Multivariate Normal Log-likelihoods in the **mvtnorm** Package [1]

Torsten Hothorn

Version 1.2-7

[1] Please cite this document as: Torsten Hothorn (2024) Multivariate Normal Log-likelihoods in the **mvtnorm** Package. R package vignette version 1.2-7, URL https://CRAN.R-project.org/package=mvtnorm.

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

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

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 factor $\mathbf{C}_i$ 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 in Chapter 4. We use the technology developed here to implement the log-likelihood and score function for situations where some variables have been observed exactly and others only in form of interval-censoring in Chapter 5 and for nonparametric maximum-likelihood estimation in unstructured Gaussian copulae in Chapter 6.

Chapter 2

Lower Triangular Matrices

`"ltMatrices.R"` 2≡

⟨ *R Header* 104 ⟩
⟨ *ltMatrices* 6a ⟩
⟨ *syMatrices* 6b ⟩
⟨ *dim ltMatrices* 6c ⟩
⟨ *dimnames ltMatrices* 7a ⟩
⟨ *names ltMatrices* 7b ⟩
⟨ *print ltMatrices* 10 ⟩
⟨ *reorder ltMatrices* 11 ⟩
⟨ *subset ltMatrices* 13 ⟩
⟨ *lower triangular elements* 15 ⟩
⟨ *diagonals ltMatrices* 17 ⟩
⟨ *diagonal matrix* 20 ⟩
⟨ *mult ltMatrices* 21a ⟩
⟨ *mult syMatrices* 25 ⟩
⟨ *solve ltMatrices* 29 ⟩
⟨ *logdet ltMatrices* 31b ⟩
⟨ *tcrossprod ltMatrices* 35 ⟩
⟨ *crossprod ltMatrices* 36 ⟩
⟨ *chol syMatrices* 37 ⟩
⟨ *add diagonal elements* 18 ⟩
⟨ *assign diagonal elements* 19 ⟩
⟨ *kronecker vec trick* 42 ⟩
⟨ *convenience functions* 45 ⟩
⟨ *aperm* 47 ⟩
⟨ *marginal* 48b ⟩
⟨ *conditional* 50b ⟩
⟨ *check obs* 52b ⟩
⟨ *ldmvnorm* 52a ⟩
⟨ *colSumsdnorm ltMatrices* 53b ⟩
⟨ *sldmvnorm* 56 ⟩
⟨ *ldpmvnorm* 94 ⟩
⟨ *sldpmvnorm* 96 ⟩
⟨ *standardize* 98 ⟩
⟨ *destandardize* 100 ⟩
◇

"ltMatrices.c" 3≡

```
⟨ C Header 105 ⟩
#ifndef  USE_FC_LEN_T
# define USE_FC_LEN_T
#endif
#include <Rconfig.h>
#include <R_ext/Lapack.h> /* for dtptri */
#ifndef FCONE
# define FCONE
#endif
#include <R.h>
#include <Rmath.h>
#include <Rinternals.h>
#include <Rdefines.h>
⟨ colSumsdnorm 53a ⟩
⟨ solve 27 ⟩
⟨ solve C 28 ⟩
⟨ logdet 31a ⟩
⟨ tcrossprod 34 ⟩
⟨ mult 22b ⟩
⟨ mult transpose 24 ⟩
⟨ chol 38 ⟩
⟨ vec trick 40a ⟩
◇
```

We first define and implement 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:

$$
\begin{aligned}
\mathbf{C} &= \begin{pmatrix} c_{11} & & & & 0 \\ c_{21} & c_{22} & & & \\ c_{31} & c_{32} & c_{33} & & \\ \vdots & \vdots & & \ddots & \\ c_{J1} & c_{J2} & \cdots & & c_{JJ} \end{pmatrix} \text{ matrix indexing} \\
&= \begin{pmatrix} c_{1} & & & & 0 \\ c_{2} & c_{J+1} & & & \\ c_{3} & c_{J+2} & c_{2J} & & \\ \vdots & \vdots & & \ddots & \\ c_{J} & c_{2J-1} & \cdots & & c_{J(J+1)/2} \end{pmatrix} \text{ column-major, } \texttt{byrow = FALSE} \\
&= \begin{pmatrix} & c_{1} & & & & & 0 \\ & c_{2} & & c_{3} & & & \\ & c_{4} & & c_{5} & c_{6} & & \\ & \vdots & & \vdots & & \ddots & \\ c_{J((J+1)/2-1)+1} & & c_{J((J+1)/2-1)+2} & \cdots & & c_{J(J+1)/2} \end{pmatrix} \text{ row-major, } \texttt{byrow = TRUE}
\end{aligned}
$$

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

⟨ *ltMatrices dim* 4 ⟩ ≡

```
J <- floor((1 + sqrt(1 + 4 * 2 * nrow(object))) / 2 - diag)
if (nrow(object) != J * (J - 1) / 2 + diag * J)
    stop("Dimension of object does not correspond to lower
          triangular part of a square matrix")
```

◇

Fragment referenced in 6a.

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* 5a ⟩ ≡

```
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 6a.

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* 5b ⟩ ≡

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

◇

Fragment referenced in 6a.

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

⟨ *ltMatrices* 6a ⟩ ≡

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

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

    ⟨ ltMatrices input 5b ⟩
    ⟨ ltMatrices dim 4 ⟩
    ⟨ ltMatrices names 5a ⟩

    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.

For the sake of completeness, we also add a constructor for symmetric multiple symmetric matrices

⟨ *syMatrices* 6b ⟩ ≡

```
as.syMatrices <- function(object) {
    stopifnot(inherits(object, "ltMatrices"))
    class(object)[1L] <- "syMatrices"
    return(object)
}
syMatrices <- function(object, diag = FALSE, byrow = FALSE, names = TRUE)
    as.syMatrices(ltMatrices(object = object, diag = diag, byrow = byrow, names = names))
```

◇

Fragment referenced in 2.

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

⟨ *dim ltMatrices* 6c ⟩ ≡

```
dim.ltMatrices <- function(x) {
    J <- attr(x, "J")
    return(c(attr(x, "dim")[2L], J, J)) ### ncol(unclass(x)) may trigger gc
}
dim.syMatrices <- dim.ltMatrices
```

◇

Fragment referenced in 2.

The corresponding dimnames can be extracted as

⟨ *dimnames ltMatrices* 7a ⟩ ≡

```
dimnames.ltMatrices <- function(x)
    return(list(attr(x, "dimnames")[[2L]], 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* 7b ⟩ ≡

```
names.ltMatrices <- function(x) {
    return(attr(x, "dimnames")[[1L]])
}
names.syMatrices <- names.ltMatrices
```

◇

Fragment referenced in 2.

Let's set-up an example for illustration. Throughout this document, we will compare numerical results using

```
> chk <- function(...) stopifnot(isTRUE(all.equal(...)))
```

We start with a a simple example demonstrating how to set-up `ltMatrices` objects

```
> library("mvtnorm")
> set.seed(290875)
> N <- 4L
> J <- 5L
> rn <- paste0("C_", 1:N)
> nm <- LETTERS[1:J]
> Jn <- J * (J - 1) / 2
> ## data
> xn <- matrix(runif(N * Jn), ncol = N)
> colnames(xn) <- rn
> xd <- matrix(runif(N * (Jn + J)), ncol = N)
> 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"

> lxn <- as.syMatrices(lxn)
> 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 objects of class `ltMatrices` to `array`. The method has a logical argument called `symmetric`, forcing the lower triangular matrix to by interpreted as a symmetric matrix.

⟨ *extract slots* 9 ⟩ ≡

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

◇

Fragment referenced in 10, 11, 12, 15, 17, 19, 21a, 25.

⟨ *print ltMatrices* 10 ⟩ ≡

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

    ⟨ extract slots 9 ⟩

    x <- unclass(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 <- rbind(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.

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).

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* 11 ⟩ ≡

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

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

    ⟨ extract slots 9 ⟩

    x <- unclass(x)

    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* 12 ⟩ ≡

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

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

    ⟨ extract slots 9 ⟩

    x <- unclass(x)

    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 13.

⟨ *subset ltMatrices* 13 ⟩ ≡

```
⟨ .subset ltMatrices 12 ⟩
### 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])
> ltM <- as.syMatrices(ltM)
> 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, undoing the effect of `ltMatrices`

⟨ *lower triangular elements* 15 ⟩ ≡

```
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 9 ⟩

    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* 17 ⟩ ≡

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

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

    ⟨ extract slots 9 ⟩

    x <- unclass(x)

    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* 18 ⟩ ≡

```
.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) <- dimnames(x)[[1L]]
    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* 19 ⟩ ≡

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

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

    ⟨ extract slots 9 ⟩

    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* 20 ⟩ ≡

```
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

Products $\mathbf{C}_i \mathbf{y}_i$ 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` later on.

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

⟨ *mult ltMatrices* 21a ⟩ ≡

```
### C %*% y
Mult <- function(x, y, ...)
    UseMethod("Mult")
Mult.default <- function(x, y, transpose = FALSE, ...) {
    if (!transpose) return(x %*% y)
    return(crossprod(x, y))
}
Mult.ltMatrices <- function(x, y, transpose = FALSE, ...) {

    ⟨ extract slots 9 ⟩

    stopifnot(is.numeric(y))
    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 23 ⟩

    x <- ltMatrices(x, byrow = TRUE)
    if (!is.double(x)) storage.mode(x) <- "double"
    if (!is.double(y)) 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* 21b ⟩ ≡

```
/* 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 22b, 24, 27, 28, 31a, 34, 40a.

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* 22a ⟩ ≡

```
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 22b, 24, 27, 28, 31a, 40a.

