The **libcoin** Package

Torsten Hothorn
Universität Zürich

September 27, 2023

Contents

Licence

Chapter 1

Introduction

The **libcoin** package implements a generic framework for permutation tests. We assume that we are provided with n observations

$$(\mathbf{Y}_i, \mathbf{X}_i, w_i, \text{block}_i), \quad i = 1, \dots, N.$$

The variables $\mathbf{Y}$ and $\mathbf{X}$ from sample spaces $\mathcal{Y}$ and $\mathcal{X}$ may be measured at arbitrary scales and may be multivariate as well. In addition to those measurements, case weights $w_i \in \mathbb{N}$ and a factor $\text{block}_i \in \{1, \dots, B\}$ coding for B independent blocks may be available. We are interested in testing the null hypothesis of independence of $\mathbf{Y}$ and $\mathbf{X}$

$$H_0 : D(\mathbf{Y} \mid \mathbf{X}) = D(\mathbf{Y})$$

against arbitrary alternatives. Strasser and Weber (1999) suggest to derive scalar test statistics for testing H_0 from multivariate linear statistics of a specific linear form. Let $\mathcal{A} \subseteq \{1, \dots, N\}$ denote some subset of the observation numbers and consider the linear statistic

$$\mathbf{T}(\mathcal{A}) = \text{vec}\left(\sum_{i \in \mathcal{A}} w_i g(\mathbf{X}_i) h(\mathbf{Y}_i, \{\mathbf{Y}_i \mid i \in \mathcal{A}\})^\top\right) \in \mathbb{R}^{PQ}. \tag{1.1}$$

Here, $g : \mathcal{X} \to \mathbb{R}^P$ is a transformation of $\mathbf{X}$ known as the *regression function* and $h : \mathcal{Y} \times \mathcal{Y}^n \to \mathbb{R}^Q$ is a transformation of $\mathbf{Y}$ known as the *influence function*, where the latter may depend on $\mathbf{Y}_i$ for $i \in \mathcal{A}$ in a permutation symmetric way. We will give specific examples on how to choose g and h later on.

With $\mathbf{x}_i = g(\mathbf{X}_i) \in \mathbb{R}^P$ and $\mathbf{y}_i = h(\mathbf{Y}_i, \{\mathbf{Y}_i, i \in \mathcal{A}\}) \in \mathbb{R}^Q$ we write

$$\mathbf{T}(\mathcal{A}) = \text{vec}\left(\sum_{i \in \mathcal{A}} w_i \mathbf{x}_i \mathbf{y}_i^\top\right) \in \mathbb{R}^{PQ}. \tag{1.2}$$

The **libcoin** package doesn't handle neither g nor h, this is the job of **coin** and we therefore continue with $\mathbf{x}_i$ and $\mathbf{y}_i$.

The distribution of $\mathbf{T}$ depends on the joint distribution of $\mathbf{Y}$ and $\mathbf{X}$, which is unknown under almost all practical circumstances. At least under the null hypothesis one can dispose of this dependency by fixing $\mathbf{X}_i, i \in \mathcal{A}$ and conditioning on all possible permutations $S(\mathcal{A})$ of the responses $\mathbf{Y}_i, i \in \mathcal{A}$. This principle leads to test procedures known as *permutation tests*. The conditional expectation $\boldsymbol{\mu}(\mathcal{A}) \in \mathbb{R}^{PQ}$ and covariance $\boldsymbol{\Sigma}(\mathcal{A}) \in \mathbb{R}^{PQ \times PQ}$ of $\mathbf{T}$ under H_0 given all permutations $\sigma \in S(\mathcal{A})$ of the responses are derived by Strasser and Weber (1999):

