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

Torsten Hothorn and Lukas Graz

Version 1.5-0

[1]Please cite this document as: Torsten Hothorn () Multivariate Normal Log-likelihoods in the **mvtnorm** Package. R package vignette version 1.5-0, URL DOI:10.32614/CRAN.package.mvtnorm.

Starting with version 1.5-0, some improvements resulting from AI-assisted code review have been added. The corresponding sections and code chunks have been labelled with "AI" such that human- and machine-generated code can be distinguished.

Contents

1 Introduction **1**

2 Lower Triangular Matrices **2**
2.1 Multiple Lower Triangular Matrices . . . 3
2.2 Printing . . . 10
2.3 Reordering . . . 11
2.4 Subsetting . . . 13
2.5 Diagonal Elements . . . 19
2.6 Multiplication . . . 23
2.7 Solving Linear Systems . . . 30
2.8 Log-determinants . . . 35
2.9 Crossproducts . . . 36
2.10 Cholesky Factorisation . . . 40
2.11 Kronecker Products . . . 47
2.12 Convenience Functions . . . 50

3 Marginal and Conditional Normal Distributions **57**
3.1 Continuous Log-likelihoods . . . 65
3.2 Application Example . . . 72

4 Multivariate Normal Log-likelihoods **73**
4.1 Algorithm . . . 74
4.2 Score Function . . . 86
4.3 Log-Scale Implementation (AI written) . . . 98
4.3.1 Log-Scale Algorithm . . . 99
4.3.2 Log-Scale Score Function . . . 104

5 Maximum-likelihood Example **111**

6 Continuous-discrete Likelihoods **119**

7 Unstructured Gaussian Copula Estimation **129**

8 Joint Mean and Covariance Estimation **135**

9 User Interface **138**

10 Reduced Rank Covariance Matrices **161**
10.1 Score Function (AI assisted) . . . 164

11 Package Infrastructure **166**

Licence

Copyright (C) 2022– Torsten Hothorn, 2026– Lukas Graz

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/>.

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 4. An example demonstrating maximum-likelihood estimation of Cholesky factors in the presence of interval-censored observations is discussed in Chapter 5. 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 6 and for nonparametric maximum-likelihood estimation in unstructured Gaussian copulae in Chapter 7. An attempt to provide useRs with a simple and (hopefully) bullet proof interface is documented in Chapter 9.

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"` 1≡

⟨ *R Header* 201 ⟩
⟨ *.lt* 4 ⟩
⟨ *ltMatrices* 7 ⟩
⟨ *syMatrices* 8 ⟩
⟨ *dim ltMatrices* 9 ⟩
⟨ *dimnames ltMatrices* 10 ⟩
⟨ *names ltMatrices* 11 ⟩
⟨ *is.ltMatrices* 12 ⟩
⟨ *as.ltMatrices* 167 ⟩
⟨ *print ltMatrices* 14 ⟩
⟨ *reorder ltMatrices* 15 ⟩
⟨ *subset ltMatrices* 17 ⟩
⟨ *lower triangular elements* 18 ⟩
⟨ *diagonals ltMatrices* 19 ⟩
⟨ *diagonal matrix* 22 ⟩
⟨ *mult ltMatrices* 23 ⟩
⟨ *mult syMatrices* 30 ⟩
⟨ *solve ltMatrices* 33 ⟩
⟨ *logdet ltMatrices* 35 ⟩
⟨ *tcrossprod ltMatrices* 36 ⟩
⟨ *crossprod ltMatrices* 37 ⟩
⟨ *crossprod tcrossprod methods* 38 ⟩
⟨ *chol syMatrices* 39 ⟩
⟨ *invchol syMatrices* 41 ⟩
⟨ *add diagonal elements* 20 ⟩
⟨ *assign diagonal elements* 21 ⟩
⟨ *kronecker vec trick* 47 ⟩
⟨ *convenience functions* 51 ⟩
⟨ *perm* 55 ⟩
⟨ *marginal* 60 ⟩
⟨ *conditional* 63 ⟩
⟨ *check obs* 65 ⟩
⟨ *colSumsdnorm ltMatrices* 67 ⟩
◇

"ltMatrices.c" 2≡

```
⟨ C Header 202 ⟩
#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 66 ⟩
⟨ solve 31 ⟩
⟨ solve C 32 ⟩
⟨ logdet 34 ⟩
⟨ mult 29 ⟩
⟨ chol 40 ⟩
⟨ invchol workhorse 43 ⟩
⟨ invchol 42 ⟩
◇
```

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:

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

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

⟨ *ltMatrices dim* 3 ⟩ ≡

```
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 7.

Before we begin assigning names, we set-up an internal shortcut for `lower.tri`, which is a bit more memory and speed efficient

⟨ *.lt* 4 ⟩ ≡

```
.lt <- function(J, diag = FALSE) {
    j <- seq_len(J)
    j1 <- rep(c(TRUE, FALSE), J)[- 2 * J]
    j2 <- c(rbind(rev(j), j))[- 2 * J]
    ret <- matrix(rep(j1, j2), nrow = J)
    if (!diag) diag(ret) <- FALSE
    ret
}
.ut <- function(J, diag = FALSE) {
    !.lt(J = J, diag = !diag)
}
```

◇

Fragment referenced in 1.

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* 5 ⟩ ≡

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

if (!nonames) {
    idx <- matrix(1:J, nrow = J, ncol = J)
    idx <- idx[.lt(J, diag = diag)]
    j <- seq_len(J - !(diag + 0L))
    idx2 <- rep(j, rev(j))
    rn <- paste(names[idx], names[idx2], sep = ".")
    if (byrow)
       rn <- rn[.c2r(J, diag = diag)]
    rownames(object) <- rn
} else {        ### add later
    warning("ltMatrices objects should be properly named")
}
```

◇

Fragment referenced in 7.

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* 6 ⟩ ≡

```
if (is.ltMatrices(object)) {
    cls <- class(object)                ### keep inheriting classes
    ret <- .reorder(object, byrow = byrow)
    class(ret) <- class(object)
    return(ret)
}
```

◇

Fragment referenced in 7.

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

⟨ *ltMatrices* 7 ⟩ ≡

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

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

    ⟨ ltMatrices input 6 ⟩
    ⟨ ltMatrices dim 3 ⟩
    ⟨ ltMatrices names 5 ⟩

    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 1.

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

⟨ *syMatrices* 8 ⟩ ≡

```
as.syMatrices <- function(x) {
    if (is.syMatrices(x))
        return(x)
    if (is.ltMatrices(x)) {
        class(x) <- gsub("ltMatrices", "syMatrices", class(x))
        return(x)
    }
    as.syMatrices(as.ltMatrices(x * .lt(nrow(x), diag = TRUE)))
}
syMatrices <- function(object, diag = FALSE, byrow = FALSE, names = TRUE) {
    if (inherits(object, "syMatrices"))
        class(object) <- gsub("syMatrices", "ltMatrices", class(object))
    as.syMatrices(ltMatrices(object = object, diag = diag, byrow = byrow,
                             names = names))
}
```

◇

Fragment referenced in 1.

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

⟨ *dim ltMatrices* 9 ⟩ ≡

```
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 1.

The corresponding dimnames can be extracted as

⟨ *dimnames ltMatrices* 10 ⟩ ≡

```
dimnames.ltMatrices <- function(x)
    return(list(attr(x, "dimnames")[[2L]], attr(x, "rcnames"), attr(x, "rcnames")))
dimnames.syMatrices <- dimnames.ltMatrices
```

◇

Fragment referenced in 1.

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

⟨ *names ltMatrices* 11 ⟩ ≡

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

◇

Fragment referenced in 1.

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 9.

⟨ *is.ltMatrices* 12 ⟩ ≡

```
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 1.

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 1.0000000         .         .         .         .
B 0.5123660 1.0000000         .         .         .
C 0.0584725 0.9095137 1.0000000         .         .
D 0.3944872 0.6612143 0.2335259 1.0000000         .
E 0.5164752 0.2979867 0.0751775 0.8182123 1.0000000

, , C_2

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

, , C_3

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

, , C_4

          A         B         C         D         E
A 1.0000000         .         .         .         .
B 0.4877241 1.0000000         .         .         .
C 0.0593046 0.7625270 1.0000000         .         .
D 0.0005227 0.1995700 0.4705089 1.0000000         .
E 0.4913541 0.2849431 0.0059611 0.8901458 1.0000000

> 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* 13 ⟩ ≡

```
diag <- attr(x, "diag")
byrow <- attr(x, "byrow")
d <- dim(x)
N <- d[1L]
J <- d[2L]
dn <- dimnames(x)
```

◇

Fragment referenced in 14, 15, 16, 18, 19, 21, 23, 30.

⟨ *print ltMatrices* 14 ⟩ ≡

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

    ⟨ extract slots 13 ⟩

    x <- unclass(x)

    L <- matrix(1L, nrow = J, ncol = J)
    diag(L) <- 2L
    if (byrow) {
        L[.ut(J, diag = diag)] <- floor(2L + 1:(J * (J - 1) / 2L + diag * J))
        L <- t(L)
    } else {
        L[.lt(J, diag = diag)] <- floor(2L + 1:(J * (J - 1) / 2L + diag * J))
    }
    if (symmetric) {
        L[.ut(J)] <- 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]
    if (!isFALSE(dn[[2L]]) && !isFALSE(dn[[3L]]))
        dimnames(ret) <- dn[3:1]
    return(ret)
}

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

print.ltMatrices <- function(x, zero.print = ".", ...) {
    if (is.null(attr(x, "dimnames")[[2L]]))
        attr(x, "dimnames")[[2L]] <- as.character(seq_len(dim(x)[1L]))
    print(as.table(as.array(x)), zero.print = zero.print, ...)
}

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

◇

Fragment referenced in 1.

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. This is quite fast but we still try to avoid it, for example in `Mult`, Section 2.6.

⟨ *reorder ltMatrices* 15 ⟩ ≡

```
.r2c <- function(J, diag = FALSE) {

    rL <- cL <- diag(0L, nrow = J)
    lt <- .lt(J, diag = diag)
    ut <- !lt
    diag(ut) <- !diag(ut)
    x <- seq_len(J * (J + c(-1, 1)[diag + 1L]) / 2)
    cL[ut] <- seq_len(length(x))
    x[t(cL)[lt]]
}

.c2r <- function(J, diag = FALSE) {

    rL <- cL <- diag(0L, nrow = J)
    lt <- .lt(J, diag = diag)
    ut <- !lt
    diag(ut) <- !diag(ut)
    x <- seq_len(J * (J + c(-1, 1)[diag + 1L]) / 2)
    rL[lt] <- seq_len(length(x))
    x[t(rL)[ut]]
}

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

    ### only we call this function
    ### stopifnot(is.ltMatrices(x))
    if (attr(x, "byrow") == byrow) return(x)

    ⟨ extract slots 13 ⟩

    x <- unclass(x)

    if (byrow) { ### row -> col order
        return(ltMatrices(x[.r2c(J, diag = diag), , drop = FALSE],
                          diag = diag, byrow = FALSE, names = dn[[2L]]))
    }
    return(ltMatrices(x[.c2r(J, diag = diag), , drop = FALSE],
                      diag = diag, byrow = TRUE, names = dn[[2L]]))
}
```

◇

Fragment referenced in 1.

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* 16 ⟩ ≡

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

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

    ⟨ extract slots 13 ⟩

    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(.ut(J, diag = diag))
        if (byrow) {
            L[.ut(J, diag = diag)] <- 1:Jp
            L <- L + t(L)
            diag(L) <- diag(L) / 2
            L <- L[j, j, drop = FALSE]
            L <- L[.ut(nrow(L), diag = diag)]
        } else {
            L[.lt(J, diag = diag)] <- 1:Jp
            L <- L + t(L)
            diag(L) <- diag(L) / 2
            L <- L[j, j, drop = FALSE]
            L <- L[.lt(nrow(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 17.

⟨ *subset ltMatrices* 17 ⟩ ≡

```
⟨ .subset ltMatrices 16 ⟩
### 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 1.

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)
> print(try(ltM[i, j], silent = TRUE))

[1] "Error in `[.ltMatrices`(ltM, i, j) : invalid subset argument j\n"
attr(,"class")
```

```
[1] "try-error"
attr(,"condition")
<simpleError in `[.ltMatrices`(ltM, i, j): invalid subset argument 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* 18 ⟩ ≡

```
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 13 ⟩

    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 1.

```
> ## 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 13 ⟩

    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 1.

```
> 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) {

    ### only we call this function
    # 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 <- D <- diag(J)
    lt <- .lt(J, diag = TRUE)
    L[lt] <- seq_len(J * (J + 1) / 2)

    ### by default new diagonal elements are 1
    ret <- matrix(D[lt], nrow = J * (J + 1) / 2, ncol = N)
    colnames(ret) <- dimnames(x)[[1L]]
    diag(lt) <- FALSE
    ret[L[lt],] <- unclass(x)

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

    ret
}
```

◇

Fragment referenced in 1.

⟨ *assign diagonal elements* 21 ⟩ ≡

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

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

    ⟨ extract slots 13 ⟩

    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 1.

```
> 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 1.

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

, , 1

  1 2 3 4 5
1 1 . . . .
2 . 1 . . .
3 . . 1 . .
4 . . . 1 .
5 . . . . 1

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

, , 1

  1 2 3 4 5
1 1 . . . .
2 . 2 . . .
3 . . 3 . .
4 . . . 4 .
5 . . . . 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 the 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 3 for defining `cond_mvnorm` later on.

We have three use cases for this matrix-vector multiplication: (1) If the number of rows of the matrix `y` is J, each column is interpreted as $\mathbf{y}_i \in \mathbb{R}^J$. (2) If the number of rows is J^2, the returned matrix is $J^2 \times N$ with each columns representing $\text{vec}(\mathbf{C}_i\mathbf{Y}_i)$ for the $J \times J$ matrix $\mathbf{Y}_i = (\mathbf{y}_1 \mid \mathbf{y}_2 \mid \cdots \mid \mathbf{y}_J)$. (3) If the number of rows of the matrix `y` is $J(J+1)/2$, we interpret the ith column of `y` as the lower triangular elements of $\mathbf{C}_i$ (in column-major storage) for computing $\mathbf{C}_i\mathbf{C}_i^\top$ (`transpose = FALSE`) or $\mathbf{C}_i^\top\mathbf{C}_i$ (`transpose = TRUE`) later on in Section 2.9. Again, this is dangerous but helps to speed-up crossproducts considerably.

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* 23 ⟩ ≡

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

    ⟨ extract slots 13 ⟩
    ### dtpmv requires diagonal elements being present
    if (!diag) diagonals(x) <- diagonals(x)

    ### for crossprod(x), use case (3)
    if (is.null(y)) {
        stopifnot(!byrow)
        stopifnot(transpose)
        y <- Lower_tri(x, diag = TRUE)
    } else {
        stopifnot(NROW(y) == J || NROW(y) == J^2)
    }

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

    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(J), as.logical(diag), as.logical(transpose))

    if (NROW(y) == d[2L] && !isFALSE(dn[[2L]]))
        rownames(ret) <- dn[[2L]]
    if (length(dn[[1L]]) == N && !isFALSE(dn[[1L]]))
        colnames(ret) <- dn[[1L]]
    return(ret)
}
```

◇

Fragment referenced in 1.

We start with some options for the LAPACK workhorses, regarding the presence of non-unit diagonal elements (which, however, have to be part of $\mathbf{C}$) and whether $\mathbf{C}$ comes in row- or column-major order.

⟨ *lapack options* 24 ⟩ ≡

```
char di, uplo;
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';
}

SEXP abyrow = PROTECT(getAttrib(C, Rf_install("byrow")));
if (TYPEOF(abyrow) != LGLSXP) error("non-logical byrow attribute");
if (LOGICAL(abyrow)[0]) {
    uplo = 'U';
} else {
    uplo = 'L';
}

◇
```

Fragment referenced in 29, 31, 32.

The underlying C code allows $\mathbf{C}_i$ (here called `C`) to be in row-major or column-major order. In column-major $\mathbf{Cy}$, we have `uplo = "L"` (lower) and `tr = "U"` (take $\mathbf{C}$ at face value). If $\mathbf{C}$ comes in row-major order, we pretent it is an upper triangular matrix (`uplo = "U"`) and and compute $\mathbf{C}^\top\mathbf{y}$ (`tr = "T"`).

⟨ *row column* 25 ⟩ ≡

```
char tr = 'N';
Rboolean Rtranspose = asLogical(transpose);
/* t(C) instead of C for column-major*/
if (Rtranspose && !LOGICAL(abyrow)[0])
    tr = 'T';
/* C instead for row-major*/
if (!Rtranspose && LOGICAL(abyrow)[0])
    tr = 'T';
◇
```

Fragment referenced in 29.

We extract dimensions next

⟨ *RC input* 26 ⟩ ≡

```
/* 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 29, 31, 32, 34.

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* 27 ⟩ ≡

```
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 29, 31, 34.

We can now implement the main part of the multiplication. We first differentiate between the three use cases. For (1), we loop over the columns of `C` and / or the columns of `y`. For (2), we additionally loop over the J vectors of length J defining the columns of `y`. For (3), we only loop over the vectors of lengths $J, J-1, \ldots, 1$ and also corresponding sub-matrices of $\mathbf{C}$ such that the lower triangular elements of the symmetric crossproduct $\mathbf{C}\mathbf{C}^\top$ emerge.

⟨ *mult workhorse* 28 ⟩ ≡

```
/* use case (1) */
if (LENGTH(y) == iJ * iN) {
    nry = 1;
    lty = 0;
/* use case (2) */
} else if (LENGTH(y) == iJ * iJ * iN) {
    nry = iJ;
    lty = 0;
/* use case (3) */
} else if (LENGTH(y) == iJ * (iJ + 1) / 2 * iN) {
    if (!(Rtranspose && !LOGICAL(abyrow)[0]))
        error("matrix multiplication not implemented");
    nry = iJ;
    lty = 1;
} else {
    error("incorrect dimension of y argument");
}

thisJ = iJ;
for (i = 0; i < iN; i++) {
    incr = 0;
    for (j = 0; j < nry; j++) {
        if (lty) thisJ = iJ - j;
        F77_CALL(dtpmv)(&uplo, &tr, &di, &thisJ,
                        dC + incr, dans, &iONE FCONE FCONE FCONE);
        dans += (lty ? iJ - j : iJ);
        if (lty)
            incr += thisJ;
    }
    dC += p;
}
```

◇

Fragment referenced in 29.

We are now ready to implement the C workhorse, using LAPACK function `dtrmv`.

⟨ *mult* 29 ⟩ ≡

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

    SEXP ans;
    double *dans;
    int i, j, nry, lty, thisJ, incr;
    int iONE = 1;

    ⟨ RC input 26 ⟩
    /* diagonal elements are always present */
    if (!Rdiag) len += iJ;
    ⟨ C length 27 ⟩
    ⟨ lapack options 24 ⟩
    ⟨ row column 25 ⟩

    PROTECT(ans = duplicate(y));
    dans = REAL(ans);

    ⟨ mult workhorse 28 ⟩

    UNPROTECT(2);
    return(ans);
}
```

◇

Fragment referenced in 2.

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

```
> ### matrix-vector
> 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)
> ### matrix-matrix
> Y <- matrix(runif(N * J^2), nrow = J^2)
> a <- Mult(lxn, Y)
> b <- do.call("rbind", lapply(1:ncol(y),
+     function(i) t(A[,,i] %*% matrix(Y[,i,drop = FALSE], nrow = J))))
> chk(a, t(b), check.attributes = FALSE)
> ### matrix-vector
> 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)
> ### matrix-matrix
> a <- Mult(lxd, Y)
> b <- do.call("rbind", lapply(1:ncol(y),
+     function(i) t(A[,,i] %*% matrix(Y[,i,drop = FALSE], nrow = J))))
> chk(a, t(b), check.attributes = FALSE)
```

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

and 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 $\mathbf{A}\mathbf{y} = \mathbf{C}\mathbf{y} + \mathbf{C}^\top\mathbf{y} - \text{diag}(\mathbf{C})\mathbf{y}$ using `Mult` applied to the lower trianular part:

⟨ *mult syMatrices* 30 ⟩ ≡

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

    ⟨ extract slots 13 ⟩

    x <- as.ltMatrices(x)
    stopifnot(is.numeric(y))
    if (!is.matrix(y)) y <- matrix(y, nrow = J, ncol = N)
    N <- ifelse(d[1L] == 1, ncol(y), N)
    ### <NOTE> not yet implemented for J x J matrices in y, see
    ### Mult.ltMatrices </NOTE>
    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 1.

```
> 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, check.attributes = FALSE)
+ })
> 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 $J(J-1)/2+J\text{diag}\times N$), and `y` is the $J \times N$ matrix $(\mathbf{y}_1 \mid \mathbf{y}_2 \mid \cdots \mid \mathbf{y}_N)$. This function returns the $J \times N$ matrix $(\mathbf{x}_1 \mid \mathbf{x}_2 \mid \cdots \mid \mathbf{x}_N)$ of solutions.

If `y` is not given, $\mathbf{C}_i^{-1}$ is returned in 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 set-up a dedicated `C` function for computing $\mathbf{C}_i\mathbf{x}_i = \mathbf{y}_i$

⟨ *solve* 31 ⟩ ≡

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

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

    ⟨ RC input 26 ⟩
    /* diagonal elements are always present */
    if (!Rdiag) len += iJ;
    ⟨ C length 27 ⟩
    ⟨ lapack options 24 ⟩


    char tr = 'N';
    Rboolean Rtranspose = asLogical(transpose);
    /* t(C) instead of C for column-major*/
    if (Rtranspose && !LOGICAL(abyrow)[0])
        tr = 'T';
    /* C instead for row-major*/
    if (!Rtranspose && LOGICAL(abyrow)[0])
        tr = 'T';

    PROTECT(ans = duplicate(y));
    dans = REAL(ans);

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

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

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

◇

Fragment referenced in 2.

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

⟨ *solve C* 32 ⟩ ≡

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

    SEXP ans;
    int i, info;

    ⟨ RC input 26 ⟩
    /* diagonal elements are always present */
    if (!Rdiag) len += iJ;
    ⟨ lapack options 24 ⟩

    PROTECT(ans = duplicate(C));
    /* dC is defined in RC input but otherwise not used */
    dC = REAL(ans);

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

        /* compute inverse */
        F77_CALL(dtptri)(&uplo, &di, &iJ, dC, &info FCONE FCONE);
        if (info != 0)
            error("Cannot solve ltMatrices");

        dC += len;
    }

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

◇

Fragment referenced in 2.

with R interface

⟨ *solve ltMatrices* 33 ⟩ ≡

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

    diag <- attr(a, "diag")
    ### dtptri and dtpsv require diagonal elements being present
    if (!diag) diagonals(a) <- diagonals(a)
    d <- dim(a)
    J <- d[2L]
    dn <- dimnames(a)
    if (!is.double(a)) storage.mode(a) <- "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, a, 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, a,
                 as.integer(d[1L]), as.integer(J), as.logical(diag),
                 as.logical(FALSE))
    ### chol = solve(invchol); invchol = solve(chol)
    if (is.chol(ret)) {
        class(ret) <- gsub("chol", "invchol", class(ret))
    } else if (is.invchol(ret)) {
        class(ret) <- gsub("invchol", "chol", class(ret))
    }

    if (diag)
        return(ret)

    ### ret always includes diagonal elements, remove here
    ret <- ltMatrices(Lower_tri(ret, diag = FALSE),
                      byrow = attr(a, "byrow"), names = dn[[2L]])
    return(ret)
}
```

◇

Fragment referenced in 1.