The C workhorse is now

⟨ *mult* 22b ⟩ ≡

```
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 21b ⟩
    ⟨ C length 22a ⟩

    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 3.

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)
```

For $\mathbf{C}_i^\top \mathbf{y}_i$ (`transpose = TRUE`), we add a dedicated `C` function paying attention to the lower triangular structure of `x`. This function assumes `x` in column-major order, so we coerce this object when necessary:

⟨ *mult ltMatrices transpose* 23 ⟩ ≡

```
if (transpose) {
    x <- ltMatrices(x, byrow = FALSE)
    if (!is.double(x)) storage.mode(x) <- "double"
    if (!is.double(y)) storage.mode(y) <- "double"

    ret <- .Call(mvtnorm_R_ltMatrices_Mult_transpose, 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 21a.

before moving to `C` for the low-level computations:

⟨ *mult transpose* 24 ⟩ ≡

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

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

    ⟨ RC input 21b ⟩
    ⟨ C length 22a ⟩

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

◇

Fragment referenced in 3.

and wrap-up with some tests for computing $\mathbf{C}_i^\top \mathbf{y}_i$

```
> 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))
```

Now we can add a `Mult` method for multiple symmetric matrices, noting that for a symmetric matrix $\mathbf{C} = \mathbf{A} + \mathbf{A}^\top - \mathrm{diag}(\mathbf{A})$ with lower triangular part $\mathbf{A}$ (including the diagonal) we can compute $\mathbf{Cy} = \mathbf{Ay} + \mathbf{A}^\top\mathbf{y} - \mathrm{diag}(\mathbf{A})\mathbf{y}$ using `Mult` applied to the lower trianular part:

⟨ *mult syMatrices* 25 ⟩ ≡

```
Mult.syMatrices <- function(x, y, ...) {

    ⟨ extract slots 9 ⟩

    class(x)[1L] <- "ltMatrices"
    stopifnot(is.numeric(y))
    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])
    stopifnot(ncol(y) == N)

    ret <- Mult(x, y) + Mult(x, y, transpose = TRUE) - y * c(diagonals(x))
    return(ret)
}
```

◇

Fragment referenced in 2.

```
> J <- 5
> N1 <- 10
> ex <- expression({
+   C <- syMatrices(matrix(runif(N2 * J * (J + c(-1, 1)[DIAG + 1L] ) / 2), ncol = N2),
+                   diag = DIAG)
+   x <- matrix(runif(N1 * J), nrow = J)
+   Ca <- as.array(C)
+   p1 <- do.call("cbind", lapply(1:N1, function(i)
+       Ca[,,c(1,i)[(N2 > 1) + 1]] %*% x[,i]))
+   p2 <- Mult(C, x)
+   chk(p1, p2)
+ })
> N2 <- N1
> DIAG <- TRUE
> eval(ex)
> N2 <- 1
> DIAG <- TRUE
> eval(ex)
> N2 <- 1
> DIAG <- FALSE
> eval(ex)
> N2 <- N1
> DIAG <- FALSE
> eval(ex)
```

2.7 Solving Linear Systems

Computing $\mathbf{C}_i^{-1}$ or solving $\mathbf{C}_i\mathbf{x}_i = \mathbf{y}_i$ for $\mathbf{x}_i$ for all $i = 1, \ldots, N$ is another important task. 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 $J(J-1)/2+J\mathrm{diag}\times N$), 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 the same order as the orginal matrix $\mathbf{C}_i$. If all $\mathbf{C}_i$ have unit diagonals, so will $\mathbf{C}_i^{-1}$.

We start with some options for the LAPACK workhorses

⟨ *lapack options* 26 ⟩ ≡

```
char di, lo = 'L', tr = 'N';
if (Rdiag) {
    /* non-unit diagonal elements */
    di = 'N';
} else {
    /* unit diagonal elements; NOTE: these diagonals 1s ARE always present but
       ignored in the computations */
    di = 'U';
}

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

◇

Fragment referenced in 27, 28.

and set-up a dedicated C function for computing $\mathbf{C}_i\mathbf{x}_i = \mathbf{y}_i$

⟨ *solve* 27 ⟩ ≡

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

    SEXP ans;
    double *dans, *dy;
    int i, j, info, ONE = 1;

    ⟨ RC input 21b ⟩
    /* diagonal elements are always present */
    if (!Rdiag) len += iJ;
    ⟨ C length 22a ⟩
    ⟨ lapack options 26 ⟩

    dy = REAL(y);
    PROTECT(ans = allocMatrix(REALSXP, iJ, iN));
    dans = REAL(ans);
    memcpy(dans, dy, iJ * iN * sizeof(double));

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

        /* solve linear system */
        F77_CALL(dtpsv)(&lo, &tr, &di, &iJ, dC, dans, &ONE FCONE FCONE FCONE);
        dans += iJ;
        dC += p;
    }

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

◇

Fragment referenced in 3.

and then for computing $\mathbf{C}_i^{-1}$ explicitly

⟨ *solve C* 28 ⟩ ≡

```
SEXP R_ltMatrices_solve_C (SEXP C, SEXP N, SEXP J, SEXP diag, SEXP transpose)
{

    SEXP ans;
    double *dans;
    int i, j, info, jj, idx, ONE = 1;

    ⟨ RC input 21b ⟩
    /* diagonal elements are always present */
    if (!Rdiag) len += iJ;
    ⟨ C length 22a ⟩
    ⟨ lapack options 26 ⟩

    PROTECT(ans = allocMatrix(REALSXP, len, iN));
    dans = REAL(ans);
    memcpy(dans, dC, iN * len * sizeof(double));

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

        /* compute inverse */
        F77_CALL(dtptri)(&lo, &di, &iJ, dans, &info FCONE FCONE);
        if (info != 0)
            error("Cannot solve ltmatices");

        dans += len;
    }

    UNPROTECT(1);
    /* note: ans always includes diagonal elements */
    return(ans);
}
```

◇

Fragment referenced in 3.

with R interface

⟨ *solve ltMatrices* 29 ⟩ ≡

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

    byrow_orig <- attr(a, "byrow")

    x <- ltMatrices(a, byrow = FALSE)
    diag <- attr(x, "diag")
    ### dtptri and dtpsv require diagonal elements being present
    if (!diag) diagonals(x) <- diagonals(x)
    d <- dim(x)
    J <- d[2L]
    dn <- dimnames(x)
    if (!is.double(x)) storage.mode(x) <- "double"

    if (!missing(b)) {
        if (!is.matrix(b)) b <- matrix(b, nrow = J, ncol = d[1L])
        stopifnot(nrow(b) == J)
        N <- ifelse(d[1L] == 1, ncol(b), d[1L])
        stopifnot(ncol(b) == N)
        if (!is.double(b)) 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 <- .Call(mvtnorm_R_ltMatrices_solve_C, x,
                 as.integer(d[1L]), 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 Log-determinants

For computing the log-determinant $\log(\det(\mathbf{C}_i)) = \sum_{j=1}^{J} \log(\mathrm{diag}(\mathbf{C}_i)_j)$ we sum over the log-diagonal entries of a lower triangular matrix in C, both when the data are stored in row- and column-major order:

⟨ *logdet* 31a ⟩ ≡

```
SEXP R_ltMatrices_logdet (SEXP C, SEXP N, SEXP J, SEXP diag, SEXP byrow) {

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

    ⟨ RC input 21b ⟩
    Rboolean Rbyrow = asLogical(byrow);
    ⟨ C length 22a ⟩

    PROTECT(ans = allocVector(REALSXP, iN));
    dans = REAL(ans);

    for (i = 0; i < iN; i++) {
        dans[i] = 0.0;
        if (Rdiag) {
            k = 1;
            for (j = 0; j < iJ; j++) {
                dans[i] += log(dC[k - 1]);
                k += (Rbyrow ? j + 2 : iJ - j);
            }
            dC += p;
        }
    }

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

◇

Fragment referenced in 3.

The R interface now simply calls this low-level function

⟨ *logdet ltMatrices* 31b ⟩ ≡

```
logdet <- function(x) {

    if (!inherits(x, "ltMatrices"))
        stop("x is not an ltMatrices object")

    byrow <- attr(x, "byrow")
    diag <- attr(x, "diag")
    d <- dim(x)
    J <- d[2L]
    dn <- dimnames(x)
    if (!is.double(x)) storage.mode(x) <- "double"

    ret <- .Call(mvtnorm_R_ltMatrices_logdet, x,
                 as.integer(d[1L]), as.integer(J), as.logical(diag),
                 as.logical(byrow))
    names(ret) <- dn[[1L]]
    return(ret)
}
```

◇

Fragment referenced in 2.

We test the functionality by extracting the diagonal elements from different matrices and summing over their logarithms

```
> chk(logdet(lxn), colSums(log(diagonals(lxn))))
> chk(logdet(lxd[1,]), colSums(log(diagonals(lxd[1,]))))
> chk(logdet(lxd), colSums(log(diagonals(lxd))))
> lxd2 <- ltMatrices(lxd, byrow = !attr(lxd, "byrow"))
> chk(logdet(lxd2), colSums(log(diagonals(lxd2))))
```

2.9 Crossproducts

Compute $\mathbf{C}_i\mathbf{C}_i^\top$ or $\text{diag}(\mathbf{C}_i\mathbf{C}_i^\top)$ (`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* 32a ⟩ ≡

```
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 32b, 33a.

⟨ *tcrossprod diagonal only* 32b ⟩ ≡

```
PROTECT(ans = allocMatrix(REALSXP, iJ, iN));
dans = REAL(ans);
for (n = 0; n < iN; n++) {
    ⟨ first element 32a ⟩
    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 34.

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

⟨ *tcrossprod full* 33a ⟩ ≡

```
nrow = iJ * (iJ + 1) / 2;
PROTECT(ans = allocMatrix(REALSXP, nrow, iN));
dans = REAL(ans);
for (n = 0; n < INTEGER(N)[0]; n++) {
    ⟨ first element 32a ⟩
    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 34.

and put both cases together

⟨ *IDX* 33b ⟩ ≡

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

Fragment referenced in 34, 40a.

⟨ *tcrossprod* 34 ⟩ ≡

```
⟨ IDX 33b ⟩

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 21b ⟩

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

    if (Rdiag_only) {
        ⟨ tcrossprod diagonal only 32b ⟩
    } else {
        ⟨ tcrossprod full 33a ⟩
    }
    UNPROTECT(1);
    return(ans);
}
```

◇

Fragment referenced in 3.

with R interface

⟨ *tcrossprod ltMatrices* 35 ⟩ ≡

```
### 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)
    N <- d[1L]
    J <- d[2L]
    dn <- dimnames(x)

    x <- ltMatrices(x, byrow = FALSE)
    if (!is.double(x)) 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 <- as.syMatrices(ltMatrices(ret, byrow = byrow_orig))
    }
    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* 36 ⟩ ≡

```
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)))
```

2.10 Cholesky Factorisation

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

⟨ *chol syMatrices* 37 ⟩ ≡

```
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]
    if (!is.double(x)) 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* 38 ⟩ ≡

```
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 3.

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))
```

2.11 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)* 39 ⟩ ≡

```
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 40a.

⟨ *vec trick* 40a ⟩ ≡

```
⟨ IDX 33b ⟩

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 21b ⟩
    ⟨ C length 22a ⟩
    dS = REAL(S);
    dA = REAL(A);

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

    ⟨ t(C) S t(A) 39 ⟩

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

Fragment referenced in 3.