$$\begin{aligned}
\boldsymbol{\mu}(\mathcal{A}) &= \mathbb{E}(\mathbf{T}(\mathcal{A}) \mid S(\mathcal{A})) = \text{vec}\left(\left(\sum_{i \in \mathcal{A}} w_i \mathbf{x}_i\right) \mathbb{E}(h \mid S(\mathcal{A}))^\top\right), \\
\boldsymbol{\Sigma}(\mathcal{A}) &= \mathbb{V}(\mathbf{T}(\mathcal{A}) \mid S(\mathcal{A})) \\
&= \frac{\mathbf{w}_\centerdot}{\mathbf{w}_\centerdot(\mathcal{A}) - 1} \mathbb{V}(h \mid S(\mathcal{A})) \otimes \left(\sum_{i \in \mathcal{A}} w_i \mathbf{x}_i \otimes w_i \mathbf{x}_i^\top\right) \qquad (1.3) \\
&- \frac{1}{\mathbf{w}_\centerdot(\mathcal{A}) - 1} \mathbb{V}(h \mid S(\mathcal{A})) \otimes \left(\sum_{i \in \mathcal{A}} w_i \mathbf{x}_i\right) \otimes \left(\sum_{i \in \mathcal{A}} w_i \mathbf{x}_i\right)^\top
\end{aligned}$$

where $\mathbf{w}_\bullet(\mathcal{A}) = \sum_{i \in \mathcal{A}} w_i$ denotes the sum of the case weights, and $\otimes$ is the Kronecker product. The conditional expectation of the influence function is

$$\mathbb{E}(h \mid S(\mathcal{A})) = \mathbf{w}_\bullet(\mathcal{A})^{-1} \sum_{i \in \mathcal{A}} w_i \mathbf{y}_i \in \mathbb{R}^Q$$

with corresponding $Q \times Q$ covariance matrix

$$\mathbb{V}(h \mid S(\mathcal{A})) = \mathbf{w}_\bullet(\mathcal{A})^{-1} \sum_{i \in \mathcal{A}} w_i \left(\mathbf{y}_i - \mathbb{E}(h \mid S(\mathcal{A}))\right) \left(\mathbf{y}_i - \mathbb{E}(h \mid S(\mathcal{A}))\right)^\top .$$

With $A_b = \{i \mid \text{block}_i = b\}$ we get $\mathbf{T} = \sum_{b=1}^B T(\mathcal{A}_b)$, $\boldsymbol{\mu} = \sum_{b=1}^B \boldsymbol{\mu}(\mathcal{A}_b)$ and $\boldsymbol{\Sigma} = \sum_{b=1}^B \boldsymbol{\Sigma}(\mathcal{A}_b)$.

Having the conditional expectation and covariance at hand we are able to standardize a linear statistic $\mathbf{T} \in \mathbb{R}^{PQ}$ of the form (1.2). Univariate test statistics c mapping an observed linear statistic $\mathbf{t} \in \mathbb{R}^{PQ}$ into the real line can be of arbitrary form. An obvious choice is the maximum of the absolute values of the standardized linear statistic

$$c_\text{max}(\mathbf{t}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \max \left| \frac{\mathbf{t} - \boldsymbol{\mu}}{\text{diag}(\boldsymbol{\Sigma})^{1/2}} \right|$$

utilizing the conditional expectation $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$. The application of a quadratic form $c_\text{quad}(\mathbf{t}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (\mathbf{t} - \boldsymbol{\mu}) \boldsymbol{\Sigma}^+ (\mathbf{t} - \boldsymbol{\mu})^\top$ is one alternative, although computationally more expensive because the Moore-Penrose inverse $\boldsymbol{\Sigma}^+$ of $\boldsymbol{\Sigma}$ is involved.

The definition of one- and two-sided p-values used for the computations in the **libcoin** package is

$$\begin{array}{rcll} P(c(\mathbf{T}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) & \le & c(\mathbf{t}, \boldsymbol{\mu}, \boldsymbol{\Sigma})) & \text{(less)} \\ P(c(\mathbf{T}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) & \ge & c(\mathbf{t}, \boldsymbol{\mu}, \boldsymbol{\Sigma})) & \text{(greater)} \\ P(|c(\mathbf{T}, \boldsymbol{\mu}, \boldsymbol{\Sigma})| & \le & |c(\mathbf{t}, \boldsymbol{\mu}, \boldsymbol{\Sigma})|) & \text{(two-sided).} \end{array}$$

Note that for quadratic forms only two-sided p-values are available and that in the one-sided case maximum type test statistics are replaced by

$$\min \left(\frac{\mathbf{t} - \boldsymbol{\mu}}{\text{diag}(\boldsymbol{\Sigma})^{1/2}} \right) \quad \text{(less) and } \max \left(\frac{\mathbf{t} - \boldsymbol{\mu}}{\text{diag}(\boldsymbol{\Sigma})^{1/2}} \right) \quad \text{(greater).}$$

This single source file implements and documents the **libcoin** package following the literate programming paradigm. The keynote lecture on literate programming by Donald E. Knuth given at useR! 2016 in Stanford very much motivated this little experiment.

