (int m = 0; m < iM; m++) {

            ⟨ init logLik loop 53c ⟩

            ⟨ inner logLik loop 55b ⟩

            ⟨ increment 55c ⟩

            if (W != R_NilValue)
                dW += iJ - 1;
        }

        ⟨ output 55d ⟩

        ⟨ move on 56a ⟩
    }

    if (W == R_NilValue)
        PutRNGstate();

    UNPROTECT(1);
    return(ans);
}
```

◇

Fragment referenced in 51b.

The R user interface consists of some checks and a call to C. Note that we need to specify both `w` and `M` in case we want a new set of weights for each observation.

⟨ *init random seed, reset on exit* 60a ⟩ ≡

```
### from stats:::simulate.lm
if (!exists(".Random.seed", envir = .GlobalEnv, inherits = FALSE))
    runif(1)
if (is.null(seed))
    RNGstate <- get(".Random.seed", envir = .GlobalEnv)
else {
    R.seed <- get(".Random.seed", envir = .GlobalEnv)
    set.seed(seed)
    RNGstate <- structure(seed, kind = as.list(RNGkind()))
    on.exit(assign(".Random.seed", R.seed, envir = .GlobalEnv))
}
```

◇

Fragment referenced in 61, 72.

⟨ *check and / or set integration weights* 60b ⟩ ≡

```
if (!is.null(w) && J > 1) {
    stopifnot(is.matrix(w))
    stopifnot(nrow(w) == J - 1)
    if (is.null(M))
        M <- ncol(w)
    stopifnot(ncol(w) %in% c(M, M * N))
    storage.mode(w) <- "double"
} else {
    if (J > 1) {
        if (is.null(M)) stop("either w or M must be specified")
    } else {
        M <- 1L
    }
}
```

◇

Fragment referenced in 61, 72.

Sometimes we want to evaluate the log-likelihood based on $\mathbf{L} = \mathbf{C}^{-1}$, the Cholesky factor of the precision (not the covariance) matrix. In this case, we explicitly invert $\mathbf{L}$ to give $\mathbf{C}$ (both matrices are lower triangular, so this is fast).

⟨ *Cholesky of precision* 60c ⟩ ≡

```
stopifnot(xor(missing(chol), missing(invchol)))
if (missing(chol)) chol <- solve(invchol)
```

◇

Fragment referenced in 61, 72.

⟨ *lpmvnorm* 61 ⟩ ≡

```
lpmvnorm <- function(lower, upper, mean = 0, center = NULL, chol, invchol, logLik = TRUE, M = NULL,
                     w = NULL, seed = NULL, tol = .Machine$double.eps, fast = FALSE) {

    ⟨ init random seed, reset on exit 60a ⟩

    ⟨ Cholesky of precision 60c ⟩

    ⟨ input checks 52 ⟩

    ⟨ standardise 53a ⟩

    ⟨ check and / or set integration weights 60b ⟩

    ret <- .Call(mvtnorm_R_lpmvnorm, ac, bc, unclass(C), as.double(center),
                 as.integer(N), as.integer(J), w, as.integer(M), as.double(tol),
                 as.logical(logLik), as.logical(fast));
    return(ret)
}
◇
```

Fragment referenced in 51a.

Coming back to our simple example, we get (with 25000 simple Monte-Carlo iterations)

```
> phat

 [1] 0.2369329 0.2337179 0.2842052 0.3915213 0.4662496 0.0000000 0.5900784
 [8] 0.4618524 0.4872819 0.0000000

> exp(lpmvnorm(a, b, chol = lx, M = 25000, logLik = FALSE, fast = TRUE))

 [1] 2.366926e-01 2.341369e-01 2.834803e-01 3.938926e-01 4.658150e-01
 [6] 8.881784e-21 5.911462e-01 4.597514e-01 4.879485e-01 8.881784e-21

> exp(lpmvnorm(a, b, chol = lx, M = 25000, logLik = FALSE, fast = FALSE))

 [1] 2.377131e-01 2.372235e-01 2.831741e-01 3.875320e-01 4.659937e-01
 [6] 8.881784e-21 5.895400e-01 4.624243e-01 4.871073e-01 8.881784e-21
```

Next we generate some data and compare our implementation to `pmvnorm` using quasi-Monte-Carlo integration. The `pmvnorm` function uses randomised Korobov rules. The experiment here applies generalised Halton sequences. Plain Monte-Carlo (`w = NULL`) will also work but produces more variable results. Results will depend a lot on appropriate choices and it is the users responsibility to make sure things work as intended. If you are unsure, you should use `pmvnorm` which provides a well-tested configuration.

```
> M <- 10000
> if (require("qrng", quietly = TRUE)) {
+     ### quasi-Monte-Carlo
+     W <- t(ghalton(M, d = J - 1))
+ } else {
+     ### Monte-Carlo
+     W <- matrix(runif(M * (J - 1)), nrow = J - 1)
+ }
> ### Genz & Bretz, 2001, without early stopping (really?)
> pGB <- lpmvnormR(a, b, chol = lx, logLik = FALSE,
```

```
+                 algorithm = GenzBretz(maxpts = M, abseps = 0, releps = 0))
> ### Genz 1992 with quasi-Monte-Carlo, fast pnorm
> pGqf <- exp(lpmvnorm(a, b, chol = lx, w = W, M = M, logLik = FALSE, fast = TRUE))
> ### Genz 1992, original Monte-Carlo, fast pnorm
> pGf <- exp(lpmvnorm(a, b, chol = lx, w = NULL, M = M, logLik = FALSE, fast = TRUE))
> ### Genz 1992 with quasi-Monte-Carlo, R::pnorm
> pGqs <- exp(lpmvnorm(a, b, chol = lx, w = W, M = M, logLik = FALSE, fast = FALSE))
> ### Genz 1992, original Monte-Carlo, R::pnorm
> pGs <- exp(lpmvnorm(a, b, chol = lx, w = NULL, M = M, logLik = FALSE, fast = FALSE))
> cbind(pGB, pGqf, pGf, pGqs, pGs)

            pGB         pGqf          pGf         pGqs          pGs
 [1,] 0.2368918 2.369290e-01 2.344954e-01 2.369297e-01 2.360153e-01
 [2,] 0.2341507 2.340099e-01 2.319416e-01 2.340103e-01 2.347435e-01
 [3,] 0.2841044 2.841303e-01 2.850959e-01 2.841316e-01 2.870079e-01
 [4,] 0.3918357 3.921465e-01 3.931626e-01 3.921469e-01 3.904457e-01
 [5,] 0.4671062 4.668249e-01 4.678817e-01 4.668242e-01 4.690837e-01
 [6,] 0.0000000 2.220446e-20 2.220446e-20 2.220446e-20 2.220446e-20
 [7,] 0.5901670 5.902059e-01 5.907621e-01 5.902056e-01 5.929013e-01
 [8,] 0.4613023 4.619428e-01 4.611888e-01 4.619434e-01 4.630231e-01
 [9,] 0.4872195 4.870317e-01 4.863298e-01 4.870324e-01 4.820740e-01
[10,] 0.0000000 2.220446e-20 2.220446e-20 2.220446e-20 2.220446e-20
```

The three versions agree nicely. We now check if the code also works for univariate problems