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* 40b ⟩ ≡

```
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]
if (!is.double(C)) storage.mode(C) <- "double"
◇
```

Fragment referenced in 42.

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

⟨ *check S argument* 41a ⟩ ≡

```
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 <- matrix(as.array(S), ncol = dS[1L])
} else {
    stopifnot(is.matrix(S))
    stopifnot(nrow(S) == J^2)
    if (dC[1] != 1L) {
        stopifnot(dC[1L] == ncol(S))
    } else {
        N <- ncol(S)
    }
}
if (!is.double(S)) storage.mode(S) <- "double"
```

◇

Fragment referenced in 42.

A is an `ltMatrices` object

⟨ *check A argument* 41b ⟩ ≡

```
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]
    if (!is.double(A)) 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 42.

We put everything together in function `vectrick`

⟨ *kronecker vec trick* 42 ⟩ ≡

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

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

    ⟨ check C argument 40b ⟩
    ⟨ check S argument 41a ⟩
    ⟨ check A argument 41b ⟩

    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)
> SD2a <- -vectrick(a, b)
> chk(SD2, SD2a)
> 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)
> SD2a <- -vectrick(a, S)
> chk(SD2, SD2a)
> chk(c(SD0), c(SD2))
> ### N > 1
```

```
> N <- 4L
> 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)
```

2.12 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* 43 ⟩ ≡

```
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 45.

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* 44 ⟩ ≡

```
### 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 45.

and now the convenience functions are one-liners:

⟨ *convenience functions* 45 ⟩ ≡

```
⟨ D times C 43 ⟩
⟨ L times D 44 ⟩

### 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)
```

We also add an `aperm` method for class `ltMatrices`

⟨ *aperm* 47 ⟩ ≡

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

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

2.13 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* 48a ⟩ ≡

```
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 48b, 50b.

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* 48b ⟩ ≡

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

    ⟨ mc input checks 48a ⟩

    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* 49 ⟩ ≡

```
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 50b.

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* 50a ⟩ ≡

```
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 50b.

⟨ *conditional* 50b ⟩ ≡

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

    which <- which_given
    ⟨ mc input checks 48a ⟩

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

    ⟨ cond simple 50a ⟩
    ⟨ cond general 49 ⟩

    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])
```

2.14 Continuous Log-likelihoods

With $\mathbf{Z} \sim \mathbb{N}_J(0, \mathbf{I}_J)$ and $\mathbf{Y} = \mathbf{C}_i\mathbf{Z} + \boldsymbol{\mu}_i \sim \mathbb{N}_J(\boldsymbol{\mu}_i, \mathbf{C}_i\mathbf{C}_i^\top)$ we want to evaluate the log-likelihood contributions for observations $\mathbf{y}_1, \ldots, \mathbf{y}_N$ in a function called `ldmvnorm`

⟨ *ldmvnorm* 52a ⟩ ≡

```
ldmvnorm <- function(obs, mean = 0, chol, invchol, logLik = TRUE) {

    stopifnot(xor(missing(chol), missing(invchol)))
    if (!is.matrix(obs)) obs <- matrix(obs, ncol = 1L)
    p <- ncol(obs)

    if (!missing(chol)) {
        ⟨ ldmvnorm chol 54a ⟩
    } else {
        ⟨ ldmvnorm invchol 54b ⟩
    }

    names(logretval) <- colnames(obs)
    if (logLik) return(sum(logretval))
    return(logretval)
}
```

◇

Fragment referenced in 2.

We first check if the observations $\mathbf{y}_1, \ldots, \mathbf{y}_N$ are given in an $J \times N$ matrix `obs` with corresponding means $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_N$ in `means`.

⟨ *check obs* 52b ⟩ ≡

```
.check_obs <- function(obs, mean, J, N) {

    nr <- nrow(obs)
    nc <- ncol(obs)
    if (nc != N)
        stop("obs and (inv)chol have non-conforming size")
    if (nr != J)
        stop("obs and (inv)chol have non-conforming size")
    if (identical(unique(mean), 0)) return(obs)
    if (length(mean) == J)
        return(obs - c(mean))
    if (!is.matrix(mean))
        stop("obs and mean have non-conforming size")
    if (nrow(mean) != nr)
        stop("obs and mean have non-conforming size")
    if (ncol(mean) != nc)
        stop("obs and mean have non-conforming size")
    return(obs - mean)
}
```

◇

Fragment referenced in 2.

With $\boldsymbol{\Sigma}_i = \mathbf{C}_i \mathbf{C}_i^\top$ the log-likelihood function for $\mathbf{Y}_i = \mathbf{y}_i$ is

$$\ell_i(\boldsymbol{\mu}_i, \mathbf{C}_i) = -\frac{k}{2} \log(2\pi) - \frac{1}{2} \log \mid \boldsymbol{\Sigma}_i \mid -\frac{1}{2} (\mathbf{y}_i - \boldsymbol{\mu}_i)^\top \boldsymbol{\Sigma}_i^{-1} (\mathbf{y}_i - \boldsymbol{\mu}_i)$$

Because $\log \mid \mathbf{\Sigma}_i \mid = \log \mid \mathbf{C}_i \mathbf{C}_i^\top \mid = 2 \log \mid \mathbf{C}_i \mid = 2 \sum_{j=1}^J \log \operatorname{diag}(\mathbf{C}_i)_j$ we get the simpler expression

$$\ell_i(\boldsymbol{\mu}_i, \mathbf{C}_i) \quad = \quad -\frac{k}{2} \log(2\pi) - \sum_{j=1}^J \log \operatorname{diag}(\mathbf{C}_i)_j - \frac{1}{2} (\mathbf{y}_i - \boldsymbol{\mu}_i)^\top \mathbf{C}_i^{-\top} \mathbf{C}_i^{-1} (\mathbf{y} - \boldsymbol{\mu}_i). \tag{2.1}$$

We need to compute `colSums(dnorm(z, log = TRUE))` quite often. This turns out to be time-consuming and memory intensive, so we provide a small internal helper function focusing on the necessary computations.

⟨ *colSumsdnorm* 53a ⟩ ≡

```
SEXP R_ltMatrices_colSumsdnorm (SEXP z, SEXP N, SEXP J) {
    /* number of columns */
    int iN = INTEGER(N)[0];
    /* number of rows */
    int iJ = INTEGER(J)[0];
    SEXP ans;
    double *dans, Jl2pi, *dz;

    Jl2pi = iJ * log(2 * PI);
    PROTECT(ans = allocVector(REALSXP, iN));
    dans = REAL(ans);
    dz = REAL(z);

    for (int i = 0; i < iN; i++) {
        dans[i] = 0.0;
        for (int j = 0; j < iJ; j++)
            dans[i] += pow(dz[j], 2);
        dans[i] = - 0.5 * (Jl2pi + dans[i]);
        dz += iJ;
    }

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

◇

Fragment referenced in 3.

⟨ *colSumsdnorm ltMatrices* 53b ⟩ ≡

```
.colSumsdnorm <- function(z) {
    stopifnot(is.numeric(z))
    if (!is.matrix(z))
        z <- matrix(z, nrow = 1, ncol = length(z))
    ret <- .Call(mvtnorm_R_ltMatrices_colSumsdnorm, z, ncol(z), nrow(z))
    names(ret) <- colnames(z)
    return(ret)
}
```

◇

Fragment referenced in 2.

The main part is now

⟨ *ldmvnorm chol* 54a ⟩ ≡

```
if (missing(chol))
    stop("either chol or invchol must be given")
## chol is given
if (!inherits(chol, "ltMatrices"))
    stop("chol is not an object of class ltMatrices")
N <- dim(chol)[1L]
N <- ifelse(N == 1, p, N)
J <- dim(chol)[2L]
obs <- .check_obs(obs = obs, mean = mean, J = J, N = N)
z <- solve(chol, obs)
logretval <- .colSumsdnorm(z)
if (attr(chol, "diag"))
    logretval <- logretval - logdet(chol)
```

◇

Fragment referenced in 52a.

where we can use the efficient implementations of `solve` and `logdet`.

If $\mathbf{L}_i = \mathbf{C}_i^{-1}$ is given, we obtain

$$\ell_i(\boldsymbol{\mu}_i, \mathbf{L}_i) \quad = \quad -\frac{k}{2}\log(2\pi) + \sum_{j=1}^{J} \log \operatorname{diag}(\mathbf{L}_i)_j - \frac{1}{2}(\mathbf{y}_i - \boldsymbol{\mu}_i)^\top \mathbf{L}_i^\top \mathbf{L}_i (\mathbf{y} - \boldsymbol{\mu}_i).$$

⟨ *ldmvnorm invchol* 54b ⟩ ≡