Chapter 2

R Code

2.1 R User Interface

`"libcoin.R"` 3a≡

```
⟨ R Header 154a ⟩
⟨ LinStatExpCov 4 ⟩
⟨ LinStatExpCov1d 6 ⟩
⟨ LinStatExpCov2d 8 ⟩
⟨ vcov LinStatExpCov 10 ⟩
⟨ doTest 12 ⟩
⟨ Contrasts 14 ⟩
◇
```

The **libcoin** package implements two R functions, `LinStatExpCov()` and `doTest()` for the computation of linear statistics, their expectation and covariance as well as for the computation of test statistics and p-values. There are two interfaces: One (labelled “1d”) when the data is available as matrices `X` and `Y`, both with the same number of rows N. The second interface (labelled “2d”) handles the case when the data is available in aggregated form; details will be explained later.

⟨ *LinStatExpCov Prototype* 3b ⟩ ≡

```
(X, Y, ix = NULL, iy = NULL, weights = integer(0),
 subset = integer(0), block = integer(0), checkNAs = TRUE,
 varonly = FALSE, nresample = 0, standardise = FALSE,
 tol = sqrt(.Machine$double.eps))◇
```

Fragment referenced in 4, 17.
Uses: `block` 26f, 27b, `subset` 26ade, `weights` 25b.

⟨ *LinStatExpCov* 4 ⟩ ≡

```
LinStatExpCov <-
function⟨ LinStatExpCov Prototype 3b ⟩
{
    if (missing(X) && !is.null(ix) && is.null(iy)) {
        X <- ix
        ix <- NULL
    }

    if (missing(X)) X <- integer(0)

    ## <FIXME> for the time being only!!! </FIXME>
##    if (length(subset) > 0) subset <- sort(subset)

    if (is.null(ix) && is.null(iy))
        .LinStatExpCov1d(X = X, Y = Y,
                         weights = weights, subset = subset,
                         block = block, checkNAs = checkNAs,
                         varonly = varonly, nresample = nresample,
                         standardise = standardise, tol = tol)
    else if (!is.null(ix) && !is.null(iy))
        .LinStatExpCov2d(X = X, Y = Y, ix = ix, iy = iy,
                         weights = weights, subset = subset,
                         block = block, checkNAs = checkNAs,
                         varonly = varonly, nresample = nresample,
                         standardise = standardise, tol = tol)
    else
        stop("incorrect call to ", sQuote("LinStatExpCov()"))
}
◇
```

Fragment referenced in 3a.
Uses: `block` 26f, 27b, `subset` 26ade, `weights` 25b, `weights,` 25cd.

2.1.1 One-Dimensional Case ("1d")

We assume that $\mathbf{x}_i$ and $\mathbf{y}_i$ for $i = 1, \dots, N$ are available as numeric matrices `X` and `Y` with N rows as well as P and Q columns, respectively. The special case of a dummy matrix `X` with P columns can also be represented by a factor at P levels. The vector of case weights `weights` can be stored as `integer` or `double` (possibly resulting from an aggregation of $N >$ `INT_MAX` observations). The subset vector `subset` may contain the elements $1, \dots, N$ as `integer` or `double` (for $N >$ `INT_MAX`) and can be longer than N. The `subset` vector MUST be sorted. `block` is a factor at B levels of length N.

