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

Torsten Hothorn

Version 1.3-1

[1]Please cite this document as: Torsten Hothorn (2024) Multivariate Normal Log-likelihoods in the **mvtnorm** Package. R package vignette version 1.3-1, URL DOI:10.32614/CRAN.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

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

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 $\mathbf{C}_i$ of the given covariance matrix is computed in FORTRAN first each time this function is called. 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. An attempt to provide useRs with a simple and (hopefully) bullet proof interface is documented in Chapter 7.

The development of this infrastructure was motivated by the necessity to evaluate probabilities (1.1) arising in the likelihood of multivariate conditional transformation models (Klein et al., 2022) for discrete or censored observations. Some forms of the likelihood for such nonparanormal models are discussed in Hothorn (2024).

Chapter 2

Lower Triangular Matrices

`"ltMatrices.R"` 2≡

⟨ *R Header* 128 ⟩
⟨ *ltMatrices* 6a ⟩
⟨ *syMatrices* 6b ⟩
⟨ *dim ltMatrices* 6c ⟩
⟨ *dimnames ltMatrices* 7a ⟩
⟨ *names ltMatrices* 7b ⟩
⟨ *is.ltMatrices* 7c ⟩
⟨ *as.ltMatrices* 112b ⟩
⟨ *print ltMatrices* 11 ⟩
⟨ *reorder ltMatrices* 12 ⟩
⟨ *subset ltMatrices* 14 ⟩
⟨ *lower triangular elements* 17 ⟩
⟨ *diagonals ltMatrices* 19 ⟩
⟨ *diagonal matrix* 22 ⟩
⟨ *mult ltMatrices* 23a ⟩
⟨ *mult syMatrices* 27 ⟩
⟨ *solve ltMatrices* 31 ⟩
⟨ *logdet ltMatrices* 33b ⟩
⟨ *tcrossprod ltMatrices* 37 ⟩
⟨ *crossprod ltMatrices* 38 ⟩
⟨ *chol syMatrices* 39 ⟩
⟨ *add diagonal elements* 20 ⟩
⟨ *assign diagonal elements* 21 ⟩
⟨ *kronecker vec trick* 44 ⟩
⟨ *convenience functions* 48 ⟩
⟨ *aperm* 51a, ... ⟩
⟨ *marginal* 52b ⟩
⟨ *conditional* 55 ⟩
⟨ *check obs* 57b ⟩
⟨ *colSumsdnorm ltMatrices* 58b ⟩
◇

"ltMatrices.c" 3≡

```
⟨ C Header 129 ⟩
#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 58a ⟩
⟨ solve 29 ⟩
⟨ solve C 30 ⟩
⟨ logdet 33a ⟩
⟨ tcrossprod 36 ⟩
⟨ mult 24b ⟩
⟨ mult transpose 26 ⟩
⟨ chol 40 ⟩
⟨ vec trick 42a ⟩
◇
```

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`, in an $J(J-1)/2 \times N$ matrix.

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

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

$$
= \begin{pmatrix} c_{1} & & & & 0 \\ c_{2} & c_{J+1} & & & \\ c_{3} & c_{J+2} & c_{2J} & & \\ \vdots & \vdots & & \ddots & \\ c_{J} & c_{2J-1} & \cdots & & c_{J(J+1)/2} \end{pmatrix} \text{ column-major, } \texttt{byrow = FALSE}
$$

$$
= \begin{pmatrix} & c_{1} & & & & & 0 \\ & c_{2} & & c_{3} & & & \\ & c_{4} & & c_{5} & c_{6} & & \\ & \vdots & & \vdots & & \ddots & \\ c_{J((J+1)/2-1)+1} & & c_{J((J+1)/2-1)+2} & \cdots & & c_{J(J+1)/2} \end{pmatrix} \text{ row-major, } \texttt{byrow = TRUE}
$$

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

⟨ *ltMatrices dim* 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)]
} # else {      ### add later
    # warning("ltMatrices objects should be properly named")
# }
```

◇

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 (is.ltMatrices(object)) {
    cls <- class(object)                ### keep inheriting classes
    ret <- .reorder(object, byrow = byrow)
    class(ret) <- class(object)
    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 multiple symmetric matrices

⟨ *syMatrices* 6b ⟩ ≡

```
as.syMatrices <- function(x) {
    if (is.syMatrices(x))
        return(x)
    x <- as.ltMatrices(x)       ### make sure "ltMatrices"
                                ### is first class
    class(x)[1L] <- "syMatrices"
    return(x)
}
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.

Finally, let's add two functions for checking the class and a function for coersing classes inheriting from `ltMatrices` to the latter, the same for `syMatrices`. Furthermode, `as.ltMatrices` coerces objects inheriting from `syMatrices` or `ltMatrices` to class `ltMatrices` (that is, `chol` or `invchol` is removed from the class list, unlike a call to the constructor `ltMatrices`). A `default` method is added in Chapter 7.

⟨ *is.ltMatrices* 7c ⟩ ≡

```
is.ltMatrices <- function(x) inherits(x, "ltMatrices")
is.syMatrices <- function(x) inherits(x, "syMatrices")
as.ltMatrices <- function(x) UseMethod("as.ltMatrices")
as.ltMatrices.syMatrices <- function(x) {
    cls <- class(x)
    class(x) <- cls[which(cls == "syMatrices"):length(cls)]
    class(x)[1L] <- "ltMatrices"
    return(x)
}
as.ltMatrices.ltMatrices <- function(x) {
    cls <- class(x)
    class(x) <- cls[which(cls == "ltMatrices"):length(cls)]
    return(x)
}
```

◇

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 1.00000 0.0000 0.00000 0.0000 0
B 0.51237 1.0000 0.00000 0.0000 0
C 0.05847 0.9095 1.00000 0.0000 0
D 0.39449 0.6612 0.23353 1.0000 0
E 0.51648 0.2980 0.07518 0.8182 1

, , C_2

       A      B      C      D E
A 1.0000 0.0000 0.0000 0.0000 0
B 0.8591 1.0000 0.0000 0.0000 0
C 0.3744 0.1023 1.0000 0.0000 0
D 0.1165 0.7957 0.8931 1.0000 0
E 0.1948 0.4730 0.2378 0.2146 1

, , C_3

       A      B      C      D E
A 1.0000 0.0000 0.0000 0.0000 0
B 0.4530 1.0000 0.0000 0.0000 0
C 0.9046 0.9270 1.0000 0.0000 0
D 0.4490 0.1326 0.4154 1.0000 0
E 0.9575 0.4917 0.7161 0.2938 1

, , C_4

          A      B        C      D E
A 1.0000000 0.0000 0.000000 0.0000 0
B 0.4877241 1.0000 0.000000 0.0000 0
C 0.0593046 0.7625 1.000000 0.0000 0
D 0.0005227 0.1996 0.470509 1.0000 0
E 0.4913541 0.2849 0.005961 0.8901 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 1.00000 0.5124 0.05847 0.3945 0.51648
B 0.51237 1.0000 0.90951 0.6612 0.29799
C 0.05847 0.9095 1.00000 0.2335 0.07518
D 0.39449 0.6612 0.23353 1.0000 0.81821
E 0.51648 0.2980 0.07518 0.8182 1.00000

, , C_2

       A      B      C      D      E
A 1.0000 0.8591 0.3744 0.1165 0.1948
B 0.8591 1.0000 0.1023 0.7957 0.4730
C 0.3744 0.1023 1.0000 0.8931 0.2378
D 0.1165 0.7957 0.8931 1.0000 0.2146
E 0.1948 0.4730 0.2378 0.2146 1.0000

, , C_3

       A      B      C      D      E
A 1.0000 0.4530 0.9046 0.4490 0.9575
B 0.4530 1.0000 0.9270 0.1326 0.4917
C 0.9046 0.9270 1.0000 0.4154 0.7161
D 0.4490 0.1326 0.4154 1.0000 0.2938
E 0.9575 0.4917 0.7161 0.2938 1.0000

, , C_4
```

```
          A      B        C         D        E
A 1.0000000 0.4877 0.059305 0.0005227 0.491354
B 0.4877241 1.0000 0.762527 0.1995700 0.284943
C 0.0593046 0.7625 1.000000 0.4705089 0.005961
D 0.0005227 0.1996 0.470509 1.0000000 0.890146
E 0.4913541 0.2849 0.005961 0.8901458 1.000000
```

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* 10 ⟩ ≡

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

◇

Fragment referenced in 11, 12, 13, 17, 19, 21, 23a, 27.

⟨ *print ltMatrices* 11 ⟩ ≡

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

    ⟨ extract slots 10 ⟩

    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* 12 ⟩ ≡

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

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

    ⟨ extract slots 10 ⟩

    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* 13 ⟩ ≡

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

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

    ⟨ extract slots 10 ⟩

    x <- unclass(x)

    if (!missing(j)) {

        if (is.character(j)) {
            stopifnot(all(j %in% dn[[2L]]))
            j <- match(j, dn[[2L]])
        }

        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 14.

⟨ *subset ltMatrices* 14 ⟩ ≡

```
⟨ .subset ltMatrices 13 ⟩
### if j is not ordered, result is not a lower triangular matrix
"[.ltMatrices" <- function(x, i, j, ..., drop = FALSE) {
    if (!missing(j)) {
        if (is.character(j)) {
            stopifnot(all(j %in% dimnames(x)[[2L]]))
            j <- match(j, dimnames(x)[[2L]])
        }
        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) {
    x <- as.syMatrices(x)
    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, names = nm)[i, j])
> b <- as.array(ltMatrices(xn, byrow = FALSE, names = nm))[j, j, i]
> chk(a, b)
> a <- as.array(ltMatrices(xn, byrow = TRUE, names = nm)[i, j])
> b <- as.array(ltMatrices(xn, byrow = TRUE, names = nm))[j, j, i]
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = FALSE,
+                          diag = TRUE, names = nm)[i, j])
> b <- as.array(ltMatrices(xd, byrow = FALSE,
+                          diag = TRUE, names = nm))[j, j, i]
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = TRUE, diag = TRUE,
+                          names = nm)[i, j])
> b <- as.array(ltMatrices(xd, byrow = TRUE, diag = TRUE,
+                          names = nm))[j, j, i]
> chk(a, b)
```

We start with both indices being positive integers

```
> i <- colnames(xn)[1:2]
> j <- 2:4
> ## subset
> a <- as.array(ltMatrices(xn, byrow = FALSE, names = nm)[i, j])
> b <- as.array(ltMatrices(xn, byrow = FALSE, names = nm))[j, j, i]
```

```
> chk(a, b)
> a <- as.array(ltMatrices(xn, byrow = TRUE, names = nm)[i, j])
> b <- as.array(ltMatrices(xn, byrow = TRUE, names = nm))[j, j, i]
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = FALSE,
+                          diag = TRUE, names = nm)[i, j])
> b <- as.array(ltMatrices(xd, byrow = FALSE,
+                          diag = TRUE, names = nm))[j, j, i]
> chk(a, b)
> a <- as.array(ltMatrices(xd, byrow = TRUE, diag = TRUE,
+                          names = nm)[i, j])
> b <- as.array(ltMatrices(xd, byrow = TRUE, diag = TRUE,
+                          names = nm))[j, j, i]
> chk(a, b)
```

proceed with characters

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

a different subset

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

```
+                           names = nm))[j, j, i]
> chk(a, b)
```

and characters again

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

and finally with with negative subsets

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

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

```
> ## subset
> j <- nm[sample(1:J)]
> ltM <- ltMatrices(xn, byrow = FALSE, names = nm)
> try(ltM[i, j])
> ltM <- as.syMatrices(ltM)
> a <- as.array(ltM[i, j])
> b <- as.array(ltM)[j, j, i]
> 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`. Note that ordering of the rows of this matrix depend on the `byrow` attribute of `x`, unless the `byrow` to this function is used to overwrite it explicitly

⟨ *lower triangular elements* 17 ⟩ ≡

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

    if (is.syMatrices(x))
        x <- as.ltMatrices(x)
    adiag <- diag
    x <- ltMatrices(x, byrow = byrow)

    ⟨ extract slots 10 ⟩

    if (diag == adiag)
        return(unclass(x)[,,drop = FALSE]) ### remove attributes

    if (!diag && adiag) {
        diagonals(x) <- 1
        return(unclass(x)[,,drop = FALSE]) ### remove attributes
    }

    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
```

```
> 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
```

```
> 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
```

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

```
       [,1]    [,2]
2.1 -0.7071 -0.7071
3.1  0.4364  0.4364
4.1 -0.4481 -0.4481
3.2 -0.9258 -0.9258
4.2  0.9189  0.9189
4.3 -0.9974 -0.9974
```

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* 19 ⟩ ≡

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

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

    ⟨ extract slots 10 ⟩

    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 which was set-up with constant $c_{jj} = 1$ diagonal elements.

⟨ *add diagonal elements* 20 ⟩ ≡

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

    stopifnot(is.ltMatrices(x))

    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* 21 ⟩ ≡

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

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

    ⟨ extract slots 10 ⟩

    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) {

    x <- as.ltMatrices(x)
    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* 22 ⟩ ≡

```
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 Section 2.13 for defining `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`$= \mathbf{C}_i$

⟨ *mult ltMatrices* 23a ⟩ ≡

```
### 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 10 ⟩

    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 25 ⟩

    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* 23b ⟩ ≡

```
/* 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 24b, 26, 29, 30, 33a, 36, 42a.