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)
```

We run the same checks again switching between row- and column-major order.

```
> tn <- lxn
> td <- lxd
> lxn <- ltMatrices(lxn, byrow = !attr(lxn, "byrow"))
> lxd <- ltMatrices(lxd, byrow = !attr(lxd, "byrow"))
> ## 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)]))
> chk(solve(lxn[1,], y, transpose = TRUE),
+     t(as.array(solve(lxn[1,]))[,,1]) %*% y)
> lxn <- tn
> lxd <- td
```

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* 34 ⟩ ≡

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

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

    ⟨ RC input 26 ⟩
    Rboolean Rbyrow = asLogical(byrow);
    ⟨ C length 27 ⟩

    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 2.

The R interface now simply calls this low-level function

⟨ *logdet ltMatrices* 35 ⟩ ≡

```
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 1.

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 compute $\mathbf{C}_i \mathbf{C}_i^\top$ or $\text{diag}(\mathbf{C}_i \mathbf{C}_i^\top)$ (`diag_only = TRUE`) for $i = 1, \ldots, N$. These are symmetric matrices, so we store them as a lower triangular matrix using a different class name `syMatrices`. We reuse `Mult` 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 and computation of the full $J \times J$ crossproduct matrix

⟨ *tcrossprod ltMatrices* 36 ⟩ ≡

```
### 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) {

    ### crossprod(x) = t(x) %*% x
    if (transpose) {
        if (diag_only)
            return(marginSums(as.array(x^2), c(2L, 3L)))
        byrow_orig <- attr(x, "byrow")
        x <- ltMatrices(x, byrow = FALSE)
        ret <- syMatrices(Mult(x, transpose = TRUE),
                          diag = TRUE, byrow = FALSE,
                          names = dimnames(x)[[2L]])
        return(syMatrices(ret, byrow = byrow_orig))
    }

    ### tcrossprod(x) = x %*% t(x)
    if (diag_only)
        return(marginSums(as.array(x^2), c(1L, 3L)))
    ### t(x) for all i = 1, ..., N
    ax <- matrix(aperm(as.array(x),  c(2L, 1L, 3L)), ncol = dim(x)[1L])
    byrow_orig <- attr(x, "byrow")
    x <- ltMatrices(x, byrow = FALSE) ### not needed but Mult() is faster
    ret <- syMatrices(Mult(x, ax)[.lt(dim(x)[2L], diag = TRUE),,drop = FALSE],
                      diag = TRUE, byrow = FALSE,
                      names = dimnames(x)[[2L]])
    return(syMatrices(ret, byrow = byrow_orig))
}

### <FIXME> remove Tcrossprod / Crossprod from interface (mlt/tram may use
### it) </FIXME>
Tcrossprod <- function(x, diag_only = FALSE)
    .Tcrossprod(x = x, diag_only = diag_only, transpose = FALSE)
◇
```

Fragment referenced in 1.

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* 37 ⟩ ≡

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

◇

Fragment referenced in 1.

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)))
```

We now also add methods implementing a standard user interface

⟨ *crossprod tcrossprod methods* 38 ⟩ ≡

```
crossprod.ltMatrices <- function(x, y = NULL, ...) {

    if (is.null(y))
        return(Crossprod(x = x, ...))
    return(Mult(x, y, transpose = TRUE))
}

crossprod.syMatrices <- crossprod.ltMatrices

tcrossprod.ltMatrices <- function(x, y = NULL, ...) {

    if (is.null(y))
        return(Tcrossprod(x = x, ...))
    return(Mult(x, y, transpose = FALSE))
}

tcrossprod.syMatrices <- tcrossprod.ltMatrices

'%*%.ltMatrices' <- function(x, y)
    Mult(x, y)

'%*%.syMatrices' <- function(x, y)
    Mult(x, y)
```

◇

Fragment referenced in 1.

which allows coding closer to the standard

```
> ### 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)
> 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)
> ## 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)
> 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)
```

Multiplication also works in this simple way

```
> a <- 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 <- 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(lxn[rep(1, N),] %*% y, lxn[1,] %*% y, check.attributes = FALSE)
> ### recycle y
> chk(lxn %*% y[,1], 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) 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)
> a <- crossprod(lxn, y)
> 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 <- crossprod(lxd, y)
> 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(crossprod(lxn[rep(1, N),], y),
+     crossprod(lxn[1,], y), check.attributes = FALSE)
> ### recycle y
> chk(crossprod(lxn, y[,1]),
+     crossprod(lxn, y[,rep(1, N)]))
```

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, ...) {

    d <- dim(x)
    x <- .adddiag(x)
    if (!is.double(x)) storage.mode(x) <- "double"

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

    ### chol is lower triangular
    class(ret) <- gsub("syMatrices", "ltMatrices", class(ret))

    return(ret)
}
```

◇

Fragment referenced in 1.

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, which requires matrices in column-major order, however, allows to switch between lower and upper triangular matrices. Thus the row-major order for our lower triangular matrices can be computed by pretending to be upper triangular.

⟨ *chol* 40 ⟩ ≡

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

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

    SEXP abyrow = getAttrib(Sigma, Rf_install("byrow"));
    if (TYPEOF(abyrow) != LGLSXP) error("non-logical byrow attribute");
    if (LOGICAL(abyrow)[0])
        uplo = 'U';

    /* duplicate preserves classes, so ltMatrices is returned */
    ans = PROTECT(duplicate(Sigma));
    dans = REAL(ans);

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

        F77_CALL(dpptrf)(&uplo, &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");
        }

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

◇

Fragment referenced in 2.

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

```
> Sigma <- tcrossprod(lxd)
> chk(chol(Sigma), lxd)
> chk(as.array(chol(Sigma)),
+     as.array(chol(syMatrices(Sigma, byrow = !attr(Sigma, "byrow")))))
> Sigma <- tcrossprod(lxn)
> ## Sigma and chol(Sigma) always have diagonal, lxn doesn't
> chk(as.array(chol(Sigma)), as.array(lxn))
> ### check singular matrices
> x <- matrix(runif(21), ncol = 7)
> S <- crossprod(x)
```

with meaningful error messages

```
> ## IGNORE_RDIFF_BEGIN
```

«cholerr» print(try(chol(S), silent = TRUE)) print(try(chol(as.syMatrices(S)), silent = TRUE))

```
> ## IGNORE_RDIFF_END
```

Alternatively, we might want to compute the decomposition $\mathbf{\Sigma}_i = \mathbf{L}_i^{-1}\mathbf{L}_i^{-\top}$ for multiple symmetric matrices $\mathbf{\Sigma}_i$, stored as a matrix in class `syMatrices`. We begin, in analogy to `chol`, setting up a new generic `invchol` with methods for `syMatrices` and also a default method handling matrices.

We first recall the interpretation of the diagonal and off-diagonal elements of $\mathbf{L}$ (without the i index here) from the log-likelihood (3.6)

$$\begin{aligned}
\ell(\boldsymbol{\nu}, \mathbf{L}) &= -\frac{J}{2}\log(2\pi) + \sum_{j=1}^{J} \log \operatorname{diag}(\mathbf{L})_j - \frac{1}{2}(\mathbf{L}\mathbf{y} - \boldsymbol{\nu})^\top(\mathbf{L}\mathbf{y} - \boldsymbol{\nu}) \\
&\propto \sum_{j=1}^{J} \log \operatorname{diag}(\mathbf{L})_j - \frac{1}{2}(\boldsymbol{\lambda}_j^\top \mathbf{y} - \nu_j)^2 \\
&= \sum_{j=1}^{J} \log(\lambda_{jj}) - \frac{1}{2}\left(\sum_{j=1}^{j-1} \lambda_{jj} y_j + \lambda_{jj} y_j - \nu_j\right)^2 .
\end{aligned}$$

where $\lambda_{jj} = \operatorname{diag}(\mathbf{L})_j$ and $\boldsymbol{\lambda}_j$ is the jth row of $\mathbf{L}$ of length j. That is, the joint log-likelihood is the sum the J log-likelihoods corresponding to the linear regressions of Y_j on $Y_1, \ldots, Y_{j-1}$, with residual standard error $\operatorname{diag}(\mathbf{L})_j^{-1} = \lambda_{jj}^{-1}$ and least-square regression coefficients $\beta_j = \lambda_{jj}\lambda_{jj}$ (NB: Veccia approximations use the same setup, but constrain some elements of β to zero).

Thus, in order to obtain $\mathbf{L}$, we only have to compute the whole series of linear regressions from $\mathbf{\Sigma}$. We proceed by induction: For $j = 1$, we simply get $\lambda_{11} = \operatorname{diag}(\mathbf{\Sigma})_1^{-1/2}$. For j, with $\mathbf{S} = \sum_{i=}^{N}(\mathbf{y}_i - \bar{\mathbf{y}})(\mathbf{y}_i - \bar{\mathbf{y}})^\top$ we can write $\boldsymbol{\beta}_j = \mathbf{S}^{-1}_{1:(j-1),1:(j-1)}\mathbf{S}_j$ for $j = 2, \ldots, J$ based on the partioned $j \times j$ submatrix

$$\mathbf{S}_{\mathbf{j},\mathbf{j}} = \begin{pmatrix} \mathbf{S}^{-1}_{1:(j-1),1:(j-1)} & \mathbf{S}_j \\ \mathbf{S}_j^\top & \sigma_j^2 \end{pmatrix}$$

of $\mathbf{\Sigma}$. This matrix corresponds to the crossproduct $\mathbf{S}^{-1}_{1:(j-1),1:(j-1)} = \mathbf{X}^\top\mathbf{X}$ of the $j-1$ regressors and $\mathbf{X}^\top Y = \mathbf{S}_j$ when using the classical $Y = \mathbf{X}^\top\boldsymbol{\beta} + \sigma\varepsilon$ notation with $\varepsilon \sim \mathbb{N}_1(0, 1)$.

We already have that $\mathbf{S}_{1:(j-1),1:(j-1)} = \mathbf{L}^{-1}_{1:(j-1),1:(j-1)}\mathbf{L}^{-\top}_{1:(j-1),1:(j-1)}$, so

$$\boldsymbol{\beta}_j = \mathbf{L}^\top_{1:(j-1),1:(j-1)}\mathbf{L}_{1:(j-1),1:(j-1)}\mathbf{S}_j.$$

We also know that the variance of Y_j was estimated as $\sigma_j^2 = \operatorname{diag}(\mathbf{S})_j$ and thus

$$\begin{aligned}
\sigma_j^2 &= \boldsymbol{\beta}_j^\top \mathbf{S}_{1:(j-1),1:(j-1),1:(j-1),1:(j-1)}\boldsymbol{\beta}_j + \lambda_{jj}^{-2} \\
&= \mathbf{S}_j^\top \mathbf{L}^\top_{1:(j-1),1:(j-1)}\mathbf{L}_{1:(j-1),1:(j-1)}\mathbf{S}_j + \lambda_{jj}^{-2} \\
\iff \quad \lambda_{jj} &= (\sigma_j^2 - \|\mathbf{L}_{1:(j-1),1:(j-1)}\mathbf{S}_j\|_2^2)^{-1/2}
\end{aligned}$$

and we can update $\mathbf{L}$ as

$$\mathbf{L}_{\mathbf{j},\mathbf{j}} = \begin{pmatrix} \mathbf{L}_{1:(j-1),1:(j-1)} & \boldsymbol{\beta}_j\lambda_{jj} \\ \boldsymbol{\beta}_j\lambda_{jj} & \lambda_{jj} \end{pmatrix}.$$

We proceed until $j = J$ and return $\mathbf{L}$.

⟨ *invchol syMatrices* 41 ⟩ ≡

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

invchol.default <- function(x, ...)
    invchol(as.syMatrices(as.matrix(x)))

invchol.syMatrices <- function(x, ...) {

    byrow_orig <- attr(x, "byrow")
    d <- dim(x)
    x <- .adddiag(x)

    ### x is of class syMatrices, coerse to ltMatrices first and re-arrange
    ### second
    class(x) <- gsub("syMatrices", "ltMatrices", class(x))
    x <- ltMatrices(x, byrow = TRUE)

    if (!is.double(x)) storage.mode(x) <- "double"

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

    ret <- ltMatrices(ret, byrow = byrow_orig)

    return(ret)
}
```

◇

Fragment referenced in 1.

The heavy lifting requires some C code

⟨ *invchol* 42 ⟩ ≡

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

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

    ans = PROTECT(duplicate(Sigma));
    dans = REAL(ans);

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

        C_invchol(iJ, dans, &info);

        if (info != 0) {
            if (info > 0)
                error("the leading minor of order %d is not positive definite",
                      info);
        }

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

◇

Fragment referenced in 2.

⟨ *invchol workhorse* 43 ⟩ ≡

```
void C_invchol (int J, double* ans, int* info) {

    int i, j, k;
    int start = 1, str;
    int end = 0;
    double sd = 0.0;
    double *sigma, *x;

    info[0] = 0;

    if (ans[0] < DBL_EPSILON) {
       info[0] = 1;
    } else {
        ans[0] = 1 / sqrt(ans[0]);

        for (j = 1; j < J; j++) {
            sigma = ans + start;
            sd = 0.0;
            /* compute L mS + its 2norm */
            /* <FIXME> use dtpmv <FIXME> */
            for (i = j - 1; i >= 0; i--) {
                sigma[i] *= ans[end];
                for (k = 1; k <= i; k++)
                    sigma[i] += ans[end - k] * sigma[i - k];
                sd += pow(sigma[i], 2);
                end -= i + 1;
            }
            /* compute lambda_jj^-1 */
            sd = sigma[j] - sd;
            if (sd < DBL_EPSILON) {
                info[0] = j;
                break;
            }
            sd = sqrt(sd);
            str = 0;
            /* compute L^\top (L mS) */
            /* <FIXME> use dtpmv <FIXME> */
            for (i = 0; i < j; i++) {
                x = ans + str + i;
                sigma[i] *= x[0];
                for (k = i + 1; k < j; k++) {
                    x += k;
                    sigma[i] += x[0] * sigma[k];
                }
                str += i + 1;
            }
            /* jth row of L */
            for (i = 0; i < j; i++)
                sigma[i] = - sigma[i] / sd;
            sigma[j] = 1 / sd;
            start += j + 1;
            end = start - 1;
        }
    }
}
```

◇

Fragment referenced in 2.

```
> Sigma <- tcrossprod(lxd)
> chk(invchol(Sigma), solve(lxd))
> Sigma <- tcrossprod(lxn)
> ## Sigma and chol(Sigma) always have diagonal, lxn doesn't
> chk(as.array(invchol(Sigma)), as.array(solve(lxn)))
> ### check singular matrices
> x <- matrix(runif(21), ncol = 7)
> S <- crossprod(x)
```

with meaningful error messages

```
> ## IGNORE_RDIFF_BEGIN

> print(try(solve(S), silent = TRUE))

[1] "Error in solve.default(S) : \n  system is computationally singular: reciprocal condition num
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in solve.default(S): system is computationally singular: reciprocal condition number

> print(try(invchol(S), silent = TRUE))

[1] "Error in invchol.syMatrices(as.syMatrices(as.matrix(x))) : \n  the leading minor of order 3
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in invchol.syMatrices(as.syMatrices(as.matrix(x))): the leading minor of order 3 is

> ## IGNORE_RDIFF_END
```

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 $\text{vec}(\mathbf{S})^\top(\mathbf{A}^\top \otimes \mathbf{C}) = \text{vec}(\mathbf{C}^\top\mathbf{S}\mathbf{A}^\top)^\top$. 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* 44 ⟩ ≡

```
C <- as.ltMatrices(C)
C_byrow_orig <- attr(C, "byrow")
dC <- dim(C)
nm <- attr(C, "rcnames")
N <- dC[1L]
J <- dC[2L]
◇
```

Fragment referenced in 47.

`S` can be an `ltMatrices` object, an `syMatrices` object, or a $J^2 \times N$ matrix featuring columns of vectorised $J \times J$, but not necessarily symmetric, matrices. In case `S` is missing, we interpret it as `diag(J)` later on

⟨ *check S argument* 45 ⟩ ≡

```
SltM <- is.ltMatrices(S)
if (SltM || is.syMatrices(S)) {
    S_byrow_orig <- attr(S, "byrow")
    stopifnot(S_byrow_orig == C_byrow_orig)
    dS <- dim(S)
    stopifnot(dC[2L] == dS[2L])
    if (dC[1] != 1L) {
        stopifnot(dC[1L] == dS[1L])
    } else {
        N <- dS[1L]
    }
    ### we'll use matrix-matrix multiplication in Mult
    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)
    }
}
```

◇

Fragment referenced in 47.

`A` is an `ltMatrices` object

⟨ *check A argument* 46 ⟩ ≡

```
if (missing(A)) {
    A <- C
} else {
    A <- as.ltMatrices(A)
    A_byrow_orig <- attr(A, "byrow")
    stopifnot(C_byrow_orig == A_byrow_orig)
    dA <- dim(A)
    stopifnot(dC[2L] == dA[2L])
}
```

◇

Fragment referenced in 47.

We put everything together in function `vectrick` by leveraging the matrix-matrix multiplication in `Mult`

⟨ *kronecker vec trick* 47 ⟩ ≡

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

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

    if (missing(S) && missing(A)) {
        if (transpose[1L] && !transpose[2L])
            return(crossprod(C)) ### t(C) %*% C
        if (!transpose[1L] && transpose[2L])
            return(tcrossprod(C)) ### C %*% t(C)
    }

    ⟨ check C argument 44 ⟩

    if (missing(S)) {  ### S = diag(J), interpreted as syMatrices obj
        SltM <- TRUE
        CS <- as.array(C)
        if (transpose[1L]) CS <- aperm(CS, perm = c(2L, 1L, 3L))
        CS <- matrix(CS, nrow = J^2)
    } else {
        ⟨ check S argument 45 ⟩
        ### CS = C %*% S or CS = t(C) %*% S
        CS <- Mult(C, S, transpose = transpose[1L])
    }

    ⟨ check A argument 46 ⟩

    ### CS %*% t(A) = t(A %*% t(CS)) or
    ### CS %*% A  = t(t(A) %*% t(CS))
    CS <- CS[TR <- c(t(L <- matrix(seq_len(J^2), nrow = J))),,drop = FALSE]
    ret <- Mult(A, CS, transpose = !transpose[2L])[TR,,drop = FALSE]

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

    ret <- ltMatrices(ret[L[.lt(J, diag = TRUE)],,drop = FALSE],
                      diag = TRUE, byrow = FALSE, names = nm)
    ret <- ltMatrices(ret, byrow = attr(C, "byrow"))
    return(ret)
}
```

◇

Fragment referenced in 1.

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* 48 ⟩ ≡

```
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 51.

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

⟨ *D times C* 49 ⟩ ≡

```
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)

    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$double.eps
    if (any(D > 1 / .Machine$double.eps))
        D[D > 1 / .Machine$double.eps] <-  (1 / .Machine$double.eps) / 2

    if (attr(x, "byrow")) {
       idx <- rep(1:J, 1:J)
    } else {
       idx <- do.call("c", sapply(seq_len(J),
           function(j) seq(from = j, to = J, by = 1L)))
    }
    x[] <- unclass(x) * D[idx,,drop = FALSE]

    ret <- as.chol(x)
    return(ret)
}
```

◇

Fragment referenced in 51.

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* 50 ⟩ ≡

```
### 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)

    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$double.eps
    if (any(D > 1 / .Machine$double.eps))
        D[D > 1 / .Machine$double.eps] <-  (1 / .Machine$double.eps) / 2

    if (attr(x, "byrow")) {
       idx <- do.call("c",sapply(seq_len(J), function(j) seq_len(j)))
    } else {
       idx <- rep(1:J, J:1)
    }
    x[] <- unclass(x) * D[idx,,drop = FALSE]

    ret <- as.invchol(x)
    return(ret)
}
```

◇

Fragment referenced in 51.

and now the convenience functions are one-liners:

⟨ *convenience functions* 51 ⟩ ≡

⟨ *chol classes* 48 ⟩
⟨ *D times C* 49 ⟩
⟨ *L times D* 50 ⟩

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

### Sigma -> C
cov2chol <- function(x)
    as.chol(chol(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))

### Sigma -> L
cov2invchol <- function(x)
    as.invchol(invchol(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 1.

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 `perm` generic

⟨ *perm generic* 52 ⟩ ≡

```
perm <- function(a, perm, ...)
    UseMethod("perm")
```
◇

Fragment referenced in 55.

and a 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 `perm`, 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

⟨ *perm checks* 53 ⟩ ≡

```
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 54.

and then implement the two methods

⟨ *perm chol invchol* 54 ⟩ ≡

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

    ⟨ perm checks 53 ⟩

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

    ⟨ perm checks 53 ⟩

    return(as.invchol(invchol(invchol2cov(a)[,perm])))
}
```

◇

Fragment referenced in 55.

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

⟨ *perm* 55 ⟩ ≡

```
⟨ perm generic 52 ⟩
⟨ perm chol invchol 54 ⟩

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

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

◇

Fragment referenced in 1.