⟨ *Check weights, subset, block* 5a ⟩ ≡

```
if (is.null(weights)) weights <- integer(0)

if (length(weights) > 0) {
    if (!((N == length(weights)) && all(weights >= 0)))
        stop("incorrect weights")
    if (checkNAs) stopifnot(!anyNA(weights))
}

if (is.null(subset)) subset <- integer(0)

if (length(subset) > 0 && checkNAs) {
    rs <- range(subset)
    if (anyNA(rs)) stop("no missing values allowed in subset")
    if (!((rs[2] <= N) && (rs[1] >= 1L)))
        stop("incorrect subset")
}

if (is.null(block)) block <- integer(0)

if (length(block) > 0) {
    if (!((N == length(block)) && is.factor(block)))
        stop("incorrect block")
    if (checkNAs) stopifnot(!anyNA(block))
}
◇
```

Fragment referenced in 6, 8, 15b.
Uses: `block` 26f, 27b, `N` 23bc, `subset` 26ade, `weights` 25b.

Missing values are only allowed in `X` and `Y`, all other vectors must not contain `NA`s. Missing values are dealt with by excluding the corresponding observations from the subset vector.

⟨ *Handle Missing Values* 5b ⟩ ≡

```
ms <- !complete.cases(X, Y)
if (all(ms))
    stop("all observations are missing")
if (any(ms)) {
    if (length(subset) > 0) {
        if (all(subset %in% which(ms)))
            stop("all observations are missing")
        subset <- subset[!(subset %in% which(ms))]
    } else {
        subset <- seq_len(N)[-which(ms)]
    }
}
◇
```

Fragment referenced in 6.
Uses: `N` 23bc, `subset` 26ade.

The logical argument `varonly` triggers the computation of the diagonal elements of the covariance matrix $\mathbf{\Sigma}$ only. `nresample` permuted linear statistics under the null hypothesis H_0 are returned on the original and standardised scale (the latter only when `standardise` is `TRUE`). Variances smaller than `tol` are treated as being zero.

⟨ *LinStatExpCov1d* 6 ⟩ ≡

```
.LinStatExpCov1d <-
function(X, Y, weights = integer(0), subset = integer(0), block = integer(0),
         checkNAs = TRUE, varonly = FALSE, nresample = 0, standardise = FALSE,
         tol = sqrt(.Machine$double.eps))
{
    if (NROW(X) != NROW(Y))
        stop("dimensions of X and Y don't match")
    N <- NROW(X)

    if (is.integer(X)) {
        if (is.null(attr(X, "levels")) || checkNAs) {
            rg <- range(X)
            if (anyNA(rg))
                stop("no missing values allowed in X")
            stopifnot(rg[1] > 0) # no missing values allowed here!
            if (is.null(attr(X, "levels")))
                attr(X, "levels") <- seq_len(rg[2])
        }
    }

    if (is.factor(X) && checkNAs)
        stopifnot(!anyNA(X))

    ⟨ Check weights, subset, block 5a ⟩

    if (checkNAs) {
        ⟨ Handle Missing Values 5b ⟩
    }

    ret <- .Call(R_ExpectationCovarianceStatistic, X, Y, weights, subset,
                 block, as.integer(varonly), as.double(tol))
    ret$varonly <- as.logical(ret$varonly)
    ret$Xfactor <- as.logical(ret$Xfactor)
    if (nresample > 0) {
        ret$PermutedLinearStatistic <-
            .Call(R_PermutedLinearStatistic, X, Y, weights, subset,
                  block, as.double(nresample))
        if (standardise)
            ret$StandardisedPermutedLinearStatistic <-
                .Call(R_StandardisePermutedLinearStatistic, ret)
    }
    class(ret) <- c("LinStatExpCov1d", "LinStatExpCov")
    ret
}
◇
```

Fragment referenced in 3a.
Uses: `block` 26f, 27b, `N` 23bc, `NROW` 130b, `R_ExpectationCovarianceStatistic` 31a, `R_PermutedLinearStatistic` 37, `subset` 26ade, `weights` 25b, `weights,` 25cd.