```
> ### test univariate problem
> ### call pmvnorm
> pGB <- lpmvnormR(a[1,,drop = FALSE], b[1,,drop = FALSE], chol = lx[,1],
+                 logLik = FALSE,
+                 algorithm = GenzBretz(maxpts = M, abseps = 0, releps = 0))
> ### call lpmvnorm
> pGq <- exp(lpmvnorm(a[1,,drop = FALSE], b[1,,drop = FALSE], chol = lx[,1],
+                     logLik = FALSE))
> ### ground truth
> ptr <- pnorm(b[1,] / c(unclass(lx[,1]))) - pnorm(a[1,] / c(unclass(lx[,1])))
> cbind(c(ptr), pGB, pGq)

                      pGB       pGq
 [1,] 0.9999758 0.9999758 0.9999758
 [2,] 0.6108928 0.6108928 0.6108928
 [3,] 0.9076043 0.9076043 0.9076043
 [4,] 0.8979932 0.8979932 0.8979932
 [5,] 0.9589363 0.9589363 0.9589363
 [6,] 0.7863435 0.7863435 0.7863435
 [7,] 0.9982537 0.9982537 0.9982537
 [8,] 0.8745388 0.8745388 0.8745388
 [9,] 0.9386051 0.9386051 0.9386051
[10,] 0.9119778 0.9119778 0.9119778
```

Because the default `fast = FALSE` was used here, all results are identical.

3.2 Score Function

In addition to the log-likelihood, we would also like to have access to the scores with respect to $\mathbf{C}_i$. Because every element of $\mathbf{C}_i$ only enters once, the chain rule rules, so to speak.

We need the derivatives of d, e, y, and f with respect to the c parameters

⟨ *chol scores* 63a ⟩ ≡

```
double dp_c[Jp], ep_c[Jp], fp_c[Jp], yp_c[(iJ - 1) * Jp];
◇
```

Fragment referenced in 63e.

and the derivates with respect to the mean

⟨ *mean scores* 63b ⟩ ≡

```
double dp_m[Jp], ep_m[Jp], fp_m[Jp], yp_m[(iJ - 1) * Jp];
◇
```

Fragment referenced in 63e.

and the derivates with respect to lower (`a`)

⟨ *lower scores* 63c ⟩ ≡

```
double dp_l[Jp], ep_l[Jp], fp_l[Jp], yp_l[(iJ - 1) * Jp];
◇
```

Fragment referenced in 63e.

and the derivates with respect to upper (`b`)

⟨ *upper scores* 63d ⟩ ≡

```
double dp_u[Jp], ep_u[Jp], fp_u[Jp], yp_u[(iJ - 1) * Jp];
◇
```

Fragment referenced in 63e.

and we start allocating the necessary memory. The output object contains the likelihood contributions (first row), the scores with respect to the mean (next J rows), with respect to the lower integration limits (next J rows), with respect to the upper integration limits (next J rows) and finally with respect to the off-diagonal elements of the Cholesky factor (last $J(J-1)/2$ rows).

⟨ *score output object* 63e ⟩ ≡

```
int Jp = iJ * (iJ + 1) / 2;
⟨ chol scores 63a ⟩
⟨ mean scores 63b ⟩
⟨ lower scores 63c ⟩
⟨ upper scores 63d ⟩
double dtmp, etmp, Wtmp, ytmp, xx;

PROTECT(ans = allocMatrix(REALSXP, Jp + 1 + 3 * iJ, iN));
dans = REAL(ans);
for (j = 0; j < LENGTH(ans); j++) dans[j] = 0.0;
◇
```

Fragment referenced in 69.

For each $i = 1, \ldots, N$, do

1. Input $\mathbf{C}_i$ (`chol`), $\mathbf{a}_i$ (`lower`), $\mathbf{b}_i$ (`upper`), and control parameters α, ϵ, and $M_{\max}$ (`M`).

2. Standardise integration limits $a_j^{(i)}/c_{jj}^{(i)}$, $b_j^{(i)}/c_{jj}^{(i)}$, and rows $c_{\jmath j}^{(i)}/c_{jj}^{(i)}$ for $1 \leq \jmath < j < J$.

   Note: We later need derivatives wrt $c_{jj}^{(i)}$, so we compute derivates wrt $a_j^{(i)}$ and $b_j^{(i)}$ and post-differentiate later.

3. Initialise intsum = varsum = 0, $M = 0$, $d_1 = \Phi\left(a_1^{(i)}\right)$, $e_1 = \Phi\left(b_1^{(i)}\right)$ and $f_1 = e_1 - d_1$.

   We start initialised the score wrt to $c_{11}^{(i)}$ (the parameter is non-existent here due to standardisation)

   ⟨ *score c11* 64a ⟩ ≡

   ```
   dp_c[0] = (R_FINITE(da[0]) ? dnorm(da[0], x0, 1.0, 0L) * (da[0] - x0 - dcenter[0]) : 0);
   ep_c[0] = (R_FINITE(db[0]) ? dnorm(db[0], x0, 1.0, 0L) * (db[0] - x0 - dcenter[0]) : 0);
   fp_c[0] = ep_c[0] - dp_c[0];
   ◇
   ```

   Fragment referenced in 64c, 69.

   ⟨ *score a, b* 64b ⟩ ≡

   ```
   dp_m[0] = (R_FINITE(da[0]) ? dnorm(da[0], x0, 1.0, 0L) : 0);
   ep_m[0] = (R_FINITE(db[0]) ? dnorm(db[0], x0, 1.0, 0L) : 0);
   dp_l[0] = dp_m[0];
   ep_u[0] = ep_m[0];
   dp_u[0] = 0;
   ep_l[0] = 0;
   fp_m[0] = ep_m[0] - dp_m[0];
   fp_l[0] = -dp_m[0];
   fp_u[0] = ep_m[0];
   ◇
   ```

   Fragment referenced in 64c, 69.

4. Repeat

   ⟨ *init score loop* 64c ⟩ ≡

   ```
   ⟨ init logLik loop 53c ⟩
   ⟨ score c11 64a ⟩
   ⟨ score a, b 64b ⟩
   ◇
   ```

   Fragment referenced in 69.

   (a) Generate uniform $w_1, \ldots, w_{J-1} \in [0, 1]$.

   (b) For $j = 2, \ldots, J$ set

   $$y_{j-1} \quad = \quad \Phi^{-1}\left(d_{j-1} + w_{j-1}(e_{j-1} - d_{j-1})\right)$$

   We again either generate w_{j-1} on the fly or use pre-computed weights (`w`). We first compute the scores with respect to the already existing parameters.

⟨ *update yp for chol* 65a ⟩ ≡

```
ytmp = exp(- dnorm(y[j - 1], 0.0, 1.0, 1L)); // = 1 / dnorm(y[j - 1], 0.0, 1.0, 0L)

for (k = 0; k < Jp; k++) yp_c[k * (iJ - 1) + (j - 1)] = 0.0;

for (idx = 0; idx < (j + 1) * j / 2; idx++) {
    yp_c[idx * (iJ - 1) + (j - 1)] = ytmp;
    yp_c[idx * (iJ - 1) + (j - 1)] *= (dp_c[idx] + Wtmp * (ep_c[idx] - dp_c[idx]));
}
```

◇

Fragment referenced in 68a.

⟨ *update yp for means, lower and upper* 65b ⟩ ≡