Chapter 3

Marginal and Conditional Normal Distributions

Marginal and conditional distributions from distributions $\mathbf{Y}_i \sim \mathbb{N}_J\left(\boldsymbol{\mu}_i, \mathbf{C}_i\mathbf{C}_i^\top\right)$ (`chol` argument for $\mathbf{C}_i$ for $i = 1, \ldots, N$) or $\mathbf{Y}_i \sim \mathbb{N}_J\left(\boldsymbol{\mu}_i, \mathbf{L}_i^{-1}\mathbf{L}_i^{-\top}\right)$ (`invchol` argument for $\mathbf{L}_i$ for $i = 1, \ldots, N$) shall be computed, where $\boldsymbol{\mu}_i$ is either the mean vector or is given by $\boldsymbol{\mu}_i = \mathbf{L}_i^{-1}\boldsymbol{\nu}_i = \mathbf{C}_i\boldsymbol{\nu}_i$.

The different parameterisations are useful in different contexts. Dropping the subscript i for notational convenience, we can write the marginal mean vector and the covariance matrix as

$$\begin{aligned}\mathbb{E}(\mathbf{Y}) &= \boldsymbol{\mu} \\ \mathbb{V}(\mathbf{Y}) &= \boldsymbol{\Sigma} = \mathbf{C}\mathbf{C}^\top = \mathbf{L}^{-1}\mathbf{L}^{-\top}.\end{aligned}$$

Of course, $Y_j \sim \mathbb{N}_1\left(\mu_j, \sigma_j^2\right)$, where σ_j^2 is the jth diagonal element of $\boldsymbol{\Sigma}$. In contrast to this "marginal parameterisation" where parameters have a direct interpretation in terms of univariate marginal distributions, a "conditional parameterisation" in terms of $\boldsymbol{\nu}$ and $\mathbf{L}$ is helpful when one is interested in the conditional distributions

$$Y_j \mid Y_1 = y_1, \ldots, Y_{j-1} = y_{j-1} \sim \mathbb{N}_1\left(\mathbf{L}_{jj}^{-1}\nu_j - \sum_{j=1}^{j-1}\mathbf{L}_{jj}^{-1}\mathbf{L}_{jj}y_j, \mathbf{L}_{jj}^{-2}\right)$$

for each $j = 2, \ldots, J$, for a proof see (3.3). Here, the intercept is given by $\mathbf{L}_{jj}^{-1}\nu_j$ and the regression coefficients by $\mathbf{L}_{jj}^{-1}\mathbf{L}_{jj}$. Based on these parameters, the joint density is given as

$$\begin{aligned}f(\mathbf{y} \mid \boldsymbol{\nu}, \mathbf{L}) &= (2\pi)^{-\frac{J}{2}} \det(\mathbf{L}) \exp\left(-\frac{1}{2}\|\mathbf{L}\mathbf{y} - \boldsymbol{\nu}\|_2^2\right) \\ &= (2\pi)^{-\frac{J}{2}} \prod_{j=1}^{J} \mathbf{L}_{jj} \exp\left(-\frac{1}{2}\|\mathbf{L}\mathbf{y} - \boldsymbol{\nu}\|_2^2\right) \\ &= f(y_1 \mid \nu_1, \mathbf{L}_{11}) \times f(y_2|y_1, \nu_2, \mathbf{L}_{2\cdot}) \times \cdots \times f(y_J \mid y_1, \ldots, y_{J-1}, \nu_J, \mathbf{L}_{J\cdot})\end{aligned} \tag{3.1}$$

which nicely decomposes into a product of conditional densities.

Before we begin, we implement three convenience functions for extracting marginal means from any parameterisation using infrastructure later to be developed in Chapter 9. We start with marginal means

⟨ *mvnorm marginal means* 56 ⟩ ≡

```
mean.mvnorm <- function(x, ...) {
    if (!is.null(x$mean)) return(x$mean)
    if (!is.null(x$invcholmean)) return(.i2m(x))
    return(0)
}
```

◇

Fragment referenced in 166.

and covariances

⟨ *mvnorm vcov* 57 ⟩ ≡

```
vcov.mvnorm <- function(object, ...) {
    if (is.invchol(object$scale)) return(invchol2cov(object$scale))
    return(chol2cov(object$scale))
}
```

◇

Fragment referenced in 166.

and also add a function for computing intercepts, regression coefficients and residual standard errors

⟨ *mvnorm regression coefs* 58 ⟩ ≡

```
coef.mvnorm <- function(object, which = dim(object$scale)[2L], ...) {

    if (dim(object$scale)[[2L]] == 1) {
        if (!is.invchol(object$scale)) object$scale <- chol2invchol(object$scale)
        if (!is.null(object$invcholmean)) object$mean <- mean(object)
        if (is.null(object$mean)) return(NULL)
        ret <- object$mean
        attr(ret, "sigma") <- 1 / c(unclass(object$scale))
        return(ret)
    }

    if (is.character(which))
        which <- which(dimnames(object$scale)[[2L]] == which)

    stopifnot((length(which) == 1L) &&
              (which > 1) &&
              (which <= dim(object$scale)[2L]))

    if (!is.invchol(object$scale)) object$scale <- chol2invchol(object$scale)

    diag <- attr(object$scale, "diag")
    j <- which - 1L
    start <- j * (j + 1) / 2
    idx <- start + seq_len(which)
    ret <- Lower_tri(ltMatrices(object$scale, byrow = TRUE, diag = diag),
                     diag = diag)[idx,,drop = FALSE]
    rownames(ret) <- dimnames(object$scale)[[2L]][seq_len(which)]
    if (!is.null(nm <- dimnames(object$scale)[[1L]]))
        colnames(ret) <- nm
    dg <- ret[nrow(ret),]
    cf <- - ret[-nrow(ret),,drop = FALSE]
    if (!is.null(object$mean)) object$invcholmean <- .m2i(object)
    if (!is.null(object$invcholmean))
        cf <- rbind("(Intercept)" = object$invcholmean[which,,drop = TRUE], cf)
    cf <- cf / matrix(dg, nrow = nrow(cf), ncol = ncol(cf), byrow = TRUE)
    cf <- cf[,,drop = TRUE]
    attr(cf, "sigma") <- 1 / dg
    return(cf)
}
```

◇

Fragment referenced in 166.

We begin computations for marginal and conditional distributions with some simple argument checks

⟨ *mc input checks* 59 ⟩ ≡

```
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 60, 63.

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* 60 ⟩ ≡

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

    ⟨ mc input checks 59 ⟩

    if (which[1] == 1L && (length(which) == 1L ||
                           all(diff(which) == 1L))) {
        ### which is 1:j
        tmp <- x[,which]
    } else {
        if (missing(chol)) {
            cv <- invchol2cov(x)
            tmp <- cov2invchol(cv[,which])
        } else {
            cv <- chol2cov(x)
            tmp <- cov2chol(cv[,which])
        }
    }

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

    ret
}
```

◇

Fragment referenced in 1.

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

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

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* 61 ⟩ ≡

```
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(A <- base::chol(Pw)) ### Pw = A A^\top
### arg invchol is missing, masking mvtnorm::invchol
### which we can not access bc R CMD check is not happy about it
chol <- getFromNamespace("invchol", "mvtnorm")(Pw) ### Pw = A A^\top
g0 <- matrix(0, nrow = J, ncol = NCOL(given))
g0[which,] <- given
S <- Crossprod(chol) ### P^{-1}_jj = A^-top A^-1
if (ncol(g0) %in% c(1L, N)) {
    mean <- - S %*% (P %*% g0)[-which,,drop = FALSE]
} else { ### compare to Mult() with ncol(y) !%in% (1, N)
    ### <FIXME> check dimension and if t() is needed
    mean <- t(sapply(seq_len(ncol(g0)), function(i)
        - S %*% (P %*% g0[,i])[-which,,drop = FALSE]))
}
```

◇

Fragment referenced in 63.

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

$$\mathbf{L} = \quad = \quad \begin{pmatrix} \mathbf{L}_{\mathbf{j},\mathbf{j}} & 0 \\ \mathbf{L}_{-\mathbf{j},\mathbf{j}} & \mathbf{L}_{-\mathbf{j},-\mathbf{j}} \end{pmatrix}$$

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

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

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

$$\mathbf{L}_{-\mathbf{j},\mathbf{j}}(\mathbf{y}_\mathbf{j} - \boldsymbol{\mu}_\mathbf{j}) \tag{3.2}$$

and the `center = TRUE` argument triggers this values to be returned.

An alternative derivation of this result is based on (3.1), the ratio of which and the density of the marginal distribution of $\mathbf{Y_j} \sim \mathbb{N}_j \left(\mathbf{L_j}^{-1} \boldsymbol{\nu}_\mathbf{j}, \mathbf{L_j}^{-1} \mathbf{L_j}^{-\top} \right)$ is

$$\begin{aligned} f(\mathbf{y_{-j}} \mid \mathbf{y_j}, \boldsymbol{\nu}, \mathbf{L}) = \frac{f(\mathbf{y} \mid \boldsymbol{\nu}, \mathbf{L})}{f(\mathbf{y_j} \mid \boldsymbol{\nu}_\mathbf{j}, \mathbf{L_{j,j}})} &= \frac{(2\pi)^{-\frac{J}{2}} \prod_{j=1}^{J} \mathbf{L}_{jj} \exp\left(-\frac{1}{2} \|\mathbf{L}\mathbf{y} - \boldsymbol{\nu}\|_2^2\right)}{(2\pi)^{-\frac{j}{2}} \prod_{j=1}^{j} \mathbf{L}_{jj} \exp\left(-\frac{1}{2} \|\mathbf{L_j}\mathbf{y_j} - \boldsymbol{\nu}_\mathbf{j}\|_2^2\right)} \\ &= (2\pi)^{-\frac{J-j}{2}} \prod_{j=j+1}^{J} \mathbf{L}_{jj} \exp\left(-\frac{1}{2} (\|\mathbf{L}\mathbf{y} - \boldsymbol{\nu}\|_2^2 - \|\mathbf{L_j}\mathbf{y_j} - \boldsymbol{\nu}_\mathbf{j}\|_2^2) \right) \\ &= (2\pi)^{-\frac{J-j}{2}} \prod_{j=j+1}^{J} \mathbf{L}_{jj} \exp\left(\|\mathbf{L_{-j,-j}}\mathbf{y_{-j}} - (\boldsymbol{\nu}_\mathbf{-j} - \mathbf{L_{-j,j}}\mathbf{y_j})\|_2^2 \right) \end{aligned}$$

which corresponds to the distribution

$$\mathbf{Y_{-j}} \mid \mathbf{y_j} \sim \mathbb{N}_{J-j} \left(\mathbf{L}^{-1}_{\mathbf{-j,-j}} (\boldsymbol{\nu}_\mathbf{-j} - \mathbf{L_{-j,j}}\mathbf{y_j}), \mathbf{L}^{-1}_{\mathbf{-j,-j}} \mathbf{L}^{-\top}_{\mathbf{-j,-j}} \right). \tag{3.3}$$

Noting that $\boldsymbol{\nu} = \mathbf{L}\boldsymbol{\mu}$ and thus $\boldsymbol{\nu}_\mathbf{-j} = \mathbf{L_{-j,-j}}\boldsymbol{\mu}_\mathbf{j} + \mathbf{L_{-j,j}}\boldsymbol{\mu}_\mathbf{-j}$, we obtain for the conditional mean

$$\begin{aligned} & \mathbf{L}^{-1}_{\mathbf{-j,-j}} (\boldsymbol{\nu}_\mathbf{-j} - \mathbf{L_{-j,j}}\mathbf{y_j}) \\ = \; & \mathbf{L}^{-1}_{\mathbf{-j,-j}} (\mathbf{L_{-j,-j}}\boldsymbol{\mu}_\mathbf{j} + \mathbf{L_{-j,j}}\boldsymbol{\mu}_\mathbf{-j} - \mathbf{L_{-j,j}}\mathbf{y_j}) \\ = \; & \boldsymbol{\mu}_\mathbf{j} + \mathbf{L}^{-1}_{\mathbf{-j,-j}} \mathbf{L_{-j,j}} \boldsymbol{\mu}_\mathbf{-j} - \mathbf{L}^{-1}_{\mathbf{-j,-j}} \mathbf{L_{-j,j}} \mathbf{y_j} \\ = \; & \boldsymbol{\mu}_\mathbf{j} - \mathbf{L}^{-1}_{\mathbf{-j,-j}} \mathbf{L_{-j,j}} (\mathbf{y_j} - \boldsymbol{\mu}_\mathbf{j}) \end{aligned}$$

The implementation reads

⟨ *cond simple* 62 ⟩ ≡

```
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 63.

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* 63 ⟩ ≡

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

    which <- which_given
    ⟨ mc input checks 59 ⟩

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

    ⟨ cond simple 62 ⟩

    ### 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 = perm(as.chol(chol), perm = perm),
                           which_given = 1:length(which), given = given,
                           center = center))
        return(cond_mvnorm(invchol = perm(as.invchol(invchol), perm = perm),
                           which_given = 1:length(which), given = given,
                           center = center))
    }

    ⟨ cond general 61 ⟩

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

    return(list(mean = mean, invchol = invchol(S)))
}
```

◇

Fragment referenced in 1.

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
> colnames(cm) <- dimnames(lxd)[[1L]][1L]
> 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_given = 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
> colnames(cm) <- dimnames(lxd)[[1L]][1L]
> 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_given = 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
> colnames(cm) <- dimnames(lxd)[[1L]][1L]
> 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_given = 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
> colnames(cm) <- dimnames(lxd)[[1L]][1L]
> 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_given = j, given = y)
> chk(c(cm), c(cmv$mean))
> chk(cS, as.array(tcrossprod(solve(cmv$invchol)))[,,1])
```

3.1 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* 64 ⟩ ≡

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

    stopifnot(xor(missing(chol), missing(invchol)))
    if (missing(mean) && missing(invcholmean)) invcholmean <- 0
    stopifnot(xor(missing(mean), missing(invcholmean)))
    if (!is.matrix(obs)) obs <- matrix(obs, ncol = 1L)
    p <- ncol(obs)

    if (!missing(chol)) {
        ⟨ ldmvnorm chol 68 ⟩
    } else {
        ⟨ ldmvnorm invchol 69 ⟩
    }

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

◇

Fragment referenced in 72.

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` and return the mean-centered observations. Alternatively, when the scaled mean $\boldsymbol{\nu}_i = \mathbf{L}_i \boldsymbol{\mu}_i$ (`invcholmean`) is given, we only check the dimensions.

⟨ *check obs* 65 ⟩ ≡

```
.check_obs_mean <- 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)
}
.check_obs_invcholmean <- function(obs, invcholmean, 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 (is.null(invcholmean))
        return(matrix(0, nrow = J, ncol = N))
    if (identical(unique(invcholmean), 0))
        return(matrix(0, nrow = J, ncol = N))

    if (!is.matrix(invcholmean))
        invcholmean <- matrix(invcholmean, nrow = J)
    nr <- nrow(invcholmean)
    nc <- ncol(invcholmean)
    if (!(nc %in% c(1L, N)))
        stop("obs and (inv)chol have non-conforming size")
    if (nr != J)
        stop("invcholmean and (inv)chol have non-conforming size")

    if (ncol(invcholmean) == N)
        return(invcholmean)
    return(matrix(invcholmean, nrow = J, ncol = N))
}
```

◇

Fragment referenced in 1.

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

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

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

$$\ell_i(\boldsymbol{\mu}_i, \mathbf{C}_i) \quad = \quad -\frac{J}{2} \log(2\pi) - \sum_{j=1}^J \log \operatorname{diag}(\mathbf{C}_i)_j - \frac{1}{2} (\mathbf{y}_i - \boldsymbol{\mu}_i)^\top \mathbf{C}_i^{-\top} \mathbf{C}_i^{-1} (\mathbf{y}_i - \boldsymbol{\mu}_i). \tag{3.4}$$

With $\boldsymbol{\nu}_i = \mathbf{C}_i^{-1} \boldsymbol{\mu}_i$ (argument `invcholmean`) given, we get

$$\ell_i(\boldsymbol{\nu}_i, \mathbf{C}_i) \quad = \quad -\frac{J}{2} \log(2\pi) - \sum_{j=1}^J \log \operatorname{diag}(\mathbf{C}_i)_j - \frac{1}{2} (\mathbf{C}_i^{-1} \mathbf{y}_i - \boldsymbol{\nu}_i)^\top (\mathbf{C}_i^{-1} \mathbf{y}_i - \boldsymbol{\nu}_i). \tag{3.5}$$

The most attractive expression however is based on the parameterisation using $\mathbf{L}_i = \mathbf{C}_i^{-1}$ and $\boldsymbol{\nu}_i$ because we get

$$\ell_i(\boldsymbol{\nu}_i, \mathbf{L}_i) \quad = \quad -\frac{J}{2} \log(2\pi) + \sum_{j=1}^J \log \operatorname{diag}(\mathbf{L}_i)_j - \frac{1}{2} (\mathbf{L}_i \mathbf{y}_i - \boldsymbol{\nu}_i)^\top (\mathbf{L}_i \mathbf{y}_i - \boldsymbol{\nu}_i) \tag{3.6}$$

which is simultaneously log-concave in both parameters (Barratt and Boyd, 2023).

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* 66 ⟩ ≡

```
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 2.

⟨ *colSumsdnorm ltMatrices* 67 ⟩ ≡

```
.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 1.

The main part is now

⟨ *ldmvnorm chol* 68 ⟩ ≡

```
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]
if (missing(invcholmean)) {
    ## NOTE: obs is now mean-centered
    obs <- .check_obs_mean(obs = obs, mean = mean, J = J, N = N)
    z <- solve(chol, obs)
} else {
    invcholmean <- .check_obs_invcholmean(obs = obs, invcholmean = invcholmean,
                                          J = J, N = N)
    z <- solve(chol, obs) - invcholmean
}
logretval <- .colSumsdnorm(z)
if (attr(chol, "diag"))
    logretval <- logretval - logdet(chol)
```

◇

Fragment referenced in 64.

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

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

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

⟨ *ldmvnorm invchol* 69 ⟩ ≡

```
## 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]
if (missing(invcholmean)) {
    ## NOTE: obs is now mean-centered
    obs <- .check_obs_mean(obs = obs, mean = mean, J = J, N = N)
    z <- Mult(invchol, obs)
} else {
    invcholmean <- .check_obs_invcholmean(obs = obs, invcholmean = invcholmean,
                                          J = J, N = N)
    z <- Mult(invchol, obs) - invcholmean
}
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 64.

The score function with respect to `obs` is

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

with respect to the scaled mean $\boldsymbol{\nu}_i$ we get

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

and with respect to `invchol` we have

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

The score function with respect to `chol` post-processes the above score using the vec-trick (Section 2.11). For the log-likelihood (3.4), 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* 70 ⟩ ≡

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

    stopifnot(xor(missing(chol), missing(invchol)))
    if (missing(mean) && missing(invcholmean)) invcholmean <- 0
    stopifnot(xor(missing(mean), missing(invcholmean)))
    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]
        if (missing(invcholmean)) {
            obs <- .check_obs_mean(obs = obs, mean = mean, J = J, N = N)
            ## NOTE: obs is mean-centered now
            Mix <- Mult(invchol, obs)
        } else {
            invcholmean <- .check_obs_invcholmean(obs = obs, invcholmean = invcholmean,
                                                  J = J, N = N)
            Mix <- Mult(invchol, obs) - invcholmean
        }
        sobs <- - Mult(invchol, Mix, transpose = TRUE)

        ### score wrt invchol
        idx <- matrix(1:J, nrow = J, ncol = J)
        idx <- idx[.lt(J, diag  = attr(invchol, "diag"))]
        j <- seq_len(J - !(attr(invchol, "diag") + 0L))
        idx2 <- rep(j, rev(j))
        ret <- - obs[idx2,,drop = FALSE] * Mix[idx,,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
        }
        if (missing(invcholmean)) {
            ret <- list(obs = sobs, invchol = ret)
        } else {
            ret <- list(obs = sobs, invcholmean = Mix, invchol = ret)
        }
        if (logLik)
            ### note: obs is obs - mean if mean was present
            ret$logLik <- ldmvnorm(obs = obs, invcholmean = invcholmean,
                                   invchol = invchol, logLik = FALSE)
        return(ret)
    }

    invchol <- solve(chol)
    ret <- sldmvnorm(obs = obs, mean = mean, invcholmean = invcholmean,
                     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 72.

3.2 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(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_given = j, given = Y[j,])
> ll3 <- sum(dnorm(md$invchol %*% Y[j,], log = TRUE)) +
+        sum(log(diagonals(md$invchol))) +
+        sum(dnorm(cd$invchol %*% (Y[-j,] - cd$mean), log = TRUE)) +
+        sum(log(diagonals(cd$invchol)))
> chk(ll1, ll3)
```

Chapter 4

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* 71 ⟩ ≡

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

    ### only needed for input checks
    stopifnot(missing(invcholmean))

    ### get access to internal mvtnorm functions
    .check_obs_mean <- mvtnorm:::.check_obs_mean

    ⟨ input checks 74 ⟩

    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 $\mathbf{C}_i$.

4.1 Algorithm

"lpmvnorm.R" 72≡

⟨ *R Header* 201 ⟩
⟨ *lpmvnorm* 99 ⟩
⟨ *slpmvnorm* 127 ⟩
⟨ *ldmvnorm* 64 ⟩
⟨ *sldmvnorm* 70 ⟩
⟨ *ldpmvnorm* 156 ⟩
⟨ *sldpmvnorm* 158 ⟩
⟨ *deperma* 163 ⟩
⟨ *standardize* 164 ⟩
⟨ *destandardize* 165 ⟩
⟨ *RR lognormdiff (AI)* 195 ⟩
⟨ *lpRR* 197 ⟩
⟨ *slpRR* 200 ⟩
◇

"lpmvnorm.c" 73≡

```
⟨ C Header 202 ⟩
#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>
⟨ lognormdiff (log scale, AI) 129 ⟩
⟨ R lpmvnorm (log scale, AI) 140 ⟩
⟨ R slpmvnorm (log scale, AI) 153 ⟩
◇
```

Sections 4.1 and 4.2 document the original probability-scale implementation of Genz (1992) for reference (so its top-level fragments are marked "Fragment never referenced" by `nuweb`). The compiled package uses the log-scale implementation presented in Section 4.3 instead.

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* 74 ⟩ ≡

```
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 (!missing(mean)) {
    lower <- .check_obs_mean(lower, mean, J = J, N = N)
    upper <- .check_obs_mean(upper, mean, J = J, N = N)
}

if (!missing(invcholmean)) {
    invcholmean <- .check_obs_invcholmean(lower, invcholmean, J = J, N = N)
    center <- - invcholmean
}

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

◇

Fragment referenced in 71, 99, 127.