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* 24a ⟩ ≡

```
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 24b, 26, 29, 33a, 42a.

The C workhorse is now

⟨ *mult* 24b ⟩ ≡

```
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 23b ⟩
    ⟨ C length 24a ⟩

    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 $\mathtt{x} = \mathbf{C}_i$. This function assumes `x` in column-major order, so we coerce this object when necessary:

⟨ *mult ltMatrices transpose* 25 ⟩ ≡

```
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 23a.

before moving to C for the low-level computations:

⟨ *mult transpose* 26 ⟩ ≡

```
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 23b ⟩
    ⟨ C length 24a ⟩

    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 $\mathbf{A} = \mathbf{C} + \mathbf{C}^\top - \text{diag}(\mathbf{C})$ with lower triangular part $\mathbf{C}$ (including the diagonal) we can compute $\mathbf{Ay} = \mathbf{Cy} + \mathbf{C}^\top\mathbf{y} - \text{diag}(\mathbf{C})\mathbf{y}$ using `Mult` applied to the lower trianular part:

⟨ *mult syMatrices* 27 ⟩ ≡

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

    ⟨ extract slots 10 ⟩

    x <- as.ltMatrices(x)
    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* 28 ⟩ ≡

```
char di, lo = 'L';
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';
}
```

◇

Fragment referenced in 29, 30.

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

⟨ *solve* 29 ⟩ ≡

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

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

    ⟨ RC input 23b ⟩
    /* diagonal elements are always present */
    if (!Rdiag) len += iJ;
    ⟨ C length 24a ⟩
    ⟨ lapack options 28 ⟩

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

    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* 30 ⟩ ≡

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

    SEXP ans;
    double *dans;
    int i, info;

    ⟨ RC input 23b ⟩
    /* diagonal elements are always present */
    if (!Rdiag) len += iJ;
    ⟨ lapack options 28 ⟩

    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* 31 ⟩ ≡

```
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* 33a ⟩ ≡

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

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

    ⟨ RC input 23b ⟩
    Rboolean Rbyrow = asLogical(byrow);
    ⟨ C length 24a ⟩

    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* 33b ⟩ ≡

```
logdet <- function(x) {

    if (!is.ltMatrices(x))
        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

We want to ompute $\mathbf{C}_i\mathbf{C}_i^\top$ 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* 34a ⟩ ≡

```
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 34b, 35a.

⟨ *tcrossprod diagonal only* 34b ⟩ ≡

```
PROTECT(ans = allocMatrix(REALSXP, iJ, iN));
dans = REAL(ans);
for (n = 0; n < iN; n++) {
    ⟨ first element 34a ⟩
    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 36.

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

⟨ *tcrossprod full* 35a ⟩ ≡

```
nrow = iJ * (iJ + 1) / 2;
PROTECT(ans = allocMatrix(REALSXP, nrow, iN));
dans = REAL(ans);
for (n = 0; n < INTEGER(N)[0]; n++) {
    ⟨ first element 34a ⟩
    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 36.

and put both cases together

⟨ *IDX* 35b ⟩ ≡

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

Fragment referenced in 36, 42a.

⟨ *tcrossprod* 36 ⟩ ≡

```
⟨ IDX 35b ⟩

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 23b ⟩

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

    if (Rdiag_only) {
        ⟨ tcrossprod diagonal only 34b ⟩
    } else {
        ⟨ tcrossprod full 35a ⟩
    }
    UNPROTECT(1);
    return(ans);
}
```

◇

Fragment referenced in 3.

with R interface

⟨ *tcrossprod ltMatrices* 37 ⟩ ≡

```
### 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 (!is.ltMatrices(x)) {
        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* 38 ⟩ ≡

```
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* 39 ⟩ ≡

```
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* 40 ⟩ ≡

```
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)* 41 ⟩ ≡

```
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 42a.

⟨ *vec trick* 42a ⟩ ≡

```
⟨ IDX 35b ⟩

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 23b ⟩
    ⟨ C length 24a ⟩
    dS = REAL(S);
    dA = REAL(A);

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

    ⟨ t(C) S t(A) 41 ⟩

    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* 42b ⟩ ≡

```
C <- as.ltMatrices(C)
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]   ### works because of as.ltMatrices(c)
if (!is.double(C)) storage.mode(C) <- "double"
◇
```

Fragment referenced in 44.

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

⟨ *check S argument* 43a ⟩ ≡

```
SltM <- is.ltMatrices(S)
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 44.

`A` is an `ltMatrices` object

⟨ *check A argument* 43b ⟩ ≡

```
if (missing(A)) {
    A <- C
} else {
    A <- as.ltMatrices(A)
    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 44.

We put everything together in function `vectrick`

⟨ *kronecker vec trick* 44 ⟩ ≡

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

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

    ⟨ check C argument 42b ⟩
    ⟨ check S argument 43a ⟩
    ⟨ check A argument 43b ⟩

    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 $i = 1, \ldots, N$.

Before we start, let us put a label on lower triangular matrices, such that we can differentiate between $\mathbf{C}$ and $\mathbf{L}$.

⟨ *chol classes* 45 ⟩ ≡

```
is.chol <- function(x) inherits(x, "chol")
as.chol <- function(x) {
    stopifnot(is.ltMatrices(x))
    if (is.chol(x)) return(x)
    if (is.invchol(x))
        return(invchol2chol(x))
    class(x) <- c("chol", class(x))
    return(x)
}
is.invchol <- function(x) inherits(x, "invchol")
as.invchol <- function(x) {
    stopifnot(is.ltMatrices(x))
    if (is.invchol(x)) return(x)
    if (is.chol(x))
        return(chol2invchol(x))
    class(x) <- c("invchol", class(x))
    return(x)
}
```

◇

Fragment referenced in 48.

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

⟨ *D times C* 46 ⟩ ≡

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

    if (is.invchol(x)) stop("Dchol cannot work with invchol objects")

    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]]

    ### for some parameter configurations logdet(ret) would
    ### be -Inf; make sure this does't happen
    if (any(D < .Machine$double.eps))
        D[D < .Machine$double.eps] <- 2 * .Machine$double.eps

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

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

◇

Fragment referenced in 48.

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* 47 ⟩ ≡

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

    if (is.chol(x)) stop("invcholD cannot work with chol objects")

    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]]

    ### for some parameter configurations logdet(ret) would
    ### be -Inf; make sure this does't happen
    if (any(D < .Machine$double.eps))
        D[D < .Machine$double.eps] <- 2 * .Machine$double.eps

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

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

◇

Fragment referenced in 48.

and now the convenience functions are one-liners:

⟨ *convenience functions* 48 ⟩ ≡

⟨ *chol classes* 45 ⟩
⟨ *D times C* 46 ⟩
⟨ *L times D* 47 ⟩

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

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

### C -> L
chol2invchol <- function(x)
    as.invchol(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`, implementing the parameters ($\tilde{\mathbf{C}}_i$ or $\tilde{\mathbf{L}}_i$) for permuted versions of the random vectors $\mathbf{Y}_i$. Let π denote a permutation of $1, \ldots, J$ and Π the corresponding permutation matrix. Then, we have $\Pi\mathbf{Y}_i \sim \mathbb{N}_J(\mathbf{0}_J, \Pi\mathbf{C}_i\mathbf{C}_i^\top\Pi^\top)$. Unfortunately, $\Pi\mathbf{C}_i$ is no longer lower triangular, so we have to find the Cholesky decompositon $\tilde{\mathbf{C}}_i\tilde{\mathbf{C}}_i^\top$ of $\Pi\mathbf{C}_i\mathbf{C}_i^\top\Pi^\top$. Of course, $\tilde{\mathbf{L}}_i = \tilde{\mathbf{C}}_i^{-1}$.

The function `aperm`, with argument `perm` $= \pi$, now computes the Cholesky factor $\tilde{\mathbf{C}}_i$ of the permuted covariance matrix, or the inverse thereof (in case `x` is of class `invchol`). We start with some tests

⟨ *aperm checks* 50 ⟩ ≡

```
J <- dim(a)[2L]
if (missing(perm)) return(a)
if (is.character(perm))
    perm <- match(perm, dimnames(a)[[2L]])
stopifnot(all(perm %in% 1:J))

args <- list(...)
if (length(args) > 0L)
    warning("Additional arguments", names(args), "ignored")
◇
```

Fragment referenced in 51a.

and then implement the two methods

⟨ *aperm* 51a ⟩ ≡

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

    ⟨ aperm checks 50 ⟩

    return(as.chol(chol(chol2cov(a)[,perm])))
}
aperm.invchol <- function(a, perm, ...) {

    ⟨ aperm checks 50 ⟩

    return(chol2invchol(chol(invchol2cov(a)[,perm])))
}
◇
```

Fragment defined by 51ab.
Fragment referenced in 2.

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

We finally add a method for class `ltMatrices`, for which we actually cannot provide a reasonable result, and for symmetric matrices, where we simply fall-back on subsetting

⟨ *aperm* 51b ⟩ ≡

```
aperm.ltMatrices <- function(a, perm, ...)
    stop("Cannot permute objects of class ltMatrices,
          consider calling as.chol() or as.invchol() first")

aperm.syMatrices <- function(a, perm, ...)
    return(a[,perm])
◇
```

Fragment defined by 51ab.
Fragment referenced in 2.

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* 52a ⟩ ≡

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

stopifnot(is.ltMatrices(x))

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

if (missing(which)) return(x)

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

◇

Fragment referenced in 52b, 55.

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* 52b ⟩ ≡

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

    ⟨ mc input checks 52a ⟩

    if (which[1] == 1L && (length(which) == 1L ||
                           all(diff(which) == 1L))) {
        ### which is 1:j
        tmp <- x[,which]
    } else {
        if (missing(chol)) x <- invchol2chol(x)
        ### note: aperm would work but computes
        ### Cholesky of J^2, here only length(which)^2
        ### is needed
        tmp <- base::chol(chol2cov(x)[,which])
        if (missing(chol)) tmp <- chol2invchol(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* 53 ⟩ ≡

```
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 55.

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* 54 ⟩ ≡

```
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))
    ### if ncol(given) is not N = dim(L)[1L] > 1, then
    ### solve() below won't work and we loop over
    ### columns of centerm
    if (dim(L)[1L] > 1 && ncol(given) != N) {
        centerm <- lapply(1:ncol(centerm), function(j)
            matrix(centerm[,j], nrow = J, ncol = N)[-which,,drop = FALSE]
        )
    } else {
        centerm <- centerm[-which,,drop = FALSE]
    }
    L <- L[,-which]
    ct <- centerm
    if (!is.matrix(ct)) ct <- do.call("rbind", ct)
    if (is.matrix(centerm)) {
        m <- -solve(L, centerm)
    } else {
        m <- do.call("rbind", lapply(centerm, function(cm) -solve(L, cm)))
    }
    if (missing(invchol)) {
        if (center)
            return(list(center = ct, chol = solve(L)))
        return(list(mean = m, chol = solve(L)))
    }
    if (center)
        return(list(center = ct, invchol = L))
    return(list(mean = m, invchol = L))
}
```

◇

Fragment referenced in 55.

Note that we could have avoided the general case altogether by first computing a Cholesky decomposition of the permuted covariance matrix (such that the conditioning variables come first). The code above only decomposes the marginal (and thus lower-dimensional) covariance. However, we didn't implement the `center = TRUE` case, so we can fall back on the permuted version if this option is requested. Putting everything together gives

⟨ *conditional* 55 ⟩ ≡

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

    which <- which_given
    ⟨ mc input checks 52a ⟩

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

    ⟨ cond simple 54 ⟩

    ### general with center = TRUE => permute first and go simple
    if (center) {
        perm <- c(which, (1:J)[!(1:J) %in% which])
        if (!missing(chol))
        return(cond_mvnorm(chol = aperm(as.chol(chol), perm = perm),
                           which_given = 1:length(which), given = given,
                           center = center))
        return(cond_mvnorm(invchol = aperm(as.invchol(invchol), perm = perm),
                           which_given = 1:length(which), given = given,
                           center = center))
    }

    ⟨ cond general 53 ⟩

    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* 57a ⟩ ≡

```
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 59a ⟩
    } else {
        ⟨ ldmvnorm invchol 59b ⟩
    }

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

◇

Fragment referenced in 64a.

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* 57b ⟩ ≡

```
.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* 58a ⟩ ≡

```
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 * M_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* 58b ⟩ ≡

```
.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* 59a ⟩ ≡

```
if (missing(chol))
    stop("either chol or invchol must be given")
## chol is given
if (!is.ltMatrices(chol))       ### NOTE: replace with is.chol
    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 57a.