```
## invchol is given
if (!inherits(invchol, "ltMatrices"))
    stop("invchol is not an object of class ltMatrices")
N <- dim(invchol)[1L]
N <- ifelse(N == 1, p, N)
J <- dim(invchol)[2L]
obs <- .check_obs(obs = obs, mean = mean, J = J, N = N)
## NOTE: obs is (J x N)
## dnorm takes rather long
z <- Mult(invchol, obs)
logretval <- .colSumsdnorm(z)
## note that the second summand gets recycled the correct number
## of times in case dim(invchol)[1L] == 1 but ncol(obs) > 1
if (attr(invchol, "diag"))
    logretval <- logretval + logdet(invchol)
```

◇

Fragment referenced in 52a.

The score function with respect to `obs` is

$$\frac{\partial \ell_i(\boldsymbol{\mu}_i, \mathbf{L}_i)}{\partial \mathbf{y}_i} = -\mathbf{L}_i^\top \mathbf{L}_i \mathbf{y}_i$$

and with respect to `invchol` we have

$$\frac{\partial \ell_i(\boldsymbol{\mu}_i, \mathbf{L}_i)}{\partial \mathbf{L}_i} = -2\mathbf{L}_i \mathbf{y}_i \mathbf{y}_i^\top + \operatorname{diag}(\mathbf{L}_i)^{-1}.$$

The score function with respect to `chol` post-processes the above score using the vec trick (Section 2.11). For the log-likelihood (2.1), the score with respect to $\mathbf{C}_i$ is the sum of the score functions of the two terms. We start with the simpler first term

$$\frac{\partial - \sum_{j=1}^{J} \log \operatorname{diag}(\mathbf{C}_i)_j}{\partial \mathbf{C}_i} = -\operatorname{diag}(\mathbf{C}_i)^{-1}$$

The second term gives (we omit the mean for the sake of simplicity)

$$\begin{aligned}
\frac{\partial - \mathbf{y}_i^\top \mathbf{C}_i^{-\top} \mathbf{C}_i^{-1} \mathbf{y}_i}{\partial \mathbf{C}_i} &= - \frac{\partial \mathbf{y}_i^\top \mathbf{A}^\top \mathbf{A} \mathbf{y}_i}{\partial \mathbf{A}}\bigg|_{\mathbf{A}=\mathbf{C}_i^{-1}} \frac{\partial \mathbf{A}^{-1}}{\partial \mathbf{A}}\bigg|_{\mathbf{A}=\mathbf{C}_i} \\
&= -2\operatorname{vec}(\mathbf{C}_i^{-1} \mathbf{y}_i \mathbf{y}_i^\top)^\top (-1)(\mathbf{C}_i^{-\top} \otimes \mathbf{C}_i^{-1}) \\
&= 2\operatorname{vec}(\mathbf{C}_i^{-\top} \mathbf{C}_i^{-1} \mathbf{y}_i \mathbf{y}_i^\top \mathbf{C}_i^{-\top})^\top
\end{aligned}$$

In `sldmvnorm`, we compute the score with respect to $\mathbf{L}_i$ and use the above relationship to compute the score with respect to $\mathbf{C}_i$.

⟨ *sldmvnorm* 56 ⟩ ≡

```
sldmvnorm <- function(obs, mean = 0, chol, invchol, logLik = TRUE) {

    stopifnot(xor(missing(chol), missing(invchol)))
    if (!is.matrix(obs)) obs <- matrix(obs, ncol = 1L)

    if (!missing(invchol)) {

        N <- dim(invchol)[1L]
        N <- ifelse(N == 1, ncol(obs), N)
        J <- dim(invchol)[2L]
        obs <- .check_obs(obs = obs, mean = mean, J = J, N = N)

        Mix <- Mult(invchol, obs)
        sobs <- - Mult(invchol, Mix, transpose = TRUE)

        Y <- matrix(obs, byrow = TRUE, nrow = J, ncol = N * J)
        ret <- - matrix(Mix[, rep(1:N, each = J)] * Y, ncol = N)

        M <- matrix(1:(J^2), nrow = J, byrow = FALSE)
        ret <- ltMatrices(ret[M[lower.tri(M, diag = attr(invchol, "diag"))],,drop = FALSE],
                          diag = attr(invchol, "diag"), byrow = FALSE)
        ret <- ltMatrices(ret,
                          diag = attr(invchol, "diag"), byrow = attr(invchol, "byrow"))
        if (attr(invchol, "diag")) {
            ### recycle properly
            diagonals(ret) <- diagonals(ret) + c(1 / diagonals(invchol))
        } else {
            diagonals(ret) <- 0
        }
        ret <- list(obs = sobs, invchol = ret)
        if (logLik)
            ret$logLik <- ldmvnorm(obs = obs, mean = mean, invchol = invchol, logLik = FALSE)
        return(ret)
    }

    invchol <- solve(chol)
    ret <- sldmvnorm(obs = obs, mean = mean, invchol = invchol)
    ### this means: ret$chol <- - vectrick(invchol, ret$invchol, invchol)
    ret$chol <- - vectrick(invchol, ret$invchol)
    ret$invchol <- NULL
    return(ret)
}
```

◇

Fragment referenced in 2.

2.15 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 <- 1000L
> J <- 50L
```

```
> 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 `ldmvnorm` function now also has `chol` and `invchol` arguments such that we can use

```
> ll3 <- ldmvnorm(obs = Y, invchol = lt)
> chk(ll1, ll3)
```

Note that argument `obs` in `ldmvnorm` is an $J \times N$ matrix whereas the traditional interface in `dmvnorm` expects an $N \times J$ matrix `x`. The reason is that `Mult` or `solve` work with $J \times N$ 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` function, so a prototype might look like

⟨ *lpmvnormR* 58 ⟩ ≡

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

    ⟨ input checks 60a ⟩

    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 simplified 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 <- 5L
> N <- 10L
> 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, by making the code more modular and, most importantly, by providing score functions for all arguments $\mathbf{a}_i$, $\mathbf{b}_i$, and $\mathbf{C}_i$.

3.1 Algorithm

"lpmvnorm.R" 59a≡

⟨ *R Header* 104 ⟩
⟨ *lpmvnorm* 69 ⟩
⟨ *slpmvnorm* 82 ⟩
◇

"lpmvnorm.c" 59b≡

```
⟨ C Header 105 ⟩
#ifndef  USE_FC_LEN_T
# define USE_FC_LEN_T
#endif
#include <Rconfig.h>
#include <R_ext/BLAS.h> /* for dtrmm */
#ifndef FCONE
# define FCONE
#endif
#include <R.h>
#include <Rmath.h>
#include <Rinternals.h>
#include <Rdefines.h>
⟨ pnorm fast 64a ⟩
⟨ pnorm slow 64b ⟩
⟨ R lpmvnorm 67 ⟩
⟨ R slpmvnorm 79 ⟩
◇
```

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* 60a ⟩ ≡

```
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 58, 69, 82.

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

⟨ *standardise* 60b ⟩ ≡

```
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)
    C <- Dchol(chol, D = 1 / dchol)
    if (J > 1) { ### else: univariate problem; C is no longer used
        uC <- Lower_tri(C)
    } else {
        uC <- unclass(C)
    }
} else {
    ac <- lower
    bc <- upper
    uC <- Lower_tri(chol)
}
```

◇

Fragment referenced in 69, 82.

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* 61a ⟩ ≡

```
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 67, 79.

4. Repeat

⟨ *init logLik loop* 61b ⟩ ≡

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

◇

Fragment referenced in 67, 73b.

(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* 61c ⟩ ≡

```
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 62d, 77a.

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

⟨ *compute x* 62a ⟩ ≡

```
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 62d, 77a.

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

⟨ *update d, e* 62b ⟩ ≡

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

◇

Fragment referenced in 62d, 77a.

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

⟨ *update f* 62c ⟩ ≡

```
start += j;
f *= emd;
```

◇

Fragment referenced in 62d, 77a.

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

⟨ *inner logLik loop* 62d ⟩ ≡

```
for (j = 1; j < iJ; j++) {

    ⟨ compute y 61c ⟩
    ⟨ compute x 62a ⟩
    ⟨ update d, e 62b ⟩
    ⟨ update f 62c ⟩
}
```

◇

Fragment referenced in 67.

(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* 63a ⟩ ≡

```
intsum += f;
```
◇

Fragment referenced in 67.

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* 63b ⟩ ≡

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

Fragment referenced in 67.

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

⟨ *move on* 63c ⟩ ≡

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

Fragment referenced in 67, 79.

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* 64a ⟩ ≡

```
/* 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 = -2.92345273673194627762e-06;
const double g10 = 1.34797733516989204361e-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 59b.

⟨ *pnorm slow* 64b ⟩ ≡

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

Fragment referenced in 59b.

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

⟨ *pnorm* 64c ⟩ ≡

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

Fragment referenced in 67, 79.

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* 65a ⟩ ≡

```
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 67, 79.

⟨ *dimensions* 65b ⟩ ≡

```
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 67, 79.

⟨ *setup return object* 65c ⟩ ≡

```
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 67.

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

⟨ *univariate problem* 66a ⟩ ≡

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

◇

Fragment referenced in 67.

⟨ *init center* 66b ⟩ ≡

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

◇

Fragment referenced in 67, 79.

We put the code together in a dedicated C function

⟨ *R slpmvnorm variables* 66c ⟩ ≡

```
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;
```

◇

Fragment referenced in 67, 79.

⟨ *R lpmvnorm* 67 ⟩ ≡

```
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) {

    ⟨ R slpmvnorm variables 66c ⟩
    double l0, lM, x0, intsum;
    int p, len;

    Rboolean RlogLik = asLogical(logLik);

    ⟨ pnorm 64c ⟩
    ⟨ dimensions 65b ⟩
    ⟨ W length 65a ⟩
    ⟨ init center 66b ⟩

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

    ⟨ setup return object 65c ⟩

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

    ⟨ univariate problem 66a ⟩

    if (W == R_NilValue)
        GetRNGstate();

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

        x0 = 0;
        ⟨ initialisation 61a ⟩

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

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

            ⟨ init logLik loop 61b ⟩
            ⟨ inner logLik loop 62d ⟩
            ⟨ increment 63a ⟩

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

        ⟨ output 63b ⟩
        ⟨ move on 63c ⟩
    }

    if (W == R_NilValue)
        PutRNGstate();

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

◇

Fragment referenced in 59b.

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* 68a ⟩ ≡

```
### 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 69, 82.

⟨ *check and / or set integration weights* 68b ⟩ ≡

```
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))
    if (!is.double(w)) 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 69, 82.

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* 68c ⟩ ≡

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

◇

Fragment referenced in 69, 82.

⟨ *lpmvnorm* 69 ⟩ ≡

```
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 68a ⟩
    ⟨ Cholesky of precision 68c ⟩
    ⟨ input checks 60a ⟩
    ⟨ standardise 60b ⟩
    ⟨ check and / or set integration weights 68b ⟩

    ret <- .Call(mvtnorm_R_lpmvnorm, ac, bc, uC, 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 59a.

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 <- 10000L
> 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* 71a ⟩ ≡

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

◇

Fragment referenced in 72a.

and the derivates with respect to the mean

⟨ *mean scores* 71b ⟩ ≡

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

◇

Fragment referenced in 72a.

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

⟨ *lower scores* 71c ⟩ ≡

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

◇

Fragment referenced in 72a.

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

⟨ *upper scores* 71d ⟩ ≡

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

◇

Fragment referenced in 72a.

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* 72a ⟩ ≡

```
int Jp = iJ * (iJ + 1) / 2;
⟨ chol scores 71a ⟩
⟨ mean scores 71b ⟩
⟨ lower scores 71c ⟩
⟨ upper scores 71d ⟩
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 79.