Here is a simple example. We have five groups and a uniform outcome (rounded to one digit) and want to test independence of group membership and outcome. The simplest way is to set-up the dummy matrix explicitly:

```
> isequal <-
+ function(a, b) {
+     attributes(a) <- NULL
+     attributes(b) <- NULL
+     if (!isTRUE(all.equal(a, b))) {
+         print(a, digits = 10)
+         print(b, digits = 10)
+         FALSE
+     } else
+         TRUE
+ }
> library("libcoin")
> set.seed(290875)
> x <- gl(5, 20)
> y <- round(runif(length(x)), 1)
> ls1 <- LinStatExpCov(X = model.matrix(~ x - 1), Y = matrix(y, ncol = 1))
> ls1$LinearStatistic

[1]  8.8  9.5 10.3  9.8 10.5

> tapply(y, x, sum)

   1    2    3    4    5
 8.8  9.5 10.3  9.8 10.5
```

The linear statistic is simply the sum of the response in each group. Alternatively, we can compute the same object without setting-up the dummy matrix:

```
> ls2 <- LinStatExpCov(X = x, Y = matrix(y, ncol = 1))
> all.equal(ls1[-grep("Xfactor", names(ls1))],
+           ls2[-grep("Xfactor", names(ls2))])

[1] TRUE
```

The results are identical, except for a logical indicating that a factor was used to represent the dummy matrix `X`.

2.1.2 Two-Dimensional Case ("2d")

Sometimes the data takes only a few unique values and considerable computational speedups can be achieved taking this information into account. Let `ix` denote an integer vector with elements $0, \dots, L_x$ of length N and `iy` an integer vector with elements $0, \dots, L_y$, also of length N. The matrix `X` is now of dimension $(L_x + 1) \times P$ and the matrix `Y` of dimension $(L_y + 1) \times Q$. The combination of `X` and `ix` means that the ith observation corresponds to the row `X[ix[i] + 1,]`. This looks cumbersome in R notation but is a very efficient way of dealing with missing values at C level. By convention, elements of `ix` being zero denote a missing value (`NA`s are not allowed in `ix` and `iy`). Thus, the first row of `X` corresponds to a missing value. If the first row is simply zero, missing values do not contribute to any of the sums computed later. Even more important is the fact that all entities, such as linear statistics etc., can be computed from the two-way tabulation (therefore the abbrevation "2d") over the N elements of `ix` and `iy`. Once such a table was computed, the remaining computations can be performed in dimension $L_x \times L_y$, typically much smaller than N.

⟨ *LinStatExpCov2d* 8 ⟩ ≡

```
.LinStatExpCov2d <-
function(X = numeric(0), Y, ix, iy, weights = integer(0), subset = integer(0),
         block = integer(0), checkNAs = TRUE, varonly = FALSE, nresample = 0,
         standardise = FALSE, tol = sqrt(.Machine$double.eps))
{
    IF <- function(x) is.integer(x) || is.factor(x)

    if (!((length(ix) == length(iy)) && IF(ix) && IF(iy)))
        stop("incorrect ix and/or iy")
    N <- length(ix)

    ⟨ Check ix 9a ⟩

    ⟨ Check iy 9b ⟩

    if (length(X) > 0) {
        if (!(NROW(X) == (length(attr(ix, "levels")) + 1) &&
              all(complete.cases(X))))
            stop("incorrect X")
    }

    if (!(NROW(Y) == (length(attr(iy, "levels")) + 1) &&
          all(complete.cases(Y))))
        stop("incorrect Y")

    ⟨ Check weights, subset, block 5a ⟩

    ret <- .Call(R_ExpectationCovarianceStatistic_2d, X, ix, Y, iy,
                 weights, subset, block, as.integer(varonly), as.double(tol))
    ret$varonly <- as.logical(ret$varonly)
    ret$Xfactor <- as.logical(ret$Xfactor)
    if (nresample > 0) {
        ret$PermutedLinearStatistic <-
            .Call(R_PermutedLinearStatistic_2d, X, ix, Y, iy, block, nresample, ret$Table)
        if (standardise)
            ret$StandardisedPermutedLinearStatistic <-
                .Call(R_StandardisePermutedLinearStatistic, ret)
    }
    class(ret) <- c("LinStatExpCov2d", "LinStatExpCov")
    ret
}
◇
```

Fragment referenced in 3a.
Uses: `block` 26f, 27b, `N` 23bc, `NROW` 130b, `R_ExpectationCovarianceStatistic_2d` 41a, `R_PermutedLinearStatistic_2d` 48, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc.