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* 59b ⟩ ≡

```
## invchol is given
if (!is.ltMatrices(invchol))    ### NOTE: replace with is.invchol
    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 57a.

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* 61 ⟩ ≡

```
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 <- ret[M[lower.tri(M, diag = attr(invchol, "diag"))],,drop = FALSE]
        if (!is.null(dimnames(invchol)[[1L]]))
            colnames(ret) <- dimnames(invchol)[[1]]
        ret <- ltMatrices(ret,
                          diag = attr(invchol, "diag"), byrow = FALSE,
                          names = dimnames(invchol)[[2L]])
        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$chol <- as.chol(- vectrick(invchol, ret$invchol))
    ret$invchol <- NULL
    return(ret)
}
```

◇

Fragment referenced in 64a.

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 $\Sigma_i^{-1} = \mathbf{L}_i\mathbf{L}_i^\top$. 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* 63 ⟩ ≡

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

    ⟨ input checks 65 ⟩

    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] 0.2369 0.2337 0.2842 0.3915 0.4662 0.0000 0.5901 0.4619 0.4873 0.0000
```

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" 64a≡

⟨ *R Header* 128 ⟩
⟨ *lpmvnorm* 74 ⟩
⟨ *slpmvnorm* 86 ⟩
⟨ *ldmvnorm* 57a ⟩
⟨ *sldmvnorm* 61 ⟩
⟨ *ldpmvnorm* 98 ⟩
⟨ *sldpmvnorm* 100 ⟩
⟨ *deperma* 104b ⟩
⟨ *standardize* 106 ⟩
⟨ *destandardize* 108 ⟩
◇

"lpmvnorm.c" 64b≡

⟨ *C Header* 129 ⟩
```
#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* 69b ⟩
⟨ *pnorm slow* 69c ⟩
⟨ *R lpmvnorm* 72 ⟩
⟨ *R slpmvnorm* 83 ⟩
◇

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* 65 ⟩ ≡

```
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(is.ltMatrices(chol))          ### NOTE: replace with is.chol
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)) {
    if (ncol(mean) == 1L)
        mean <- mean[,rep(1, N),drop = FALSE]
    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 63, 74, 86.

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

⟨ *standardise* 66a ⟩ ≡

```
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 74, 86.

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* 66b ⟩ ≡

```
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 72, 83.

4. Repeat

⟨ *init logLik loop* 66c ⟩ ≡

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

◇

Fragment referenced in 72, 77c.

(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 67a ⟩ ≡

```
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 68b, 81b.

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

⟨ *compute* x 67b ⟩ ≡

```
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 68b, 81b.

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

⟨ *update* d, e 67c ⟩ ≡

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

◇

Fragment referenced in 68b, 81b.

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

⟨ *update f* 68a ⟩ ≡

```
start += j;
f *= emd;
```
◇

Fragment referenced in 68b, 81b.

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

⟨ *inner logLik loop* 68b ⟩ ≡

```
for (j = 1; j < iJ; j++) {
    ⟨ compute y 67a ⟩
    ⟨ compute x 67b ⟩
    ⟨ update d, e 67c ⟩
    ⟨ update f 68a ⟩
}
```
◇

Fragment referenced in 72.

(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* 68c ⟩ ≡

```
intsum += f;
```
◇

Fragment referenced in 72.

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* 68d ⟩ ≡

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

Fragment referenced in 72.

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

⟨ *move on* 69a ⟩ ≡

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

Fragment referenced in 72, 83.

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

⟨ *pnorm fast* 69b ⟩ ≡

```
/* see https://doi.org/10.2139/ssrn.2842681  */
const double g2 =  -0.0150234471495426236132;
const double g4 = 0.00066609851170101874728 9;
const double g6 = 5.07937324518981103694e-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 64b.

⟨ *pnorm slow* 69c ⟩ ≡

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

Fragment referenced in 64b.

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

⟨ *pnorm* 70a ⟩ ≡

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

Fragment referenced in 72, 83.

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* 70b ⟩ ≡

```
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 72, 83.

⟨ *dimensions* 70c ⟩ ≡

```
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 72, 83.

⟨ *setup return object* 71a ⟩ ≡

```
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 72.

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

⟨ *univariate problem* 71b ⟩ ≡

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

◇

Fragment referenced in 72.

⟨ *init center* 71c ⟩ ≡

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

◇

Fragment referenced in 72, 83.

We put the code together in a dedicated C function

⟨ *R slpmvnorm variables* 71d ⟩ ≡

```
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 72, 83.

⟨ *R lpmvnorm* 72 ⟩ ≡

```
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 71d ⟩
    double l0, lM, x0, intsum;
    int p, len;

    Rboolean RlogLik = asLogical(logLik);

    ⟨ pnorm 70a ⟩
    ⟨ dimensions 70c ⟩
    ⟨ W length 70b ⟩
    ⟨ init center 71c ⟩

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

    ⟨ setup return object 71a ⟩

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

    ⟨ univariate problem 71b ⟩

    if (W == R_NilValue)
        GetRNGstate();

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

        x0 = 0;
        ⟨ initialisation 66b ⟩

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

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

            ⟨ init logLik loop 66c ⟩
            ⟨ inner logLik loop 68b ⟩
            ⟨ increment 68c ⟩

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

        ⟨ output 68d ⟩
        ⟨ move on 69a ⟩
    }

    if (W == R_NilValue)
        PutRNGstate();

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

◇

Fragment referenced in 64b.

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* 73a ⟩ ≡

```
### 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 74, 86.

⟨ *check and / or set integration weights* 73b ⟩ ≡

```
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 74, 86.

Sometimes we want to evaluate the log-likelihood based on $\mathbf{L} = \mathbf{C}^{-1}$, the inverse Cholesky factor of 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* 73c ⟩ ≡

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

◇

Fragment referenced in 74, 86.

⟨ *lpmvnorm* 74 ⟩ ≡

```
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 73a ⟩
    ⟨ Cholesky of precision 73c ⟩
    ⟨ input checks 65 ⟩
    ⟨ standardise 66a ⟩
    ⟨ check and / or set integration weights 73b ⟩

    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 64a.

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

```
> phat

 [1] 0.2369 0.2337 0.2842 0.3915 0.4662 0.0000 0.5901 0.4619 0.4873 0.0000

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

 [1] 2.367e-01 2.341e-01 2.835e-01 3.939e-01 4.658e-01 8.882e-21 5.911e-01
 [8] 4.598e-01 4.879e-01 8.882e-21

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

 [1] 2.377e-01 2.372e-01 2.832e-01 3.875e-01 4.660e-01 8.882e-21 5.895e-01
 [8] 4.624e-01 4.871e-01 8.882e-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 user's 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, 2002, 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,] 0.2369 2.369e-01 2.345e-01 2.369e-01 2.360e-01
 [2,] 0.2342 2.340e-01 2.319e-01 2.340e-01 2.347e-01
 [3,] 0.2841 2.841e-01 2.851e-01 2.841e-01 2.870e-01
 [4,] 0.3918 3.921e-01 3.932e-01 3.921e-01 3.904e-01
 [5,] 0.4671 4.668e-01 4.679e-01 4.668e-01 4.691e-01
 [6,] 0.0000 2.220e-20 2.220e-20 2.220e-20 2.220e-20
 [7,] 0.5902 5.902e-01 5.908e-01 5.902e-01 5.929e-01
 [8,] 0.4613 4.619e-01 4.612e-01 4.619e-01 4.630e-01
 [9,] 0.4872 4.870e-01 4.863e-01 4.870e-01 4.821e-01
[10,] 0.0000 2.220e-20 2.220e-20 2.220e-20 2.220e-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,] 1.0000 1.0000 1.0000
 [2,] 0.6109 0.6109 0.6109
 [3,] 0.9076 0.9076 0.9076
 [4,] 0.8980 0.8980 0.8980
 [5,] 0.9589 0.9589 0.9589
 [6,] 0.7863 0.7863 0.7863
 [7,] 0.9983 0.9983 0.9983
 [8,] 0.8745 0.8745 0.8745
 [9,] 0.9386 0.9386 0.9386
[10,] 0.9120 0.9120 0.9120
```

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* 76a ⟩ ≡

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

◇

Fragment referenced in 76e.

and the derivates with respect to the mean

⟨ *mean scores* 76b ⟩ ≡

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

◇

Fragment referenced in 76e.

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

⟨ *lower scores* 76c ⟩ ≡

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

◇

Fragment referenced in 76e.

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

⟨ *upper scores* 76d ⟩ ≡

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

◇

Fragment referenced in 76e.

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* 76e ⟩ ≡

```
int Jp = iJ * (iJ + 1) / 2;
⟨ chol scores 76a ⟩
⟨ mean scores 76b ⟩
⟨ lower scores 76c ⟩
⟨ upper scores 76d ⟩
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 83.

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 $c_{j\jmath}^{(i)}/c_{jj}^{(i)}$ 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* 77a ⟩ ≡

   ```
   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 77c, 83.

   ⟨ *score a, b* 77b ⟩ ≡

   ```
   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 77c, 83.

4. Repeat

   ⟨ *init score loop* 77c ⟩ ≡

   ⟨ *init logLik loop* 66c ⟩
   ⟨ *score c11* 77a ⟩
   ⟨ *score a, b* 77b ⟩
   ◇

   Fragment referenced in 83.

   (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* 78a ⟩ ≡

```
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 81b.

⟨ *update yp for means, lower and upper* 78b ⟩ ≡

```
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 81b.

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

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

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

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

⟨ *score wrt new chol off-diagonals* 79a ⟩ ≡

```
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 81b.

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

⟨ *score wrt new chol diagonal* 79b ⟩ ≡

```
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 81b.

⟨ *new score means, lower and upper* 80a ⟩ ≡

```
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 81b.

We next update scores for parameters introduced for smaller j

⟨ *update score for chol* 80b ⟩ ≡

```
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 81b.

⟨ *update score means, lower and upper* 81a ⟩ ≡

```
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 81b.

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

⟨ *inner score loop* 81b ⟩ ≡

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

    ⟨ compute y 67a ⟩
    ⟨ compute x 67b ⟩
    ⟨ update d, e 67c ⟩
    ⟨ update yp for chol 78a ⟩
    ⟨ update yp for means, lower and upper 78b ⟩
    ⟨ score wrt new chol off-diagonals 79a ⟩
    ⟨ score wrt new chol diagonal 79b ⟩
    ⟨ new score means, lower and upper 80a ⟩
    ⟨ update score for chol 80b ⟩
    ⟨ update score means, lower and upper 81a ⟩
    ⟨ update f 68a ⟩

}
```

◇

Fragment referenced in 83.

(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* 82a ⟩ ≡

```
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 83.

⟨ *init dans* 82b ⟩ ≡

```
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 83.

We put everything together in C

⟨ *R slpmvnorm* 83 ⟩ ≡

```
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 71d ⟩
    double intsum;
    int p, idx;
    ⟨ dimensions 70c ⟩
    ⟨ pnorm 70a ⟩
    ⟨ W length 70b ⟩
    ⟨ init center 71c ⟩
    int start, j, k;
    double tmp, e, d, f, emd, x, x0, y[(iJ > 1 ? iJ - 1 : 1)];

    ⟨ score output object 76e ⟩

    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 66b ⟩
        ⟨ score c11 77a ⟩
        ⟨ score a, b 77b ⟩
        ⟨ init dans 82b ⟩

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

        for (int m = 0; m < iM; m++) {
            ⟨ init score loop 77c ⟩
            ⟨ inner score loop 81b ⟩
            ⟨ score output 82a ⟩
            if (W != R_NilValue)
                dW += iJ - 1;
        }

        ⟨ move on 69a ⟩
        dans += Jp + 1 + 3 * iJ;
    }

    if (W == R_NilValue)
        PutRNGstate();

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

◇

Fragment referenced in 64b.

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* 84a ⟩ ≡

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

◇

Fragment referenced in 86.

⟨ *post differentiate lower score* 84b ⟩ ≡

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

◇

Fragment referenced in 86.

⟨ *post differentiate upper score* 84c ⟩ ≡

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

◇

Fragment referenced in 86.

⟨ *post differentiate chol score* 84d ⟩ ≡

```
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 86.

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* 85a ⟩ ≡

```
if (!missing(invchol)) {
    ret <- ltMatrices(ret, diag = TRUE, byrow = TRUE,
                      names = dimnames(chol)[[2L]])
    ### this means vectrick(chol, ret, chol)
    ret <- - unclass(vectrick(chol, ret))
}
```

◇

Fragment referenced in 86.

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* 85b ⟩ ≡

```
if (!attr(chol, "diag"))
    ### remove scores for constant diagonal elements
    ret[idx,] <- 0
ret <- ltMatrices(ret, diag = TRUE, byrow = TRUE,
                  names = dimnames(chol)[[2L]])
```

◇

Fragment referenced in 86.