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 $c_{jj}^{(i)}/c_{jj}^{(i)}$ 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* 72b ⟩ ≡

   ```
   if (LENGTH(center)) {
       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);
   } else {
       dp_c[0] = (R_FINITE(da[0]) ? dnorm(da[0], x0, 1.0, 0L) * (da[0] - x0) : 0);
       ep_c[0] = (R_FINITE(db[0]) ? dnorm(db[0], x0, 1.0, 0L) * (db[0] - x0) : 0);
   }
   fp_c[0] = ep_c[0] - dp_c[0];
   ◇
   ```

   Fragment referenced in 73b, 79.

   ⟨ *score a, b* 73a ⟩ ≡

   ```
   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 73b, 79.

4. Repeat

⟨ *init score loop* 73b ⟩ ≡

```
⟨ init logLik loop 61b ⟩
⟨ score c11 72b ⟩
⟨ score a, b 73a ⟩
◇
```

Fragment referenced in 79.

(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* 73c ⟩ ≡

```
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 77a.

⟨ *update yp for means, lower and upper* 74 ⟩ ≡

```
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 77a.

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

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

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

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

⟨ *score wrt new chol off-diagonals* 75a ⟩ ≡

```
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 77a.

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

⟨ *score wrt new chol diagonal* 75b ⟩ ≡

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

◇

Fragment referenced in 77a.

⟨ *new score means, lower and upper* 75c ⟩ ≡

```
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 77a.

We next update scores for parameters introduced for smaller j

⟨ *update score for chol* 76a ⟩ ≡

```
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 77a.

⟨ *update score means, lower and upper* 76b ⟩ ≡

```
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 77a.

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

⟨ *inner score loop* 77a ⟩ ≡

```
for (j = 1; j < iJ; j++) {

    ⟨ compute y 61c ⟩
    ⟨ compute x 62a ⟩
    ⟨ update d, e 62b ⟩
    ⟨ update yp for chol 73c ⟩
    ⟨ update yp for means, lower and upper 74 ⟩
    ⟨ score wrt new chol off-diagonals 75a ⟩
    ⟨ score wrt new chol diagonal 75b ⟩
    ⟨ new score means, lower and upper 75c ⟩
    ⟨ update score for chol 76a ⟩
    ⟨ update score means, lower and upper 76b ⟩
    ⟨ update f 62c ⟩

}
```

◇

Fragment referenced in 79.

(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* 77b ⟩ ≡

```
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 79.

⟨ *init dans* 77c ⟩ ≡

```
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];
}
```
◇

Fragment referenced in 79.

We put everything together in C

⟨ *R slpmvnorm* 79 ⟩ ≡

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

    ⟨ R slpmvnorm variables 66c ⟩
    double intsum;
    int p, idx;

    ⟨ dimensions 65b ⟩
    ⟨ pnorm 64c ⟩
    ⟨ W length 65a ⟩
    ⟨ init center 66b ⟩

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

    ⟨ score output object 72a ⟩

    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 61a ⟩
        ⟨ score c11 72b ⟩
        ⟨ score a, b 73a ⟩
        ⟨ init dans 77c ⟩

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

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

            ⟨ init score loop 73b ⟩
            ⟨ inner score loop 77a ⟩
            ⟨ score output 77b ⟩

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

        ⟨ move on 63c ⟩

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

    if (W == R_NilValue)
        PutRNGstate();

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

◇

Fragment referenced in 59b.

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* 80a ⟩ ≡

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

◇

Fragment referenced in 82.

⟨ *post differentiate lower score* 80b ⟩ ≡

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

◇

Fragment referenced in 82.

⟨ *post differentiate upper score* 80c ⟩ ≡

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

◇

Fragment referenced in 82.

⟨ *post differentiate chol score* 80d ⟩ ≡

```
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 82.

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 2.11)

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

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

⟨ *post differentiate invchol score* 81a ⟩ ≡

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

◇

Fragment referenced in 82.

If the diagonal elements are 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* 81b ⟩ ≡

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

◇

Fragment referenced in 82.

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

⟨ *slpmvnorm* 82 ⟩ ≡

```
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 68a ⟩
    ⟨ Cholesky of precision 68c ⟩
    ⟨ input checks 60a ⟩
    ⟨ standardise 60b ⟩
    ⟨ check and / or set integration weights 68b ⟩

    ret <- .Call(mvtnorm_R_slpmvnorm, ac, bc, uC, 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 ### NOTE: division by zero MAY happen,
                                      ### catch outside

    ⟨ post differentiate mean score 80a ⟩
    ⟨ post differentiate lower score 80b ⟩
    ⟨ post differentiate upper score 80c ⟩

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

    ⟨ post differentiate chol score 80d ⟩
    ⟨ post differentiate invchol score 81a ⟩
    ⟨ post process score 81b ⟩

    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 59a.

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

```
> J <- 5L
> N <- 4L
> 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 <- 10000L
> 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
> ### ATLAS and M1mac print 0 as something < .Machine$double.eps
> round(Sigma <- diag(sqrt(1:J / 2)) %*% R %*% diag(sqrt(1:J / 2)), 7)

          [,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 $\mathbf{Y} = \mathbf{C}\mathbf{Z} + \boldsymbol{\mu} \sim \mathbb{N}_J(\boldsymbol{\mu}, \mathbf{C}\mathbf{C}^\top)$

```
> N <- 100L
> 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 $\boldsymbol{\mu}$ 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)
+     -ldmvnorm(obs = Yc, chol = C)
+ }
> sc <- function(parm) {
+     C <- ltMatrices(parm, diag = TRUE, byrow = BYROW)
+     -rowSums(unclass(sldmvnorm(obs = 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

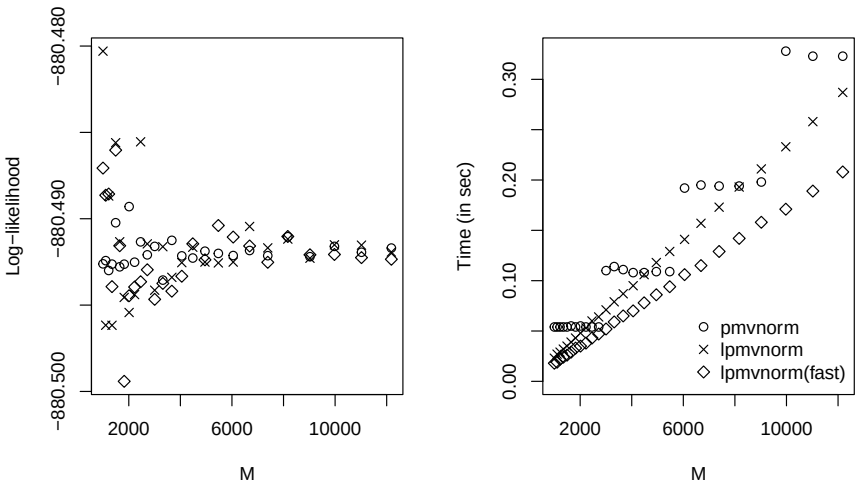

Figure 4.1: Evaluated log-likelihoods (left) and timings (right).

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

[1] 880.4956

> ### ATLAS gives -880.4908, M1mac gives -880.4911
> round(lpmvnormR(lwr, upr, mean = mn, chol = lt,
+                 algorithm = GenzBretz(maxpts = M, abseps = 0, releps = 0)), 3)

[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 $\boldsymbol{\mu}$ 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

```
> ### ATLAS print issues
> round(Tcrossprod(lt), 7)  ### 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

> round(Tcrossprod(C), 7)   ### interval-censored obs

, , 1

          1         2         3         4
1 0.4495644 0.1794540 0.3638631 0.0350230
2 0.1794540 1.1167747 0.0940557 1.0202488
3 0.3638631 0.0940557 1.5368386 0.0734113
4 0.0350230 1.0202488 0.0734113 1.9029791

> round(Shat, 7)            ### "exact" obs

          1         2         3         4
1 0.4665566 0.1810443 0.3422224 0.0160918
2 0.1810443 0.9438534 0.0899225 0.8430953
3 0.3422224 0.0899225 1.3605492 0.0810409
4 0.0160918 0.8430953 0.0810409 1.6330153
```

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 <- try(t(chol(H)))
> ### some check on r-oldrel-macos-arm64
> if (inherits(L, "try-error"))
+     L <- t(chol(H + 1e-4 * diag(nrow(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 different from 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

Continuous-discrete Likelihoods

We sometimes are faced with outcomes measured at different levels of precision. Some variables might have been observed very exactly, and therefore we might want to use the log-Lebesque density for defining the log-likelihood. Other variables might be available as relatively wide intervals only, and thus the log-likelihood is a log-probability. We can use the infrastructure developed so far to compute a joint likelihood. Let's assume we have are interested in the joint distribution of $(\mathbf{Y}_i, \mathbf{X}_i)$ and we observed $\mathbf{Y}_i = \mathbf{y}_i$ (that is, exact observations of $\mathbf{Y}$) and $\mathbf{a}_i < \mathbf{X}_i \leq \mathbf{b}_i$ (that is, interval-censored observations for $\mathbf{X}_i$). We define the log-likelihood based on the joint normal distribution $(\mathbf{Y}_i, \mathbf{X}_i) \sim \mathbb{N}_J((\boldsymbol{\mu}_i, \boldsymbol{\eta}_i)^\top, \mathbf{C}_i\mathbf{C}_i^\top)$ as

$$\ell_i(\boldsymbol{\mu}_i, \boldsymbol{\eta}_i, \mathbf{C}_i) = \ell_i(\boldsymbol{\mu}_i, \mathbf{C}_i) + \log(\mathbb{P}(\mathbf{a}_i < \mathbf{X}_i \leq \mathbf{b}_i \mid \mathbf{C}_i, \boldsymbol{\eta}_i, \mathbf{Y}_i = \mathbf{y}_i)).$$

The trick here is to decompose the joint likelihood into a product of the marginal Lebesque density of $\mathbf{Y}_i$ and the conditional probability of $\mathbf{X}_i$ given $\mathbf{Y}_i = \mathbf{y}_i$.