```
for (k = 0; k < iJ; k++)
    yp_m[k * (iJ - 1) + (j - 1)] = 0.0;

for (idx = 0; idx < j; idx++) {
    yp_m[idx * (iJ - 1) + (j - 1)] = ytmp;
    yp_m[idx * (iJ - 1) + (j - 1)] *= (dp_m[idx] + Wtmp * (ep_m[idx] - dp_m[idx]));
}
for (k = 0; k < iJ; k++)
    yp_l[k * (iJ - 1) + (j - 1)] = 0.0;

for (idx = 0; idx < j; idx++) {
    yp_l[idx * (iJ - 1) + (j - 1)] = ytmp;
    yp_l[idx * (iJ - 1) + (j - 1)] *= (dp_l[idx] + Wtmp * (dp_u[idx] - dp_l[idx]));
}
for (k = 0; k < iJ; k++)
    yp_u[k * (iJ - 1) + (j - 1)] = 0.0;

for (idx = 0; idx < j; idx++) {
    yp_u[idx * (iJ - 1) + (j - 1)] = ytmp;
    yp_u[idx * (iJ - 1) + (j - 1)] *= (ep_l[idx] + Wtmp * (ep_u[idx] - ep_l[idx]));
}
```

◇

Fragment referenced in 68a.

$$x_{j-1} = \sum_{\jmath=1}^{j-1} c_{j\jmath}^{(i)} y_j$$

$$\begin{aligned} d_j &= \Phi\left(a_j^{(i)} - x_{j-1}\right) \\ e_j &= \Phi\left(b_j^{(i)} - x_{j-1}\right) \end{aligned}$$

$$f_j = (e_j - d_j) f_{j-1}.$$

The scores with respect to $c_{j\jmath}^{(i)}, \jmath = 1, \ldots, j-1$ are

⟨ *score wrt new chol off-diagonals* 66a ⟩ ≡

```
dtmp = dnorm(da[j], x, 1.0, 0L);
etmp = dnorm(db[j], x, 1.0, 0L);

for (k = 0; k < j; k++) {
    idx = start + j + k;
    if (LENGTH(center)) {
        dp_c[idx] = dtmp * (-1.0) * (y[k] - dcenter[k]);
        ep_c[idx] = etmp * (-1.0) * (y[k] - dcenter[k]);
    } else {
        dp_c[idx] = dtmp * (-1.0) * y[k];
        ep_c[idx] = etmp * (-1.0) * y[k];
    }
    fp_c[idx] = (ep_c[idx] - dp_c[idx]) * f;
}
```

◇

Fragment referenced in 68a.

and the score with respect to (the here non-existing) $c_{jj}^{(i)}$ is

⟨ *score wrt new chol diagonal* 66b ⟩ ≡

```
idx = (j + 1) * (j + 2) / 2 - 1;
dp_c[idx] = (R_FINITE(da[j]) ? dtmp * (da[j] - x - dcenter[j]) : 0);
ep_c[idx] = (R_FINITE(db[j]) ? etmp * (db[j] - x - dcenter[j]) : 0);
fp_c[idx] = (ep_c[idx] - dp_c[idx]) * f;
```

◇

Fragment referenced in 68a.

⟨ *new score means, lower and upper* 66c ⟩ ≡

```
dp_m[j] = (R_FINITE(da[j]) ? dtmp : 0);
ep_m[j] = (R_FINITE(db[j]) ? etmp : 0);
dp_l[j] = dp_m[j];
ep_u[j] = ep_m[j];
dp_u[j] = 0;
ep_l[j] = 0;
fp_l[j] = - dp_m[j] * f;
fp_u[j] = ep_m[j] * f;
fp_m[j] = fp_u[j] + fp_l[j];
```

◇

Fragment referenced in 68a.

We next update scores for parameters introduced for smaller j

⟨ *update score for chol* 67a ⟩ ≡

```
for (idx = 0; idx < j * (j + 1) / 2; idx++) {
    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_c[idx * (iJ - 1) + k];

    dp_c[idx] = dtmp * (-1.0) * xx;
    ep_c[idx] = etmp * (-1.0) * xx;
    fp_c[idx] = (ep_c[idx] - dp_c[idx]) * f + emd * fp_c[idx];
}
```

◇

Fragment referenced in 68a.

⟨ *update score means, lower and upper* 67b ⟩ ≡

```
for (idx = 0; idx < j; idx++) {
    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_m[idx * (iJ - 1) + k];

    dp_m[idx] = dtmp * (-1.0) * xx;
    ep_m[idx] = etmp * (-1.0) * xx;
    fp_m[idx] = (ep_m[idx] - dp_m[idx]) * f + emd * fp_m[idx];
}

for (idx = 0; idx < j; idx++) {
    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_l[idx * (iJ - 1) + k];

    dp_l[idx] = dtmp * (-1.0) * xx;
    dp_u[idx] = etmp * (-1.0) * xx;
    fp_l[idx] = (dp_u[idx] - dp_l[idx]) * f + emd * fp_l[idx];
}

for (idx = 0; idx < j; idx++) {
    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_u[idx * (iJ - 1) + k];

    ep_l[idx] = dtmp * (-1.0) * xx;
    ep_u[idx] = etmp * (-1.0) * xx;
    fp_u[idx] = (ep_u[idx] - ep_l[idx]) * f + emd * fp_u[idx];
}
```

◇

Fragment referenced in 68a.

We put everything together in a loop starting with the second dimension

⟨ *inner score loop* 68a ⟩ ≡

```
for (j = 1; j < iJ; j++) {
    ⟨ compute y 54a ⟩
    ⟨ compute x 54b ⟩
    ⟨ update d, e 54c ⟩
    ⟨ update yp for chol 65a ⟩
    ⟨ update yp for means, lower and upper 65b ⟩
    ⟨ score wrt new chol off-diagonals 66a ⟩
    ⟨ score wrt new chol diagonal 66b ⟩
    ⟨ new score means, lower and upper 66c ⟩
    ⟨ update score for chol 67a ⟩
    ⟨ update score means, lower and upper 67b ⟩
    ⟨ update f 55a ⟩
}
```

◇

Fragment referenced in 69.

(c) Set intsum = intsum+f_J, varsum = varsum+f_J^2, $M = M+1$, and error = $\sqrt{(\text{varsum}/M - (\text{intsum}/M)^2)/M}$.
We refrain from early stopping and error estimation.

Until error $< \epsilon$ or $M = M_{\max}$

5. Output $\hat{p}_i = \text{intsum}/M$.
We return $\log \hat{p}_i$ for each i, or we immediately sum-up over i.

⟨ *score output* 68b ⟩ ≡

```
dans[0] += f;
for (j = 0; j < Jp; j++)
    dans[j + 1] += fp_c[j];
for (j = 0; j < iJ; j++) {
    idx = Jp + j + 1;
    dans[idx] += fp_m[j];
    dans[idx + iJ] += fp_l[j];
    dans[idx + 2 * iJ] += fp_u[j];
}
```

◇

Fragment referenced in 69.

We put everything together in C

⟨ *R slpmvnorm* 69 ⟩ ≡

```
SEXP R_slpmvnorm(SEXP a, SEXP b, SEXP C, SEXP center, SEXP N, SEXP J, SEXP W,
                 SEXP M, SEXP tol, SEXP fast) {