`ix` and `iy` can be factors but without any missing values

⟨ *Check ix* 9a ⟩ ≡

```
if (is.null(attr(ix, "levels"))) {
    rg <- range(ix)
    if (anyNA(rg))
        stop("no missing values allowed in ix")
    stopifnot(rg[1] >= 0)
    attr(ix, "levels") <- seq_len(rg[2])
} else {
    ## lev can be data.frame (see inum::inum)
    lev <- attr(ix, "levels")
    if (!is.vector(lev)) lev <- seq_len(NROW(lev))
    attr(ix, "levels") <- lev
    if (checkNAs) stopifnot(!anyNA(ix))
}
◇
```

Fragment referenced in 8, 15b.
Uses: `NROW` 130b.

⟨ *Check iy* 9b ⟩ ≡

```
if (is.null(attr(iy, "levels"))) {
    rg <- range(iy)
    if (anyNA(rg))
        stop("no missing values allowed in iy")
    stopifnot(rg[1] >= 0)
    attr(iy, "levels") <- seq_len(rg[2])
} else {
    ## lev can be data.frame (see inum::inum)
    lev <- attr(iy, "levels")
    if (!is.vector(lev)) lev <- seq_len(NROW(lev))
    attr(iy, "levels") <- lev
    if (checkNAs) stopifnot(!anyNA(iy))
}
◇
```

Fragment referenced in 8, 15b.
Uses: `NROW` 130b.

In our small example, we can set-up the data in the following way

```
> X <- rbind(0, diag(nlevels(x)))
> ix <- unclass(x)
> ylev <- sort(unique(y))
> Y <- rbind(0, matrix(ylev, ncol = 1))
> iy <- .bincode(y, breaks = c(-Inf, ylev, Inf))
> ls3 <- LinStatExpCov(X = X, ix = ix, Y = Y, iy = iy)
> all.equal(ls1[-grep("Table", names(ls1))],
+           ls3[-grep("Table", names(ls3))])

[1] TRUE

> ### works also with factors
> ls3 <- LinStatExpCov(X = X, ix = factor(ix), Y = Y, iy = factor(iy))
> all.equal(ls1[-grep("Table", names(ls1))],
+           ls3[-grep("Table", names(ls3))])

[1] TRUE
```

Similar to the one-dimensional case, we can also omit the `X` matrix here

```
> ls4 <- LinStatExpCov(ix = ix, Y = Y, iy = iy)
> all.equal(ls3[-grep("Xfactor", names(ls3))],
+           ls4[-grep("Xfactor", names(ls4))])
```

```
[1] TRUE
```

It is important to note that all computations are based on the tabulations

```
> ls3$Table
```

```
, , 1

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
[1,]    0    0    0    0    0    0    0    0    0     0     0     0
[2,]    0    0    4    4    1    2    3    0    1     2     3     0
[3,]    0    2    2    1    2    2    5    0    1     1     3     1
[4,]    0    1    1    4    0    1    5    2    0     2     3     1
[5,]    0    0    2    2    4    2    2    1    3     2     1     1
[6,]    0    1    3    1    1    1    2    2    2     6     1     0
```

```
> xtabs(~ ix + iy)
```

```
   iy
ix  1 2 3 4 5 6 7 8 9 10 11
  1 0 4 4 1 2 3 0 1 2  3  0
  2 2 2 1 2 2 5 0 1 1  3  1
  3 1 1 4 0 1 5 2 0 2  3  1
  4 0 2 2 4 2 2 1 3 2  1  1
  5 1 3 1 1 1 2 2 2 6  1  0
```

where the former would record missing values in the first row / column.

2.1.3 Methods and Tests

Objects of class `"LinStatExpCov"` returned by `LinStatExpCov()` contain the symmetric covariance matrix as a vector of the lower triangular elements. The `vcov` method allows to extract the full covariance matrix from such an object.