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* 75 ⟩ ≡

```
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 99, 127.

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

   Here, `center` is the negative rescaled mean (3.2) allowing to evaluate probabilities for conditional multivariate normal distributions in `lpmvnorm` and corresponding scores in `slpmvnorm`. This feature is not present in the orginal formulation of the algorithm.

⟨ *initialisation* 76 ⟩ ≡

```
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 95, 118.

4. Repeat

⟨ *init logLik loop* 77 ⟩ ≡

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

◇

Fragment referenced in 95, 107.

(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* 78 ⟩ ≡

```
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 82, 115.

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

⟨ *compute x* 79 ⟩ ≡

```
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 82, 115, 135, 150.

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

⟨ *update d, e* 80 ⟩ ≡

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

◇

Fragment referenced in 82, 115.

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

⟨ *update f* 81 ⟩ ≡

```
start += j;
f *= emd;
```
◇

Fragment referenced in 82, 115.

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

⟨ *inner logLik loop* 82 ⟩ ≡

```
for (j = 1; j < iJ; j++) {
    ⟨ compute y 78 ⟩
    ⟨ compute x 79 ⟩
    ⟨ update d, e 80 ⟩
    ⟨ update f 81 ⟩
}
```
◇

Fragment referenced in 95.

(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* 83 ⟩ ≡

```
intsum += f;
```
◇

Fragment referenced in 95.

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* 84 ⟩ ≡

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

Fragment referenced in 95.

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

⟨ *move on* 85 ⟩ ≡

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

Fragment referenced in 95, 118, 140, 153.

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* 86 ⟩ ≡

```
/* 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 never referenced.

⟨ *pnorm slow* 87 ⟩ ≡

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

Fragment never referenced.

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

⟨ *pnorm* 88 ⟩ ≡

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

Fragment referenced in 95, 118.

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* 89 ⟩ ≡

```
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 95, 118, 140, 153.

⟨ *dimensions* 90 ⟩ ≡

```
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 95, 118, 140, 153.

⟨ *setup return object* 91 ⟩ ≡

```
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 95, 140.

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

⟨ *univariate problem* 92 ⟩ ≡

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

Fragment referenced in 95.

⟨ *init center* 93 ⟩ ≡

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

Fragment referenced in 95, 118, 140, 153.

We put the code together in a dedicated C function

⟨ *R slpmvnorm variables* 94 ⟩ ≡

```
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 95, 118.

⟨ *R lpmvnorm* 95 ⟩ ≡

```
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 94 ⟩
    double lO, lM, x0, intsum;
    int p, len;

    Rboolean RlogLik = asLogical(logLik);

    ⟨ pnorm 88 ⟩
    ⟨ dimensions 90 ⟩
    ⟨ W length 89 ⟩
    ⟨ init center 93 ⟩

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

    ⟨ setup return object 91 ⟩

    qO = qnorm(dtol, 0.0, 1.0, 1L, 0L);
    lO = log(dtol);

    ⟨ univariate problem 92 ⟩

    if (W == R_NilValue)
        GetRNGstate();

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

        x0 = 0;
        ⟨ initialisation 76 ⟩

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

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

            ⟨ init logLik loop 77 ⟩
            ⟨ inner logLik loop 82 ⟩
            ⟨ increment 83 ⟩

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

        ⟨ output 84 ⟩
        ⟨ move on 85 ⟩
    }

    if (W == R_NilValue)
        PutRNGstate();

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

◇

Fragment never referenced.

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* 96 ⟩ ≡

```
### 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 99, 127, 175.

⟨ *check and / or set integration weights* 97 ⟩ ≡

```
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 99, 127.

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* 98 ⟩ ≡

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

◇

Fragment referenced in 99, 127.

FIXME: `center` is obsolate as `center = x` gives the same result as `invcholmean = - x`. Also in `slpmvnorm`.

⟨ *lpmvnorm* 99 ⟩ ≡

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

    ⟨ tol+fast 128 ⟩
    ⟨ init random seed, reset on exit 96 ⟩
    ⟨ Cholesky of precision 98 ⟩
    ⟨ input checks 74 ⟩
    ⟨ standardise 75 ⟩
    ⟨ check and / or set integration weights 97 ⟩

    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 72.

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))

 [1] 0.2367 0.2341 0.2835 0.3939 0.4658 0.0000 0.5911 0.4598 0.4879 0.0000
```

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
+     ### byrow = TRUE because adding / removing dimensions
+     ### keeps the MC points for the remaining dimensions constant
+     W <- matrix(runif(M * (J - 1)), nrow = J - 1, byrow = TRUE)
+ }
> ### 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
> pGq <- exp(lpmvnorm(a, b, chol = lx, w = W, M = M, logLik = FALSE))
> ### Genz 1992, original Monte-Carlo
> pG <- exp(lpmvnorm(a, b, chol = lx, w = NULL, M = M, logLik = FALSE))
> cbind(pGB, pGq, pG)
```

```
         pGB    pGq     pG
 [1,] 0.2369 0.2369 0.2360
 [2,] 0.2342 0.2342 0.2349
 [3,] 0.2841 0.2841 0.2826
 [4,] 0.3919 0.3914 0.3981
 [5,] 0.4675 0.4666 0.4685
 [6,] 0.0000 0.0000 0.0000
 [7,] 0.5902 0.5901 0.5903
 [8,] 0.4623 0.4618 0.4610
 [9,] 0.4872 0.4875 0.4864
[10,] 0.0000 0.0000 0.0000
```

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
```

All results are identical.

4.2 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* 100 ⟩ ≡

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

◇

Fragment referenced in 104.

and the derivates with respect to the mean

⟨ *mean scores* 101 ⟩ ≡

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

◇

Fragment referenced in 104.

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

⟨ *lower scores* 102 ⟩ ≡

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

◇

Fragment referenced in 104.

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

⟨ *upper scores* 103 ⟩ ≡

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

◇

Fragment referenced in 104.

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* 104 ⟩ ≡

```
int Jp = iJ * (iJ + 1) / 2;
⟨ chol scores 100 ⟩
⟨ mean scores 101 ⟩
⟨ lower scores 102 ⟩
⟨ upper scores 103 ⟩
double dtmp, ndtmp, etmp, netmp, 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 118, 153.

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

1. Input $\mathbf{C}_i$ (`chol`), $\mathbf{a}_i$ (`lower`), $\mathbf{b}_i$ (`upper`), and control parameters α, ϵ, and $M_{\max}$ (`M`).
2. Standardise integration limits $a_j^{(i)}/c_{jj}^{(i)}$, $b_j^{(i)}/c_{jj}^{(i)}$, and rows $c_{jj}^{(i)}/c_{jj}^{(i)}$ for $1 \leq \jmath < j < J$.

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

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

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

   ⟨ *score c11* 105 ⟩ ≡

```
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 107, 118.

   ⟨ *score a, b* 106 ⟩ ≡

```
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 107, 118.

4. Repeat

   ⟨ *init score loop* 107 ⟩ ≡

   ⟨ *init logLik loop* 77 ⟩
   ⟨ *score c11* 105 ⟩
   ⟨ *score a, b* 106 ⟩
   ◇

   Fragment referenced in 118.

   (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* 108 ⟩ ≡

```
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 115.

⟨ *update yp for means, lower and upper* 109 ⟩ ≡

```
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 115, 150.

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

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

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

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

⟨ *score wrt new chol off-diagonals* 110 ⟩ ≡

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

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

◇

Fragment referenced in 115.

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

⟨ *score wrt new chol diagonal* 111 ⟩ ≡

```
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 115.

⟨ *new score means, lower and upper* 112 ⟩ ≡

```
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 115.

We next update scores for parameters introduced for smaller j

⟨ *update score for chol* 113 ⟩ ≡

```
for (idx = 0; idx < j * (j + 1) / 2; idx++) {
    idxJ1 = idx * (iJ - 1);
    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_c[idxJ1 + k];

    dp_c[idx] = ndtmp * xx;
    ep_c[idx] = netmp * xx;
    fp_c[idx] = (ep_c[idx] - dp_c[idx]) * f + emd * fp_c[idx];
}
```

◇

Fragment referenced in 115.

⟨ *update score means, lower and upper* 114 ⟩ ≡

```
for (idx = 0; idx < j; idx++) {
    idxJ1 = idx * (iJ - 1);
    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_m[idxJ1 + k];

    dp_m[idx] = ndtmp * xx;
    ep_m[idx] = netmp * xx;
    fp_m[idx] = (ep_m[idx] - dp_m[idx]) * f + emd * fp_m[idx];

    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_l[idxJ1 + k];

    dp_l[idx] = ndtmp * xx;
    dp_u[idx] = netmp * xx;
    fp_l[idx] = (dp_u[idx] - dp_l[idx]) * f + emd * fp_l[idx];

    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_u[idxJ1 + k];

    ep_l[idx] = ndtmp * xx;
    ep_u[idx] = netmp * xx;
    fp_u[idx] = (ep_u[idx] - ep_l[idx]) * f + emd * fp_u[idx];
}
```

◇

Fragment referenced in 115.

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

⟨ *inner score loop* 115 ⟩ ≡

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

    ⟨ compute y 78 ⟩
    ⟨ compute x 79 ⟩
    ⟨ update d, e 80 ⟩
    ⟨ update yp for chol 108 ⟩
    ⟨ update yp for means, lower and upper 109 ⟩
    ⟨ score wrt new chol off-diagonals 110 ⟩
    ⟨ score wrt new chol diagonal 111 ⟩
    ⟨ new score means, lower and upper 112 ⟩
    ⟨ update score for chol 113 ⟩
    ⟨ update score means, lower and upper 114 ⟩
    ⟨ update f 81 ⟩

}
```

◇

Fragment referenced in 118.

(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* 116 ⟩ ≡

```
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 118.

⟨ *init dans* 117 ⟩ ≡

```
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 118.

We put everything together in C

⟨ *R slpmvnorm* 118 ⟩ ≡

```
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 94 ⟩
    double intsum;
    int p, idx, idxJ1;
    ⟨ dimensions 90 ⟩
    ⟨ pnorm 88 ⟩
    ⟨ W length 89 ⟩
    ⟨ init center 93 ⟩
    int start, j, k;
    double tmp, e, d, f, emd, x, x0, y[(iJ > 1 ? iJ - 1 : 1)];

    ⟨ score output object 104 ⟩

    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 76 ⟩
        ⟨ score c11 105 ⟩
        ⟨ score a, b 106 ⟩
        ⟨ init dans 117 ⟩

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

        for (int m = 0; m < iM; m++) {
            ⟨ init score loop 107 ⟩
            ⟨ inner score loop 115 ⟩
            ⟨ score output 116 ⟩
            if (W != R_NilValue)
                dW += iJ - 1;
        }

        ⟨ move on 85 ⟩
        dans += Jp + 1 + 3 * iJ;
    }

    if (W == R_NilValue)
        PutRNGstate();

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

◇

Fragment never referenced.

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* 119 ⟩ ≡

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

◇

Fragment referenced in 127.

⟨ *post differentiate lower score* 120 ⟩ ≡

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

◇

Fragment referenced in 127.

⟨ *post differentiate upper score* 121 ⟩ ≡

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

◇

Fragment referenced in 127.

⟨ *post differentiate chol score* 122 ⟩ ≡

```
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 127.

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* 123 ⟩ ≡

```
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 127.

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* 124 ⟩ ≡

```
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 127.

If the model was parameterised in terms of $\boldsymbol{\nu}_i = \mathbf{C}_i^{-1}\boldsymbol{\mu}_i$, we compute the score for $\boldsymbol{\nu}_i$ based on the score for $\boldsymbol{\mu}_i$

⟨ *invcholmean score* 125 ⟩ ≡

```
if (!missing(invcholmean))
    sinvcholmean <- Mult(chol, smean, transpose = TRUE)
```

◇

Fragment referenced in 127.

On the probability scale, `R_slpmvnorm` returns plain sums over the M integration points, which are normalised to log-likelihood contributions and scores of the log-likelihood (the compiled package uses the log-scale version from Section 4.3 instead):

⟨ *R normalise scores* 126 ⟩ ≡

```
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 have happened for observations with
### probability < tol; the log-lik is constant in these cases
### and thus the derivative = 0
ret[m < tol] <- 0
```

◇

Fragment never referenced.

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

⟨ *slpmvnorm* 127 ⟩ ≡

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

    ⟨ tol+fast 128 ⟩
    ⟨ init random seed, reset on exit 96 ⟩
    ⟨ Cholesky of precision 98 ⟩
    ⟨ input checks 74 ⟩
    ⟨ standardise 75 ⟩
    ⟨ check and / or set integration weights 97 ⟩

    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));

    ⟨ R normalise scores (log scale, AI) 154 ⟩

    ⟨ post differentiate mean score 119 ⟩
    ⟨ post differentiate lower score 120 ⟩
    ⟨ post differentiate upper score 121 ⟩

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

    ⟨ post differentiate chol score 122 ⟩
    ⟨ post differentiate invchol score 123 ⟩
    ⟨ post process score 124 ⟩
    ⟨ invcholmean score 125 ⟩

    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"
        if (!missing(invcholmean)) ret$invcholmean <- sinvcholmean
        return(ret)
    }

    return(ret)
}
```

◇

Fragment referenced in 72.

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] -0.10403 -0.08311 -0.10447 -0.20040

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

[1] -0.10403 -0.08311 -0.10447 -0.20040

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

$logLik
[1] -0.10403 -0.08311 -0.10447 -0.20040

$mean
    [,1]    [,2]   [,3]   [,4]
1 0.1139 0.04481 0.1149 0.2677
```

```
$lower
     [,1]    [,2]    [,3]    [,4]
1 -0.1514 -0.1043 -0.1522 -0.2816

$upper
    [,1]    [,2]    [,3]   [,4]
1 0.0375 0.05945 0.03724 0.0139

$chol
       [,1]    [,2]    [,3]    [,4]
1.1 -0.2903 -0.2833 -0.2904 -0.2972
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] -0.2903 -0.2833 -0.2904 -0.2972
```

4.3 Log-Scale Implementation (AI written)

The probability-scale implementation in Sections 4.1 and 4.2 runs into numerical problems when integration intervals lie far in the tails, are very narrow, or span higher dimensions. First, evaluating $e_j - d_j = \Phi(b_j^{(i)} - x_{j-1}) - \Phi(a_j^{(i)} - x_{j-1})$ directly suffers from cancellation when the two probabilities are close and returns zero once both Φ-values saturate to 0 or 1 in double precision; in addition, clamping $d_{j-1} + w_{j-1}(e_{j-1} - d_{j-1})$ to $[\text{tol}, 1 - \text{tol}]$ silently truncates y_{j-1} in the tails. Second, multiplying $f_j = (e_j - d_j) f_{j-1}$ across dimensions can shrink intsum $= \sum_{m=1}^{M} f_{J,m}$ below `tol` (or underflow it to zero), clamping the log-likelihood at $\log(\text{tol}) - \log M$ and setting all scores to zero even when the true log-probability and its gradients are finite.

We avoid both issues by carrying out the computations on the log scale: interval probabilities are evaluated directly as log-differences, products f_j become sums $\ell_{f,j}$ combined across integration points $m = 1, \ldots, M$ via a running maximum, and in `R_slpmvnorm` all intermediate derivatives are normalised so that `fp` accumulates the score of $\log f_j$ additively. We omit the observation index i in the following for notational convenience.

Arguments `tol` and `fast` remain in the signatures of `lpmvnorm`, `slpmvnorm`, `R_lpmvnorm`, and `R_slpmvnorm` for backwards compatibility and are unused. Unchanged fragments from Sections 4.1 and 4.2 are reused directly below, we issue a warning if users change the defaults

⟨ *tol+fast* 128 ⟩ ≡

```
if (tol != .Machine$double.eps)
    warning("argument tol has been ignored")
if (fast)
    warning("argument fast has been ignored")
```

◇

Fragment referenced in 99, 127.

4.3.1 Log-Scale Algorithm

We replace `C_pnorm_fast` and `C_pnorm_slow` with `C_lognormdiff`, which computes the log-probability of an interval $(a, b]$ shifted by μ (argument `m`) directly on the log scale. Writing ℓ_d and ℓ_e for the log-probabilities of the two limits ($\ell_d \le \ell_e$ for $a \le b$), we evaluate

$$\begin{aligned} \ell_{e-d} &= \log\bigl(\Phi(b-\mu) - \Phi(a-\mu)\bigr) \\ &= \ell_e + \log\bigl(1 - \exp(\ell_d - \ell_e)\bigr) \end{aligned}$$

in the lower tail when $a < \mu$ (`lt = 1`) and, using $\Phi(b-\mu) - \Phi(a-\mu) = \bar\Phi(a-\mu) - \bar\Phi(b-\mu)$ with $\bar\Phi = 1 - \Phi$, in the upper tail when $a \ge \mu$ (`lt = 0`). For $x = \ell_e - \ell_d$, the term $\log(1 - \exp(-x))$ is evaluated via R's `pexp(x, 1.0, 1L, 1L)`, which implements the accurate algorithm of Mächler (2012) and returns $-\infty$ for $x \le 0$ (the same idea is used by `.lognormdiff` in Chapter 10). Besides returning ℓ_{e-d}, `C_lognormdiff` stores the tail indicator `lt` and the larger log-probability `le` (ℓ_e) for the subsequent quantile inversion.

⟨ *lognormdiff (log scale, AI)* 129 ⟩ ≡

```
/*
    log(Phi(b - m) - Phi(a - m)) = le + log(1 - exp(ld - le))

    where ld <= le are the log-probabilities of the two limits in the lower
    tail (lt = 1) or, for a >= m, by reflection Phi(b) - Phi(a) =
    Phi(-a) - Phi(-b), in the upper tail (lt = 0). The same idea is
    implemented by .lognormdiff() for lpRR().
*/

static double C_lognormdiff (double a, double b, double m, int *lt, double *le) {

    double ld;

    *lt = (a < m);
    ld  = pnorm((*lt ? a : b), m, 1.0, *lt, 1L);
    *le = pnorm((*lt ? b : a), m, 1.0, *lt, 1L);
    if (*le == R_NegInf) return(R_NegInf);
    /* pexp(x, log) = log(1 - exp(-x)) for x > 0 and -Inf otherwise */
    return(*le + pexp(*le - ld, 1.0, 1L, 1L));
}
```

◇

Fragment referenced in 73.

Steps 1 and 2 of the algorithm (`input checks` and `standardise`) remain unchanged. At step 3, we initialise the first dimension on the log scale (with shift $x_0 = -\text{center}_1$ when `center` is given, and $x_0 = 0$ otherwise) and set the running maximum `lmax` to `R_NegInf`

⟨ *initialisation (log scale, AI)* 130 ⟩ ≡

```
x0 = 0.0;
if (LENGTH(center))
    x0 = -dcenter[0];
lemd0 = C_lognormdiff(da[0], db[0], x0, &lt0, &le0);
lmax = R_NegInf;
```

◇

Fragment referenced in 140, 153.

At step 4, each integration point $m = 1, \ldots, M$ starts with

⟨ *init logLik loop (log scale, AI)* 131 ⟩ ≡

```
lt = lt0;
le = le0;
lemd = lemd0;
lf = lemd0;
start = 0;
◇
```

Fragment referenced in 140, 143.

Within step 4(b), we clamp w_{j-1} (`Wtmp`) away from 0 and 1 by `DBL_EPSILON` so that y_{j-1} remains finite even if a quasi-Monte-Carlo sequence or random draw hits 0 or 1. To avoid evaluating $d_{j-1}+w_{j-1}(e_{j-1}-d_{j-1})$ on the probability scale, we rewrite it in terms of the larger tail probability stored in `le`. In the lower tail (`lt = 1`), $d_{j-1}+w_{j-1}(e_{j-1}-d_{j-1}) = e_{j-1}-(1-w_{j-1})(e_{j-1}-d_{j-1})$ gives the log-probability

$$\log\bigl(d_{j-1} + w_{j-1}(e_{j-1} - d_{j-1})\bigr) \quad = \quad \ell_e + \log\bigl(1 - (1 - w_{j-1})\exp(\ell_{e-d} - \ell_e)\bigr),$$

while in the upper tail (`lt = 0`), $1-(d_{j-1}+w_{j-1}(e_{j-1}-d_{j-1})) = (1-d_{j-1})-w_{j-1}(e_{j-1}-d_{j-1})$ gives the same form with $1-w_{j-1}$ replaced by w_{j-1}, which we then pass to `qnorm` on the log scale

⟨ *compute y (log scale, AI)* 132 ⟩ ≡

```
Wtmp = (W == R_NilValue ? unif_rand() : dW[j - 1]);
/* y must be finite */
Wtmp = fmin2(fmax2(Wtmp, DBL_EPSILON), 1.0 - DBL_EPSILON);
/* d + w (e - d) = e - (1 - w) (e - d);
   reflection replaces w by 1 - w */
tmp = le + log1p(-(lt ? 1.0 - Wtmp : Wtmp) * exp(lemd - le));
y[j - 1] = qnorm(tmp, 0.0, 1.0, lt, 1L);
◇
```

Fragment referenced in 135, 150.

Computing x_{j-1} is unchanged, after which we update $\ell_{e-d,j}$ and accumulate $\ell_{f,j} = \sum_{s=1}^{j} \ell_{e-d,s}$

⟨ *update d, e (log scale, AI)* 133 ⟩ ≡

```
lemd = C_lognormdiff(da[j], db[j], x, &lt, &le);
◇
```

Fragment referenced in 135, 150.

⟨ *update f (log scale, AI)* 134 ⟩ ≡

```
start += j;
lf += lemd;
◇
```

Fragment referenced in 135, 150.

We put these steps together in the inner loop, stopping early as soon as $\ell_f = -\infty$ so that subsequent iterations do not attempt to invert an empty or zero-probability interval

⟨ *inner logLik loop (log scale, AI)* 135 ⟩ ≡

```
/* stop early when f = 0 */
for (j = 1; j < iJ && lf > R_NegInf; j++) {

    ⟨ compute y (log scale, AI) 132 ⟩
    ⟨ compute x 79 ⟩
    ⟨ update d, e (log scale, AI) 133 ⟩
    ⟨ update f (log scale, AI) 134 ⟩

}
```

◇

Fragment referenced in 140.

To combine the log-integrands $\ell_{f,m}$ across $m = 1, \ldots, M$ without underflow or overflow, we track their running maximum $\ell^\star = \max_m \ell_{f,m}$ (`lmax`) and rescale `intsum` whenever a larger $\ell_{f,m}$ arrives, skipping draws with $\ell_{f,m} = -\infty$. Each finite draw contributes $\exp(\ell_{f,m} - \ell^\star) \in [0, 1]$ with the maximum contributing 1, so intsum $= \sum_m \exp(\ell_{f,m} - \ell^\star) \in [1, M]$ whenever at least one draw has finite log-integrand. At step 5, we output $\log \hat{p} = \ell^\star + \log(\text{intsum}) - \log M$, which gives $-\infty$ when all draws have $\ell_{f,m} = -\infty$

⟨ *increment (log scale, AI)* 136 ⟩ ≡

```
/* intsum = sum_m exp(lf_m - lmax), lmax = max_m lf_m */
if (lf > lmax) {
    intsum *= exp(lmax - lf);
    lmax = lf;
}
if (lf != R_NegInf)
    intsum += exp(lf - lmax);
```

◇

Fragment referenced in 140.

⟨ *output (log scale, AI)* 137 ⟩ ≡

```
dans[0] += lmax + log(intsum) - lM;
if (!RlogLik)
    dans += lL;
```

◇

Fragment referenced in 140.

For the univariate case $J = 1$, setting `iM = 1` executes a single iteration with an empty inner loop, which handles both `R_lpmvnorm` and `R_slpmvnorm` without a separate special case (such as `init dans` in Section 4.2)

⟨ *univariate problem (log scale, AI)* 138 ⟩ ≡

```
if (iJ == 1) iM = 1;    /* one draw with an empty inner loop */
lM = log((double) iM);
```

◇

Fragment referenced in 140, 153.

We put everything together in `R_lpmvnorm`, reusing the unchanged setup and pointer-advance fragments from Section 4.1

⟨ *R slpmvnorm variables (log scale, AI)* 139 ⟩ ≡

```
SEXP ans;
double *da, *db, *dC, *dW, *dans;
double *dcenter;
double le0, lemd0, lmax;
int lt0;
```

◇

Fragment referenced in 140, 153.

⟨ *R lpmvnorm (log scale, AI)* 140 ⟩ ≡

```
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 (log scale, AI) 139 ⟩

    double lM, x0, intsum;
    int p, len;

    Rboolean RlogLik = asLogical(logLik);

    ⟨ dimensions 90 ⟩
    ⟨ W length 89 ⟩
    ⟨ init center 93 ⟩

    int start, j, k, lt;
    double tmp, Wtmp, le, lf, lemd, x, y[(iJ > 1 ? iJ - 1 : 1)];

    ⟨ setup return object 91 ⟩

    ⟨ univariate problem (log scale, AI) 138 ⟩

    if (W == R_NilValue)
        GetRNGstate();

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

        intsum = 0.0;
        ⟨ initialisation (log scale, AI) 130 ⟩

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

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

            ⟨ init logLik loop (log scale, AI) 131 ⟩
            ⟨ inner logLik loop (log scale, AI) 135 ⟩
            ⟨ increment (log scale, AI) 136 ⟩

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

        ⟨ output (log scale, AI) 137 ⟩
        ⟨ move on 85 ⟩

    }

    if (W == R_NilValue)
        PutRNGstate();

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

◇

Fragment referenced in 73.