    SEXP ans;
    double *da, *db, *dC, *dW, *dans, dtol = REAL(tol)[0];
    double *dcenter;
    double mdtol = 1.0 - dtol;
    double d0, e0, emd0, f0, q0, intsum;
    int p, idx;

    ⟨ dimensions 57c ⟩
    ⟨ pnorm 57a ⟩
    ⟨ W length 57b ⟩
    ⟨ init center 58c ⟩

    int start, j, k;
    double tmp, e, d, f, emd, x, x0, y[iJ - 1];

    ⟨ score output object 63e ⟩

    q0 = qnorm(dtol, 0.0, 1.0, 1L, 0L);

    /* univariate problem */
    if (iJ == 1) iM = 0;

    if (W == R_NilValue)
        GetRNGstate();

    for (int i = 0; i < iN; i++) {

        ⟨ initialisation 53b ⟩
        ⟨ score c11 64a ⟩
        ⟨ score a, b 64b ⟩

        if (iM == 0) {
            dans[0] = intsum;
            dans[1] = fp_c[0];
            dans[2] = fp_m[0];
            dans[3] = fp_l[0];
            dans[4] = fp_u[0];
        }

        if (W != R_NilValue && pW == 0)
            dW = REAL(W);

        for (int m = 0; m < iM; m++) {

            ⟨ init score loop 64c ⟩
            ⟨ inner score loop 68a ⟩
            ⟨ score output 68b ⟩

            if (W != R_NilValue)
                dW += iJ - 1;
        }

        ⟨ move on 56a ⟩

        dans += Jp + 1 + 3 * iJ;
    }

    if (W == R_NilValue)
        PutRNGstate();

    UNPROTECT(1);
    return(ans);
}
```

◇

Fragment referenced in 51b.

The R code is now essentially identical to `lpmvnorm`, however, we need to undo the effect of standardisation once the scores have been computed

⟨ *post differentiate mean score* 70a ⟩ ≡

```
Jp <- J * (J + 1) / 2;
smean <- - ret[Jp + 1:J, , drop = FALSE]
if (attr(chol, "diag"))
    smean <- smean / c(dchol)
◇
```

Fragment referenced in 72.

⟨ *post differentiate lower score* 70b ⟩ ≡

```
slower <- ret[Jp + J + 1:J, , drop = FALSE]
if (attr(chol, "diag"))
    slower <- slower / c(dchol)
◇
```

Fragment referenced in 72.

⟨ *post differentiate upper score* 70c ⟩ ≡

```
supper <- ret[Jp + 2 * J + 1:J, , drop = FALSE]
if (attr(chol, "diag"))
    supper <- supper / c(dchol)
◇
```

Fragment referenced in 72.

⟨ *post differentiate chol score* 70d ⟩ ≡

```
if (J == 1) {
    idx <- 1L
} else {
    idx <- cumsum(c(1, 2:J))
}
if (attr(chol, "diag")) {
    ret <- ret / c(dchol[rep(1:J, 1:J),]) ### because 1 / dchol already there
    ret[idx,] <- -ret[idx,]
}
◇
```

Fragment referenced in 72.

We sometimes parameterise models in terms of $\mathbf{L} = \mathbf{C}^{-1}$, the Cholesky factor of the precision matrix. The log-likelihood operates on $\mathbf{C}$, so we need to post-differentiate the score function. We have

$$\mathbf{A} = \frac{\partial \mathbf{L}^{-1}}{\partial \mathbf{L}} = -\mathbf{L}^{-\top} \otimes \mathbf{L}^{-1}$$

and computing $\mathbf{sA}$ for a score vector $\mathbf{s}$ with respect to $\mathbf{L}$ can be implemented by the "vec trick" (Section 2.10)

$$\mathbf{sA} = \mathbf{L}^{-\top}\mathbf{S}\mathbf{L}^{-\top}$$

where $\mathbf{s} = \text{vec}(\mathbf{S})$.

⟨ *post differentiate invchol score* 71a ⟩ ≡

```
if (!missing(invchol)) {
    ret <- ltMatrices(ret, diag = TRUE, byrow = TRUE)
    ret <- - unclass(vectrick(chol, ret, chol))
}
```

◇

Fragment referenced in 72.

If the diagonal elements were constants, we set them to zero. The function always returns an object of class `ltMatrices` with explicit diagonal elements (use `Lower_tri(, diag = FALSE)` to extract the lower triangular elements such that the scores match the input)

⟨ *post process score* 71b ⟩ ≡

```
if (!attr(chol, "diag"))
    ### remove scores for constant diagonal elements
    ret[idx,] <- 0
ret <- ltMatrices(ret, diag = TRUE, byrow = TRUE)
```

◇

Fragment referenced in 72.

We can now finally put everything together in a single score function.

⟨ *slpmvnorm* 72 ⟩ ≡

```
slpmvnorm <- function(lower, upper, mean = 0, center = NULL, chol, invchol, logLik = TRUE, M = NULL,
                      w = NULL, seed = NULL, tol = .Machine$double.eps, fast = FALSE) {

    ⟨ init random seed, reset on exit 60a ⟩

    ⟨ Cholesky of precision 60c ⟩

    ⟨ input checks 52 ⟩

    ⟨ standardise 53a ⟩

    ⟨ check and / or set integration weights 60b ⟩

    ret <- .Call(mvtnorm_R_slpmvnorm, ac, bc, unclass(C), as.double(center), as.integer(N),
                 as.integer(J), w, as.integer(M), as.double(tol), as.logical(fast));

    ll <- log(pmax(ret[1L,], tol)) - log(M)
    intsum <- ret[1L,]
    m <- matrix(intsum, nrow = nrow(ret) - 1, ncol = ncol(ret), byrow = TRUE)
    ret <- ret[-1L,,drop = FALSE] / m

    ⟨ post differentiate mean score 70a ⟩
    ⟨ post differentiate lower score 70b ⟩
    ⟨ post differentiate upper score 70c ⟩

    ret <- ret[1:Jp, , drop = FALSE]

    ⟨ post differentiate chol score 70d ⟩

    ⟨ post differentiate invchol score 71a ⟩

    ⟨ post process score 71b ⟩

    ret <- ltMatrices(ret, byrow = byrow_orig)

    if (logLik) {
        ret <- list(logLik = ll,
                    mean = smean,
                    lower = slower,
                    upper = supper,
                    chol = ret)
        if (!missing(invchol)) names(ret)[names(ret) == "chol"] <- "invchol"
        return(ret)
    }

    return(ret)
}
```

◇

Fragment referenced in 51a.

Let's look at an example, where we use `numDeriv::grad` to check the results

```
> J <- 5
> N <- 4
> S <- crossprod(matrix(runif(J^2), nrow = J))
```

```
> prm <- t(chol(S))[lower.tri(S, diag = TRUE)]
> ### define C
> mC <- ltMatrices(matrix(prm, ncol = 1), diag = TRUE)
> a <- matrix(runif(N * J), nrow = J) - 2
> b <- a + 4
> a[2,] <- -Inf
> b[3,] <- Inf
> M <- 10000
> W <- matrix(runif(M * (J - 1)), ncol = M)
> lli <- c(lpmvnorm(a, b, chol = mC, w = W, M = M, logLik = FALSE))
> fC <- function(prm) {
+     C <- ltMatrices(matrix(prm, ncol = 1), diag = TRUE)
+     lpmvnorm(a, b, chol = C, w = W, M = M)
+ }
> sC <- slpmvnorm(a, b, chol = mC, w = W, M = M)
> chk(lli, sC$logLik)
> if (require("numDeriv", quietly = TRUE))
+     chk(grad(fC, unclass(mC)), rowSums(unclass(sC$chol)), check.attributes = FALSE)
```

We can do the same when $\mathbf{L}$ (and not $\mathbf{C}$) is given