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

⟨ *slpmvnorm* 86 ⟩ ≡

```
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 73a ⟩
    ⟨ Cholesky of precision 73c ⟩
    ⟨ input checks 65 ⟩
    ⟨ standardise 66a ⟩
    ⟨ check and / or set integration weights 73b ⟩

    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 84a ⟩
    ⟨ post differentiate lower score 84b ⟩
    ⟨ post differentiate upper score 84c ⟩

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

    ⟨ post differentiate chol score 84d ⟩
    ⟨ post differentiate invchol score 85a ⟩
    ⟨ post process score 85b ⟩

    ret <- ltMatrices(ret, byrow = byrow_orig)

    rownames(smean) <- rownames(slower) <-
        rownames(supper) <- dimnames(chol)[[2L]]

    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 64a.

Let's look at an example, where we use `numDeriv::grad` to check the results (this functionality from package **numDeriv** was absolutely instrumental for this project)

```
> 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] -0.01166 -0.08617 -0.01240 -0.03105

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

[1] -0.01166 -0.08617 -0.01240 -0.03105

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

$logLik
[1] -0.01166 -0.08617 -0.01240 -0.03105

$mean
     [,1]  [,2]    [,3]    [,4]
1 0.02222 0.214 0.02642 0.08861
```

```
$lower
      [,1]    [,2]     [,3]     [,4]
1 -0.03222 -0.2145 -0.03536 -0.09096

$upper
      [,1]      [,2]     [,3]     [,4]
1 0.009995 0.0004369 0.008944 0.002351

$chol
       [,1]    [,2]    [,3]    [,4]
1.1 -0.1041 -0.2994 -0.1076 -0.1787
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] -0.1041 -0.2994 -0.1076 -0.1787
```

Chapter 4

Maximum-likelihood Example

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

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

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

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
0.9685 2.1269 2.9634 3.9826

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

```
        1       2       3       4
1 0.46656 0.18104 0.34222 0.01609
2 0.18104 0.94385 0.08992 0.84310
3 0.34222 0.08992 1.36055 0.08104
4 0.01609 0.84310 0.08104 1.63302
```

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] 517.9

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

, , 1

        1        2      3      4
1 0.68306  0.00000 0.0000 0.0000
2 0.26505  0.93465 0.0000 0.0000
3 0.50102 -0.04587 1.0523 0.0000
4 0.02356  0.89535 0.1048 0.9054
```

```
> ll(op$par)

[1] 517.9

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

        1        2      3      4
1 0.68305  0.00000 0.0000 0.0000
2 0.26505  0.93467 0.0000 0.0000
3 0.50102 -0.04587 1.0523 0.0000
4 0.02356  0.89535 0.1048 0.9054

> ll(cML)

[1] 517.9

> ## true C matrix
> lt

, , 1

       1       2      3     4
1 0.7071  0.0000 0.0000 0.000
2 0.2500  0.9682 0.0000 0.000
3 0.6124 -0.1581 1.0488 0.000
4 0.0000  1.0954 0.1651 0.879
```

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 comparison 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) {
```

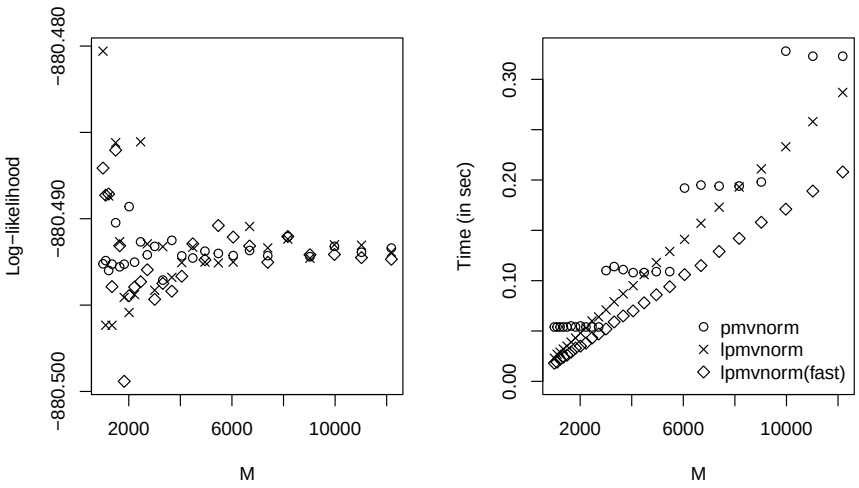

Figure 4.1: Evaluated log-likelihoods (left) and timings (right).

```
+     W <- NULL
+     if (require("qrng", quietly = TRUE))
+         W <- t(ghalton(m, d = J - 1))
+     st <- system.time(ret <- lpmvnorm(lwr, upr, mean = mn,
+                                       chol = lt, w = W, M = m))
+     return(c(st["user.self"], ll = ret))
+ })
> lHf <- sapply(M, function(m) {
+     W <- NULL
+     if (require("qrng", quietly = TRUE))
+         W <- t(ghalton(m, d = J - 1))
+     st <- system.time(ret <- lpmvnorm(lwr, upr, mean = mn, chol = lt,
+                                       w = W, M = m, fast = TRUE))
+     return(c(st["user.self"], ll = ret))
+ })
```

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

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

```
> M <- 500
> if (require("qrng", quietly = TRUE)) {
+     ### quasi-Monte-Carlo
+     W <- t(ghalton(M, d = J - 1))
+ } else {
+     ### Monte-Carlo
+     W <- matrix(runif(M * (J - 1)), nrow = J - 1)
+ }
> ll <- function(parm, J) {
+      m <- parm[1:J]             ### mean parameters
```

```
+      parm <- parm[-(1:J)]       ### chol parameters
+      C <- matrix(c(parm), ncol = 1L)
+      C <- ltMatrices(C, diag = TRUE, byrow = BYROW)
+      -lpmvnorm(lower = lwr, upper = upr, mean = m, chol = C,
+                w = W, M = M, logLik = TRUE)
+ }
```

We can check the correctness of our log-likelihood function

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

[1] 880.5

> ### 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] -880.5

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

[1] -880.5

> 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] 875.4
```

```
> op <- optim(start, fn = ll, gr = sc, J = J, method = "L-BFGS-B",
+             lower = llim, control = list(trace = FALSE))
> op$value ## compare with

[1] 874.2

> ll(prm, J = J)

[1] 880.5
```

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 0.67050  0.00000 0.00000 0.0000
2 0.26764  1.02232 0.00000 0.0000
3 0.54268 -0.05007 1.11348 0.0000
4 0.05223  0.98430 0.08473 0.9614

> lt

, , 1

       1       2      3     4
1 0.7071  0.0000 0.0000 0.000
2 0.2500  0.9682 0.0000 0.000
3 0.6124 -0.1581 1.0488 0.000
4 0.0000  1.0954 0.1651 0.879
```

and the estimated means

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

    1     2     3     4
0.967 2.128 2.945 3.989

> 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), 4)  ### true Sigma

, , 1

       1      2     3     4
1 0.5000 0.1768 0.433 0.000
2 0.1768 1.0000 0.000 1.061
3 0.4330 0.0000 1.500 0.000
4 0.0000 1.0607 0.000 2.000
```

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

, , 1

        1       2       3       4
1 0.44956 0.17945 0.36386 0.03502
2 0.17945 1.11677 0.09406 1.02025
3 0.36386 0.09406 1.53684 0.07341
4 0.03502 1.02025 0.07341 1.90298

> Shat                      ### "exact" obs

        1       2       3       4
1 0.46656 0.18104 0.34222 0.01609
2 0.18104 0.94385 0.08992 0.84310
3 0.34222 0.08992 1.36055 0.08104
4 0.01609 0.84310 0.08104 1.63302
```

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]  0.2602  0.2270 -0.1299
```

Alternatively, we can compute these regressions from a permuted Cholesky factor (this goes into the "simple" conditional distribution in Section 2.13)

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

[1]  0.2602  0.2270 -0.1299
```

or, as a third option, from the last row of the precision matrix of the permuted Cholesky factor

```
> x <- as.array(chol2pre(aperm(as.chol(C), perm = c(2:J, 1))))[J,,1]
> c(-x[-J] / x[J])

      2       3       4
 0.2602  0.2270 -0.1299
```

In higher dimensions, the first option is to be preferred, because it only involves computing the Cholesky decomposition of a $(J-1) \times (J-1)$ matrix, whereas the latter two options are based on a decomposition of the full $J \times J$ covariance matrix.

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
 0.3169  0.2405 -0.1657
```

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$par, fn = ll, gr = sc, J = J, method = "L-BFGS-B",
+            lower = llim, hessian = TRUE,
+            control = list(trace = FALSE))$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      12      13      14
0.05130 0.07990 0.12446 0.16090 0.07609 0.11567 0.14020 0.09622 0.10415 0.08279
```

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

```
 [1] 0.06826 0.10816 0.12670 0.14074 0.05498 0.10839 0.12442 0.14312 0.08813
[10] 0.11638 0.13340 0.09587 0.10451 0.08154
```

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] 0.08793 0.04869 0.07752
```

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

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

```
     Y2      Y3      Y4
0.08230 0.05039 0.06246
```

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

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

where $\mathbf{C}_{\mathbf{Y},i}$ is the upper part of $\mathbf{C}_i$ corresponding to the marginal distribution of $\mathbf{Y}_i$. The conditional probability of $\mathbf{X}$ given $\mathbf{Y}$ depends on all parameters, as explained in Section 2.13. 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* 97 ⟩ ≡

```
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) || NCOL(mean) == 1L)
        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 98, 100.

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* 98 ⟩ ≡

```
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 97 ⟩

    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 64a.

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

⟨ *sldpmvnorm invchol* 99 ⟩ ≡

```
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 100.

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

⟨ *sldpmvnorm* 100 ⟩ ≡

```
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 97 ⟩

    if (!missing(invchol)) {
        ⟨ sldpmvnorm invchol 99 ⟩
    }

    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$chol <- as.chol(- vectrick(invchol, ret$invchol))
    ret$invchol <- NULL
    return(ret)
}
```

◇

Fragment referenced in 64a.

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

```
> ic <- 1:2         ### position of continuous variables
> 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[ic,], lower = lwr[-ic,],
+                 upper = upr[-ic,], mean = m, chol = C,
+                 w = W[-ic,,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[ic,], lower = lwr[-ic,],
+                       upper = upr[-ic,], mean = m, chol = C,
+                       w = W[-ic,,drop = FALSE], M = M)
+     return(-c(rowSums(ret$mean),
+               rowSums(Lower_tri(ret$chol, diag = TRUE))))
+ }
```

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

, , 1

        1        2       3      4
1 0.68303  0.00000 0.00000 0.0000
2 0.26504  0.93467 0.00000 0.0000
3 0.53509 -0.05736 1.11261 0.0000
4 0.06749  0.95887 0.07775 0.9669

> ## compare with true C
> lt

, , 1

       1       2      3     4
1 0.7071  0.0000 0.0000 0.000
2 0.2500  0.9682 0.0000 0.000
3 0.6124 -0.1581 1.0488 0.000
4 0.0000  1.0954 0.1651 0.879

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

     1      2      3      4
0.9685 2.1269 2.9441 3.9898

> ## compare with true means
> mn

[1] 1 2 3 4
```

The one restriction in both `ldpmvnorm` and `sldpmvnorm` is that the continuous variables $\mathbf{Y}$ are ranked before the discrete variables $\mathbf{X}$ in the observation $(\mathbf{Y}_i, \mathbf{X}_i)$, and thus also in $(\boldsymbol{\mu}, \boldsymbol{\eta})$ and $\mathbf{C}$ (the subscript i is dropped from the parameters in the following paragraph to keep the notational complexity in check).