The R wrapper `lpmvnorm` is unchanged.

4.3.2 Log-Scale Score Function

Allocation of the score workspace is unchanged from Section 4.2. On the log scale, we store normalised derivatives instead of raw ones: for any parameter θ, the `dp` and `ep` arrays hold the derivatives of d_j and e_j divided by $e_j - d_j$, and the `fp` arrays hold $f_j^{-1}\partial f_j/\partial\theta = \partial \log f_j/\partial\theta$. Because $\log f_j = \log f_{j-1} + \log(e_j - d_j)$, the chain rule becomes additive:

$$\frac{\partial \log f_j}{\partial\theta} \quad = \quad \frac{\partial \log f_{j-1}}{\partial\theta} + \frac{1}{e_j - d_j}\left(\frac{\partial e_j}{\partial\theta} - \frac{\partial d_j}{\partial\theta}\right).$$

Moreover, evaluating the density ratios $\phi(a_j - x_{j-1})/(e_j - d_j)$ and $\phi(b_j - x_{j-1})/(e_j - d_j)$ in log-space as $\exp(\log\phi(\cdot) - \ell_{e-d,j})$ keeps them finite even in deep tails where numerator and denominator both underflow.

At the start of each draw (dimension $j = 1$, where $f_1 = e_1 - d_1$), we initialise these normalised derivatives using `dtmp` $= \exp(\log\phi(a_1 - x_0) - \ell_{e-d,1})$ and `etmp` $= \exp(\log\phi(b_1 - x_0) - \ell_{e-d,1})$

⟨ *score c11 (log scale, AI)* 141 ⟩ ≡

```
/* dp, ep, and fp are derivatives of d, e, and f divided by
   e - d, e - d, and f, respectively: fp is the score of log(f) */
dtmp = exp(dnorm(da[0], x0, 1.0, 1L) - lemd);
etmp = exp(dnorm(db[0], x0, 1.0, 1L) - lemd);
if (LENGTH(center)) {
    dp_c[0] = (R_FINITE(da[0]) ? dtmp * (da[0] - x0 - dcenter[0]) : 0);
    ep_c[0] = (R_FINITE(db[0]) ? etmp * (db[0] - x0 - dcenter[0]) : 0);
} else {
    dp_c[0] = (R_FINITE(da[0]) ? dtmp * (da[0] - x0) : 0);
    ep_c[0] = (R_FINITE(db[0]) ? etmp * (db[0] - x0) : 0);
}
fp_c[0] = ep_c[0] - dp_c[0];
◇
```

Fragment referenced in 143.

⟨ *score a, b (log scale, AI)* 142 ⟩ ≡

```
dp_m[0] = (R_FINITE(da[0]) ? dtmp : 0);
ep_m[0] = (R_FINITE(db[0]) ? etmp : 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 143.

⟨ *init score loop (log scale, AI)* 143 ⟩ ≡

```
⟨ init logLik loop (log scale, AI) 131 ⟩
⟨ score c11 (log scale, AI) 141 ⟩
⟨ score a, b (log scale, AI) 142 ⟩
◇
```

Fragment referenced in 153.

Because the `dp` and `ep` values from step $j-1$ are divided by $e_{j-1}-d_{j-1}$, recovering the unnormalised derivative of y_{j-1} requires multiplying by $(e_{j-1} - d_{j-1})/\phi(y_{j-1})$, which we evaluate in log-space as $\exp(\ell_{e-d,j-1} - \log\phi(y_{j-1}))$. Since this needs `lemd` from step $j-1$, updating `yp` must happen right after `compute y` and before `update d, e` overwrites `lemd`. The fragment `update yp for means, lower and upper` from Section 4.2 simply reuses `ytmp` and therefore remains unchanged

⟨ *update yp for chol (log scale, AI)* 144 ⟩ ≡

```
/* needs e - d from j - 1, so moved before "update d, e" */
/* = (e - d) / dnorm(y[j - 1], 0.0, 1.0, 0L) */
ytmp = exp(lemd - dnorm(y[j - 1], 0.0, 1.0, 1L));

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 150.

After updating x_{j-1} and $\ell_{e-d,j}$, we evaluate $\phi(a_j - x_{j-1})/(e_j - d_j)$ and $\phi(b_j - x_{j-1})/(e_j - d_j)$ in log-space and set the scores for parameters entering at step j

⟨ *score wrt new chol off-diagonals (log scale, AI)* 145 ⟩ ≡

```
dtmp = exp(dnorm(da[j], x, 1.0, 1L) - lemd);
ndtmp = dtmp * (-1.0);
etmp = exp(dnorm(db[j], x, 1.0, 1L) - lemd);
netmp = etmp * (-1.0);

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

Fragment referenced in 150.

⟨ *score wrt new chol diagonal (log scale, AI)* 146 ⟩ ≡

```
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];
◇
```

Fragment referenced in 150.

⟨ *new score means, lower and upper (log scale, AI)* 147 ⟩ ≡

```
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];
fp_u[j] = ep_m[j];
fp_m[j] = fp_u[j] + fp_l[j];
◇
```

Fragment referenced in 150.

We next update scores for parameters introduced for smaller j, which now accumulate additively in `fp`

⟨ *update score for chol (log scale, AI)* 148 ⟩ ≡

```
for (idx = 0; idx < j * (j + 1) / 2; idx++) {
    idxJ1 = idx * (iJ - 1);
    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_c[idxJ1 + k];

    dp_c[idx] = ndtmp * xx;
    ep_c[idx] = netmp * xx;
    fp_c[idx] += ep_c[idx] - dp_c[idx];
}
◇
```

Fragment referenced in 150.

⟨ *update score means, lower and upper (log scale, AI)* 149 ⟩ ≡

```
for (idx = 0; idx < j; idx++) {
    idxJ1 = idx * (iJ - 1);
    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_m[idxJ1 + k];

    dp_m[idx] = ndtmp * xx;
    ep_m[idx] = netmp * xx;
    fp_m[idx] += ep_m[idx] - dp_m[idx];

    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_l[idxJ1 + k];

    dp_l[idx] = ndtmp * xx;
    dp_u[idx] = netmp * xx;
    fp_l[idx] += dp_u[idx] - dp_l[idx];

    xx = 0.0;
    for (k = 0; k < j; k++)
        xx += dC[start + k] * yp_u[idxJ1 + k];

    ep_l[idx] = ndtmp * xx;
    ep_u[idx] = netmp * xx;
    fp_u[idx] += ep_u[idx] - ep_l[idx];
}
```

◇

Fragment referenced in 150.

We put everything together in the inner score loop, using the same early exit when $\ell_f = -\infty$

⟨ *inner score loop (log scale, AI)* 150 ⟩ ≡

```
/* stop early when f = 0 */
for (j = 1; j < iJ && lf > R_NegInf; j++) {

    ⟨ compute y (log scale, AI) 132 ⟩
    ⟨ update yp for chol (log scale, AI) 144 ⟩
    ⟨ update yp for means, lower and upper 109 ⟩
    ⟨ compute x 79 ⟩
    ⟨ update d, e (log scale, AI) 133 ⟩
    ⟨ score wrt new chol off-diagonals (log scale, AI) 145 ⟩
    ⟨ score wrt new chol diagonal (log scale, AI) 146 ⟩
    ⟨ new score means, lower and upper (log scale, AI) 147 ⟩
    ⟨ update score for chol (log scale, AI) 148 ⟩
    ⟨ update score means, lower and upper (log scale, AI) 149 ⟩
    ⟨ update f (log scale, AI) 134 ⟩

}
```

◇

Fragment referenced in 153.

Writing $\ell^\star = \max_m \ell_{f,m}$, the score of $\log \hat{p}$ is a weighted average of the per-draw scores $\partial \log f_{J,m}/\partial\theta$,

$$\frac{\partial \log \hat{p}}{\partial \theta} \quad = \quad \frac{\sum_{m=1}^{M} \exp(\ell_{f,m} - \ell^\star) \, \frac{\partial \log f_{J,m}}{\partial \theta}}{\sum_{m=1}^{M} \exp(\ell_{f,m} - \ell^\star)}.$$

At step 4(c), `dans[0]` accumulates the denominator and the remaining entries of `dans` accumulate the numerator with running-max rescaling, skipping draws with $\ell_{f,m} = -\infty$. After the loop over m, we divide the score entries by `dans[0]` whenever `dans[0] > 0` (leaving them at zero if all draws have zero probability) and overwrite `dans[0]` with $\ell^\star + \log\bigl(\sum_{m=1}^{M} \exp(\ell_{f,m} - \ell^\star)\bigr) - \log M$

⟨ *score output (log scale, AI)* 151 ⟩ ≡

```
/* dans = sum_m exp(lf_m - lmax) * (1, fp_m), lmax = max_m lf_m */
if (lf > lmax) {
    tmp = exp(lmax - lf);
    for (j = 0; j < Jp + 1 + 3 * iJ; j++)
        dans[j] *= tmp;
    lmax = lf;
}
if (lf != R_NegInf) {
    f = exp(lf - lmax);
    dans[0] += f;
    for (j = 0; j < Jp; j++)
        dans[j + 1] += f * fp_c[j];
    for (j = 0; j < iJ; j++) {
        idx = Jp + j + 1;
        dans[idx] += f * fp_m[j];
        dans[idx + iJ] += f * fp_l[j];
        dans[idx + 2 * iJ] += f * fp_u[j];
    }
}
◇
```

Fragment referenced in 153.

⟨ *normalise scores (log scale, AI)* 152 ⟩ ≡

```
/* scores of log(intsum), zero if intsum = 0; and log(intsum / M) */
if (dans[0] > 0) {
    for (j = 1; j < Jp + 1 + 3 * iJ; j++)
        dans[j] /= dans[0];
}
dans[0] = lmax + log(dans[0]) - lM;
◇
```

Fragment referenced in 153.

We put everything together in `R_slpmvnorm`

⟨ *R slpmvnorm (log scale, AI)* 153 ⟩ ≡

```
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 (log scale, AI) 139 ⟩

    double lM;
    int p, idx, idxJ1;
    ⟨ dimensions 90 ⟩
    ⟨ W length 89 ⟩
    ⟨ init center 93 ⟩

    int start, j, k, lt;
    double tmp, le, lf, f, lemd, x, x0, y[(iJ > 1 ? iJ - 1 : 1)];

    ⟨ score output object 104 ⟩

    ⟨ univariate problem (log scale, AI) 138 ⟩

    if (W == R_NilValue)
        GetRNGstate();

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

        ⟨ initialisation (log scale, AI) 130 ⟩

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

        for (int m = 0; m < iM; m++) {
            ⟨ init score loop (log scale, AI) 143 ⟩
            ⟨ inner score loop (log scale, AI) 150 ⟩
            ⟨ score output (log scale, AI) 151 ⟩

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

        ⟨ normalise scores (log scale, AI) 152 ⟩
        ⟨ move on 85 ⟩

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

    if (W == R_NilValue)
        PutRNGstate();

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

◇

Fragment referenced in 73.

On the R side, `R_slpmvnorm` already returns the log-likelihood contributions and the normalised scores of the log-likelihood, so `slpmvnorm` only needs to separate the first row from the score rows

⟨ *R normalise scores (log scale, AI)* 154 ⟩ ≡

```
### C returns the log-likelihood contributions and the scores of
### the log-likelihood; the latter are zero for observations with
### probability zero
ll <- ret[1L,]
ret <- ret[-1L,,drop = FALSE]
◇
```

Fragment referenced in 127.

Let's check an example far in the upper tail (with limits in $[20, 22]^3$), where both the log-likelihood and its Cholesky scores remain finite and agree with `numDeriv::grad`

```
> C_tail <- ltMatrices(c(1, 0.5, 1, -0.3, 0.4, 1), diag = TRUE, byrow = TRUE)
> lwr_tail <- matrix(c(20, 20, 20), nrow = 3)
> upr_tail <- matrix(c(22, 22, 22), nrow = 3)
> w_tail <- outer(sqrt(c(2, 3)), 1:200) %% 1
> (ll_tail <- lpmvnorm(lwr_tail, upr_tail, chol = C_tail, w = w_tail))

[1] -502

> stopifnot(is.finite(ll_tail), ll_tail < -100)
> s_tail <- slpmvnorm(lwr_tail, upr_tail, chol = C_tail, w = w_tail)
> chk(ll_tail, s_tail$logLik)
> if (require("numDeriv", quietly = TRUE)) {
+     f_tail <- function(prm)
+         lpmvnorm(lwr_tail, upr_tail,
+                  chol = ltMatrices(prm, diag = TRUE, byrow = TRUE),
+                  w = w_tail)
+     chk(grad(f_tail, c(unclass(C_tail))), c(unclass(s_tail$chol)),
+         check.attributes = FALSE, tolerance = 1e-5)
+ }
```

Chapter 5

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 <- 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        .        .        .
2 0.26505  0.93465        .        .
3 0.50102 -0.04587  1.05234        .
4 0.02356  0.89535  0.10482  0.90544
```

```
> 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       .       .       .
2  0.2500  0.9682       .       .
3  0.6124 -0.1581  1.0488       .
4       .  1.0954  0.1651  0.8790
```

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. 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) {
```

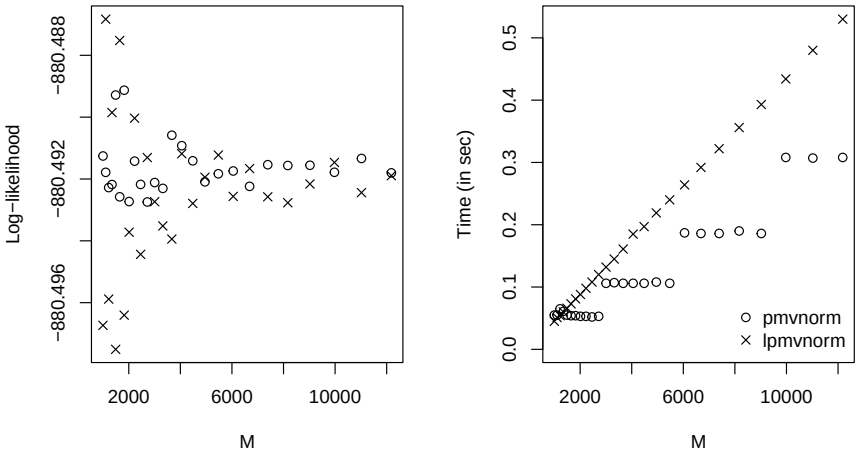

Figure 5.1: Evaluated log-likelihoods (left) and timings (right).

```
+     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))
+ })
```

The evaluated log-likelihoods and corresponding timings are given in Figure 5.1. 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, byrow = TRUE)
+ }
> 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        .        .        .
2  0.26764  1.02232        .        .
3  0.54268 -0.05007  1.11348        .
4  0.05223  0.98430  0.08473  0.96137

> lt

, , 1

        1       2       3       4
1  0.7071       .       .       .
2  0.2500  0.9682       .       .
3  0.6124 -0.1581  1.0488       .
4       .  1.0954  0.1651  0.8790
```

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_given = 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 3)

```
> c(cond_mvnorm(chol = perm(as.chol(C), perm = c(2:J, 1)),
+               which_given = 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(perm(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_given = 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 6

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 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 3. 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* 155 ⟩ ≡

```
stopifnot(xor(missing(chol), missing(invchol)))
cJ <- nrow(obs)
dJ <- nrow(lower)
N <- ncol(obs)
stopifnot(N == ncol(lower))
stopifnot(N == ncol(upper))
if (missing(mean)) {
    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 156, 158.

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* 156 ⟩ ≡

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

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

    ⟨ dp input checks 155 ⟩

    stopifnot(missing(invcholmean))

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

        ### check if 1:cJ corresponds to obs
        nm <- dimnames(invchol)[[2L]]
        rno <- rownames(obs)
        rnl <- rownames(lower)
        rnu <- rownames(upper)
        if (!is.null(rnl) && !is.null(rnu))
            stopifnot(isTRUE(all.equal(rnl, rnu)))
        if (!is.null(rnl) && !is.null(rno) && !is.null(nm))
            stopifnot(isTRUE(all.equal(c(rno, rnl), nm)))

        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 72.

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

⟨ *sldpmvnorm invchol* 157 ⟩ ≡

```
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, logLik = logLik)

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, drop = FALSE]
idx <- t(M1)[.ut(cJ, 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), drop = FALSE]
idx <- t(M3)[.ut(nrow(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$obs + dobs),
         ds[c("lower", "upper")])
if (logLik) ret$logLik <- cs$logLik + ds$logLik
ret$mean <- rbind(-ret$obs, ds$mean)
return(ret)
```

◇

Fragment referenced in 158.

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

⟨ *sldpmvnorm* 158 ⟩ ≡

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

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

    ⟨ dp input checks 155 ⟩

    ### for mixed data we cannot deal with invcholmean at the moment
    stopifnot(missing(invcholmean))

    if (!missing(invchol)) {
        ⟨ sldpmvnorm invchol 157 ⟩
    }

    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 72.

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, tolerance = 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        .        .        .
2  0.26504  0.93467        .        .
3  0.53509 -0.05736  1.11261        .
4  0.06749  0.95887  0.07775  0.96692

> ## compare with true C
> lt

, , 1

        1       2       3       4
1  0.7071       .       .       .
2  0.2500  0.9682       .       .
3  0.6124 -0.1581  1.0488       .
4       .  1.0954  0.1651  0.8790

> ## 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 `perm` 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 <- perm(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 `perm` 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 $f_3 =$`perm`, and we are interested in its Jacobian. Deriving the Jacobian 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 Jacobian of f_1 as the product of the Jacobians of f_2 and f_3:

$$\frac{\partial f_1(\mathbf{C})}{\partial \mathbf{C}} = \frac{\partial f_2(\tilde{\mathbf{C}})}{\partial \tilde{\mathbf{C}}} \frac{\partial f_3(\mathbf{C})}{\partial \mathbf{C}}. \tag{6.1}$$

The last factor is what we want to compute. It turns out that it is simpler to compute the first two Jacobians 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 Jacobian 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 (6.1), $\mathbf{B}_1 = \mathbf{B}_2\mathbf{B}_3$. 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* 159 ⟩ ≡

```
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 163.

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

⟨ *deperma input checks perm* 160 ⟩ ≡

```
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 163.

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* 161 ⟩ ≡

```
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 163.

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* 162 ⟩ ≡

```
idx <- matrix(1:J^2, nrow = J, ncol = J)        ### assuming byrow = TRUE
tidx <- c(t(idx))
ltT <- idx[.lt(J, 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 163.

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* 163 ⟩ ≡

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

    ⟨ deperma input checks chol 159 ⟩
    ⟨ deperma input checks perm 160 ⟩
    ⟨ deperma input checks schol 161 ⟩

    ⟨ deperma indices 162 ⟩

    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 72.

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 <- perm(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, tolerance = 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       .       .       .
2 0.05465 1.36452       .       .
3 0.29576 0.02194 0.61530       .
4 0.07133 0.66705 0.23615 0.66185

> ## compare with true _permuted_ C for (X, Y)
> round(as.array(perm(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 7

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 $\tilde{\mathbf{C}}(\mathbf{C}) := \mathrm{diag}(\mathbf{C}\mathbf{C}^\top)^{-1/2}\mathbf{C}$ such that $\mathbf{\Sigma} = \tilde{\mathbf{C}}\tilde{\mathbf{C}}^\top$ is a correlation matrix ($\mathrm{diag}(\mathbf{\Sigma}) = \mathbf{I}_J$). Note that directly estimating $\tilde{\mathbf{C}}$ requires $J(J+1)/2$ parameters under constraints $\mathrm{diag}(\mathbf{\Sigma}) = 1$ whereas only $J(J-1)/2$ parameters are necessary when estimating the lower triangular part of $\mathbf{C}$. The standardisation by $\mathrm{diag}(\mathbf{C}\mathbf{C}^\top)^{-1/2}$ ensures that $\mathrm{diag}(\mathbf{\Sigma}) \equiv 1$, that is, unconstained optimisation can be applied.

The function works both for unit and non-unit triangular matrices

⟨ *standardize* 164 ⟩ ≡

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

◇

Fragment referenced in 72.