```
> mL <- solve(mC)
> lliL <- c(lpmvnorm(a, b, invchol = mL, w = W, M = M, logLik = FALSE))
> chk(lli, lliL)
> fL <- function(prm) {
+     L <- ltMatrices(matrix(prm, ncol = 1), diag = TRUE)
+     lpmvnorm(a, b, invchol = L, w = W, M = M)
+ }
> sL <- slpmvnorm(a, b, invchol = mL, w = W, M = M)
> chk(lliL, sL$logLik)
> if (require("numDeriv", quietly = TRUE))
+     chk(grad(fL, unclass(mL)), rowSums(unclass(sL$invchol)),
+         check.attributes = FALSE)
```

The score function also works for univariate problems

```
> ptr <- pnorm(b[1,] / c(unclass(mC[,1]))) - pnorm(a[1,] / c(unclass(mC[,1])))
> log(ptr)

[1] -0.01165889 -0.08617272 -0.01240094 -0.03105050

> lpmvnorm(a[1,,drop = FALSE], b[1,,drop = FALSE], chol = mC[,1], logLik = FALSE)

[1] -0.01165889 -0.08617272 -0.01240094 -0.03105050

> lapply(slpmvnorm(a[1,,drop = FALSE], b[1,,drop = FALSE], chol = mC[,1], logLik =
+ TRUE), unclass)

$logLik
[1] -0.01165889 -0.08617272 -0.01240094 -0.03105050

$mean
           [,1]      [,2]       [,3]       [,4]
[1,] 0.02222249 0.2140162 0.02641782 0.08861162

$lower
```

```
            [,1]      [,2]        [,3]        [,4]
[1,] -0.03221736 -0.214453 -0.03536199 -0.09096213

$upper
           [,1]         [,2]        [,3]        [,4]
[1,] 0.00999487 0.0004368597 0.008944164 0.002350511

$chol
         [,1]       [,2]       [,3]       [,4]
1.1 -0.104149 -0.2994286 -0.1075726 -0.1787174
attr(,"J")
[1] 1
attr(,"diag")
[1] TRUE
attr(,"byrow")
[1] FALSE
attr(,"rcnames")
[1] "1"

> sd1 <- c(unclass(mC[,1]))
> (dnorm(b[1,] / sd1) * b[1,] - dnorm(a[1,] / sd1) * a[1,]) * (-1) / sd1^2 / ptr

[1] -0.1041490 -0.2994286 -0.1075726 -0.1787174
```

Chapter 4

Maximum-likelihood Example

We now discuss how this infrastructure can be used to estimate the Cholesky factor of a multivariate normal in the presence of interval-censored observations.

We first generate a covariance matrix $\Sigma = \mathbf{C}\mathbf{C}^\top$ and extract the Cholesky factor $\mathbf{C}$

```
> J <- 4
> R <- diag(J)
> R[1,2] <- R[2,1] <- .25
> R[1,3] <- R[3,1] <- .5
> R[2,4] <- R[4,2] <- .75
> (Sigma <- diag(sqrt(1:J / 2)) %*% R %*% diag(sqrt(1:J / 2)))
```

```
          [,1]      [,2]      [,3]    [,4]
[1,] 0.5000000 0.1767767 0.4330127 0.00000
[2,] 0.1767767 1.0000000 0.0000000 1.06066
[3,] 0.4330127 0.0000000 1.5000000 0.00000
[4,] 0.0000000 1.0606602 0.0000000 2.00000
```

```
> (C <- t(chol(Sigma)))
```

```
          [,1]       [,2]      [,3]      [,4]
[1,] 0.7071068  0.0000000 0.0000000 0.0000000
[2,] 0.2500000  0.9682458 0.0000000 0.0000000
[3,] 0.6123724 -0.1581139 1.0488088 0.0000000
[4,] 0.0000000  1.0954451 0.1651446 0.8790491
```

We now represent this matrix as `ltMatrices` object

```
> prm <- C[lower.tri(C, diag = TRUE)]
> lt <- ltMatrices(matrix(prm, ncol = 1L),
+                  diag = TRUE,    ### has diagonal elements
+                  byrow = FALSE)  ### prm is column-major
> BYROW <- FALSE   ### later checks
> lt <- ltMatrices(lt,
+                  byrow = BYROW)   ### convert to row-major
> chk(C, as.array(lt)[,,1], check.attributes = FALSE)
> chk(Sigma, as.array(Tcrossprod(lt))[,,1], check.attributes = FALSE)
```

We generate some data from $\mathbb{N}_J(\mathbf{0}_J, \Sigma)$ by first sampling from $\mathbf{Z} \sim \mathbb{N}_J(\mathbf{0}_J, \mathbf{I}_J)$ and then computing $\mathbf{Y} = \mathbf{C}\mathbf{Z} + \mu \sim \mathbb{N}_J(\mu, \mathbf{C}\mathbf{C}^\top)$

```
> N <- 100
> Z <- matrix(rnorm(N * J), nrow = J)
> Y <- Mult(lt, Z) + (mn <- 1:J)
```

Before we add some interval-censoring to the data, let's estimate the Cholesky factor $\mathbf{C}$ (here called `lt`) from the raw continuous data. The true mean μ and the true covariance matrix Σ can be estimated from the uncensored data via maximum likelihood as

```
> rowMeans(Y)

       1        2        3        4
0.9685377 2.1268796 2.9633561 3.9825669

> (Shat <- var(t(Y)) * (N - 1) / N)

           1          2          3          4
1 0.46655660 0.18104431 0.34222237 0.01609179
2 0.18104431 0.94385339 0.08992252 0.84309528
3 0.34222237 0.08992252 1.36054915 0.08104091
4 0.01609179 0.84309528 0.08104091 1.63301525
```

We first check if we can obtain the same results by numerial optimisation using `dmvnorm` and the scores `sldmvnorm`. The log-likelihood and the score function (for the centered means) in terms of $\mathbf{C}$ are

```
> Yc <- Y - rowMeans(Y)
> ll <- function(parm) {
+     C <- ltMatrices(parm, diag = TRUE, byrow = BYROW)
+     -sum(dmvnorm(x = Yc, chol = C, log = TRUE))
+ }
> sc <- function(parm) {
+     C <- ltMatrices(parm, diag = TRUE, byrow = BYROW)
+     -rowSums(unclass(sldmvnorm(x = Yc, chol = C)$chol))
+ }
```

The diagonal elements of $\mathbf{C}$ are positive, so we need box constraints

```
> llim <- rep(-Inf, J * (J + 1) / 2)
> llim[which(rownames(unclass(lt)) %in% paste(1:J, 1:J, sep = "."))] <- 1e-4
```

The ML-estimate of $\mathbf{C}\mathbf{C}^\top$ is now used to obtain an estimate of $\mathbf{C}$ and we check the score function for some random starting values