While the means can be simply permuted, this is not the case for the Cholesky factor $\mathbf{C}$ (see function `aperm` in Section 2.12). Of course, we can simply permute $\hat{\mathbf{C}}_i$, but we loose standard errors in this process. Alternatively, we can permute the order of variables in $\mathbf{C}$ to our liking in the log-likelihood function (while keeping the original order of the observations and for the mean parameters)

```
> ### discrete variables first
> perm <- c((1:J)[-ic], ic)
> ll_ap <- function(parm, J) {
```

```
+      m <- parm[1:J]             ### mean parameters; NOT permuted
+      parm <- parm[-(1:J)]       ### chol parameters
+      C <- matrix(c(parm), ncol = 1L)
+      C <- ltMatrices(C, diag = TRUE, byrow = BYROW)
+      Ct <- aperm(as.chol(C), perm = perm)
+      -ldpmvnorm(obs = Y[ic,], lower = lwr[-ic,],
+                 upper = upr[-ic,], mean = m, chol = Ct,
+                 w = W[-ic,,drop = FALSE], M = M)
+ }
```

Unfortunately, this distorts the score function and we need to "de-permute" the scores. We start with $\mathbf{\Sigma} = \mathbf{C}\mathbf{C}^\top$, the Cholesky decomposition of a quadratic positive definite $J \times J$ covariance matrix. There are $J \times (J+1)/2$ parameters in the lower triagular part (including the diagonal) of $\mathbf{C}$. Changing the order of the variables by a permutation π with permutation matrix Π gives a covariance $\Pi\mathbf{C}\mathbf{C}^\top\Pi^\top$. This is no longer a Cholesky decomposition, because $\Pi\mathbf{C}$ is not lower triangular. The new decomposition is

$$\Pi\mathbf{C}\mathbf{C}^\top\Pi^\top = \tilde{\mathbf{C}}\tilde{\mathbf{C}}^\top$$

($\tilde{\mathbf{C}}$ is what `aperm` computes). As $\mathbf{C}$, the Cholesky factor $\tilde{\mathbf{C}}$ is lower triangular with $J \times (J+1)/2$ parameters. We could write this operation as a function

$$\begin{aligned} f_3 &: \mathbb{R}^{J\times(J+1)/2} \to \mathbb{R}^{J\times(J+1)/2} \\ f_3(\mathbf{C}) &= \tilde{\mathbf{C}}, \end{aligned}$$

where in fact $f_3 =$`aperm`, and we are interested in its gradient. Deriving the gradient of a Cholesky decomposition might seem hopeless (it certainly did, at least to me, for a very long time), but there is a trick. Let us define two other functions:

$$\begin{aligned} f_1 &: \mathbb{R}^{J\times(J+1)/2} \to \mathbb{R}^{J\times J} \\ f_1(\mathbf{C}) &= \Pi\mathbf{C}\mathbf{C}^\top\Pi^\top \\ f_2 &: \mathbb{R}^{J\times(J+1)/2} \to \mathbb{R}^{J\times J} \\ f_2(\tilde{\mathbf{C}}) &= \tilde{\mathbf{C}}\tilde{\mathbf{C}}^\top. \end{aligned}$$

Exploiting the chain rule for the composition $f_1 = f_2 \circ f_3$, we can write the gradient of f_1 as the product of the gradients of f_2 and f_3:

$$\frac{\partial f_1(\mathbf{C})}{\partial \mathbf{C}} = \frac{\partial f_2(\tilde{\mathbf{C}})}{\partial \tilde{\mathbf{C}}} \frac{\partial f_3(\mathbf{C})}{\partial \mathbf{C}}. \tag{5.1}$$

The last factor is what we want to compute. It turns out that it is simpler to compute the first two gradients first and, in a second step, to derive the last factor. In more detail

$$\begin{aligned} \frac{\partial f_1(\mathbf{C})}{\partial \mathbf{C}} &= \frac{\partial \Pi\mathbf{C}\mathbf{C}^\top\Pi^\top}{\partial \mathbf{C}} \\ &= \frac{\partial \Pi\mathbf{C}\mathbf{C}^\top\Pi^\top}{\partial \Pi\mathbf{C}} \frac{\partial \Pi\mathbf{C}}{\mathbf{C}} \\ &= \left((\Pi\mathbf{C} \otimes \mathbf{I}_J) + (\mathbf{I}_J \otimes \Pi\mathbf{C}) \frac{\partial \mathbf{A}^\top}{\partial \mathbf{A}} \right) (\mathbf{I}_J \otimes \Pi). \end{aligned}$$

($\mathbf{A}$ is a quadratic matrix and the gradient of its transpose is a permutation matrix). This analytic expression only contains known elements and can be computed. The same applies to

$$\begin{aligned} \frac{\partial f_2(\tilde{\mathbf{C}})}{\partial \tilde{\mathbf{C}}} &= \frac{\partial \tilde{\mathbf{C}}\tilde{\mathbf{C}}^\top\Pi}{\partial \tilde{\mathbf{C}}} \\ &= (\tilde{\mathbf{C}} \otimes \mathbf{I}_J) + (\mathbf{I}_J \otimes \tilde{\mathbf{C}}) \frac{\partial \mathbf{A}^\top}{\partial \mathbf{A}} \end{aligned}$$

Both expressions treat $\mathbf{C}$ or $\tilde{\mathbf{C}}$ as full matrices, we are only interested in the score contributions by the $J \times (J+1)/2$ lower triangular elements. Using sloppy notation, we collect the relevant columns in matrices $\mathbf{B}_1 = \frac{\partial f_1(\mathbf{C})}{\partial \mathbf{C}} \in \mathbb{R}^{J^2 \times J \times (J+1)/2}$ and $\mathbf{B}_2 = \frac{\partial f_2(\tilde{\mathbf{C}})}{\partial \tilde{\mathbf{C}}} \in \mathbb{R}^{J^2 \times J \times (J+1)/2}$. For the last, unknown, factor, we write $\mathbf{B}_3 = \frac{\partial f_3(\tilde{\mathbf{C}})}{\partial \tilde{\mathbf{C}}} \in \mathbb{R}^{J \times (J+1)/2 \times J \times (J+1)/2}$ and, with formula (5.1), $\mathbf{B}_1 = \mathbf{B}_2\mathbf{B}_3$. We can then solve for $\mathbf{B}_3$ in the system $\mathbf{B}_1^\top \mathbf{B}_1 = \mathbf{B}_1^\top \mathbf{B}_2 \mathbf{B}_3$.

With `chol` $= \mathbf{C}$, `permuted_chol` $= \tilde{\mathbf{C}}$, `perm` $= \pi$ and score `score_schol` of the log-likelihood $\ell(\tilde{\mathbf{C}})$ with respect to the parameters in $\tilde{\mathbf{C}}$, we can now implement this de-permutation of the scores. Starting with some basic sanity checks, we require lower triangular matrix objects as inputs, with diagonal elements, and check if the dimensions match

⟨ *deperma input checks chol* 103a ⟩ ≡

```
stopifnot(is.ltMatrices(chol))  ### NOTE: replace with is.chol
byrow_orig <- attr(chol, "byrow")
chol <- ltMatrices(chol, byrow = FALSE)
stopifnot(is.ltMatrices(permuted_chol)) ### NOTE: replace with is.chol
permuted_chol <- ltMatrices(permuted_chol, byrow = FALSE)
stopifnot(max(abs(dim(chol) - dim(permuted_chol))) == 0)
J <- dim(chol)[2L]
stopifnot(attr(chol, "diag"))
INVCHOL <- !missing(invchol)
◇
```

Fragment referenced in 104b.

Regarding `perm`, we check if this is an actual permutation

⟨ *deperma input checks perm* 103b ⟩ ≡

```
if (missing(perm)) return(score_schol)
stopifnot(isTRUE(all.equal(sort(perm), 1:J)))
if (max(abs(perm - 1:J)) == 0) return(score_schol)
◇
```

Fragment referenced in 104b.

The scores with respect to $\tilde{\mathbf{C}}$ have been computed elsewhere, we just check the dimensions. In case we were given the scores with respect to $\mathbf{L}$, we first compute the scores with respect to $\mathbf{C}$ (as we were lazy and only derived the results for $\mathbf{C}$). As in `standardize`, the argument `score_schol` gives the score with respect to $\mathbf{C}$ and it is the user's responsibility to provide this quantity (even when `invchol` is given).

⟨ *deperma input checks schol* 103c ⟩ ≡

```
if (is.ltMatrices(score_schol)) {
    byrow_orig_s <- attr(score_schol, "byrow")
    score_schol <- ltMatrices(score_schol, byrow = FALSE)
    ### don't do this here!
    ### if (INVCHOL) score_schol <- -vectrick(permuted_invchol, score_schol)
    score_schol <- unclass(score_schol) ### this preserves byrow
}
stopifnot(is.matrix(score_schol))
N <- ncol(score_schol)
stopifnot(J * (J + 1) / 2 == nrow(score_schol))
◇
```

Fragment referenced in 104b.

We'll have to loop over $i = 1, \ldots, N$ eventually and therefore coerce all objects to objects of class `array`, there is no need to worry about row or column storage order. We set-up indices matrices and the permutation matrix Π

⟨ *deperma indices* 104a ⟩ ≡

```
idx <- matrix(1:J^2, nrow = J, ncol = J)        ### assuming byrow = TRUE
tidx <- c(t(idx))
ltT <- idx[lower.tri(idx, diag = TRUE)]
P <- matrix(0, nrow = J, ncol = J)
P[cbind(1:J, perm)] <- 1
ID <- diag(J)
IDP <- (ID %x% P)
```

◇

Fragment referenced in 104b.

and are now ready for the main course. We are gentle and also allow `invchol`= $\mathbf{L}$ as input, and we clean-up by post-differentiation at the very end in this case.

⟨ *deperma* 104b ⟩ ≡

```
deperma <- function(chol = solve(invchol),
                    permuted_chol = solve(permuted_invchol),
                    invchol, permuted_invchol, perm, score_schol) {

    ⟨ deperma input checks chol 103a ⟩
    ⟨ deperma input checks perm 103b ⟩
    ⟨ deperma input checks schol 103c ⟩

    ⟨ deperma indices 104a ⟩

    Nc <- dim(chol)[1L]
    mC <- as.array(chol)[perm,,,drop = FALSE]
    Ct <- as.array(permuted_chol)
    ret <- lapply(1:Nc, function(i) {
        B1 <- (mC[,,i] %x% ID) + (ID %x% mC[,,i])[,tidx]
        #                                        ^^^^^^^ <- d t(A) / d A
        B1 <- B1 %*% IDP
        B1 <- B1[,ltT] ### relevant columns of B1
        B2 <- (Ct[,,i] %x% ID) + (ID %x% Ct[,,i])[,tidx]
        B2 <- B2[,ltT] ### relevant columns of B2
        B3 <- try(solve(crossprod(B2), crossprod(B2, B1)))
        if (inherits(B3, "try-error"))
            stop("failure computing permutation score")
        if (Nc == 1L)
            return(crossprod(score_schol, B3))
        return(crossprod(score_schol[,i,drop = FALSE], B3))
    })
    ret <- do.call("rbind", ret)
    ret <-ltMatrices(t(ret), diag = TRUE, byrow = FALSE)
    if (INVCHOL)
        ret <- -vectrick(chol, ret)
    ret <- ltMatrices(ret, byrow = byrow_orig_s)
    return(ret)
}
```

◇

Fragment referenced in 64a.

We can now use this function to estimate the Cholesky factor for $(\mathbf{X}, \mathbf{Y})$ when the data comes as $(\mathbf{Y}, \mathbf{X})$ (which is needed because continuous variables come first in our implementation of log-likehood and score function).

```
> sc_ap <- function(parm, J) {
+     m <- parm[1:J]               ### mean parameters; NOT permuted
+     parm <- parm[-(1:J)]         ### chol parameters
+     C <- matrix(c(parm), ncol = 1L)
+     C <- ltMatrices(C, diag = TRUE, byrow = BYROW)
+     ### permutation
+     Ct <- aperm(as.chol(C), perm = perm)
+     ret <- sldpmvnorm(obs = Y[ic,], lower = lwr[-ic,],
+                       upper = upr[-ic,], mean = m, chol = Ct,
+                       w = W[-ic,,drop = FALSE], M = M)
+     ### undo permutation for chol
+     retC <- deperma(chol = C, permuted_chol = Ct,
+                     perm = perm, score_schol = ret$chol)
+     return(-c(rowSums(ret$mean),
+               rowSums(Lower_tri(retC, diag = TRUE))))
+ }
```

and the score function seems to be correct

```
> if (require("numDeriv", quietly = TRUE))
+     chk(grad(ll_ap, start, J = J), sc_ap(start, J = J),
+         check.attributes = FALSE, tol = 1e-6)
```

We can now jointly estimate all model parameters via

```
> op <- optim(start, fn = ll_ap, gr = sc_ap, J = J,
+             method = "L-BFGS-B", lower = llim,
+             control = list(trace = FALSE))
> ## estimated C for (X, Y)
> ltMatrices(matrix(op$par[-(1:J)], ncol = 1),
+            diag = TRUE, byrow = BYROW)

, , 1

       1       2      3      4
1 1.23596 0.00000 0.0000 0.0000
2 0.05465 1.36452 0.0000 0.0000
3 0.29576 0.02194 0.6153 0.0000
4 0.07133 0.66705 0.2361 0.6619

> ## compare with true _permuted_ C for (X, Y)
> round(as.array(aperm(as.chol(lt), perm = perm)), 4)

, , 1

      3     4      1      2
3 1.2247 0.000 0.0000 0.0000
4 0.0000 1.414 0.0000 0.0000
1 0.3536 0.000 0.6124 0.0000
2 0.0000 0.750 0.2887 0.5951
```

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* 106 ⟩ ≡

```
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 64a.

```
> 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

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

and thus

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

and with

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

we can write

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

thus

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

when $\mathbf{b} = \mathrm{vec}(\mathbf{B})$. 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 $\mathbf{T}$ (`score_schol`, the score with respect to $\mathbf{C}$, and it is the user's responsibility to provide this quantity).

⟨ *destandardize* 108 ⟩ ≡

```
destandardize <- function(chol = solve(invchol), invchol, score_schol)
{
    stopifnot(is.ltMatrices(chol))      ### NOTE: replace with is.chol
    J <- dim(chol)[2L]
    stopifnot(!attr(chol, "diag"))
    byrow_orig <- attr(chol, "byrow")
    chol <- ltMatrices(chol, byrow = FALSE)

    ### TODO: check byrow in score_schol?

    if (is.ltMatrices(score_schol))
        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 <- ret[M[lower.tri(M)],,drop = FALSE]
    if (!is.null(dimnames(chol)[[1L]]))
        colnames(ret) <- dimnames(chol)[[1L]]
    ret <- ltMatrices(ret,
                      diag = FALSE, byrow = FALSE,
                      names = dimnames(chol)[[2L]])
    ret <- ltMatrices(ret, byrow = byrow_orig)
    diagonals(ret) <- 0
    return(ret)
}
```

◇

Fragment referenced in 64a.