We first check the data

⟨ *dp input checks* 93 ⟩ ≡

```
stopifnot(xor(missing(chol), missing(invchol)))
cJ <- nrow(obs)
dJ <- nrow(lower)
N <- ncol(obs)
stopifnot(N == ncol(lower))
stopifnot(N == ncol(upper))
if (all(mean == 0)) {
    cmean <- 0
    dmean <- 0
} else {
    if (!is.matrix(mean))
        mean <- matrix(mean, nrow = cJ + dJ, ncol = N)
    stopifnot(nrow(mean) == cJ + dJ)
    stopifnot(ncol(mean) == N)
    cmean <- mean[1:cJ,, drop = FALSE]
    dmean <- mean[-(1:cJ),, drop = FALSE]
}
```

◇

Fragment referenced in 94, 96.

We can use `marg_mvnorm` and `cond_mvnorm` to compute the marginal and the conditional normal distributions and the joint log-likelihood is simply the sum of the two corresponding log-likelihoods.

⟨ *ldpmvnorm* 94 ⟩ ≡

```
ldpmvnorm <- function(obs, lower, upper, mean = 0, chol, invchol,
                      logLik = TRUE, ...) {

    if (missing(obs) || is.null(obs))
        return(lpmvnorm(lower = lower, upper = upper, mean = mean,
                        chol = chol, invchol = invchol, logLik = logLik, ...))
    if (missing(lower) && missing(upper) || is.null(lower) && is.null(upper))
        return(ldmvnorm(obs = obs, mean = mean,
                        chol = chol, invchol = invchol, logLik = logLik))

    ⟨ dp input checks 93 ⟩

    if (!missing(invchol)) {
        J <- dim(invchol)[2L]
        stopifnot(cJ + dJ == J)

        md <- marg_mvnorm(invchol = invchol, which = 1:cJ)
        ret <- ldmvnorm(obs = obs, mean = cmean, invchol = md$invchol,
                        logLik = logLik)

        cd <- cond_mvnorm(invchol = invchol, which_given = 1:cJ,
                          given = obs - cmean, center = TRUE)
        ret <- ret + lpmvnorm(lower = lower, upper = upper, mean = dmean,
                              invchol = cd$invchol, center = cd$center,
                              logLik = logLik, ...)
        return(ret)
    }

    J <- dim(chol)[2L]
    stopifnot(cJ + dJ == J)

    md <- marg_mvnorm(chol = chol, which = 1:cJ)
    ret <- ldmvnorm(obs = obs, mean = cmean, chol = md$chol, logLik = logLik)

    cd <- cond_mvnorm(chol = chol, which_given = 1:cJ,
                      given = obs - cmean, center = TRUE)
    ret <- ret + lpmvnorm(lower = lower, upper = upper, mean = dmean,
                          chol = cd$chol, center = cd$center,
                          logLik = logLik, ...)
    return(ret)
}
```

◇

Fragment referenced in 2.

The score function requires a little extra work. We start with the case when `invchol` is given

⟨ *sldpmvnorm invchol* 95 ⟩ ≡

```
byrow_orig <- attr(invchol, "byrow")
invchol <- ltMatrices(invchol, byrow = TRUE)

J <- dim(invchol)[2L]
stopifnot(cJ + dJ == J)

md <- marg_mvnorm(invchol = invchol, which = 1:cJ)
cs <- sldmvnorm(obs = obs, mean = cmean, invchol = md$invchol)

obs_cmean <- obs - cmean
cd <- cond_mvnorm(invchol = invchol, which_given = 1:cJ,
                  given = obs_cmean, center = TRUE)
ds <- slpmvnorm(lower = lower, upper = upper, mean = dmean,
                center = cd$center, invchol = cd$invchol,
                logLik = logLik, ...)

tmp0 <- solve(cd$invchol, ds$mean, transpose = TRUE)
tmp <- - tmp0[rep(1:dJ, each = cJ),,drop = FALSE] *
         obs_cmean[rep(1:cJ, dJ),,drop = FALSE]

Jp <- nrow(unclass(invchol))
diag <- attr(invchol, "diag")
M <- as.array(ltMatrices(1:Jp, diag = diag, byrow = TRUE))[,,1]
ret <- matrix(0, nrow = Jp, ncol = ncol(obs))
M1 <- M[1:cJ, 1:cJ]
idx <- t(M1)[upper.tri(M1, diag = diag)]
ret[idx,] <- Lower_tri(cs$invchol, diag = diag)

idx <- c(t(M[-(1:cJ), 1:cJ]))
ret[idx,] <- tmp

M3 <- M[-(1:cJ), -(1:cJ)]
idx <- t(M3)[upper.tri(M3, diag = diag)]
ret[idx,] <- Lower_tri(ds$invchol, diag = diag)

ret <- ltMatrices(ret, diag = diag, byrow = TRUE)
if (!diag) diagonals(ret) <- 0
ret <- ltMatrices(ret, byrow = byrow_orig)

### post differentiate mean
aL <- as.array(invchol)[-(1:cJ), 1:cJ,,drop = FALSE]
lst <- tmp0[rep(1:dJ, cJ),,drop = FALSE]
if (dim(aL)[3] == 1)
      aL <- aL[,,rep(1, ncol(lst)), drop = FALSE]
dim <- dim(aL)
dobs <- -margin.table(aL * array(lst, dim = dim), 2:3)

ret <- c(list(invchol = ret, obs = cs$obs + dobs),
         ds[c("lower", "upper")])
ret$mean <- rbind(-ret$obs, ds$mean)
return(ret)
◇
```

Fragment referenced in 96.

For `chol`, we compute the above code for its inverse and post-differentiate using the vec-trick

⟨ *sldpmvnorm* 96 ⟩ ≡

```
sldpmvnorm <- function(obs, lower, upper, mean = 0, chol, invchol, logLik = TRUE, ...) {

    if (missing(obs) || is.null(obs))
        return(slpmvnorm(lower = lower, upper = upper, mean = mean,
                         chol = chol, invchol = invchol, logLik = logLik, ...))
    if (missing(lower) && missing(upper) || is.null(lower) && is.null(upper))
        return(sldmvnorm(obs = obs, mean = mean,
                         chol = chol, invchol = invchol, logLik = logLik))

    ⟨ dp input checks 93 ⟩

    if (!missing(invchol)) {
        ⟨ sldpmvnorm invchol 95 ⟩
    }

    invchol <- solve(chol)
    ret <- sldpmvnorm(obs = obs, lower = lower, upper = upper,
                      mean = mean, invchol = invchol, logLik = logLik, ...)
    ### this means: ret$chol <- - vectrick(invchol, ret$invchol, invchol)
    ret$chol <- - vectrick(invchol, ret$invchol)
    ret$invchol <- NULL
    return(ret)
}
```

◇

Fragment referenced in 2.

Let's assume we observed the first two dimensions exactly in our small example, and the remaining two dimensions are only known in intervals. The log-likelihood and score function for $\boldsymbol{\mu}$ and $\mathbf{C}$ are

```
> ll_cd <- 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)
+     -ldpmvnorm(obs = Y[1:2,], lower = lwr[-(1:2),],
+                upper = upr[-(1:2),], mean = m, chol = C,
+                w = W[-(1:2),,drop = FALSE], M = M)
+ }
> sc_cd <- 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 <- sldpmvnorm(obs = Y[1:2,], lower = lwr[-(1:2),],
+                       upper = upr[-(1:2),], mean = m, chol = C,
+                       w = W[-(1:2),,drop = FALSE], M = M)
+     return(-c(rowSums(ret$mean), rowSums(unclass(ret$chol))))
+ }
```

and the score function seems to be correct

```
> if (require("numDeriv", quietly = TRUE))
```

```
+     chk(grad(ll_cd, start, J = J), sc_cd(start, J = J),
+         check.attributes = FALSE, tol = 1e-6)
```

We can now jointly estimate all model parameters via

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

iter   10 value 655.707790
final  value 655.707779
converged

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

, , 1

           1           2          3         4
1 0.68303340  0.00000000 0.00000000 0.0000000
2 0.26504369  0.93466598 0.00000000 0.0000000
3 0.53508534 -0.05736364 1.11260547 0.0000000
4 0.06748574  0.95887388 0.07774847 0.9669178

> ## compare with true C
> 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

> ## estimated means
> op$par[1:J]

       1        2        3        4
0.968533 2.126882 2.944105 3.989790

> ## compare with true means
> mn

[1] 1 2 3 4
```

Chapter 6

Unstructured Gaussian Copula Estimation

With $\mathbf{Z} \sim \mathbb{N}_J(0, \mathbf{I}_J)$ and $\mathbf{Y} = \tilde{\mathbf{C}}\mathbf{Z} \sim \mathbb{N}_J(0, \tilde{\mathbf{C}}\tilde{\mathbf{C}}^\top)$ we want to estimate the off-diagonal elements of the lower triangular unit-diagonal matrix $\mathbf{C}$. We have $\tilde{\mathbf{C}}(\mathbf{C}) := \text{diag}(\mathbf{C}\mathbf{C}^\top)^{-1/2}\mathbf{C}$ such that $\mathbf{\Sigma} = \tilde{\mathbf{C}}\tilde{\mathbf{C}}^\top$ is a correlation matrix ($\text{diag}(\mathbf{\Sigma}) = \mathbf{I}_J$). Note that directly estimating $\tilde{\mathbf{C}}$ requires $J(J+1)/2$ parameters under constraints $\text{diag}(\mathbf{\Sigma}) = 1$ whereas only $J(J-1)/2$ parameters are necessary when estimating the lower triangular part of $\mathbf{C}$. The standardisation by $\text{diag}(\mathbf{C}\mathbf{C}^\top)^{-1/2}$ ensures that $\text{diag}(\mathbf{\Sigma}) \equiv 1$, that is, unconstained optimisation can be applied.

⟨ *standardize* 98 ⟩ ≡

```
standardize <- function(chol, invchol) {
    stopifnot(xor(missing(chol), missing(invchol)))
    if (!missing(invchol)) {
        stopifnot(!attr(invchol, "diag"))
        return(invcholD(invchol))
    }
    stopifnot(!attr(chol, "diag"))
    return(Dchol(chol))
}
```

◇

Fragment referenced in 2.