```
> if (BYROW) {
+   cML <- chol(Shat)[upper.tri(Shat, diag = TRUE)]
+ } else {
+   cML <- t(chol(Shat))[lower.tri(Shat, diag = TRUE)]
+ }
> ll(cML)

[1] 517.8685

> start <- runif(length(cML))
> if (require("numDeriv", quietly = TRUE))
+     chk(grad(ll, start), sc(start), check.attributes = FALSE)
```

Finally, we hand over to `optim` and compare the results of the analytically and numerically obtained ML estimates

```
> op <- optim(start, fn = ll, gr = sc, method = "L-BFGS-B",
+             lower = llim, control = list(trace = TRUE))
```

```
iter   10 value 518.092239
iter   20 value 517.868548
final  value 517.868548
converged
```

```
> ## ML numerically
> ltMatrices(op$par, diag = TRUE, byrow = BYROW)

, , 1

           1           2         3         4
1 0.68305690  0.00000000 0.0000000 0.0000000
2 0.26505417  0.93464707 0.0000000 0.0000000
3 0.50102358 -0.04586658 1.0523442 0.0000000
4 0.02356369  0.89534692 0.1048239 0.9054404

> ll(op$par)

[1] 517.8685

> ## ML analytically
> t(chol(Shat))

           1           2        3         4
1 0.68304949  0.00000000 0.000000 0.0000000
2 0.26505300  0.93466588 0.000000 0.0000000
3 0.50102134 -0.04587167 1.052341 0.0000000
4 0.02355875  0.89534773 0.104822 0.9054419

> ll(cML)

[1] 517.8685

> ## true C matrix
> lt

, , 1

          1          2         3         4
1 0.7071068  0.0000000 0.0000000 0.0000000
2 0.2500000  0.9682458 0.0000000 0.0000000
3 0.6123724 -0.1581139 1.0488088 0.0000000
4 0.0000000  1.0954451 0.1651446 0.8790491
```

Under interval-censoring, the mean and $\mathbf{C}$ are no longer orthogonal and there is no analytic solution to the ML estimation problem. So, we add some interval-censoring represented by `lwr` and `upr` and try to estimate the model parameters via `lpmvnorm` and corresponding scores `slpmvnorm`.

```
> prb <- 1:9 / 10
> sds <- sqrt(diag(Sigma))
> ct <- sapply(1:J, function(j) qnorm(prb, mean = mn[j], sd = sds[j]))
> lwr <- upr <- Y
> for (j in 1:J) {
+     f <- cut(Y[j,], breaks = c(-Inf, ct[,j], Inf))
+     lwr[j,] <- c(-Inf, ct[,j])[f]
+     upr[j,] <- c(ct[,j], Inf)[f]
+ }
```

Let's do some sanity and performance checks first. For different values of M, we evaluate the log-likelihood using `pmvnorm` (called in `lpmvnormR`) and the simplified implementation (fast and slow). The comparion is a bit unfair, because we do not add the time needed to setup Halton sequences, but we would do this only once and use the stored values for repeated evaluations of a log-likelihood (because the optimiser expects a deterministic function to be optimised)

```
> M <- floor(exp(0:25/10) * 1000)
> lGB <- sapply(M, function(m) {
+     st <- system.time(ret <- lpmvnormR(lwr, upr, mean = mn, chol = lt, algorithm =
+                                        GenzBretz(maxpts = m, abseps = 0, releps = 0)))
+     return(c(st["user.self"], ll = ret))
+ })
> lH <- sapply(M, function(m) {
+     W <- NULL
+     if (require("qrng", quietly = TRUE))
+         W <- t(ghalton(m, d = J - 1))
+     st <- system.time(ret <- lpmvnorm(lwr, upr, mean = mn, chol = lt, w = W, M = m))
+     return(c(st["user.self"], ll = ret))
+ })
> lHf <- sapply(M, function(m) {
+     W <- NULL
+     if (require("qrng", quietly = TRUE))
+         W <- t(ghalton(m, d = J - 1))
+     st <- system.time(ret <- lpmvnorm(lwr, upr, mean = mn, chol = lt, w = W, M = m,
+                                       fast = TRUE))
+     return(c(st["user.self"], ll = ret))
+ })
```

The evaluated log-likelihoods and corresponding timings are given in Figure 4.1. It seems that for $M \geq 3000$, results are reasonably stable.

We now define the log-likelihood function. It is important to use weights via the `w` argument (or to set the `seed`) such that only the candidate parameters `parm` change with repeated calls to `ll`. We use an extremely low number of integration points `M`, let's see if this still works out.

```
> M <- 500
> if (require("qrng", quietly = TRUE)) {
+     ### quasi-Monte-Carlo
+     W <- t(ghalton(M, d = J - 1))
+ } else {
+     ### Monte-Carlo
+     W <- matrix(runif(M * (J - 1)), nrow = J - 1)
+ }
> ll <- function(parm, J) {
+      m <- parm[1:J]                ### mean parameters
+      parm <- parm[-(1:J)]        ### chol parameters
+      C <- matrix(c(parm), ncol = 1L)
+      C <- ltMatrices(C, diag = TRUE, byrow = BYROW)
+      -lpmvnorm(lower = lwr, upper = upr, mean = m, chol = C, w = W, M = M, logLik = TRUE)
+ }
```

We can check the correctness of our log-likelihood function

```
> prm <- c(mn, unclass(lt))
> ll(prm, J = J)

[1] 880.4956
```

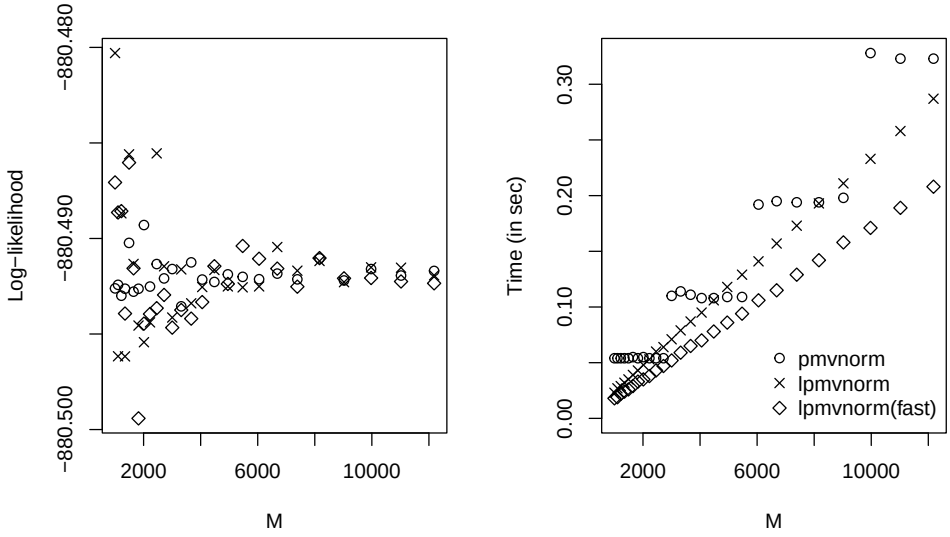

Figure 4.1: Evaluated log-likelihoods (left) and timings (right).