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. Because no stats text on multivariate problems is complete without a reference to Edgar Anderson's iris data, let's set up a model for these four classical variables

```
> data("iris", package = "datasets")
> 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      1.00000    -0.09887       0.8695      0.7819
Sepal.Width      -0.09887     1.00000      -0.2710     -0.2414
Petal.Length      0.86952    -0.27099       1.0000      0.8714
Petal.Width       0.78191    -0.24142       0.8714      1.0000
```

```
> 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)$chol)))
+ }
> start <- t(chol(CR))
> start <- start[lower.tri(start)]
> if (require("numDeriv", quietly = TRUE))
+     chk(grad(ll, start), sc(start), check.attributes = FALSE)
> op <- optim(start, fn = ll, gr = sc, method = "BFGS",
+             control = list(trace = FALSE), hessian = TRUE)
> op$value
```

```
[1] 602.5
```

```
> 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)$chol)))
+ }
> if (require("numDeriv", quietly = TRUE))
+     chk(grad(ll, start), sc(start), check.attributes = FALSE)
> op2 <- optim(start, fn = ll, gr = sc, method = "BFGS",
+              control = list(trace = FALSE), hessian = TRUE)
> S_NPML <- chol2cov(standardize(ltMatrices(op2$par)))
```

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

```
> S_ML

, , 1

        1       2       3       4
1  1.0000 -0.1139  0.8768  0.7962
2 -0.1139  1.0000 -0.2856 -0.2575
3  0.8768 -0.2856  1.0000  0.8817
4  0.7962 -0.2575  0.8817  1.0000

> S_NPML

, , 1

         1        2       3       4
1  1.00000 -0.09786  0.8735  0.7833
2 -0.09786  1.00000 -0.2726 -0.2482
3  0.87346 -0.27260  1.0000  0.8849
4  0.78328 -0.24822  0.8849  1.0000
```

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 0.00000 0.00000 0.000 0
2 0.08122 0.00000 0.000 0
3 0.13679 0.08762 0.000 0
4 0.12621 0.10787 0.101 0

, , 1

        1       2      3 4
1 0.00000 0.00000 0.0000 0
2 0.07731 0.00000 0.0000 0
3 0.14000 0.08695 0.0000 0
4 0.13691 0.11038 0.1161 0
```

Chapter 7

(Experimental) User Interface

"interface.R" 112a≡

⟨ *mvnorm* 114a ⟩
⟨ *mvnorm methods* 114b ⟩
⟨ *mvnorm simulate* 115 ⟩
⟨ *mvnorm margDist* 116 ⟩
⟨ *mvnorm condDist* 117 ⟩
⟨ *mvnorm logLik* 120c ⟩
⟨ *mvnorm lLgrad* 125 ⟩
◇

The tools provided in the previous chapters are rather low-level, so we will invest some time into setting-up a more high-level interface for representing normal models, either as $\mathbb{N}_J(\boldsymbol{\mu}, \mathbf{C}\mathbf{C}^\top)$ or $\mathbb{N}_J(\boldsymbol{\mu}, \mathbf{L}^{-1}\mathbf{L}^{-\top})$, for simulating from such models, and for evaluating the log-likelihood and corresponding score functions. The latter functionality shall also work when only incomplete (variables are missing) or censored (observations are only known as intervals) data is available.

We start with the conversion of a lower triangular matrix `x` to an `ltMatrices` object

⟨ *as.ltMatrices* 112b ⟩ ≡

```
as.ltMatrices.default <- function(x) {
    stopifnot(is.numeric(x))
    if (!is.matrix(x)) x <- matrix(x)
    DIAG <- max(abs(diag(x) - 1)) > .Machine$double.eps
    DIAG <- DIAG & (nrow(x) > 1)
    lt <- x[lower.tri(x, diag = DIAG)]
    up <- x[upper.tri(x, diag = FALSE)]
    stopifnot(max(abs(up)) < .Machine$double.eps)
    nm <- rownames(x)
    if (!is.null(nm))
        return(ltMatrices(lt, diag = DIAG, names = nm))
    return(ltMatrices(lt, diag = DIAG))
}
```

◇

Fragment referenced in 2.

and proceed defining a constructor for object respresenting, potentially multiple, multivariate normal distributions. If the Cholesky factor $\mathbf{C}$ (or multiple Cholesky factors $\mathbf{C}_1, \ldots, \mathbf{C}_N$) are

given as `chol` argument, we label them as being such objects using `as.chol`. If only a matrix is given, we convert it (if possible) to a single Cholesky factor $\mathbf{C}$. The same is done when $\mathbf{L}$ is given as `invchol` argument. Of course, only one of these arguments must be specified.

⟨ *mvnorm chol invchol* 113a ⟩ ≡

```
if (missing(chol) && missing(invchol))
    chol <- as.chol(ltMatrices(1, diag = TRUE))
stopifnot(xor(missing(chol), missing(invchol)))

if (!missing(chol)) {
    if (!is.ltMatrices(chol))
        chol <- as.ltMatrices(chol)
    scale <- as.chol(chol)
}

if (!missing(invchol)) {
    if (!is.ltMatrices(invchol))
        invchol <- as.ltMatrices(invchol)
    scale <- as.invchol(invchol)
}
ret <- list(scale = scale)
◇
```

Fragment referenced in 114a.

The mean, or multiple means, is stored as a $J \times 1$ or $J \times N$ matrix, and we check if dimensions and, possibly, names are in line with what was specified as `chol` or `invchol`

⟨ *mvnorm mean* 113b ⟩ ≡

```
if (!missing(mean)) {
    stopifnot(is.numeric(mean))
    stopifnot(NROW(mean) == dim(scale)[2L])
    if (!is.matrix(mean)) {
        mean <- matrix(mean, nrow = NROW(mean))
        rownames(mean) <- names(mean)
    }
    nm <- dimnames(scale)[[2L]]
    if (is.null(rownames(mean)))
        rownames(mean) <- nm
    if (!isTRUE(all.equal(rownames(mean), nm)))
        stop("rownames of mean do not match")
    nm <- dimnames(scale)[[1L]]
    if (!is.null(nm) && dim(scale)[[2L]] == ncol(mean)) {
        if (is.null(colnames(mean)))
            colnames(mean) <- nm
        if (!isTRUE(all.equal(colnames(mean), nm)))
            stop("colnames of mean do not match")
    }
    ret$mean <- mean
}
◇
```

Fragment referenced in 114a.

Finally, we put everything together and return an object of class `mvnorm`, featuring `mean` and `scale`. The class of the latter slot carries the information how this object is to be interpreted (as Cholesky factor or inverse thereof)

⟨ *mvnorm* 114a ⟩ ≡

```
### allow more than one distribution
mvnorm <- function(mean, chol, invchol) {

  ⟨ mvnorm chol invchol 113a ⟩
  ⟨ mvnorm mean 113b ⟩
  class(ret) <- "mvnorm"
  return(ret)
}
```

◇

Fragment referenced in 112a.

It might have been smarter to specify the scaled mean $\boldsymbol{\eta} = \mathbf{L}\boldsymbol{\mu}$ because the log-density is then jointly convex in $\boldsymbol{\eta}$ and $\mathbf{L}$ and thus a convex problem would emerge (Barratt and Boyd, 2023).

We add a `names` and `aperm` method. The latter returns a multivariate normal distribution with permuted order of the variables

⟨ *mvnorm methods* 114b ⟩ ≡

```
names.mvnorm <- function(x)
    dimnames(x$scale)[[2L]]

aperm.mvnorm <- function(a, perm, ...) {

    ret <- list(scale = aperm(a$scale, perm = perm, ...))
    if (!is.null(a$mean))
        ret$mean <- a$mean[perm,,drop = FALSE]
    class(ret) <- "mvnorm"
    ret
}
```

◇

Fragment referenced in 112a.

We are now ready to draw samples from such an object. If multiple normal distributions are contained in `object`, we return one sample each, otherwise, `nsim` samples are returned. Because most tools in this package expect data as $J \times N$ matrices, we return the data in this format. If a classical `data.frame` is preferred, `as.data.frame = TRUE` we provide one

⟨ *mvnorm simulate* 115 ⟩ ≡

```
simulate.mvnorm <- function(object, nsim = dim(object$scale)[1L], seed = NULL,
                            standardize = FALSE, as.data.frame = FALSE, ...) {

    J <- dim(object$scale)[2L]
    N <- dim(object$scale)[1L]
    if (N > 1)
        stopifnot(nsim == N)
    if (standardize) {
        if (is.chol(object$scale)) {
            object$scale <- standardize(chol = object$scale)
        } else {
            object$scale <- standardize(invchol = object$scale)
        }
    }
    Z <- matrix(rnorm(nsim * J), nrow = J)
    if (is.chol(object$scale)) {
        Y <- Mult(object$scale, Z)
    } else {
        Y <- solve(object$scale, Z)
    }
    ret <- Y
    if (!is.null(object$mean))
        ret <- ret + c(object$mean)
    rownames(ret) <- dimnames(object$scale)[[2L]]
    if (!as.data.frame)
        return(ret)
    return(as.data.frame(t(ret)))
}
```

◇

Fragment referenced in 112a.

It is maybe time for a first example, and we return to the iris dataset, ignoring the iris' species for the time being. We set-up a model in terms of the sample maximum-likelihood estimates

```
> data("iris", package = "datasets")
> vars <- names(iris)[-5L]
> N <- nrow(iris)
> m <- colMeans(iris[,vars])
> V <- var(iris[,vars]) * (N - 1) / N
> iris_mvn <- mvnorm(mean = m, chol = t(chol(V)))
> iris_var <- simulate(iris_mvn, nsim = nrow(iris))
```

Marginal and conditional distributions might be of interest, the `margDist` and `condDist` methods are simple wrappers to `marg_mvnorm` and `cond_mvnorm`

⟨ *mvnorm margDist* 116 ⟩ ≡

```
margDist <- function(object, which, ...)
    UseMethod("margDist")

margDist.mvnorm <- function(object, which, ...) {

    if (is.chol(object$scale)) {
        ret <- list(scale = as.chol(marg_mvnorm(chol = object$scale,
                                                which = which)$chol))
    } else {
        ret <- list(scale = as.invchol(marg_mvnorm(invchol = object$scale,
                                                   which = which)$invchol))
    }
    if (!is.null(object$mean))
        ret$mean <- object$mean[which,,drop = FALSE]
    class(ret) <- "mvnorm"
    return(ret)
}
```

◇

Fragment referenced in 112a.

⟨ *mvnorm condDist* 117 ⟩ ≡

```
condDist <- function(object, which_given, given, ...)
    UseMethod("condDist")

condDist.mvnorm <- function(object, which_given = 1L, given, ...) {

    if (is.chol(object$scale)) {
        ret <- cond_mvnorm(chol = object$scale, which_given = which_given,
                           given = given, ...)
        ret$scale <- as.chol(ret$chol)
        ret$chol <- NULL
    } else {
        ret <- cond_mvnorm(invchol = object$scale, which_given = which_given,
                           given = given, ...)
        ret$invchol <- as.chol(ret$invchol)
        ret$invchol <- NULL
    }
    if (!is.null(object$mean)) {
        if (is.character(which_given))
            which_given <- match(which_given, dimnames(object$scale)[[2L]])
        if (ncol(object$mean) > 1L && ncol(ret$mean) > 1)
            stop("dimensions do not match")
        if (ncol(object$mean) == 1L && ncol(ret$mean) > 1L) {
            ret$mean <- object$mean[-which_given,,drop = TRUE] + ret$mean
        } else {
            ret$mean <- object$mean[-which_given,,drop = FALSE] + c(ret$mean)
        }

    }
    class(ret) <- "mvnorm"
    return(ret)
}
```

◇

Fragment referenced in 112a.

We could now compute the marginal distribution of two Petal variables or the bivariate regressions of the two Petal variables given the observed Sepal variables. Note that the last object contains $N = 150$ different distributions

```
> j <- 3:4
> margDist(iris_mvn, which = vars[j])

$scale
, , 1

             Petal.Length Petal.Width
Petal.Length       1.7594      0.0000
Petal.Width        0.7315      0.2051


$mean
              [,1]
Petal.Length 3.758
Petal.Width  1.199
```

```
attr(,"class")
[1] "mvnorm"
```

```
> gm <- t(iris[,vars[-(j)]])
> iris_cmvn <- condDist(iris_mvn, which = vars[j], given = gm)
```

We now work towards implementating the corresponding log-likelihood function. This is a trivial task as long as all variables for all observations have been observed exactly (that is, we can interpret the data as being continuous). Here, we also want to allow imprecise, that is, interval-censored, measurements. The one constraint in `ldpmvnorm` is that the continuous variables come first, followed by the censored ones. This of course might not be in line with the variable ordering we have in mind for our model. Our log-likelihood function shall be able to evaluate the log-likelihood for arbitrary permutations of the variables and, optionally, also based on marginal distributions in case observations are missing.

The following `logLik` method for objects of class `mvnorm` is essentially a wrapper for `ldpmvnorm`, handling permutations, marginalisation, and standardisation. We begin with some sanity checks

⟨ *argchecks* 119 ⟩ ≡