```
> ### unit triangular
> C <- ltMatrices(runif(10))
> chk(as.array(chol2cov(standardize(chol = C))),
+     as.array(chol2cor(standardize(chol = C))))
> L <- solve(C)
> chk(as.array(invchol2cov(standardize(invchol = L))),
+     as.array(invchol2cor(standardize(invchol = L))))
> ### non-unit
> diagonals(C) <- 2
> chk(as.array(chol2cov(standardize(chol = C))),
+     as.array(chol2cor(standardize(chol = C))))
> L <- solve(C)
> chk(as.array(invchol2cov(standardize(invchol = L))),
+     as.array(invchol2cor(standardize(invchol = L))))
```

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

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

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* 165 ⟩ ≡

```
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[.lt(J, diag = FALSE)],,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 72.

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, ties.method = "max")) /
+        (nrow(iris) + 1)))
> (CR <- cor(t(Z)))
```

```
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length      1.00000    -0.09064       0.8690      0.7695
Sepal.Width      -0.09064     1.00000      -0.2446     -0.1908
Petal.Length      0.86896    -0.24456       1.0000      0.8577
Petal.Width       0.76951    -0.19082       0.8577      1.0000
```

```
> ll <- function(parm) {
+     C <- ltMatrices(parm)
+     Cs <- standardize(chol = C)
+     -ldmvnorm(obs = Z, chol = Cs)
+ }
> sc <- function(parm) {
+     C <- ltMatrices(parm)
+     Cs <- standardize(chol = 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] 605.5
```

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

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

```
> S_ML

, , 1

        1       2       3       4
1  1.0000 -0.1100  0.8797  0.7906
2 -0.1100  1.0000 -0.2597 -0.1957
3  0.8797 -0.2597  1.0000  0.8740
4  0.7906 -0.1957  0.8740  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 -0.27261  1.0000  0.8849
4  0.78328 -0.24823  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       .       .       . .
2 0.08219       .       . .
3 0.13663 0.08769       . .
4 0.12316 0.10956 0.10266 .

, , 1

        1       2       3 4
1       .       .       . .
2 0.07731       .       . .
3 0.14000 0.08695       . .
4 0.13691 0.11038 0.11610 .
```

However, we are still in the wrong, because we pretended we know the marginal distribution functions. A holistic approach would estimate the marginal empirical distributions and the copula parameters simultaneously.

Chapter 8

Joint Mean and Covariance Estimation

From a computational points of view, the parameterisation of the multivariate normal distribution in terms of the inverse Cholesky factor $\mathbf{L}$ and the scaled mean $\boldsymbol{\nu} = \mathbf{L}\boldsymbol{\mu}$ instead of the mean $\boldsymbol{\mu}$ is attractive, because the log-density is then jointly concave in $\boldsymbol{\nu}$ and $\mathbf{L}$ and thus a convex optimisation problem would emerge (Barratt and Boyd, 2023). This also carries over to the interval-censored likelihood because probabilities of log-concave densities are again log-concave (Prékopa, 1973).

The package implements the log-likelihood contributions as $\ell_i(\boldsymbol{\mu}_i, \mathbf{L}_i)$ and derives the scores with respect to $\boldsymbol{\mu}_i$ and $\mathbf{L}_i$ (FIXME: `ldmvnorm/sldmvnorm` now directly work with $\boldsymbol{\nu}$, but the other procedures still need adaptation). With $\boldsymbol{\mu}_i = \mathbf{L}_i^{-1}\boldsymbol{\nu}$ it is simple to rewrite the log-likelihood contribution as a concave function of both parameters as

$$\ell_i^{\text{concave}}(\boldsymbol{\nu}_i, \mathbf{L}_i) = \ell_i(\mathbf{L}_i^{-1}\boldsymbol{\nu}, \mathbf{L}_i).$$

The implementation of the corresponding convex negative log-likelihood is also easy:

```
> J <- 5
> N <- 100
> ### mean
> m <- rnorm(J)
> L <- ltMatrices(prm <- runif(J * (J + 1) / 2), diag = TRUE)
> Z <- matrix(rnorm(N * J), nrow = J)
> Y <- solve(L, Z) + m
> ### scaled mean; FIXME: update example using invcholmean args
> d <- L %*% m
> nll <- function(parm) {
+     d <- parm[seq_len(J)]
+     L <- ltMatrices(parm[-seq_len(J)], diag = TRUE)
+     -ldmvnorm(obs = Y, mean = solve(L, d), invchol = L)
+ }
> start <- c(d, prm)
> nll(start)

[1] 1135

> ### identical
> -ldmvnorm(obs = Y, mean = m, invchol = L)

[1] 1135
```

However, computing the scores is a bit more work. Let's first consider the scores with respect to $\boldsymbol{\nu}_i$. We have

$$\begin{aligned}
\frac{\partial \ell_i^{\text{concave}}(\boldsymbol{\nu}_i, \mathbf{L}_i)}{\partial \boldsymbol{\nu}_i} &= \frac{\partial \ell_i(\mathbf{L}_i^{-1}\boldsymbol{\nu}_i, \mathbf{L}_i)}{\partial \boldsymbol{\nu}_i} \\
&= \left.\frac{\partial \ell_i(\boldsymbol{\mu}, \mathbf{L}_i)}{\partial \boldsymbol{\mu}}\right|_{\boldsymbol{\mu}=\mathbf{L}_i^{-1}\boldsymbol{\nu}_i} \frac{\partial \mathbf{L}_i^{-1}\boldsymbol{\nu}_i}{\partial \boldsymbol{\nu}_i} \\
&= \left.\frac{\partial \ell_i(\boldsymbol{\mu}, \mathbf{L}_i)}{\partial \boldsymbol{\mu}}\right|_{\boldsymbol{\mu}=\mathbf{L}_i^{-1}\boldsymbol{\nu}_i} \mathbf{L}_i^{-1}.
\end{aligned}$$

The first term is the score with respect to $\boldsymbol{\mu}_i$ and thus the score with respect to $\boldsymbol{\nu}_i$ is straightforward to compute. The score with respect to $\mathbf{L}_i$ is more complex. With the multivariate chain rule we have

$$\begin{aligned}
\frac{\partial \ell_i^{\text{concave}}(\boldsymbol{\nu}_i, \mathbf{L}_i)}{\partial \mathbf{L}_i} &= \frac{\partial \ell_i(\mathbf{L}_i^{-1}\boldsymbol{\nu}_i, \mathbf{L}_i)}{\partial \mathbf{L}_i} \\
&= \left.\frac{\partial \ell_i(\boldsymbol{\mu}, \mathbf{L}_i)}{\partial \boldsymbol{\mu}}\right|_{\boldsymbol{\mu}=\mathbf{L}_i^{-1}\boldsymbol{\nu}_i} \frac{\partial \mathbf{L}_i^{-1}\boldsymbol{\nu}_i}{\partial \mathbf{L}_i} + \left.\frac{\partial \ell_i(\boldsymbol{\mu}, \mathbf{L}_i)}{\partial \mathbf{L}_i}\right|_{\boldsymbol{\mu}=\mathbf{L}_i^{-1}\boldsymbol{\nu}_i} \\
&= \left.\frac{\partial \ell_i(\boldsymbol{\mu}, \mathbf{L}_i)}{\partial \boldsymbol{\mu}}\right|_{\boldsymbol{\mu}=\mathbf{L}_i^{-1}\boldsymbol{\nu}_i} \frac{\partial \mathbf{L}_i^{-1}\boldsymbol{\nu}_i}{\partial \mathbf{L}_i^{-1}} \frac{\partial \mathbf{L}_i^{-1}}{\partial \mathbf{L}_i} + \left.\frac{\partial \ell_i(\boldsymbol{\mu}, \mathbf{L}_i)}{\partial \mathbf{L}_i}\right|_{\boldsymbol{\mu}=\mathbf{L}_i^{-1}\boldsymbol{\nu}_i} \\
&= \underbrace{\left.\frac{\partial \ell_i(\boldsymbol{\mu}, \mathbf{L}_i)}{\partial \boldsymbol{\mu}}\right|_{\boldsymbol{\mu}=\mathbf{L}_i^{-1}\boldsymbol{\nu}_i}}_{=:\boldsymbol{\mu}_i'} (\boldsymbol{\nu}_i^\top \otimes \mathbf{I}_J)(-1)(\mathbf{L}_i^{-\top} \otimes \mathbf{L}_i^{-1}) + \left.\frac{\partial \ell_i(\boldsymbol{\mu}, \mathbf{L}_i)}{\partial \mathbf{L}_i}\right|_{\boldsymbol{\mu}=\mathbf{L}_i^{-1}\boldsymbol{\nu}_i} \\
&= \text{vec}(\boldsymbol{\mu}_i'\boldsymbol{\nu}_i^\top)(-1)(\mathbf{L}_i^{-\top} \otimes \mathbf{L}_i^{-1}) + \left.\frac{\partial \ell_i(\boldsymbol{\mu}, \mathbf{L}_i)}{\partial \mathbf{L}_i}\right|_{\boldsymbol{\mu}=\mathbf{L}_i^{-1}\boldsymbol{\nu}_i}
\end{aligned}$$

where we apply the vec-trick twice in the last line.

The negative gradient is now

```
> nsc <- function(parm) {
+     d <- parm[seq_len(J)]
+     L <- ltMatrices(parm[-seq_len(J)], diag = TRUE)
+     ret <- sldmvnorm(obs = Y, mean = solve(L, d), invchol = L)
+     C <- solve(L)
+
+     J <- dim(L)[2L]
+     M <- matrix(seq_len(J^2), nrow = J, byrow = FALSE)
+     idx <- M[lower.tri(M, diag = TRUE)]
+
+     X <- -ret$obs
+     Y <- matrix(d, nrow = nrow(X), ncol = ncol(X))
+     A <- X[rep(1:nrow(X), times = nrow(X)),,drop = FALSE] *
+          Y[rep(1:nrow(Y), each = nrow(X)),,drop = FALSE]
+
+     scL <- - vectrick(C, A)
+     scL <- scL[idx,,drop = FALSE]
+     scL <- rowSums(unclass(scL) + unclass(ret$invchol))
+     - c(rowSums(solve(L, -ret$obs, transpose = TRUE)),
+         scL)
+ }
> chk(unname(nsc(start)), grad(nll, start))
```

Similarily, when the model is parameterized by $\boldsymbol{\nu}_i$ and $\mathbf{C}_i = \mathbf{L}_i^{-1}$, we can compute the log-likelihood as

```
> C <- ltMatrices(prm <- runif(J * (J + 1) / 2), diag = TRUE)
> Z <- matrix(rnorm(N * J), nrow = J)
> Y <- C %*% Z + m
> ### scaled mean
> d <- solve(C, m)
> nll <- function(parm) {
+     d <- parm[seq_len(J)]
+     C <- ltMatrices(parm[-seq_len(J)], diag = TRUE)
+     -ldmvnorm(obs = Y, mean = C %*% d, chol = C)
+ }
> start <- c(d, prm)
> nll(start)
```

```
[1] 144.1
```

```
> ### identical
> -ldmvnorm(obs = Y, mean = m, chol = C)
```

```
[1] 144.1
```

By the same arguments as above, the score with respect to $\boldsymbol{\nu}_i$ is

$$\begin{aligned} \frac{\partial \ell_i^{\text{concave}}(\boldsymbol{\nu}_i, \mathbf{C}_i)}{\partial \boldsymbol{\nu}_i} &= \frac{\partial \ell_i(\mathbf{C}_i \boldsymbol{\nu}_i, \mathbf{C}_i)}{\partial \boldsymbol{\nu}_i} \\ &= \left. \frac{\partial \ell_i(\boldsymbol{\mu}, \mathbf{C}_i)}{\partial \boldsymbol{\mu}} \right|_{\boldsymbol{\mu} = \mathbf{C}_i \boldsymbol{\nu}_i} \mathbf{C}_i. \end{aligned}$$

and the score with respect to $\mathbf{C}_i$ is

$$\begin{aligned} \frac{\partial \ell_i^{\text{concave}}(\boldsymbol{\nu}_i, \mathbf{C}_i)}{\partial \mathbf{C}_i} &= \frac{\partial \ell_i(\mathbf{C}_i \boldsymbol{\nu}_i, \mathbf{C}_i)}{\partial \mathbf{C}_i} \\ &= \boldsymbol{\mu}_i' \boldsymbol{\nu}_i^\top + \left. \frac{\partial \ell_i(\boldsymbol{\mu}, \mathbf{C}_i)}{\partial \mathbf{C}_i} \right|_{\boldsymbol{\mu} = \mathbf{C}_i \boldsymbol{\nu}_i} \end{aligned}$$

```
> nsc <- function(parm) {
+     d <- parm[seq_len(J)]
+     C <- ltMatrices(parm[-seq_len(J)], diag = TRUE)
+     ret <- sldmvnorm(obs = Y, mean = C %*% d, chol = C)
+
+     J <- dim(C)[2L]
+     M <- matrix(seq_len(J^2), nrow = J, byrow = FALSE)
+     idx <- M[lower.tri(M, diag = TRUE)]
+
+     X <- -ret$obs
+     Y <- matrix(d, nrow = nrow(X), ncol = ncol(X))
+     A <- X[rep(1:nrow(X), times = nrow(X)),,drop = FALSE] *
+          Y[rep(1:nrow(Y), each = nrow(X)),,drop = FALSE]
+
+     scC <- A[idx,,drop = FALSE]
+     scC <- rowSums(unclass(scC) + unclass(ret$chol))
+     - c(rowSums(crossprod(C, -ret$obs)),
+         scC)
+ }
> chk(unname(nsc(start)), grad(nll, start))
```

Chapter 9

User Interface

"interface.R" 166≡

⟨ *mvnorm* 171 ⟩
⟨ *mean2invcholmean* 172 ⟩
⟨ *invcholmean2mean* 173 ⟩
⟨ *mvnorm methods* 174 ⟩
⟨ *mvnorm simulate* 175 ⟩
⟨ *mvnorm margDist* 176 ⟩
⟨ *mvnorm condDist* 177 ⟩
⟨ *mvnorm logLik* 180 ⟩
⟨ *mvnorm lLgrad* 191 ⟩
⟨ *mvnorm marginal means* 56 ⟩
⟨ *mvnorm vcov* 57 ⟩
⟨ *mvnorm regression coefs* 58 ⟩
◇

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 $\mathbb{N}_J(\boldsymbol{\mu}, \mathbf{L}^{-1}\mathbf{L}^{-\top})$, or $\mathbb{N}_J(\mathbf{L}^{-1}\boldsymbol{\nu}, \mathbf{L}^{-1}\mathbf{L}^{-\top})$, or $\mathbb{N}_J(\mathbf{C}\boldsymbol{\nu}, \mathbf{C}\mathbf{C}^\top)$, 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* 167 ⟩ ≡

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

◇

Fragment referenced in 1.

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* 168 ⟩ ≡

```
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 171.

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* 169 ⟩ ≡

```
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 171.

⟨ *mvnorm invcholmean* 170 ⟩ ≡

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

◇

Fragment referenced in 171.

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). We also allow the specification of the scaled mean $\boldsymbol{\nu} = \mathbf{L}\boldsymbol{\mu}$ as argument `invcholmean`:

⟨ *mvnorm* 171 ⟩ ≡

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

  ⟨ mvnorm chol invchol 168 ⟩
  if (!missing(invcholmean)) {
      stopifnot(missing(mean))
      ### note: this is not really necessary once all ls functions
      ### are able to deal with invcholmean
      ⟨ mvnorm invcholmean 170 ⟩
  } else {
      ⟨ mvnorm mean 169 ⟩
  }
  class(ret) <- "mvnorm"
  return(ret)
}
```

◇

Fragment referenced in 166.

After updating the object, it might be necessary to update $\boldsymbol{\nu}$ or $\boldsymbol{\mu}$ as well:

⟨ *mean2invcholmean* 172 ⟩ ≡

```
.m2i <- function(object) {
    if (is.chol(object$scale))
        return(solve(object$scale, object$mean))
    return(object$scale %*% object$mean)
}
```

◇

Fragment referenced in 166.

⟨ *invcholmean2mean* 173 ⟩ ≡

```
.i2m <- function(object) {
    if (is.chol(object$scale))
        return(object$scale %*% object$invcholmean)
    return(solve(object$scale, object$invcholmean))
}
```

◇

Fragment referenced in 166.

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

⟨ *mvnorm methods* 174 ⟩ ≡

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

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

    if (!is.null(a$invcholmean))
        a$mean <- .i2m(a)
    ret <- list(scale = perm(a$scale, perm = perm, ...))
    if (!is.null(a$mean))
        ret$mean <- a$mean[perm,,drop = FALSE]
    if (!is.null(a$invcholmean)) {
        ret$invcholmean <- .m2i(ret)
        ret$mean <- NULL
    }
    class(ret) <- "mvnorm"
    ret
}
```

◇

Fragment referenced in 166.

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* 175 ⟩ ≡

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

    ⟨ init random seed, reset on exit 96 ⟩

    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$scale)
        }
    }
    Z <- matrix(rnorm(nsim * J), nrow = J)
    if (!is.null(object$invcholmean))
        Z <- Z + c(object$invcholmean)
    if (is.chol(object$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 166.

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* 176 ⟩ ≡

```
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$invcholmean))
        object$mean <- .i2m(object)
    if (!is.null(object$mean))
        ret$mean <- object$mean[which,,drop = FALSE]
    if (!is.null(object$invcholmean)) {
        ret$invcholmean <- .m2i(ret)
        ret$mean <- NULL
    }

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

◇

Fragment referenced in 166.

⟨ *mvnorm condDist* 177 ⟩ ≡

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

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

    if (!is.null(object$invcholmean))
        object$mean <- .i2m(object)
    if (!is.null(object$mean)) {
        if (ncol(object$mean) == 1L && NCOL(given) > 1L) {
            given <- given - object$mean[which_given,,drop = TRUE]
        } else {
            given <- c(given) - object$mean[which_given,,drop = FALSE]
        }
    }

    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$scale <- as.invchol(ret$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$mean) > 1) {
            ret$mean <- object$mean[-which_given,,drop = FALSE] + ret$mean
        } else 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)
        }
    }
    if (!is.null(object$invcholmean)) {
        ret$invcholmean <- .m2i(ret)
        ret$mean <- NULL
    }
    class(ret) <- "mvnorm"
    return(ret)
}
```

◇

Fragment referenced in 166.

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           .
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_given = 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* 178 ⟩ ≡

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

nm <- c(nmobs, nmlu)
no <- names(object)
### we allow dimensions w/o data
stopifnot(all(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 180, 191.

and handle standardization for the two possible parameterisations, first when **C** was given and second when the inverse Cholesky factors **L** was given

⟨ *logLik standardize* 179 ⟩ ≡

```
if (is.chol(object$scale)) {
    names(args)[names(args) == "scale"] <- "chol"
    if (standardize)
       args$chol <- object$scale <- standardize(chol = args$chol)
} else {
    names(args)[names(args) == "scale"] <- "invchol"
    if (standardize)
        args$invchol <- object$scale <- standardize(invchol = args$invchol)
}
```

◇

Fragment referenced in 180.

Putting everything together in a corresponding `logLik` method finally calls `ldpmvnorm` when there are (i) only continuous observations for all dimensions, (ii) only intervals for all dimensions, (iii) continuous observations for at least the first (maybe more) dimension followed by intervals for the remaining dimensions. If this is not the case, `perm` is not `NULL` and the appropriate marginal (for continuous) and conditional distributions (intervals given continuous) distributions are computed first. This also handles the case when data is missing for some observations.

In contrast to `lLgrad.mvnorm`, which first permutes the dimensions via `perm` and thus requires a Cholesky decomposition of the permuted $J \times J$ covariance matrix, Cholesky decompositions of lower dimensional matrices are required when computing the log-likelihood. However, it is best to avoid this reshuffeling by using an appropriate order of the dimensions (continuous first, intervals second, no missing data).

⟨ *mvnorm logLik* 180 ⟩ ≡

```
logLik.mvnorm <- function(object, obs, lower, upper, standardize = FALSE,
                          ...) {
    ⟨ argchecks 178 ⟩

    ⟨ logLik standardize 179 ⟩

    if (!is.null(perm)) {
        ll <- if (!is.null(args$logLik)) args$logLik else TRUE
        ret <- 0
        ### integrate out dimensions w/o data
        if (length(nm) < length(no) && !is.null(nmlu))
            object <- margDist(object, which = nm)
        ### continuous
        if (!is.null(nmobs)) {
            ret <- ret + logLik(margDist(object, which = nmobs),
                                obs = obs, logLik = ll)
            ### set-up conditional distribution if there are any intervals
            if (!is.null(nmlu))
                object <- condDist(object, which_given = nmobs, given = obs)
        }
        ### interval given continuous
        if (!is.null(nmlu))
            ret <- ret + logLik(object, lower = lower, upper = upper, ...)
        return(ret)
    }

    ### base likelihood on mean for mixed data
    if (!is.null(args$invcholmean) &&
        (!missing(obs)) + (!missing(lower) || !missing(upper)) == 2L  ### mixed data
        ) {
        args$mean <- .i2m(object)
        args$invcholmean <- NULL
    }
    return(do.call("ldpmvnorm", args))
}
```

◇

Fragment referenced in 166.

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

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

```
[1] -223.5
```

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] -223.5
```

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 ($\boldsymbol{\mu}$ or $\boldsymbol{\nu}$ 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`. A downside of the approach taken here is that post-processing of the scores takes place on the level of observations, as we do not compute the score function but the score contributions. If one is only interested in the former, major speedups could be achieved by summing over observations before post-processing.

⟨ *llgrad chol* 181 ⟩ ≡

```
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$chol <- pc <- perm(as.chol(args$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 182 ⟩
⟨ llgrad marginalisation 183 ⟩
⟨ llgrad deperma 184 ⟩
⟨ llgrad destandarized 185 ⟩
⟨ llgrad diagonals 186 ⟩
⟨ llgrad return 187 ⟩
◇
```

Fragment referenced in 191.