```
> lpmvnormR(lwr, upr, mean = mn, chol = lt,
+          algorithm = GenzBretz(maxpts = M, abseps = 0, releps = 0))

[1] -880.491

> (llprm <- lpmvnorm(lwr, upr, mean = mn, chol = lt, w = W, M = M))

[1] -880.4956

> chk(llprm, sum(lpmvnorm(lwr, upr, mean = mn, chol = lt, w = W, M = M, logLik = FALSE)))
```

Before we hand over to the optimiser, we define the score function with respect to μ and $\mathbf{C}$

```
> sc <- function(parm, J) {
+     m <- parm[1:J]              ### mean parameters
+     parm <- parm[-(1:J)]        ### chol parameters
+     C <- matrix(c(parm), ncol = 1L)
+     C <- ltMatrices(C, diag = TRUE, byrow = BYROW)
+     ret <- slpmvnorm(lower = lwr, upper = upr, mean = m, chol = C,
+                      w = W, M = M, logLik = TRUE)
+     return(-c(rowSums(ret$mean), rowSums(unclass(ret$chol))))
+ }
```

and check the correctness numerically

```
> if (require("numDeriv", quietly = TRUE))
+     chk(grad(ll, prm, J = J), sc(prm, J = J), check.attributes = FALSE)
```

Finally, we can hand-over to `optim`. Because we need $\text{diag}(\mathbf{C}) > 0$, we use box constraints and `method = "L-BFGS-B"`. We start with the estimates obtained from the original continuous data.

```
> llim <- rep(-Inf, J + J * (J + 1) / 2)
> llim[J + which(rownames(unclass(lt)) %in% paste(1:J, 1:J, sep = "."))] <- 1e-4
> if (BYROW) {
+   start <- c(rowMeans(Y), chol(Shat)[upper.tri(Shat, diag = TRUE)])
+ } else {
+   start <- c(rowMeans(Y), t(chol(Shat))[lower.tri(Shat, diag = TRUE)])
+ }
> ll(start, J = J)

[1] 875.4005

> op <- optim(start, fn = ll, gr = sc, J = J, method = "L-BFGS-B",
+             lower = llim, control = list(trace = TRUE))

iter   10 value 874.158309
final  value 874.158301
converged

> op$value ## compare with

[1] 874.1583

> ll(prm, J = J)

[1] 880.4956
```

We can now compare the true and estimated Cholesky factor $\mathbf{C}$ of our covariance matrix $\mathbf{\Sigma} = \mathbf{C}\mathbf{C}^\top$

```
> (C <- ltMatrices(matrix(op$par[-(1:J)], ncol = 1),
+                  diag = TRUE, byrow = BYROW))

, , 1

          1           2          3         4
1 0.67049567  0.00000000 0.00000000 0.0000000
2 0.26764384  1.02232159 0.00000000 0.0000000
3 0.54267774 -0.05007103 1.11347760 0.0000000
4 0.05223456  0.98429745 0.08473411 0.9613685

> lt

, , 1

          1          2         3         4
1 0.7071068  0.0000000 0.0000000 0.0000000
2 0.2500000  0.9682458 0.0000000 0.0000000
3 0.6123724 -0.1581139 1.0488088 0.0000000
4 0.0000000  1.0954451 0.1651446 0.8790491
```

and the estimated means

```
> op$par[1:J]

        1         2         3         4
0.9669828 2.1281616 2.9454002 3.9886471

> mn
```

```
[1] 1 2 3 4
```

We can also compare the results on the scale of the covariance matrix

```
> Tcrossprod(lt)                ### true Sigma

, , 1

          1         2         3       4
1 0.5000000 0.1767767 0.4330127 0.00000
2 0.1767767 1.0000000 0.0000000 1.06066
3 0.4330127 0.0000000 1.5000000 0.00000
4 0.0000000 1.0606602 0.0000000 2.00000

> Tcrossprod(C)           ### interval-censored obs

, , 1

           1          2          3          4
1 0.44956444 0.17945404 0.36386307 0.03502305
2 0.17945404 1.11677466 0.09405566 1.02024880
3 0.36386307 0.09405566 1.53683860 0.07341127
4 0.03502305 1.02024880 0.07341127 1.90297912

> Shat                    ### "exact" obs

           1          2          3          4
1 0.46655660 0.18104431 0.34222237 0.01609179
2 0.18104431 0.94385339 0.08992252 0.84309528
3 0.34222237 0.08992252 1.36054915 0.08104091
4 0.01609179 0.84309528 0.08104091 1.63301525
```

This looks reasonably close.

Warning: Do NOT assume the choices made here (especially `M` and `W`) to be universally applicable. Make sure to investigate the accuracy depending on these parameters of the log-likelihood and score function in your application.

One could ask what this whole exercise was about statistically. We estimated a multivariate normal distribution from interval-censored data, so what? Maybe we were primarily interested in fitting a linear regression

$$\mathbb{E}(Y_1 \mid Y_j = y_j, j = 2, \ldots, J) = \alpha + \sum_{j=2}^{J} \beta_j y_j.$$

Interval-censoring in the response could have been handled by some Tobit model, but what about interval-censoring in the explanatory variables? Based on the multivariate distribution just estimated, we can obtain the regression coefficients β_j as

```
> c(cond_mvnorm(chol = C, which = 2:J, given = diag(J - 1))$mean)

[1]  0.2602003  0.2270392 -0.1298560
```

We can compare these estimated regression coefficients with those obtained from a linear model fitted to the exact observations

```
> dY <- as.data.frame(t(Y))
> colnames(dY) <- paste0("Y", 1:J)
> coef(m1 <- lm(Y1 ~ ., data = dY))[-1L]
```

```
       Y2         Y3         Y4
 0.3169117  0.2404565 -0.1656946
```

The estimates are quite close, but what about standard errors? Interval-censoring means loss of information, so we should see larger standard errors for the interval-censored data.

Let's obtain the Hessian for all parameters first

```
> H <- optim(op$par, fn = ll, gr = sc, J = J, method = "L-BFGS-B",
+            lower = llim, hessian = TRUE)$hessian
```

and next we sample from the distribution of the maximum-likelihood estimators

```
> L <- t(chol(H))
> L <- ltMatrices(L[lower.tri(L, diag = TRUE)], diag = TRUE)
> Nsim <- 50000
> Z <- matrix(rnorm(Nsim * nrow(H)), ncol = Nsim)
> rC <- solve(L, Z)[-(1:J),] + op$par[-(1:J)] ### remove mean parameters
```

The standard error in this sample should be close to the ones obtained from the inverse Fisher information

```
> c(sqrt(rowMeans((rC - rowMeans(rC))^2)))

         5          6          7          8          9         10         11
0.05129646 0.07989618 0.12445698 0.16089554 0.07609088 0.11566519 0.14020346
        12         13         14
0.09622312 0.10415427 0.08278985

> c(sqrt(diagonals(Crossprod(solve(L)))))

 [1] 0.06825507 0.10816499 0.12670329 0.14073702 0.05498052 0.10839260
 [7] 0.12441885 0.14311786 0.08812684 0.11638318 0.13340466 0.09586564
[13] 0.10450821 0.08154249
```

We now coerse the matrix `rC` to an object of class `ltMatrices`

```
> rC <- ltMatrices(rC, diag = TRUE)
```

The object `rC` contains all sampled Cholesky factors of the covariance matrix. From each of these matrices, we compute the regression coefficient, giving us a sample we can use to compute standard errors from

```
> rbeta <- cond_mvnorm(chol = rC, which = 2:J, given = diag(J - 1))$mean
> sqrt(rowMeans((rbeta - rowMeans(rbeta))^2))