```
> C <- ltMatrices(runif(10))
> all.equal(as.array(chol2cov(standardize(chol = C))),
+           as.array(chol2cor(standardize(chol = C))))

[1] TRUE

> L <- solve(C)
> all.equal(as.array(invchol2cov(standardize(invchol = L))),
+           as.array(invchol2cor(standardize(invchol = L))))

[1] TRUE
```

The log-likelihood function is $\ell_i(\mathbf{C}_i)$ (we omit i in the following) and we assume the score

$$\frac{\partial \ell(\mathbf{C})}{\partial \mathbf{C}}$$

is already available. We want to compute the score

$$\frac{\partial \ell(\tilde{\mathbf{C}})}{\partial \mathbf{C}}$$

which gives

$$\frac{\partial \ell(\tilde{\mathbf{C}})}{\partial \mathbf{C}} = \underbrace{\frac{\partial \ell(\tilde{\mathbf{C}})}{\partial \tilde{\mathbf{C}}}}_{=:\mathbf{T}} \times \frac{\partial \tilde{\mathbf{C}}(\mathbf{C})}{\partial \mathbf{C}}$$

We further have

$$\frac{\partial \tilde{\mathbf{C}}(\mathbf{C})}{\partial \mathbf{C}} = (\mathbf{C}^\top \otimes \mathbf{I}_J)\frac{\partial \text{diag}(\mathbf{C}\mathbf{C}^\top)^{-1/2}}{\partial \mathbf{C}} + (\mathbf{I}_J \otimes \text{diag}(\mathbf{C}\mathbf{C}^\top)^{-1/2})$$

and thus

$$\frac{\partial \ell(\tilde{\mathbf{C}})}{\partial \mathbf{C}} = \text{vec}(\mathbf{I}_J\mathbf{T}\mathbf{C}^\top)^\top \frac{\partial \text{diag}(\mathbf{C}\mathbf{C}^\top)^{-1/2}}{\partial \mathbf{C}} + \text{vec}(\text{diag}(\mathbf{C}\mathbf{C}^\top)^{-1/2}\mathbf{T}\mathbf{I}_J)^\top$$

and with

$$\begin{aligned}\frac{\partial \text{diag}(\mathbf{C}\mathbf{C}^\top)^{-1/2}}{\partial \mathbf{C}} &= \frac{\partial \text{diag}(\mathbf{A})^{-1/2}}{\partial \mathbf{A}}\Big|_{\mathbf{A}=\mathbf{C}\mathbf{C}^\top} \frac{\partial \mathbf{C}\mathbf{C}^\top}{\partial \mathbf{C}} \\ &= -\frac{1}{2}\text{diag}(\text{vec}(\text{diag}(\mathbf{C}\mathbf{C}^\top)^{-3/2}))\left[(\mathbf{C}\otimes\mathbf{I}_J)\frac{\partial \mathbf{C}}{\partial \mathbf{C}} + (\mathbf{I}_J \otimes \mathbf{C})\frac{\partial \mathbf{C}^\top}{\partial \mathbf{C}}\right]\end{aligned}$$

we can write

$$\text{vec}(\mathbf{I}_J\mathbf{T}\mathbf{C}^\top)^\top(-\frac{1}{2})\text{diag}(\text{vec}(\text{diag}(\mathbf{C}\mathbf{C}^\top)^{-3/2})) = -\frac{1}{2} \times \text{vec}(\mathbf{I}_J\mathbf{T}\mathbf{C}^\top)^\top \times \text{vec}(\text{diag}(\mathbf{C}\mathbf{C}^\top)^{-3/2})^\top =: \mathbf{b}^\top$$

thus

$$\begin{aligned}\frac{\partial \ell(\tilde{\mathbf{C}})}{\partial \mathbf{C}} &= \mathbf{b}^\top\left[(\mathbf{C}\otimes\mathbf{I}_J)\frac{\partial \mathbf{C}}{\partial \mathbf{C}} + (\mathbf{I}_J\otimes\mathbf{C})\frac{\partial \mathbf{C}^\top}{\partial \mathbf{C}}\right] + \text{vec}(\text{diag}(\mathbf{C}\mathbf{C}^\top)^{-1/2}\mathbf{T}\mathbf{I}_J)^\top \\ &= \text{vec}(\mathbf{I}_J\mathbf{B}\mathbf{C})^\top + \text{vec}(\mathbf{C}^\top\mathbf{B}\mathbf{I}_J)^\top\frac{\partial \mathbf{C}^\top}{\partial \mathbf{C}} + \text{vec}(\text{diag}(\mathbf{C}\mathbf{C}^\top)^{-1/2}\mathbf{T}\mathbf{I}_J)^\top\end{aligned}$$

when $\mathbf{b} = \text{vec}(\mathbf{B})$. These scores are implemented in `destandardize` with `chol` $= \mathbf{C}$ and `score_schol` $= \mathbf{T}$. If the model was parameterised in $\mathbf{L} = \mathbf{C}^{-1}$, we have `invchol` $= \mathbf{L}$, however, we would still need to compute $\mathbf{T}$ (the score with respect to $\mathbf{C}$).

⟨ *destandardize* 100 ⟩ ≡

```
destandardize <- function(chol = solve(invchol), invchol, score_schol)
{
    stopifnot(inherits(chol, "ltMatrices"))
    J <- dim(chol)[2L]
    stopifnot(!attr(chol, "diag"))
    byrow_orig <- attr(chol, "byrow")
    chol <- ltMatrices(chol, byrow = FALSE)

    if (inherits(score_schol, "ltMatrices"))
        score_schol <- matrix(as.array(score_schol),
                              nrow = dim(score_schol)[2L]^2)
    stopifnot(is.matrix(score_schol))
    N <- ncol(score_schol)
    stopifnot(J^2 == nrow(score_schol))

    CCt <- Tcrossprod(chol, diag_only = TRUE)
    DC <- Dchol(chol, D = Dinv <- 1 / sqrt(CCt))
    SDC <- solve(DC)

    IDX <- t(M <- matrix(1:J^2, nrow = J, ncol = J))
    i <- cumsum(c(1, rep(J + 1, J - 1)))
    ID <- diagonals(as.integer(J), byrow = FALSE)
    if (dim(ID)[1L] != dim(chol)[1L])
        ID <- ID[rep(1, dim(chol)[1L]),]

    B <- vectrick(ID, score_schol, chol)
    B[i,] <- B[i,] * (-.5) * c(CCt)^(-3/2)
    B[-i,] <- 0

    Dtmp <- Dchol(ID, D = Dinv)

    ret <- vectrick(ID, B, chol, transpose = c(TRUE, FALSE)) +
           vectrick(chol, B, ID)[IDX,] +
           vectrick(Dtmp, score_schol, ID)

    if (!missing(invchol)) {
        ### this means: ret <- - vectrick(chol, ret, chol)
        ret <- - vectrick(chol, ret)
    }
    ret <- ltMatrices(ret[M[lower.tri(M)],,drop = FALSE],
                      diag = FALSE, byrow = FALSE)
    ret <- ltMatrices(ret, byrow = byrow_orig)
    diagonals(ret) <- 0
    return(ret)
}
```

◇

Fragment referenced in 2.

We can now set-up the log-likelihood and score functions for a Gaussian copula model. We start with the classical approach of generating the marginal observations $\mathbf{Y}$ from the ECDF with denominator $N + 1$ and subsequent use of the Lebesque density as likelihood.

```
> data("iris")
> J <- 4
```

```
> Z <- t(qnorm(do.call("cbind", lapply(iris[1:J], rank)) / (nrow(iris) + 1)))
> (CR <- cor(t(Z)))

             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length   1.00000000 -0.09887012    0.8695177   0.7819059
Sepal.Width   -0.09887012  1.00000000   -0.2709859  -0.2414218
Petal.Length   0.86951767 -0.27098589    1.0000000   0.8713759
Petal.Width    0.78190591 -0.24142185    0.8713759   1.0000000

> ll <- function(parm) {
+     C <- ltMatrices(parm)
+     Cs <- standardize(C)
+     -ldmvnorm(obs = Z, chol = Cs)
+ }
> sc <- function(parm) {
+     C <- ltMatrices(parm)
+     Cs <- standardize(C)
+     -rowSums(Lower_tri(destandardize(chol = C,
+         score_schol = sldmvnorm(obs = Z, chol = Cs)$chol)))
+ }
> start <- t(chol(CR))
> start <- start[lower.tri(start)]
> if (require("numDeriv", quietly = TRUE))
+     chk(grad(ll, start), sc(start), check.attributes = FALSE)
> op <- optim(start, fn = ll, gr = sc, method = "BFGS", hessian = TRUE)
> op$value

[1] 602.5055

> S_ML <- chol2cov(standardize(ltMatrices(op$par)))
```

This approach is of course a bit strange, because we estimate the marginal distributions by nonparametric maximum likelihood whereas the joint distribution is estimated by plain maximum likelihood. For the latter, we can define the likelihood by boxes given by intervals obtained from the marginale ECDFs and estimate the Copula parameters by maximisation of this nonparametric likelihood.