```
args <- c(object, list(...))
nargs <- missing(obs) + missing(lower) + missing(upper)
stopifnot(nargs < 3L)

nmobs <- NULL
if (!missing(obs)) {
    if (!is.null(obs)) {
        stopifnot(is.matrix(obs))
        nmobs <- rownames(obs)
    }
}
nmlower <- nmupper <- nmlu <- NULL
if (!missing(lower)) {
    if (!is.null(lower)) {
        stopifnot(is.matrix(lower))
        nmlu <- nmlower <- rownames(lower)
    }
}
if (!missing(upper)) {
    if (!is.null(lower)) {
        stopifnot(is.matrix(upper))
        nmupper <- rownames(upper)
        if (!missing(lower)) {
            stopifnot(isTRUE(all.equal(nmlower, nmupper)))
        } else {
            nmlu <- nmupper
        }
    }
}

nm <- c(nmobs, nmlu)
no <- names(object)
stopifnot(nm %in% no)
perm <- NULL
if (!isTRUE(all.equal(nm, no)))
    perm <- c(nm, no[!no %in% nm])

if (!missing(obs)) args$obs <- obs
if (!missing(lower)) args$lower <- lower
if (!missing(upper)) args$upper <- upper
◇
```

Fragment referenced in 120c, 125.

and proceed with the workhorse when **C** was given

⟨ *logLik chol* 120a ⟩ ≡

```
names(args)[names(args) == "scale"] <- "chol"
if (standardize)
    args$chol <- standardize(chol = args$chol)
if (!is.null(perm)) {
    args$chol <- aperm(as.chol(args$chol), perm = perm)
    if (length(nm) < length(no))
        args$chol <- marg_mvnorm(chol = args$chol, which = nm)$chol
    args$mean <- args$mean[nm,,drop = FALSE]
}
return(do.call("ldpmvnorm", args))
```

◇

Fragment referenced in 120c.

For inverse Cholesky factors $\mathbf{L}$, the code is very similar, just the argument names change

⟨ *logLik invchol* 120b ⟩ ≡

```
names(args)[names(args) == "scale"] <- "invchol"
if (standardize)
    args$invchol <- standardize(invchol = args$invchol)
if (!is.null(perm)) {
    args$invchol <- aperm(as.invchol(args$invchol), perm = perm)
    if (length(nm) < length(no))
        args$invchol <- marg_mvnorm(invchol = args$invchol,
                                    which = nm)$invchol
    args$mean <- args$mean[nm,,drop = FALSE]
}
return(do.call("ldpmvnorm", args))
```

◇

Fragment referenced in 120c.

Putting everything together in a corresponding `logLik` method

⟨ *mvnorm logLik* 120c ⟩ ≡

```
logLik.mvnorm <- function(object, obs, lower, upper, standardize = FALSE,
                          ...) {
    ⟨ argchecks 119 ⟩
    if (is.chol(object$scale)) {
        ⟨ logLik chol 120a ⟩
    }
    ⟨ logLik invchol 120b ⟩
}
```

◇

Fragment referenced in 112a.

allows us to evaluate the log-likelihood of the conditional models for iris

```
> logLik(object = iris_cmvn, obs = t(iris[,vars[-j]]))

[1] -4782
```

This implementation of the log-likelihood silently handles the case when variables have been specified in a different order than hard-wired into the model

```
> logLik(object = iris_cmvn, obs = t(iris[,rev(vars[-j])]))
```

```
[1] -4782
```

The hardest task is the implementation of a score function which features the same options as the log-likelihood function and provides the gradients with respect not only to the parameters (μ and $\mathbf{C}$ or $\mathbf{L}$), but also with respect to the data objects `obs`, `lower`, and `upper`.

In essence, we have to repair the damage imposed by a series of transformations in `logLik.mvnorm`, that is, by standardisation, permutation, and marginalisation. We start with the case when $\mathbf{C}$ was given. First, we repeat all the steps performed in `logLik`, but call the score function `sldpmvnorm` instead of the log-likelihood function `ldpmvnorm`

⟨ *lLgrad chol* 121a ⟩ ≡

```
names(args)[names(args) == "scale"] <- "chol"
sc <- args$chol
if (standardize)
    args$chol <- sc <- standardize(chol = args$chol)
if (!is.null(perm)) {
    if (!attr(args$chol, "diag")) {
        diagonals(args$chol) <- 1
        sc <- args$chol
    }
    args$chol <- pc <- aperm(as.chol(args$chol), perm = perm)
    if (length(nm) < length(no))
        args$chol <- marg_mvnorm(chol = args$chol, which = nm)$chol
    args$mean <- args$mean[nm,,drop = FALSE]
}
ret <- do.call("sldpmvnorm", args)
⟨ lLgrad mean 121b ⟩
⟨ lLgrad marginalisation 122a ⟩
⟨ lLgrad deperma 122b ⟩
⟨ lLgrad destandarized 122c ⟩
⟨ lLgrad diagonals 123a ⟩
⟨ lLgrad return 123b ⟩
◇
```

Fragment referenced in 125.

The next task is to post-differentiate all scores such that the gradients with respect to the original arguments of `logLik` are obtained. We start with the gradient with respect to $\boldsymbol{\mu}$, in case it was not given

⟨ *lLgrad mean* 121b ⟩ ≡

```
### sldmvnorm returns mean score as -obs
if (is.null(ret$mean)) ret$mean <- - ret$obs
◇
```

Fragment referenced in 121a.

In case we marginalised over some variables, we have to set the omitted parameters to zero

⟨ *llgrad marginalisation* 122a ⟩ ≡

```
om <- length(no) - length(nm)
if (om > 0) {
    am <- matrix(0, nrow = om, ncol = ncol(ret$mean))
    rownames(am) <- no[!no %in% nm]
    ret$mean <- rbind(ret$mean, am)
    Jo <- dim(object$scale)[[2L]]
    pJ <- dim(args$invchol)[[2L]]
    am <- matrix(0, nrow = Jo * (Jo + 1) / 2 - pJ * (pJ + 1) / 2,
                 ncol = dim(ret$invchol)[1L])
    byrow_orig <- attr(ret$chol, "byrow")
    ret$chol <- ltMatrices(ret$chol, byrow = TRUE)
    ### rbind only works for byrow = TRUE
    ret$chol <- ltMatrices(rbind(unclass(ret$chol), am),
                           byrow = TRUE,
                           diag = TRUE,
                           names = perm)
    ret$chol <- ltMatrices(ret$chol, byrow = byrow_orig)
}
```

◇

Fragment referenced in 121a.

If the order of the variables was permuted, we compute the scores for the original ordering of the variables, as explained in Chapter 5

⟨ *llgrad deperma* 122b ⟩ ≡

```
if (!is.null(perm))
    ret$chol <- deperma(chol = sc, permuted_chol = pc,
                        perm = match(perm, no),
                        score_schol = ret$chol)
```

◇

Fragment referenced in 121a.

The effect of standardization can be removed as discussed in Chapter 6

⟨ *llgrad destandarized* 122c ⟩ ≡

```
if (standardize)
    ret$chol <- destandardize(chol = object$scale,
                              score_schol = ret$chol)
```

◇

Fragment referenced in 121a.

and it remains to remove fix diagonal elements

⟨ *llGrad diagonals* 123a ⟩ ≡

```
if (!attr(sc, "diag"))
    ret$chol <- ltMatrices(Lower_tri(ret$chol, diag = FALSE),
                           diag = FALSE,
                           byrow = attr(ret$chol, "byrow"),
                           names = dimnames(ret$chol)[[2L]])
◇
```

Fragment referenced in 121a.

and to return the results, with mean scores in the correct ordering

⟨ *llGrad return* 123b ⟩ ≡

```
ret$scale <- ret$chol
ret$chol <- NULL
ret$mean <- ret$mean[no,,drop = FALSE]
return(ret)
◇
```

Fragment referenced in 121a.

The steps are essentially the same when $\mathbf{L}$ was given, but we have to post-differentiate $\mathbf{C} = \mathbf{L}^{-1}$ with respect to $\mathbf{L}$

⟨ *llGrad invchol* 124 ⟩ ≡

```
names(args)[names(args) == "scale"] <- "invchol"
si <- args$invchol
if (standardize)
    args$invchol <- si <- standardize(invchol = args$invchol)
if (!is.null(perm)) {
    if (!attr(args$invchol, "diag")) {
        diagonals(args$invchol) <- 1
        si <- args$invchol
    }
    args$invchol <- pi <- aperm(as.invchol(args$invchol), perm = perm)
    if (length(nm) < length(no))
        args$invchol <- marg_mvnorm(invchol = args$invchol,
                                    which = nm)$invchol
    args$mean <- args$mean[nm,,drop = FALSE]
}
ret <- do.call("sldpmvnorm", args)
### sldmvnorm returns mean score as -obs
if (is.null(ret$mean)) ret$mean <- - ret$obs
om <- length(no) - length(nm)
if (om > 0) {
    am <- matrix(0, nrow = om, ncol = ncol(ret$mean))
    rownames(am) <- no[!no %in% nm]
    ret$mean <- rbind(ret$mean, am)
    Jo <- dim(object$scale)[[2L]]
    pJ <- dim(args$invchol)[[2L]]
    am <- matrix(0, nrow = Jo * (Jo + 1) / 2 - pJ * (pJ + 1) / 2,
                 ncol = dim(ret$invchol)[1L])
    byrow_orig <- attr(ret$invchol, "byrow")
    ret$invchol <- ltMatrices(ret$invchol, byrow = TRUE)
    ### rbind only works for byrow = TRUE
    ret$invchol <- ltMatrices(rbind(unclass(ret$invchol), am),
                              byrow = TRUE,
                              diag = TRUE,
                              names = perm)
    ret$invchol <- ltMatrices(ret$invchol, byrow = byrow_orig)
}
if (!is.null(perm))
    ret$invchol <- deperma(invchol = si, permuted_invchol = pi,
                           perm = match(perm, no),
                           score_schol = -vectrick(pi, ret$invchol))
if (standardize)
    ret$invchol <- destandardize(invchol = object$scale,
                                 score_schol = -vectrick(si, ret$invchol))
if (!attr(si, "diag"))
    ret$invchol <- ltMatrices(Lower_tri(ret$invchol, diag = FALSE),
                              diag = FALSE,
                              byrow = attr(ret$invchol, "byrow"),
                              names = dimnames(ret$invchol)[[2L]])
ret$scale <- ret$invchol
ret$invchol <- NULL
ret$mean <- ret$mean[no,,drop = FALSE]
return(ret)
```

◇

Fragment referenced in 125.

We can now provide the log-likelihood gradients

⟨ *mvnorm lLgrad* 125 ⟩ ≡

```
lLgrad <- function(object, ...)
    UseMethod("lLgrad")

lLgrad.mvnorm <- function(object, obs, lower, upper, standardize = FALSE,
                          ...) {
    ⟨ argchecks 119 ⟩
    if (is.chol(object$scale)) {
        ⟨ lLgrad chol 121a ⟩
    }
    ⟨ lLgrad invchol 124 ⟩
}
```

◇

Fragment referenced in 112a.

Let's use this infrastructure to set-up maximum-likelihood estimation procedures. We start implementing the log-likelihood and score functions for the iris dataset

```
> J <- length(vars)
> obs <- t(iris[, vars])
> lower <- upper <- NULL
> ll <- function(parm) {
+     C <- ltMatrices(parm[-(1:J)], diag = TRUE, names = vars)
+     x <- mvnorm(mean = parm[1:J], chol = C)
+     -logLik(object = x, obs = obs, lower = lower, upper = upper)
+ }
> sc <- function(parm) {
+     C <- ltMatrices(parm[-(1:J)], diag = TRUE, names = vars)
+     x <- mvnorm(mean = parm[1:J], chol = C)
+     ret <- lLgrad(object = x, obs = obs, lower = lower, upper = upper)
+     -c(rowSums(ret$mean), rowSums(Lower_tri(ret$scale, diag = TRUE)))
+ }
```

and can now estimate the mean and Cholesky factor of the covariance matrix. Before we start, we check if the gradient, evaluated at the sample maximum-likelihood estimates, is actually zero.