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* 182 ⟩ ≡

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

Fragment referenced in 181.

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

⟨ *llgrad marginalisation* 183 ⟩ ≡

```
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 181.

If the order of the variables was permuted, we compute the scores for the original ordering of the variables, as explained in Chapter 6

⟨ *llgrad deperma* 184 ⟩ ≡

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

Fragment referenced in 181.

The effect of standardization can be removed as discussed in Chapter 7

⟨ *llgrad destandarized* 185 ⟩ ≡

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

Fragment referenced in 181.

and it remains to remove fix diagonal elements

⟨ *lLgrad diagonals* 186 ⟩ ≡

```
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 181.

and to return the results, with mean scores in the correct ordering. Eventually, we have to perform the post-processing steps explained in Chapter 8 when the scaled mean $\boldsymbol{\nu} = \mathbf{L}\boldsymbol{\mu}$ was specified and the score is not already available from the score workhorse function `sldpmvnorm`

⟨ *lLgrad return* 187 ⟩ ≡

```
ret$scale <- ret$chol
ret$chol <- NULL
if (!is.null(ret$mean)) ret$mean <- ret$mean[no,, drop = FALSE]
### sldpmvnorm for mixed data returns mean score only
if (!is.null(object$invcholmean) && is.null(ret$invcholmean)) {
    J <- dim(sc)[2L]
    M <- matrix(seq_len(J^2), nrow = J, byrow = FALSE)
    idx <- M[.lt(J, diag = TRUE)]

    X <- ret$mean
    Y <- matrix(object$invcholmean, nrow = nrow(X), ncol = ncol(X))
    A <- X[rep(1:nrow(X), times = nrow(X)),,drop = FALSE] *
         Y[rep(1:nrow(Y), each = nrow(X)),,drop = FALSE]

    scC <- ltMatrices(A[idx,,drop = FALSE], diag = TRUE, byrow = FALSE)
    if (!attr(sc, "diag"))
        scC <- ltMatrices(Lower_tri(scC, diag = FALSE), diag = FALSE, byrow = FALSE)
    scC <- ltMatrices(scC, byrow = attr(sc, "byrow"))
    ret$scale <- ltMatrices(unclass(scC) + unclass(ret$scale),
                            diag = attr(sc, "diag"),
                            byrow = attr(sc, "byrow"),
                            names = dimnames(sc)[[2L]])
    ret$invcholmean <- Mult(sc, X, transpose = TRUE)
    ret$mean <- NULL
}
return(ret)
```

◇

Fragment referenced in 181.

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 $\mathbf{L}$. We start with marginalisation

⟨ *llgrad invchol marginalisation* 188 ⟩ ≡

```
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)
}
◇
```

Fragment referenced in 190.

turn to post-processing if necessary

⟨ *llgrad invchol post* 189 ⟩ ≡

```
### sldpmvnorm for mixed data returns mean score only
if (!is.null(object$invcholmean) && is.null(ret$invcholmean)) {
    J <- dim(si)[2L]
    M <- matrix(seq_len(J^2), nrow = J, byrow = FALSE)
    idx <- M[.lt(J, diag = TRUE)]

    X <- ret$mean
    Y <- matrix(object$invcholmean, nrow = nrow(X), ncol = ncol(X))
    A <- X[rep(1:nrow(X), times = nrow(X)),,drop = FALSE] *
         Y[rep(1:nrow(Y), each = nrow(X)),,drop = FALSE]

    scL <- - vectrick(solve(si), A)
    scL <- ltMatrices(scL[idx,,drop = FALSE], diag = TRUE, byrow = FALSE)
    if (!attr(si, "diag"))
        scL <- ltMatrices(Lower_tri(scL, diag = FALSE), diag = FALSE, byrow = FALSE)
    scL <- ltMatrices(scL, byrow = attr(si, "byrow"))
    ret$scale <- ltMatrices(unclass(scL) + unclass(ret$scale),
                            diag = attr(si, "diag"),
                            byrow = attr(si, "byrow"),
                            names = dimnames(si)[[2L]])
    ret$invcholmean <- solve(si, X, transpose = TRUE)
    ret$mean <- FALSE
}
◇
```

Fragment referenced in 190.

and finally put everything together

⟨ *llgrad invchol* 190 ⟩ ≡

```
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$invchol <- pi <- perm(as.invchol(args$invchol), perm = perm)
    if (length(nm) < length(no))
        args$invchol <- marg_mvnorm(invchol = args$invchol,
                                    which = nm)$invchol
    args$mean <- args$mean[nm,,drop = FALSE]
}
ret <- do.call("sldpmvnorm", args)
### sldmvnorm returns mean score as -obs
### return only if object had mean specified
if (is.null(ret$mean) && !is.null(args$mean))
    ret$mean <- - ret$obs

⟨ llgrad invchol marginalisation 188 ⟩

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
if (!is.null(ret$mean)) ret$mean <- ret$mean[no,, drop = FALSE]
⟨ llgrad invchol post 189 ⟩
return(ret)
◇
```

Fragment referenced in 191.

We can now provide the log-likelihood gradients

⟨ *mvnorm lLgrad* 191 ⟩ ≡

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

lLgrad.mvnorm <- function(object, obs, lower, upper, standardize = FALSE,
                          ...) {
    ⟨ argchecks 178 ⟩

    ### base likelihood on mean when necessary
    ### note that post processing the mean score is time consuming
    if (!is.null(args$invcholmean) &&
        (!is.null(perm) ||      ### permutations
         (!missing(obs)) + (!missing(lower) || !missing(upper)) == 2L  ### mixed data
        )) {
        args$mean <- .i2m(object)
        args$invcholmean <- NULL
    }

    if (is.chol(object$scale)) {
        ⟨ lLgrad chol 181 ⟩
    }
    ⟨ lLgrad invchol 190 ⟩
}
```

◇

Fragment referenced in 166.

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])
> 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)
+ }
> 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)
+     -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.

```
> ### don't start at the solution
> start <- round(c(c(iris_mvn$mean),
+                  Lower_tri(iris_mvn$scale, diag = TRUE)), 2)
> llim <- rep(-Inf, J + J * (J + 1) / 2)
> llim[J + c(diagonals(ltMatrices(seq_len(J * (J + 1) / 2), diag = TRUE)))] <- 1e-4
> op <- optim(start, fn = ll, gr = sc, method = "L-BFGS-B",
+             lower = llim,
+             control = list(trace = FALSE, maxit = 1500,
```

```
+                             factr = 1e-6)) ### noLD machines
> 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 (noLD brings up small differences)
> round(vcov(ML), 3)

, , 1

             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length        0.681      -0.042        1.266       0.513
Sepal.Width        -0.042       0.189       -0.327      -0.121
Petal.Length        1.266      -0.327        3.096       1.287
Petal.Width         0.513      -0.121        1.287       0.577

> 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
> mean(ML)[,,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
```

We now check if we can obtain corresponding results when jointly optimising the scaled mean $\boldsymbol{\nu}$ and the inverse Cholesky factor $\mathbf{L}$:

```
> ll <- function(parm, logLik = TRUE) {
+     L <- ltMatrices(parm[-(1:J)], diag = TRUE, names = vars)
+     x <- mvnorm(invcholmean = parm[1:J], invchol = L)
+     if (!logLik) return(x)
+     -logLik(object = x, obs = obs)
+ }
> sc <- function(parm) {
+     x <- ll(parm, logLik = FALSE)
+     ret <- lLgrad(object = x, obs = obs)
+     -c(rowSums(ret$invcholmean), rowSums(Lower_tri(ret$scale, diag = TRUE)))
+ }
> ### note: This is a convex problem now, so (here incorrect)
> ### starting values shouldn't matter
> opL <- optim(start, fn = ll, gr = sc, method = "L-BFGS-B",
+              lower = llim,
+              control = list(trace = FALSE, maxit = 1500,
+                             factr = 1e-6)) ### noLD machines
> MLL <- ll(opL$par, logLik = FALSE)
```

where the comparison to the analytic estimates is

```
> ### log-likelihood
> op$value

[1] 379.9

> opL$value

[1] 379.9

> ### covariance
> round(vcov(MLL), 3)

, , 1

             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length        0.681      -0.042        1.266       0.513
Sepal.Width        -0.042       0.189       -0.327      -0.121
Petal.Length        1.266      -0.327        3.096       1.287
Petal.Width         0.513      -0.121        1.287       0.577

> 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
> mean(MLL)[,,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]], probs = 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]

> 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]
```

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 need to optimise over both sets of parameters simultaneously and therefore opt for the parameterisation in terms of $\boldsymbol{\nu}$ and $\mathbf{L}$ because this leads to a convex optimisation problem also for interval-censored observations as mentioned in Chapter 8.

Extreme parameter values need no special treatment: because the log-likelihood is computed on the log scale (Section 4.3), observations whose likelihood underflows to zero on the probability scale still have finite log-likelihood contributions and scores.

```
> ll <- function(parm, logLik = TRUE) {
+     L <- ltMatrices(parm[-(1:J)], diag = TRUE, names = vars)
+     x <- mvnorm(invcholmean = parm[1:J], invchol = L)
+     if (!logLik) return(x)
+     -logLik(object = x, obs = obs, lower = lower, upper = upper)
+ }
> sc <- function(parm) {
+     x <- ll(parm, logLik = FALSE)
+     ret <- lLgrad(object = x, obs = obs, lower = lower, upper = upper)
+     -c(rowSums(ret$invcholmean, na.rm = TRUE),
+        rowSums(Lower_tri(ret$scale, diag = TRUE), na.rm = TRUE))
+ }

> start <- round(opL$par, 2)
> if (require("numDeriv", quietly = TRUE))
+     chk(grad(ll, start), sc(start), check.attributes = FALSE)
> opi <- optim(start, fn = ll, gr = sc, method = "L-BFGS-B",
+              lower = llim,
+              control = list(trace = FALSE, maxit = 1500,
+                             factr = 1e-6)) ### noLD machines
> MLi <- ll(opi$par, logLik = FALSE)
```

Because the likelihood is a product of a continuous density and a conditional probability as introduced in Chapter 6, 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)

```
> ### covariance
> round(vcov(MLi), 3)

, , 1

             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length        0.726      -0.026        1.271       0.522
Sepal.Width        -0.026       0.189       -0.327      -0.121
Petal.Length        1.271      -0.327        3.096       1.287
Petal.Width         0.522      -0.121        1.287       0.577

> round(vcov(MLL), 3)

, , 1

             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length        0.681      -0.042        1.266       0.513
```

```
Sepal.Width        -0.042       0.189      -0.327      -0.121
Petal.Length        1.266      -0.327       3.096       1.287
Petal.Width         0.513      -0.121       1.287       0.577

> round(vcov(ML), 3)

, , 1

             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length        0.681      -0.042        1.266       0.513
Sepal.Width        -0.042       0.189       -0.327      -0.121
Petal.Length        1.266      -0.327        3.096       1.287
Petal.Width         0.513      -0.121        1.287       0.577

> ### mean
> mean(MLi)[,,drop = TRUE]

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

> mean(MLL)[,,drop = TRUE]

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

> mean(ML)[,,drop = TRUE]

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

Again, we can extract the regression coefficients from these multivariate normal distributions. We start with the $\boldsymbol{\mu}$ and $\mathbf{C}$ parameterisation:

```
> ### least-squares coefficients
> coef(irislm <- lm(Petal.Width ~ Sepal.Length + Sepal.Width + Petal.Length,
+                   data = iris))

 (Intercept) Sepal.Length  Sepal.Width Petal.Length
     -0.2403      -0.2073       0.2228       0.5241

> ### residual standard deviation
> summary(irislm)$sigma

[1] 0.192

> ### compare with
> round(coef(ML, which = "Petal.Width"), 3)

 (Intercept) Sepal.Length  Sepal.Width Petal.Length
      -0.240       -0.207        0.223        0.524
attr(,"sigma")
Petal.Width
     0.1894
```

The regression coefficients and the residual variance are almost identical with those obtained from `lm()`. The same exericise, now with the $\boldsymbol{\nu}$ and $\mathbf{L}$ parameterisation for exact and interval-censored observations, gives slightly different results for the latter, due to censoring:

```
> ### nu, L for exact observations
> round(coef(MLL, which = "Petal.Width"), 3)

 (Intercept) Sepal.Length  Sepal.Width Petal.Length
      -0.240       -0.207        0.223        0.524
attr(,"sigma")
Petal.Width
     0.1894

> ### nu, L for censored observations
> round(coef(MLi, which = "Petal.Width"), 3)

 (Intercept) Sepal.Length  Sepal.Width Petal.Length
      -0.467       -0.135        0.195        0.492
attr(,"sigma")
Petal.Width
     0.1955
```

The log-likelihood and score function automagically marginalise over dimensions where all observations are $(-\infty, \infty)$. We can simply omit these dimensions from the matrices specified as `obs`, `lower`, and `upper` arguments. Let's say we have four dimensions called A to D and three observations. All observations have $A = (-1, 1)$ and $B = (-\infty, \infty)$, so in fact, the likelihood is given by the marginal distribution of A, C, D.

```
> N <- 3
> J <- 4
> L <- ltMatrices(runif(J * (J + 1) / 2), diag = TRUE, names = LETTERS[1:J])
> Z <- matrix(rnorm(J * N), nrow = J)
> Y <- solve(L, Z)
> lwrA <- matrix(-1, nrow = 1, ncol = N)
> uprA <- matrix(1, nrow = 1, ncol = N)
> rownames(lwrA) <- rownames(uprA) <- "A"
> lwrB <- matrix(-Inf, nrow = 1, ncol = N)
> uprB <- matrix(Inf, nrow = 1, ncol = N)
> rownames(lwrB) <- rownames(uprB) <- "B"
> lwr <- rbind(lwrA, lwrB)
> upr <- rbind(uprA, uprB)
> obs <- Y[rev(LETTERS[3:J]),]    ### change order of dimensions
```

With this data, we first compute the log-likelihood and score functions for the complete data, that is, including the infinite intervals for B.

```
> w <- matrix(runif(1000), nrow = 1, byrow = TRUE)
> lABCD <- logLik(mvnorm(invchol = L), obs = obs, lower = lwr, upper = upr, w = w)
> sABCD <- lLgrad(mvnorm(invchol = L), obs = obs, lower = lwr, upper = upr, w = w)
```

This is (almost) the same as omitting dimension B from the data, but of course not from the model

```
> lACD <- logLik(mvnorm(invchol = L), obs = obs, lower = lwrA, upper = uprA)
> sACD <- lLgrad(mvnorm(invchol = L), obs = obs, lower = lwrA, upper = uprA)
```

We can compare the results

```
> chk(lABCD, lACD)
> nm <- names(sABCD)
> nm <- nm[!nm %in% c("lower", "upper")]
> chk(sABCD[nm], sACD[nm])
```

noting that the scores with respect to the B data in `lower` and `upper` are missing from `sACD`

```
> chk(sABCD$lower["A",,drop = FALSE], sACD$lower)
> chk(sABCD$upper["A",,drop = FALSE], sACD$upper)
> sABCD$lower["B",]        ### zero

[1] 0 0 0

> sABCD$upper["B",]        ### zero

[1] 0 0 0
```

Omitting dimensions might be important because `lpmvnorm` introduced in Chapter 4 does not check if both `lower` and `upper` are infinite and omission thus reduces the dimensionality of the integral we evaluate numerically.

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 10

Reduced Rank Covariance Matrices

We sometimes can write the $J \times J$ covariance matrix as $\mathbf{\Sigma} = \mathbf{B}\mathbf{B}^\top + \mathbf{D}$ where $\mathbf{B}$ is $J \times K$ with $K < J$ and $\mathbf{D}$ is a J-dimensional diagonal matrix. Marsaglia (1963) and Genz and Bretz (2009), in their Chapter 2.3.1, demonstrated that the probability $\mathbb{P}(\mathbf{a} < \mathbf{Y} \leq \mathbf{b} \mid \mathbf{\Sigma})$ can be written as

$$\mathbb{P}(\mathbf{a} < \mathbf{Y} \leq \mathbf{b} \mid \mathbf{\Sigma}) = \int_{[0,1]^K} \prod_{j=1}^{J} \left(\Phi\left(\frac{b_j - \mathbf{B}_{jk}\Phi^{-1}(w_k)}{\sqrt{\mathbf{D}_{jj}}} \right) - \Phi\left(\frac{a_j - \mathbf{B}_{jk}\Phi^{-1}(w_k)}{\sqrt{\mathbf{D}_{jj}}} \right) \right) d\mathbf{w}$$

where the integration takes place with respect to $\mathbf{w} = (w_1, \ldots, w_K)^\top$ from the K- (and not J-) dimensional unit hypercube.

We start implementing low-level functionality for computing log-probabilities for such structures with some book keeping

⟨ *RR input B D* 192 ⟩ ≡

```
stopifnot(!missing(B))
if (!is.matrix(B)) B <- matrix(B, ncol = 1)
J <- nrow(B)
K <- ncol(B)
stopifnot(all(D > 0))
Dsqrt <- sqrt(D)
```

◇

Fragment referenced in 197, 200.

We use $Z = \Phi^{-1}(w)$ and optional weights and compute the products $\mathbf{B}_{jk}\Phi^{-1}(w_k)$ for all j and k with standardisation by the diagonal elements of $\mathbf{D}$.

⟨ *RR input Z, weights* 193 ⟩ ≡

```
stopifnot(nrow(Z) == K)
stopifnot(length(weights) == 1 || length(weights) == ncol(Z))
BZ <- (B / Dsqrt) %*% Z
```

◇

Fragment referenced in 197, 200.

The limits $\mathbf{a}$ and $\mathbf{b}$ are standardised by $\sqrt{\text{diag}(\mathbf{D})}$ and shifted by $\mathbf{B}_{jk}\Phi^{-1}(w_k)/\sqrt{\text{diag}(\mathbf{D})}$.

⟨ *RR limits* 194 ⟩ ≡

```
if (missing(lower)) {
    llim <- matrix(-Inf, nrow = J, ncol = ncol(Z))
} else {
    stopifnot(length(lower) == J)
    llim <- (c(lower) - mean) / Dsqrt - BZ
}
if (missing(upper)) {
    ulim <- matrix(Inf, nrow = J, ncol = ncol(Z))
} else {
    stopifnot(length(upper) == J)
    ulim <- (c(upper) - mean) / Dsqrt - BZ
}
```

◇

Fragment referenced in 197, 200.

Evaluating each factor as the plain difference $\Phi(u_{jm}) - \Phi(l_{jm})$ loses all precision once both limits sit in the same tail: for $\mathbf{D}_{jj}$ small the two probabilities saturate to the same double and the difference is exactly zero, so a single coordinate j annihilates the whole integration point m. We therefore evaluate

$$\lambda_{jm} = \log(\Phi(u_{jm}) - \Phi(l_{jm})) = \log \Phi(u_{jm}) + \log\left(1 - e^{\log \Phi(l_{jm}) - \log \Phi(u_{jm})}\right)$$

Only one tail needs code, because reflection, $\Phi(u) - \Phi(l) = \Phi(-l) - \Phi(-u)$, maps an interval in the upper tail onto one in the lower tail. Mächler (2012)'s guard for $\log(1 - \exp(d))$ switches at $-\log 2$ between the `expm1` and the `log1p` form, neither of which is accurate over the whole range. This was properly taken care of in `pexp(, log.p = TRUE)`, so we reuse this function.

⟨ *RR lognormdiff (AI)* 195 ⟩ ≡

```
.lognormdiff <- function(llim, ulim) {
    ret <- matrix(-Inf, nrow = nrow(ulim), ncol = ncol(ulim))

    up <- llim >= 0                     ### reflect the upper tail onto the lower
    if (any(up)) {
        flip <- llim[up]
        llim[up] <- -ulim[up]
        ulim[up] <- -flip
    }

    tail <- ulim <= 0                   ### interval does not contain zero
    if (any(tail)) {
        logA <- pnorm(ulim[tail], log.p = TRUE)
        d <- pnorm(llim[tail], log.p = TRUE) - logA
        d[is.na(d)] <- 0                ### both limits -Inf: empty interval
        ret[tail] <- logA + pexp(-pmin(d, 0), log.p = TRUE)
                          #^^^^ = log(1 - exp(x)) in a stable way
    }

    if (any(mid <- !tail))              ### interval spans zero: no cancellation
        ret[mid] <- log(pmax(0, pnorm(ulim[mid]) - pnorm(llim[mid])))
    ret
}
```

◇

Fragment referenced in 72.

Summing λ_{jm} over j gives the log-integrant S_m at integration point m. Exponentiating S_m directly would cap the dynamic range at $\exp(-745)$, so we subtract $S^\star = \max_m S_m$ before exponentiating and add it back afterwards. In contrast to `lpmvnorm`, we also allow weights for the summation, such that sparse grids (for example from add-on package **SparseGrid**) can be utilised; negative weights are permitted but then disable the effective sample size diagnostic.

⟨ *RR log inner (AI)* 196 ⟩ ≡

```
lambda <- .lognormdiff(llim, ulim)
### logint is S_m, logintmax is S* in the text
logint <- .colSums(m = J, n = ncol(Z), x = lambda, na.rm = FALSE)
logintmax <- max(logint)
if (is.finite(logintmax)) {
    contrib <- weights * exp(logint - logintmax)
    totcontrib <- sum(contrib)
} else {
    totcontrib <- 0
}
```

◇

Fragment referenced in 197, 200.

Finally, we wrap everything up in a function called `lpRR`

⟨ *lpRR* 197 ⟩ ≡

```
lpRR <- function(lower, upper, mean = 0, B, D = rep(1, nrow(B)),
                 Z, weights = 1 / ncol(Z), log.p = TRUE) {

    ⟨ RR input B D 192 ⟩
    ⟨ RR input Z, weights 193 ⟩
    ⟨ RR limits 194 ⟩

    ⟨ RR log inner (AI) 196 ⟩

    ret <- if (totcontrib > 0) logintmax + log(totcontrib) else -Inf
    if (log.p) return(ret)
    return(exp(ret))
}
```

◇

Fragment referenced in 72.