```
> lwr <- do.call("cbind", lapply(iris[1:J], rank, ties.method = "min")) - 1L
> upr <- do.call("cbind", lapply(iris[1:J], rank, ties.method = "max"))
> lwr <- t(qnorm(lwr / nrow(iris)))
> upr <- t(qnorm(upr / nrow(iris)))
> 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) {
+     C <- ltMatrices(parm)
+     Cs <- standardize(C)
+     -lpmvnorm(lower = lwr, upper = upr, chol = Cs, M = M, w = W)
+ }
> sc <- function(parm) {
```

```
+     C <- ltMatrices(parm)
+     Cs <- standardize(C)
+     -rowSums(Lower_tri(destandardize(chol = C,
+         score_schol = slpmvnorm(lower = lwr, upper = upr, chol = Cs,
+                                 M = M, w = W)$chol)))
+ }
> if (require("numDeriv", quietly = TRUE))
+     chk(grad(ll, start), sc(start), check.attributes = FALSE)
> op2 <- optim(start, fn = ll, gr = sc, method = "BFGS", hessian = TRUE)
> S_NPML <- chol2cov(standardize(ltMatrices(op2$par)))
```

For $N = 150$, the difference is (as expected) marginal:

```
> S_ML

, , 1

           1          2          3          4
1  1.0000000 -0.1139030  0.8768269  0.7962466
2 -0.1139030  1.0000000 -0.2856045 -0.2574850
3  0.8768269 -0.2856045  1.0000000  0.8816944
4  0.7962466 -0.2574850  0.8816944  1.0000000

> S_NPML

, , 1

            1           2          3          4
1  1.00000000 -0.09785513  0.8734599  0.7832830
2 -0.09785513  1.00000000 -0.2725997 -0.2482241
3  0.87345993 -0.27259973  1.0000000  0.8849489
4  0.78328300 -0.24822413  0.8849489  1.0000000
```

with relatively close standard errors

```
> sd_ML <- ltMatrices(sqrt(diag(solve(op$hessian))))
> diagonals(sd_ML) <- 0
> sd_NPML <- try(ltMatrices(sqrt(diag(solve(op2$hessian)))))
> if (!inherits(sd_NPML, "try-error")) {
+     diagonals(sd_NPML) <- 0
+     print(sd_ML)
+     print(sd_NPML)
+ }

, , 1

           1          2         3 4
1 0.00000000 0.00000000 0.0000000 0
2 0.08122393 0.00000000 0.0000000 0
3 0.13679345 0.08761945 0.0000000 0
4 0.12621115 0.10787495 0.1010173 0

, , 1

           1          2         3 4
1 0.00000000 0.00000000 0.0000000 0
```

```
2 0.07731078 0.00000000 0.0000000 0
3 0.13999691 0.08694828 0.0000000 0
4 0.13691328 0.11037843 0.1161017 0
```

Chapter 7

Package Infrastructure

⟨ *R Header* 104 ⟩ ≡

```
###    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, 59a.

⟨ *C Header* 105 ⟩ ≡

```
/*
    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 3, 59b.

Appendix

This document uses the following matrix derivatives

$$\begin{aligned}
\frac{\partial \mathbf{y}^\top \mathbf{A}^\top \mathbf{A}\mathbf{y}}{\partial \mathbf{A}} &= 2\mathbf{A}\mathbf{y}\mathbf{y}^\top \\
\frac{\partial \mathbf{A}^{-1}}{\partial \mathbf{A}} &= -(\mathbf{A}^{-\top} \otimes \mathbf{A}^{-1}) \\
\frac{\partial \mathbf{A}\mathbf{A}^\top}{\partial \mathbf{A}} &= (\mathbf{A} \otimes \mathbf{I}_J)\frac{\partial \mathbf{A}}{\partial \mathbf{A}} + (\mathbf{I}_J \otimes \mathbf{A})\frac{\partial \mathbf{A}^\top}{\partial \mathbf{A}} \\
&= (\mathbf{A} \otimes \mathbf{I}_J) + (\mathbf{I}_J \otimes \mathbf{A})\frac{\partial \mathbf{A}^\top}{\partial \mathbf{A}} \\
\frac{\partial \text{diag}(\mathbf{A})}{\partial \mathbf{A}} &= \text{diag}(\text{vec}(\mathbf{I}_J)) \\
\frac{\partial \mathbf{A}}{\partial \mathbf{A}} &= \text{diag}(I_{J^2}) \\
\frac{\mathbf{y}^\top \mathbf{A}\mathbf{y}}{\partial \mathbf{y}} &= \mathbf{y}^\top(\mathbf{A} + \mathbf{A}^\top)
\end{aligned}$$

and the "vec trick" $\text{vec}(\mathbf{X})^\top(\mathbf{B} \otimes \mathbf{A}^\top) = \text{vec}(\mathbf{A}\mathbf{X}\mathbf{B})^\top$.

Index

Files

Fragments

⟨ init center 66b ⟩ Referenced in 67, 79.
⟨ init dans 77c ⟩ Referenced in 79.
⟨ init logLik loop 61b ⟩ Referenced in 67, 73b.
⟨ init random seed, reset on exit 68a ⟩ Referenced in 69, 82.
⟨ init score loop 73b ⟩ Referenced in 79.
⟨ initialisation 61a ⟩ Referenced in 67, 79.
⟨ inner logLik loop 62d ⟩ Referenced in 67.
⟨ inner score loop 77a ⟩ Referenced in 79.
⟨ input checks 60a ⟩ Referenced in 58, 69, 82.
⟨ kronecker vec trick 42 ⟩ Referenced in 2.
⟨ L times D 44 ⟩ Referenced in 45.
⟨ lapack options 26 ⟩ Referenced in 27, 28.
⟨ ldmvnorm 52a ⟩ Referenced in 2.
⟨ ldmvnorm chol 54a ⟩ Referenced in 52a.
⟨ ldmvnorm invchol 54b ⟩ Referenced in 52a.
⟨ ldpmvnorm 94 ⟩ Referenced in 2.
⟨ logdet 31a ⟩ Referenced in 3.
⟨ logdet ltMatrices 31b ⟩ Referenced in 2.
⟨ lower scores 71c ⟩ Referenced in 72a.
⟨ lower triangular elements 15 ⟩ Referenced in 2.
⟨ lpmvnorm 69 ⟩ Referenced in 59a.
⟨ lpmvnormR 58 ⟩ Not referenced.
⟨ ltMatrices 6a ⟩ Referenced in 2.
⟨ ltMatrices dim 4 ⟩ Referenced in 6a.
⟨ ltMatrices input 5b ⟩ Referenced in 6a.
⟨ ltMatrices names 5a ⟩ Referenced in 6a.
⟨ marginal 48b ⟩ Referenced in 2.
⟨ mc input checks 48a ⟩ Referenced in 48b, 50b.
⟨ mean scores 71b ⟩ Referenced in 72a.
⟨ move on 63c ⟩ Referenced in 67, 79.
⟨ mult 22b ⟩ Referenced in 3.
⟨ mult ltMatrices 21a ⟩ Referenced in 2.
⟨ mult ltMatrices transpose 23 ⟩ Referenced in 21a.
⟨ mult syMatrices 25 ⟩ Referenced in 2.
⟨ mult transpose 24 ⟩ Referenced in 3.
⟨ names ltMatrices 7b ⟩ Referenced in 2.
⟨ new score means, lower and upper 75c ⟩ Referenced in 77a.
⟨ output 63b ⟩ Referenced in 67.
⟨ pnorm 64c ⟩ Referenced in 67, 79.
⟨ pnorm fast 64a ⟩ Referenced in 59b.
⟨ pnorm slow 64b ⟩ Referenced in 59b.
⟨ post differentiate chol score 80d ⟩ Referenced in 82.
⟨ post differentiate invchol score 81a ⟩ Referenced in 82.
⟨ post differentiate lower score 80b ⟩ Referenced in 82.
⟨ post differentiate mean score 80a ⟩ Referenced in 82.
⟨ post differentiate upper score 80c ⟩ Referenced in 82.
⟨ post process score 81b ⟩ Referenced in 82.
⟨ print ltMatrices 10 ⟩ Referenced in 2.
⟨ R Header 104 ⟩ Referenced in 2, 59a.
⟨ R lpmvnorm 67 ⟩ Referenced in 59b.
⟨ R slpmvnorm 79 ⟩ Referenced in 59b.
⟨ R slpmvnorm variables 66c ⟩ Referenced in 67, 79.
⟨ RC input 21b ⟩ Referenced in 22b, 24, 27, 28, 31a, 34, 40a.
⟨ reorder ltMatrices 11 ⟩ Referenced in 2.
⟨ score a, b 73a ⟩ Referenced in 73b, 79.
⟨ score c11 72b ⟩ Referenced in 73b, 79.
⟨ score output 77b ⟩ Referenced in 79.
⟨ score output object 72a ⟩ Referenced in 79.

⟨ score wrt new chol diagonal 75b ⟩ Referenced in 77a.
⟨ score wrt new chol off-diagonals 75a ⟩ Referenced in 77a.
⟨ setup return object 65c ⟩ Referenced in 67.
⟨ sldmvnorm 56 ⟩ Referenced in 2.
⟨ sldpmvnorm 96 ⟩ Referenced in 2.
⟨ sldpmvnorm invchol 95 ⟩ Referenced in 96.
⟨ slpmvnorm 82 ⟩ Referenced in 59a.
⟨ solve 27 ⟩ Referenced in 3.
⟨ solve C 28 ⟩ Referenced in 3.
⟨ solve ltMatrices 29 ⟩ Referenced in 2.
⟨ standardise 60b ⟩ Referenced in 69, 82.
⟨ standardize 98 ⟩ Referenced in 2.
⟨ subset ltMatrices 13 ⟩ Referenced in 2.
⟨ syMatrices 6b ⟩ Referenced in 2.
⟨ t(C) S t(A) 39 ⟩ Referenced in 40a.
⟨ tcrossprod 34 ⟩ Referenced in 3.
⟨ tcrossprod diagonal only 32b ⟩ Referenced in 34.
⟨ tcrossprod full 33a ⟩ Referenced in 34.
⟨ tcrossprod ltMatrices 35 ⟩ Referenced in 2.
⟨ univariate problem 66a ⟩ Referenced in 67.
⟨ update d, e 62b ⟩ Referenced in 62d, 77a.
⟨ update f 62c ⟩ Referenced in 62d, 77a.
⟨ update score for chol 76a ⟩ Referenced in 77a.
⟨ update score means, lower and upper 76b ⟩ Referenced in 77a.
⟨ update yp for chol 73c ⟩ Referenced in 77a.
⟨ update yp for means, lower and upper 74 ⟩ Referenced in 77a.
⟨ upper scores 71d ⟩ Referenced in 72a.
⟨ vec trick 40a ⟩ Referenced in 3.
⟨ W length 65a ⟩ Referenced in 67, 79.

Bibliography

Alan Genz. Numerical computation of multivariate normal probabilities. *Journal of Computational and Graphical Statistics*, 1(2), 1992. doi: 10.1080/10618600.1992.10477010. 1, 59

Alan Genz and Frank Bretz. Methods for the computation of multivariate t probabilities. *Journal of Computational and Graphical Statistics*, 11(4):950–971, 2002. doi: 10.1198/106186002394. 1, 58

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. 63