[1] 0.08792945 0.04869062 0.07752184
```

which are, as expected, slightly larger than the ones obtained from the more informative exact observations

```
> sqrt(diag(vcov(m1)))[-1L]

       Y2         Y3         Y4
0.08229627 0.05039009 0.06246094
```

Chapter 5

Package Infrastructure

⟨ *R Header* 83 ⟩ ≡

```
###    Copyright (C) 2022- Torsten Hothorn
###
###    This file is part of the 'mvtnorm' R add-on package.
###
###    'mvtnorm' is free software: you can redistribute it and/or modify
###    it under the terms of the GNU General Public License as published by
###    the Free Software Foundation, version 2.
###
###    'mvtnorm' is distributed in the hope that it will be useful,
###    but WITHOUT ANY WARRANTY; without even the implied warranty of
###    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
###    GNU General Public License for more details.
###
###    You should have received a copy of the GNU General Public License
###    along with 'mvtnorm'.  If not, see <http://www.gnu.org/licenses/>.
###
###
###    DO NOT EDIT THIS FILE
###
###    Edit 'lmvnorm_src.w' and run 'nuweb -r lmvnorm_src.w'
◇
```

Fragment referenced in 2, 51a.

⟨ *C Header* 84 ⟩ ≡

```
/*
    Copyright (C) 2022- Torsten Hothorn

    This file is part of the 'mvtnorm' R add-on package.

    'mvtnorm' is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, version 2.

    'mvtnorm' is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with 'mvtnorm'.  If not, see <http://www.gnu.org/licenses/>.


    DO NOT EDIT THIS FILE

    Edit 'lmvnorm_src.w' and run 'nuweb -r lmvnorm_src.w'
*/
◇
```

Fragment referenced in 3a, 51b.

Index

Files

Fragments

⟨ init score loop 64c ⟩ Referenced in 69.
⟨ initialisation 53b ⟩ Referenced in 59, 69.
⟨ inner logLik loop 55b ⟩ Referenced in 59.
⟨ inner score loop 68a ⟩ Referenced in 69.
⟨ input checks 52 ⟩ Referenced in 50, 61, 72.
⟨ kronecker vec trick 39b ⟩ Referenced in 2.
⟨ L times D 41b ⟩ Referenced in 42.
⟨ lower scores 63c ⟩ Referenced in 63e.
⟨ lower triangular elements 14 ⟩ Referenced in 2.
⟨ lpmvnorm 61 ⟩ Referenced in 51a.
⟨ lpmvnormR 50 ⟩ Not referenced.
⟨ ltMatrices 5a ⟩ Referenced in 2.
⟨ ltMatrices dim 3b ⟩ Referenced in 5a.
⟨ ltMatrices input 4b ⟩ Referenced in 5a.
⟨ ltMatrices names 4a ⟩ Referenced in 5a.
⟨ marginal 45b ⟩ Referenced in 2.
⟨ mc input checks 45a ⟩ Referenced in 45b, 47b.
⟨ mean scores 63b ⟩ Referenced in 63e.
⟨ move on 56a ⟩ Referenced in 59, 69.
⟨ mult 22 ⟩ Referenced in 3a.
⟨ mult ltMatrices 20b ⟩ Referenced in 2.
⟨ mult ltMatrices transpose 20a ⟩ Referenced in 20b.
⟨ names ltMatrices 6 ⟩ Referenced in 2.
⟨ new score means, lower and upper 66c ⟩ Referenced in 68a.
⟨ output 55d ⟩ Referenced in 59.
⟨ pnorm 57a ⟩ Referenced in 59, 69.
⟨ pnorm fast 56b ⟩ Referenced in 51b.
⟨ pnorm slow 56c ⟩ Referenced in 51b.
⟨ post differentiate chol score 70d ⟩ Referenced in 72.
⟨ post differentiate invchol score 71a ⟩ Referenced in 72.
⟨ post differentiate lower score 70b ⟩ Referenced in 72.
⟨ post differentiate mean score 70a ⟩ Referenced in 72.
⟨ post differentiate upper score 70c ⟩ Referenced in 72.
⟨ post process score 71b ⟩ Referenced in 72.
⟨ print ltMatrices 9 ⟩ Referenced in 2.
⟨ R Header 83 ⟩ Referenced in 2, 51a.
⟨ R lpmvnorm 59 ⟩ Referenced in 51b.
⟨ R slpmvnorm 69 ⟩ Referenced in 51b.
⟨ RC input 21a ⟩ Referenced in 22, 26, 31, 37a.
⟨ reorder ltMatrices 10 ⟩ Referenced in 2.
⟨ return objects 25 ⟩ Referenced in 26.
⟨ score a, b 64b ⟩ Referenced in 64c, 69.
⟨ score c11 64a ⟩ Referenced in 64c, 69.
⟨ score output 68b ⟩ Referenced in 69.
⟨ score output object 63e ⟩ Referenced in 69.
⟨ score wrt new chol diagonal 66b ⟩ Referenced in 68a.
⟨ score wrt new chol off-diagonals 66a ⟩ Referenced in 68a.
⟨ setup memory 23 ⟩ Referenced in 26.
⟨ setup return object 58a ⟩ Referenced in 59.
⟨ slpmvnorm 72 ⟩ Referenced in 51a.
⟨ solve 26 ⟩ Referenced in 3a.
⟨ solve ltMatrices 27 ⟩ Referenced in 2.
⟨ standardise 53a ⟩ Referenced in 61, 72.
⟨ subset ltMatrices 12 ⟩ Referenced in 2.
⟨ t(C) S t(A) 36 ⟩ Referenced in 37a.
⟨ tcrossprod 31 ⟩ Referenced in 3a.
⟨ tcrossprod diagonal only 29 ⟩ Referenced in 31.
⟨ tcrossprod full 30a ⟩ Referenced in 31.

⟨ tcrossprod ltMatrices 32 ⟩ Referenced in 2.
⟨ univariate problem 58b ⟩ Referenced in 59.
⟨ update d, e 54c ⟩ Referenced in 55b, 68a.
⟨ update f 55a ⟩ Referenced in 55b, 68a.
⟨ update score for chol 67a ⟩ Referenced in 68a.
⟨ update score means, lower and upper 67b ⟩ Referenced in 68a.
⟨ update yp for chol 65a ⟩ Referenced in 68a.
⟨ update yp for means, lower and upper 65b ⟩ Referenced in 68a.
⟨ upper scores 63d ⟩ Referenced in 63e.
⟨ vec trick 37a ⟩ Referenced in 3a.
⟨ W length 57b ⟩ Referenced in 59, 69.

Bibliography

Alan Genz. Numerical computation of multivariate normal probabilities. *Journal of Computational and Graphical Statistics*, 1:141–149, 1992. doi: 10.1080/10618600.1992.10477010. 1, 51

Alan Genz and Frank Bretz. Methods for the computation of multivariate t-probabilities. *Journal of Computational and Graphical Statistics*, 11:950–971, 2002. doi: 10.1198/106186002394. 1, 50

Ivan Matić, Radoš Radoičić, and Dan Stefanica. A sharp Pólya-based approximation to the normal CDF. *Applied Mathematics and Computation*, 322:111–122, 2018. doi: 10.2139/ssrn.2842681. 56