We test this functionality for dimensions $J = 6$ and $K = 3$. That is, the integration problem reduces from a $J - 1 = 5$ dimensional one in `lpmvnorm` to a three dimensional one in `lpRR`. We first compare the two log-probabilities, computed with high accuracy

```
> J <- 6
> K <- 3
> B <- matrix(rnorm(J * K), nrow = J)
> D <- runif(J)
> S <- tcrossprod(B) + diag(D)
> Linv <- t(chol(S))
> Linv <- ltMatrices(Linv[lower.tri(Linv, diag = TRUE)], diag = TRUE)
> a <- -(2 + runif(J))
> b <- 2 + runif(J)
```

```
> M <- 1e6
> dim(w <- matrix(runif((J - 1) * M), nrow = J - 1, byrow = TRUE))
[1]       5 1000000
> lpmvnorm(lower = a, upper = b, chol = Linv, w = w)
[1] -1.348
> dim(Z <- matrix(rnorm(K * M), nrow = K))
[1]       3 1000000
> lpRR(lower = a, upper = b, B = B, D = D, Z = Z)
[1] -1.35
```

10.1 Score Function (AI assisted)

Only integration points with a finite S_m contribute, so we keep just those before evaluating the score; on those every λ_{jm} is finite too, which is what makes the exponentials below safe.

⟨ *RR keep finite (AI)* 198 ⟩ ≡

```
finite <- is.finite(logint)
contrib <- contrib[finite]
lambda <- lambda[, finite, drop = FALSE]
ulim   <- ulim[, finite, drop = FALSE]
llim   <- llim[, finite, drop = FALSE]
Z      <- Z[, finite, drop = FALSE]
◇
```

Fragment referenced in 200.

The score function with respect to $\mathbf{a}$, $\mathbf{b}$, $\boldsymbol{\mu}$, $\mathbf{B}$, and $\mathbf{D}$ follows from the same quantities. Writing $\pi_m = w_m \exp(S_m - S^\star)/\sum_{m'} w_{m'} \exp(S_{m'} - S^\star)$ for the normalised integration point weights, the scores are sums of π_m against the two ratios $\phi(u_{jm})/(\Phi(u_{jm}) - \Phi(l_{jm}))$ and its counterpart at l_{jm}, each formed as $\exp(\log\phi - \lambda)$ rather than as a quotient – $\log\phi(x) = -x^2/2 - \log(2\pi)/2$ is exact for every x – so the ratio stays accurate even where numerator and denominator both underflow. The $\mathbf{D}$-score additionally multiplies an already-zero weight by u_{jm} or l_{jm}, which is $\pm\infty$ whenever `lower` or `upper` was missing; the two products below are guarded so that $0 \times \infty$ evaluates to its correct limit zero rather than `NaN`.

⟨ *RR score (AI)* 199 ⟩ ≡

```
lphi <- function(x) -x^2 / 2 - log(2 * base::pi) / 2
normw <- contrib / totcontrib   ### pi_m, sums to one
wphiu <- exp(lphi(ulim) - lambda) * rep(normw, each = J)
wphil <- exp(lphi(llim) - lambda) * rep(normw, each = J)

dl <- -rowSums(wphil) / Dsqrt
du <- rowSums(wphiu) / Dsqrt
dm <- -du - dl
dB <- -((wphiu - wphil) %*% t(Z)) / Dsqrt

puu <- wphiu * ulim; puu[!is.finite(ulim)] <- 0
pll <- wphil * llim; pll[!is.finite(llim)] <- 0
dD <- -(rowSums(puu) - rowSums(pll)) / (2 * D)
◇
```

Fragment referenced in 200.

⟨ *slpRR* 200 ⟩ ≡

```
slpRR <- function(lower, upper, mean = 0, B, D = rep(1, nrow(B)),
                  Z, weights = 1 / ncol(Z), log.p = TRUE) {
    ⟨ RR input B D 192 ⟩
    ⟨ RR input Z, weights 193 ⟩
    ⟨ RR limits 194 ⟩
    ⟨ RR log inner (AI) 196 ⟩

    ⟨ RR keep finite (AI) 198 ⟩
    ⟨ RR score (AI) 199 ⟩

    fct <- if (log.p) 1 else exp(logintmax + log(totcontrib))
    list(lower = fct * dl, upper = fct * du, mean = fct * dm,
         B = fct * dB, D = fct * dD)
}
```

◇

Fragment referenced in 72.

We can now compare the gradients for **a**, **b** and the mean in our small example. We cannot expect them to be equal but close

```
> smv <- slpmvnorm(lower = a, upper = b, chol = Linv, w = w)
> sRR <- slpRR(lower = a, upper = b, B = B, D = D, Z = Z)
> chk(c(smv$lower), sRR$lower, tolerance = 1e-2)
> chk(c(smv$upper), sRR$upper, tolerance = 1e-2)
> chk(c(smv$mean), sRR$mean, tolerance = 1e-2 * 2)
```

The gradient with respect to **B** and **D** are finally checked against their numerical approximations

```
> Z <- matrix(rnorm(K * 1000), nrow = K)
> lB <- function(B) lpRR(lower = a, upper = b, B = B, D = D, Z = Z)
> gB <- grad(lB, B)
> sRR <- slpRR(lower = a, upper = b, B = B, D = D, Z = Z)
> chk(gB, c(sRR$B), tolerance = 1e-3)
> lD <- function(D) lpRR(lower = a, upper = b, B = B, D = D, Z = Z)
> gD <- grad(lD, D)
> chk(gD, c(sRR$D), tolerance = 1e-3)
> ### while we are at it, check lower and again
> llwr <- function(a) lpRR(lower = a, upper = b, B = B, D = D, Z = Z)
> glwr <- grad(llwr, a)
> chk(glwr, c(sRR$lower))
> lupr <- function(b) lpRR(lower = a, upper = b, B = B, D = D, Z = Z)
> gupr <- grad(lupr, b)
> chk(gupr, c(sRR$upper))
```

Chapter 11

Package Infrastructure

⟨ *R Header* 201 ⟩ ≡

```
###    Copyright (C) 2022- Torsten Hothorn, 2026-- Lukas Graz
###
###    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 -rn lmvnorm_src.w'
```

◇

Fragment referenced in 1, 72.

⟨ *C Header* 202 ⟩ ≡

```
/*
    Copyright (C) 2022- Torsten Hothorn, 2026-- Lukas Graz

    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 -rn lmvnorm_src.w'
*/
◇
```

Fragment referenced in 2, 73.

Appendix

This document uses the following matrix derivatives for vectors $\mathbf{y} \in \mathbb{R}^J$ and matrices $\mathbf{A} \in \mathbb{R}^{J \times J}$, $\mathbf{B} \in \mathbb{R}^{K \times J}$, and $\mathbf{X} \in \mathbb{R}^{J \times K}$:

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

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

Index

Files

"interface.R" Defined by 166.
"lpmvnorm.c" Defined by 73.
"lpmvnorm.R" Defined by 72.
"ltMatrices.c" Defined by 2.
"ltMatrices.R" Defined by 1.

Fragments

⟨ .lt 4 ⟩ Referenced in 1.
⟨ .subset ltMatrices 16 ⟩ Referenced in 17.
⟨ add diagonal elements 20 ⟩ Referenced in 1.
⟨ argchecks 178 ⟩ Referenced in 180, 191.
⟨ as.ltMatrices 167 ⟩ Referenced in 1.
⟨ assign diagonal elements 21 ⟩ Referenced in 1.
⟨ C Header 202 ⟩ Referenced in 2, 73.
⟨ C length 27 ⟩ Referenced in 29, 31, 34.
⟨ check A argument 46 ⟩ Referenced in 47.
⟨ check and / or set integration weights 97 ⟩ Referenced in 99, 127.
⟨ check C argument 44 ⟩ Referenced in 47.
⟨ check obs 65 ⟩ Referenced in 1.
⟨ check S argument 45 ⟩ Referenced in 47.
⟨ chol 40 ⟩ Referenced in 2.
⟨ chol classes 48 ⟩ Referenced in 51.
⟨ chol scores 100 ⟩ Referenced in 104.
⟨ chol syMatrices 39 ⟩ Referenced in 1.
⟨ Cholesky of precision 98 ⟩ Referenced in 99, 127.
⟨ colSumsdnorm 66 ⟩ Referenced in 2.
⟨ colSumsdnorm ltMatrices 67 ⟩ Referenced in 1.
⟨ compute x 79 ⟩ Referenced in 82, 115, 135, 150.
⟨ compute y 78 ⟩ Referenced in 82, 115.
⟨ compute y (log scale, AI) 132 ⟩ Referenced in 135, 150.
⟨ cond general 61 ⟩ Referenced in 63.
⟨ cond simple 62 ⟩ Referenced in 63.
⟨ conditional 63 ⟩ Referenced in 1.
⟨ convenience functions 51 ⟩ Referenced in 1.
⟨ crossprod ltMatrices 37 ⟩ Referenced in 1.
⟨ crossprod tcrossprod methods 38 ⟩ Referenced in 1.
⟨ D times C 49 ⟩ Referenced in 51.
⟨ deperma 163 ⟩ Referenced in 72.
⟨ deperma indices 162 ⟩ Referenced in 163.
⟨ deperma input checks chol 159 ⟩ Referenced in 163.
⟨ deperma input checks perm 160 ⟩ Referenced in 163.
⟨ deperma input checks schol 161 ⟩ Referenced in 163.

⟨ destandardize 165 ⟩ Referenced in 72.
⟨ diagonal matrix 22 ⟩ Referenced in 1.
⟨ diagonals ltMatrices 19 ⟩ Referenced in 1.
⟨ dim ltMatrices 9 ⟩ Referenced in 1.
⟨ dimensions 90 ⟩ Referenced in 95, 118, 140, 153.
⟨ dimnames ltMatrices 10 ⟩ Referenced in 1.
⟨ dp input checks 155 ⟩ Referenced in 156, 158.
⟨ extract slots 13 ⟩ Referenced in 14, 15, 16, 18, 19, 21, 23, 30.
⟨ increment 83 ⟩ Referenced in 95.
⟨ increment (log scale, AI) 136 ⟩ Referenced in 140.
⟨ init center 93 ⟩ Referenced in 95, 118, 140, 153.
⟨ init dans 117 ⟩ Referenced in 118.
⟨ init logLik loop 77 ⟩ Referenced in 95, 107.
⟨ init logLik loop (log scale, AI) 131 ⟩ Referenced in 140, 143.
⟨ init random seed, reset on exit 96 ⟩ Referenced in 99, 127, 175.
⟨ init score loop 107 ⟩ Referenced in 118.
⟨ init score loop (log scale, AI) 143 ⟩ Referenced in 153.
⟨ initialisation 76 ⟩ Referenced in 95, 118.
⟨ initialisation (log scale, AI) 130 ⟩ Referenced in 140, 153.
⟨ inner logLik loop 82 ⟩ Referenced in 95.
⟨ inner logLik loop (log scale, AI) 135 ⟩ Referenced in 140.
⟨ inner score loop 115 ⟩ Referenced in 118.
⟨ inner score loop (log scale, AI) 150 ⟩ Referenced in 153.
⟨ input checks 74 ⟩ Referenced in 71, 99, 127.
⟨ invchol 42 ⟩ Referenced in 2.
⟨ invchol syMatrices 41 ⟩ Referenced in 1.
⟨ invchol workhorse 43 ⟩ Referenced in 2.
⟨ invcholmean score 125 ⟩ Referenced in 127.
⟨ invcholmean2mean 173 ⟩ Referenced in 166.
⟨ is.ltMatrices 12 ⟩ Referenced in 1.
⟨ kronecker vec trick 47 ⟩ Referenced in 1.
⟨ L times D 50 ⟩ Referenced in 51.
⟨ lapack options 24 ⟩ Referenced in 29, 31, 32.
⟨ ldmvnorm 64 ⟩ Referenced in 72.
⟨ ldmvnorm chol 68 ⟩ Referenced in 64.
⟨ ldmvnorm invchol 69 ⟩ Referenced in 64.
⟨ ldpmvnorm 156 ⟩ Referenced in 72.
⟨ lLgrad chol 181 ⟩ Referenced in 191.
⟨ lLgrad deperma 184 ⟩ Referenced in 181.
⟨ lLgrad destandarized 185 ⟩ Referenced in 181.
⟨ lLgrad diagonals 186 ⟩ Referenced in 181.
⟨ lLgrad invchol 190 ⟩ Referenced in 191.
⟨ lLgrad invchol marginalisation 188 ⟩ Referenced in 190.
⟨ lLgrad invchol post 189 ⟩ Referenced in 190.
⟨ lLgrad marginalisation 183 ⟩ Referenced in 181.
⟨ lLgrad mean 182 ⟩ Referenced in 181.
⟨ lLgrad return 187 ⟩ Referenced in 181.
⟨ logdet 34 ⟩ Referenced in 2.
⟨ logdet ltMatrices 35 ⟩ Referenced in 1.
⟨ logLik standardize 179 ⟩ Referenced in 180.
⟨ lognormdiff (log scale, AI) 129 ⟩ Referenced in 73.
⟨ lower scores 102 ⟩ Referenced in 104.
⟨ lower triangular elements 18 ⟩ Referenced in 1.
⟨ lpmvnorm 99 ⟩ Referenced in 72.
⟨ lpmvnormR 71 ⟩ Not referenced.
⟨ lpRR 197 ⟩ Referenced in 72.
⟨ ltMatrices 7 ⟩ Referenced in 1.
⟨ ltMatrices dim 3 ⟩ Referenced in 7.

⟨ ltMatrices input 6 ⟩ Referenced in 7.
⟨ ltMatrices names 5 ⟩ Referenced in 7.
⟨ marginal 60 ⟩ Referenced in 1.
⟨ mc input checks 59 ⟩ Referenced in 60, 63.
⟨ mean scores 101 ⟩ Referenced in 104.
⟨ mean2invcholmean 172 ⟩ Referenced in 166.
⟨ move on 85 ⟩ Referenced in 95, 118, 140, 153.
⟨ mult 29 ⟩ Referenced in 2.
⟨ mult ltMatrices 23 ⟩ Referenced in 1.
⟨ mult syMatrices 30 ⟩ Referenced in 1.
⟨ mult workhorse 28 ⟩ Referenced in 29.
⟨ mvnorm 171 ⟩ Referenced in 166.
⟨ mvnorm chol invchol 168 ⟩ Referenced in 171.
⟨ mvnorm condDist 177 ⟩ Referenced in 166.
⟨ mvnorm invcholmean 170 ⟩ Referenced in 171.
⟨ mvnorm lLgrad 191 ⟩ Referenced in 166.
⟨ mvnorm logLik 180 ⟩ Referenced in 166.
⟨ mvnorm margDist 176 ⟩ Referenced in 166.
⟨ mvnorm marginal means 56 ⟩ Referenced in 166.
⟨ mvnorm mean 169 ⟩ Referenced in 171.
⟨ mvnorm methods 174 ⟩ Referenced in 166.
⟨ mvnorm regression coefs 58 ⟩ Referenced in 166.
⟨ mvnorm simulate 175 ⟩ Referenced in 166.
⟨ mvnorm vcov 57 ⟩ Referenced in 166.
⟨ names ltMatrices 11 ⟩ Referenced in 1.
⟨ new score means, lower and upper 112 ⟩ Referenced in 115.
⟨ new score means, lower and upper (log scale, AI) 147 ⟩ Referenced in 150.
⟨ normalise scores (log scale, AI) 152 ⟩ Referenced in 153.
⟨ output 84 ⟩ Referenced in 95.
⟨ output (log scale, AI) 137 ⟩ Referenced in 140.
⟨ perm 55 ⟩ Referenced in 1.
⟨ perm checks 53 ⟩ Referenced in 54.
⟨ perm chol invchol 54 ⟩ Referenced in 55.
⟨ perm generic 52 ⟩ Referenced in 55.
⟨ pnorm 88 ⟩ Referenced in 95, 118.
⟨ pnorm fast 86 ⟩ Not referenced.
⟨ pnorm slow 87 ⟩ Not referenced.
⟨ post differentiate chol score 122 ⟩ Referenced in 127.
⟨ post differentiate invchol score 123 ⟩ Referenced in 127.
⟨ post differentiate lower score 120 ⟩ Referenced in 127.
⟨ post differentiate mean score 119 ⟩ Referenced in 127.
⟨ post differentiate upper score 121 ⟩ Referenced in 127.
⟨ post process score 124 ⟩ Referenced in 127.
⟨ print ltMatrices 14 ⟩ Referenced in 1.
⟨ R Header 201 ⟩ Referenced in 1, 72.
⟨ R lpmvnorm 95 ⟩ Not referenced.
⟨ R lpmvnorm (log scale, AI) 140 ⟩ Referenced in 73.
⟨ R normalise scores 126 ⟩ Not referenced.
⟨ R normalise scores (log scale, AI) 154 ⟩ Referenced in 127.
⟨ R slpmvnorm 118 ⟩ Not referenced.
⟨ R slpmvnorm (log scale, AI) 153 ⟩ Referenced in 73.
⟨ R slpmvnorm variables 94 ⟩ Referenced in 95, 118.
⟨ R slpmvnorm variables (log scale, AI) 139 ⟩ Referenced in 140, 153.
⟨ RC input 26 ⟩ Referenced in 29, 31, 32, 34.
⟨ reorder ltMatrices 15 ⟩ Referenced in 1.
⟨ row column 25 ⟩ Referenced in 29.
⟨ RR input B D 192 ⟩ Referenced in 197, 200.
⟨ RR input Z, weights 193 ⟩ Referenced in 197, 200.

⟨ RR keep finite (AI) 198 ⟩ Referenced in 200.
⟨ RR limits 194 ⟩ Referenced in 197, 200.
⟨ RR log inner (AI) 196 ⟩ Referenced in 197, 200.
⟨ RR lognormdiff (AI) 195 ⟩ Referenced in 72.
⟨ RR score (AI) 199 ⟩ Referenced in 200.
⟨ score a, b 106 ⟩ Referenced in 107, 118.
⟨ score a, b (log scale, AI) 142 ⟩ Referenced in 143.
⟨ score c11 105 ⟩ Referenced in 107, 118.
⟨ score c11 (log scale, AI) 141 ⟩ Referenced in 143.
⟨ score output 116 ⟩ Referenced in 118.
⟨ score output (log scale, AI) 151 ⟩ Referenced in 153.
⟨ score output object 104 ⟩ Referenced in 118, 153.
⟨ score wrt new chol diagonal 111 ⟩ Referenced in 115.
⟨ score wrt new chol diagonal (log scale, AI) 146 ⟩ Referenced in 150.
⟨ score wrt new chol off-diagonals 110 ⟩ Referenced in 115.
⟨ score wrt new chol off-diagonals (log scale, AI) 145 ⟩ Referenced in 150.
⟨ setup return object 91 ⟩ Referenced in 95, 140.
⟨ sldmvnorm 70 ⟩ Referenced in 72.
⟨ sldpmvnorm 158 ⟩ Referenced in 72.
⟨ sldpmvnorm invchol 157 ⟩ Referenced in 158.
⟨ slpmvnorm 127 ⟩ Referenced in 72.
⟨ slpRR 200 ⟩ Referenced in 72.
⟨ solve 31 ⟩ Referenced in 2.
⟨ solve C 32 ⟩ Referenced in 2.
⟨ solve ltMatrices 33 ⟩ Referenced in 1.
⟨ standardise 75 ⟩ Referenced in 99, 127.
⟨ standardize 164 ⟩ Referenced in 72.
⟨ subset ltMatrices 17 ⟩ Referenced in 1.
⟨ syMatrices 8 ⟩ Referenced in 1.
⟨ tcrossprod ltMatrices 36 ⟩ Referenced in 1.
⟨ tol+fast 128 ⟩ Referenced in 99, 127.
⟨ univariate problem 92 ⟩ Referenced in 95.
⟨ univariate problem (log scale, AI) 138 ⟩ Referenced in 140, 153.
⟨ update d, e 80 ⟩ Referenced in 82, 115.
⟨ update d, e (log scale, AI) 133 ⟩ Referenced in 135, 150.
⟨ update f 81 ⟩ Referenced in 82, 115.
⟨ update f (log scale, AI) 134 ⟩ Referenced in 135, 150.
⟨ update score for chol 113 ⟩ Referenced in 115.
⟨ update score for chol (log scale, AI) 148 ⟩ Referenced in 150.
⟨ update score means, lower and upper 114 ⟩ Referenced in 115.
⟨ update score means, lower and upper (log scale, AI) 149 ⟩ Referenced in 150.
⟨ update yp for chol 108 ⟩ Referenced in 115.
⟨ update yp for chol (log scale, AI) 144 ⟩ Referenced in 150.
⟨ update yp for means, lower and upper 109 ⟩ Referenced in 115, 150.
⟨ upper scores 103 ⟩ Referenced in 104.
⟨ W length 89 ⟩ Referenced in 95, 118, 140, 153.

Bibliography

Barratt S, Boyd S (2023). "Covariance Prediction via Convex Optimization." *Optimization and Engineering*, **24**(3), 2045–2078. doi10.1007/s11081-022-09765-w. 68, 135

Genz A (1992). "Numerical Computation of Multivariate Normal Probabilities." *Journal of Computational and Graphical Statistics*, **1**(2), 141–149. doi10.1080/10618600.1992.10477010. 1, 75

Genz A, Bretz F (2002). "Methods for the Computation of Multivariate t Probabilities." *Journal of Computational and Graphical Statistics*, **11**(4), 950–971. doi10.1198/106186002394. 1, 74

Genz A, Bretz F (2009). *Computation of Multivariate Normal and t Probabilities*. Lecture Notes in Statistics. Springer-Verlag, Heidelberg, Germany. ISBN 978-3-642-01688-2. 161

Hothorn T (2024). "On Nonparanormal Likelihoods." *Technical report*, arXiv 2408.17346. doi10.48550/arXiv.2408.17346. 1

Klein N, Hothorn T, Barbanti L, Kneib T (2022). "Multivariate Conditional Transformation Models." *Scandinavian Journal of Statistics*, **49**, 116–142. doi10.1111/sjos.12501. 1

Mächler M (2012). *Accurately Computing* $\log(1 - \exp(-|a|))$ *— Assessed by the Rmpfr Package*. doi10.32614/CRAN.package.Rmpfr. R package vignette, URL https://CRAN.R-project.org/package=Rmpfr/vignettes/log1mexp-note.pdf. 99, 162

Marsaglia G (1963). "Expressing the Normal Distribution with Covariance Matrix $A+B$ in Terms of One with Covariance Matrix A." *Biometrika*, **50**(3–4), 535–538. 161

Matić I, Radoičić R, Stefanica D (2018). "A Sharp Pólya-Based Approximation to the Normal CDF." *Applied Mathematics and Computation*, **322**, 111–122. doi10.2139/ssrn.2842681. 80

Prékopa A (1973). "On Logarithmic Concave Measures and Functions." *Acta Scientiarum Mathematicarum*, **34**, 335–343. 135