```
> start <- c(c(iris_mvn$mean), Lower_tri(iris_mvn$scale, diag = TRUE))
> max(abs(sc(start))) < sqrt(.Machine$double.eps)
```

```
[1] TRUE
```

```
> op <- optim(start, fn = ll, gr = sc, method = "L-BFGS-B",
+             lower = llim, control = list(trace = FALSE))
> Chat <- ltMatrices(op$par[-(1:J)], diag = TRUE, names = vars)
> ML <- mvnorm(mean = op$par[1:J], chol = Chat)
```

Quite unsurprisingly, the results are practically equivalent to the analytically available maximum-likelihood estimators in this case

```
> ### covariance
> chol2cov(ML$scale)
```

```
, , 1

             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length      0.68112    -0.04215       1.2658      0.5128
Sepal.Width      -0.04215     0.18871      -0.3275     -0.1208
Petal.Length      1.26582    -0.32746       3.0955      1.2870
Petal.Width       0.51283    -0.12083       1.2870      0.5771

> V

             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length      0.68112    -0.04215       1.2658      0.5128
Sepal.Width      -0.04215     0.18871      -0.3275     -0.1208
Petal.Length      1.26582    -0.32746       3.0955      1.2870
Petal.Width       0.51283    -0.12083       1.2870      0.5771

> ### mean
> ML$mean[,,drop = TRUE]

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.843        3.057        3.758        1.199

> m

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.843        3.057        3.758        1.199
```

Now, this was a lot of work to replace `mean` and `var` with something more fancy, and we would of course not go down this way in real life. But how about a more complex situation where one (or more) variables are only known up to intervals? Let's present the first variable is such a case

```
> v1 <- vars[1]
> q1 <- quantile(iris[[v1]], prob = 1:4 / 5)
> head(f1 <- cut(iris[[v1]], breaks = c(-Inf, q1, Inf)))

[1] (5,5.6]  (-Inf,5] (-Inf,5] (-Inf,5] (-Inf,5] (5,5.6]
Levels: (-Inf,5] (5,5.6] (5.6,6.1] (6.1,6.52] (6.52, Inf]
```

The only necessary modification to our code is the specification of `lower` and `upper` bounds for these intervals, and the removal of the first variable from the "exact continuous" observations `obs`. The rest of the machinery *doesn't need any update at all.* Note that the mean and covariance parameters are no longer orthogonal (as in the toy example above), so we do have to optimise over both sets of parameters simultaneously.

```
> lower <- matrix(c(-Inf, q1)[f1], nrow = 1)
> upper <- matrix(c(q1, Inf)[f1], nrow = 1)
> rownames(lower) <- rownames(upper) <- v1
> obs <- obs[!rownames(obs) %in% v1,,drop = FALSE]
> if (require("numDeriv", quietly = TRUE))
+     chk(grad(ll, start), sc(start), check.attributes = FALSE)
> opi <- optim(start, fn = ll, gr = sc, method = "L-BFGS-B",
+              lower = llim, control = list(trace = FALSE))
> Chati <- ltMatrices(opi$par[-(1:J)], diag = TRUE, names = vars)
> MLi <- mvnorm(mean = opi$par[1:J], chol = Chati)
```

Because the likelihood is a product of a continuous density and a conditional probability as introduced in Chapter 5, the two in-sample log-likelihoods are not comparable. However, the parameters of the two estimated normal distributions can be compared directly (and are rather close in our case)

```
> op$value

[1] 379.9

> opi$value

[1] 472.2

> ### covariance
> chol2cov(MLi$scale)

, , 1

             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length      0.72585    -0.02555       1.2710      0.5221
Sepal.Width      -0.02555     0.18871      -0.3274     -0.1208
Petal.Length      1.27103    -0.32742       3.0950      1.2867
Petal.Width       0.52211    -0.12081       1.2867      0.5770

> chol2cov(ML$scale)

, , 1

             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length      0.68112    -0.04215       1.2658      0.5128
Sepal.Width      -0.04215     0.18871      -0.3275     -0.1208
Petal.Length      1.26582    -0.32746       3.0955      1.2870
Petal.Width       0.51283    -0.12083       1.2870      0.5771

> ### mean
> MLi$mean[,,drop = TRUE]

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.760        3.057        3.758        1.199

> ML$mean[,,drop = TRUE]

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.843        3.057        3.758        1.199
```

We close this chapter with a word of warning: If more than one variable is censored, the `M` and `w` arguments to `lpmvnorm` and `slpmvnorm` have to be specified in `logLik` and `lLgrad` as additional arguments (...) *AND MUST BE IDENTICAL* in both calls.

Chapter 8

Package Infrastructure

⟨ *R Header* 128 ⟩ ≡

```
###    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, 64a.

⟨ *C Header* 129 ⟩ ≡

```
/*
    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, 64b.

Appendix

This document uses the following matrix derivatives

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

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

⟨ diagonal matrix 22 ⟩ Referenced in 2.
⟨ diagonals ltMatrices 19 ⟩ Referenced in 2.
⟨ dim ltMatrices 6c ⟩ Referenced in 2.
⟨ dimensions 70c ⟩ Referenced in 72, 83.
⟨ dimnames ltMatrices 7a ⟩ Referenced in 2.
⟨ dp input checks 97 ⟩ Referenced in 98, 100.
⟨ extract slots 10 ⟩ Referenced in 11, 12, 13, 17, 19, 21, 23a, 27.
⟨ first element 34a ⟩ Referenced in 34b, 35a.
⟨ IDX 35b ⟩ Referenced in 36, 42a.
⟨ increment 68c ⟩ Referenced in 72.
⟨ init center 71c ⟩ Referenced in 72, 83.
⟨ init dans 82b ⟩ Referenced in 83.
⟨ init logLik loop 66c ⟩ Referenced in 72, 77c.
⟨ init random seed, reset on exit 73a ⟩ Referenced in 74, 86.
⟨ init score loop 77c ⟩ Referenced in 83.
⟨ initialisation 66b ⟩ Referenced in 72, 83.
⟨ inner logLik loop 68b ⟩ Referenced in 72.
⟨ inner score loop 81b ⟩ Referenced in 83.
⟨ input checks 65 ⟩ Referenced in 63, 74, 86.
⟨ is.ltMatrices 7c ⟩ Referenced in 2.
⟨ kronecker vec trick 44 ⟩ Referenced in 2.
⟨ L times D 47 ⟩ Referenced in 48.
⟨ lapack options 28 ⟩ Referenced in 29, 30.
⟨ ldmvnorm 57a ⟩ Referenced in 64a.
⟨ ldmvnorm chol 59a ⟩ Referenced in 57a.
⟨ ldmvnorm invchol 59b ⟩ Referenced in 57a.
⟨ ldpmvnorm 98 ⟩ Referenced in 64a.
⟨ lLgrad chol 121a ⟩ Referenced in 125.
⟨ lLgrad deperma 122b ⟩ Referenced in 121a.
⟨ lLgrad destandarized 122c ⟩ Referenced in 121a.
⟨ lLgrad diagonals 123a ⟩ Referenced in 121a.
⟨ lLgrad invchol 124 ⟩ Referenced in 125.
⟨ lLgrad marginalisation 122a ⟩ Referenced in 121a.
⟨ lLgrad mean 121b ⟩ Referenced in 121a.
⟨ lLgrad return 123b ⟩ Referenced in 121a.
⟨ logdet 33a ⟩ Referenced in 3.
⟨ logdet ltMatrices 33b ⟩ Referenced in 2.
⟨ logLik chol 120a ⟩ Referenced in 120c.
⟨ logLik invchol 120b ⟩ Referenced in 120c.
⟨ lower scores 76c ⟩ Referenced in 76e.
⟨ lower triangular elements 17 ⟩ Referenced in 2.
⟨ lpmvnorm 74 ⟩ Referenced in 64a.
⟨ lpmvnormR 63 ⟩ 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 52b ⟩ Referenced in 2.
⟨ mc input checks 52a ⟩ Referenced in 52b, 55.
⟨ mean scores 76b ⟩ Referenced in 76e.
⟨ move on 69a ⟩ Referenced in 72, 83.
⟨ mult 24b ⟩ Referenced in 3.
⟨ mult ltMatrices 23a ⟩ Referenced in 2.
⟨ mult ltMatrices transpose 25 ⟩ Referenced in 23a.
⟨ mult syMatrices 27 ⟩ Referenced in 2.
⟨ mult transpose 26 ⟩ Referenced in 3.
⟨ mvnorm 114a ⟩ Referenced in 112a.
⟨ mvnorm chol invchol 113a ⟩ Referenced in 114a.

⟨ mvnorm condDist 117 ⟩ Referenced in 112a.
⟨ mvnorm lLgrad 125 ⟩ Referenced in 112a.
⟨ mvnorm logLik 120c ⟩ Referenced in 112a.
⟨ mvnorm margDist 116 ⟩ Referenced in 112a.
⟨ mvnorm mean 113b ⟩ Referenced in 114a.
⟨ mvnorm methods 114b ⟩ Referenced in 112a.
⟨ mvnorm simulate 115 ⟩ Referenced in 112a.
⟨ names ltMatrices 7b ⟩ Referenced in 2.
⟨ new score means, lower and upper 80a ⟩ Referenced in 81b.
⟨ output 68d ⟩ Referenced in 72.
⟨ pnorm 70a ⟩ Referenced in 72, 83.
⟨ pnorm fast 69b ⟩ Referenced in 64b.
⟨ pnorm slow 69c ⟩ Referenced in 64b.
⟨ post differentiate chol score 84d ⟩ Referenced in 86.
⟨ post differentiate invchol score 85a ⟩ Referenced in 86.
⟨ post differentiate lower score 84b ⟩ Referenced in 86.
⟨ post differentiate mean score 84a ⟩ Referenced in 86.
⟨ post differentiate upper score 84c ⟩ Referenced in 86.
⟨ post process score 85b ⟩ Referenced in 86.
⟨ print ltMatrices 11 ⟩ Referenced in 2.
⟨ R Header 128 ⟩ Referenced in 2, 64a.
⟨ R lpmvnorm 72 ⟩ Referenced in 64b.
⟨ R slpmvnorm 83 ⟩ Referenced in 64b.
⟨ R slpmvnorm variables 71d ⟩ Referenced in 72, 83.
⟨ RC input 23b ⟩ Referenced in 24b, 26, 29, 30, 33a, 36, 42a.
⟨ reorder ltMatrices 12 ⟩ Referenced in 2.
⟨ score a, b 77b ⟩ Referenced in 77c, 83.
⟨ score c11 77a ⟩ Referenced in 77c, 83.
⟨ score output 82a ⟩ Referenced in 83.
⟨ score output object 76e ⟩ Referenced in 83.
⟨ score wrt new chol diagonal 79b ⟩ Referenced in 81b.
⟨ score wrt new chol off-diagonals 79a ⟩ Referenced in 81b.
⟨ setup return object 71a ⟩ Referenced in 72.
⟨ sldmvnorm 61 ⟩ Referenced in 64a.
⟨ sldpmvnorm 100 ⟩ Referenced in 64a.
⟨ sldpmvnorm invchol 99 ⟩ Referenced in 100.
⟨ slpmvnorm 86 ⟩ Referenced in 64a.
⟨ solve 29 ⟩ Referenced in 3.
⟨ solve C 30 ⟩ Referenced in 3.
⟨ solve ltMatrices 31 ⟩ Referenced in 2.
⟨ standardise 66a ⟩ Referenced in 74, 86.
⟨ standardize 106 ⟩ Referenced in 64a.
⟨ subset ltMatrices 14 ⟩ Referenced in 2.
⟨ syMatrices 6b ⟩ Referenced in 2.
⟨ t(C) S t(A) 41 ⟩ Referenced in 42a.
⟨ tcrossprod 36 ⟩ Referenced in 3.
⟨ tcrossprod diagonal only 34b ⟩ Referenced in 36.
⟨ tcrossprod full 35a ⟩ Referenced in 36.
⟨ tcrossprod ltMatrices 37 ⟩ Referenced in 2.
⟨ univariate problem 71b ⟩ Referenced in 72.
⟨ update d, e 67c ⟩ Referenced in 68b, 81b.
⟨ update f 68a ⟩ Referenced in 68b, 81b.
⟨ update score for chol 80b ⟩ Referenced in 81b.
⟨ update score means, lower and upper 81a ⟩ Referenced in 81b.
⟨ update yp for chol 78a ⟩ Referenced in 81b.
⟨ update yp for means, lower and upper 78b ⟩ Referenced in 81b.
⟨ upper scores 76d ⟩ Referenced in 76e.
⟨ vec trick 42a ⟩ Referenced in 3.

⟨ W length 70b ⟩ Referenced in 72, 83.

Bibliography

Shane Barratt and Stephen Boyd. Covariance prediction via convex optimization. *Optimization and Engineering*, 24(3):2045–2078, 2023. doi: 10.1007/s11081-022-09765-w. 113

Alan Genz. Numerical computation of multivariate normal probabilities. *Journal of Computational and Graphical Statistics*, 1(2), 1992. doi: 10.1080/10618600.1992.10477010. 1, 65

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, 63

Torsten Hothorn. On nonparanormal likelihoods. Technical report, arXiv 2408.17346, 2024. 1

Nadja Klein, Torsten Hothorn, Luisa Barbanti, and Thomas Kneib. Multivariate conditional transformation models. *Scandinavian Journal of Statistics*, 49:116–142, 2022. doi: 10.1111/sjos.12501. 1

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. 69