⟨ *vcov LinStatExpCov* 10 ⟩ ≡

```
vcov.LinStatExpCov <-
function(object, ...)
{
    if (object$varonly)
        stop("cannot extract covariance matrix")
    drop(.Call(R_unpack_sym, object$Covariance, NULL, 0L))
}
```

◇

Fragment referenced in 3a.
Uses: `R_unpack_sym` 139.

```
> ls1$Covariance
```

```
 [1]  1.3572364 -0.3393091 -0.3393091 -0.3393091 -0.3393091  1.3572364
 [7] -0.3393091 -0.3393091 -0.3393091  1.3572364 -0.3393091 -0.3393091
[13]  1.3572364 -0.3393091  1.3572364
```

```
> vcov(ls1)
```

```
           [,1]       [,2]       [,3]       [,4]       [,5]
[1,]  1.3572364 -0.3393091 -0.3393091 -0.3393091 -0.3393091
[2,] -0.3393091  1.3572364 -0.3393091 -0.3393091 -0.3393091
[3,] -0.3393091 -0.3393091  1.3572364 -0.3393091 -0.3393091
[4,] -0.3393091 -0.3393091 -0.3393091  1.3572364 -0.3393091
[5,] -0.3393091 -0.3393091 -0.3393091 -0.3393091  1.3572364
```

The most important task is, however, to compute test statistics and p-values. `doTest()` allows to compute the statistics c_max (taking `alternative` into account) and c_quad along with the corresponding p-values. If `nresample = 0` was used in the call to `LinStatExpCov()`, p-values are obtained from the limiting asymptotic distribution whenever such a thing is available at reasonable costs. Otherwise, the permutation p-value is returned (along with the permuted test statistics when `PermutedStatistics` is `TRUE`). The p-values (`lower = FALSE`) or $(1-p)$-values (`lower = TRUE`) can be computed on the log-scale.

⟨ *doTest Prototype* 11 ⟩ ≡

```
(object, teststat = c("maximum", "quadratic", "scalar"),
 alternative = c("two.sided", "less", "greater"), pvalue = TRUE,
 lower = FALSE, log = FALSE, PermutedStatistics = FALSE,
 minbucket = 10L, ordered = TRUE, maxselect = object$Xfactor,
 pargs = GenzBretz())◇
```

Fragment referenced in 12, 18.

⟨ *doTest* 12 ⟩ ≡

```
### note: lower = FALSE => p-value; lower = TRUE => 1 - p-value
doTest <-
function⟨ doTest Prototype 11 ⟩
{
    teststat <- match.arg(teststat, choices = c("maximum", "quadratic", "scalar"))
    if (!any(teststat == c("maximum", "quadratic", "scalar")))
        stop("incorrect teststat")
    alternative <- alternative[1]
    if (!any(alternative == c("two.sided", "less", "greater")))
        stop("incorrect alternative")

    if (maxselect)
        stopifnot(object$Xfactor)

    if (teststat == "quadratic" || maxselect) {
        if (alternative != "two.sided")
            stop("incorrect alternative")
    }

    test <- which(c("maximum", "quadratic", "scalar") == teststat)
    if (test == 3) {
        if (length(object$LinearStatistic) != 1)
            stop("scalar test statistic not applicable")
        test <- 1L # scalar is maximum internally
    }
    alt <- which(c("two.sided", "less", "greater") == alternative)

    if (!pvalue && (NCOL(object$PermutedLinearStatistic) > 0))
        object$PermutedLinearStatistic <- matrix(NA_real_, nrow = 0, ncol = 0)

    if (!maxselect) {
        if (teststat == "quadratic") {
            ret <- .Call(R_QuadraticTest, object, as.integer(pvalue), as.integer(lower),
                         as.integer(log), as.integer(PermutedStatistics))
        } else {
            ret <- .Call(R_MaximumTest, object, as.integer(alt), as.integer(pvalue),
                         as.integer(lower), as.integer(log), as.integer(PermutedStatistics),
                         as.integer(pargs$maxpts), as.double(pargs$releps),
                         as.double(pargs$abseps))
            if (teststat == "scalar") {
                var <- if (object$varonly) object$Variance else object$Covariance
                ret$TestStatistic <- object$LinearStatistic - object$Expectation
                ret$TestStatistic <-
                    if (var > object$tol) ret$TestStatistic / sqrt(var) else NaN
            }
        }
    } else {
        ret <- .Call(R_MaximallySelectedTest, object, as.integer(ordered), as.integer(test),
                     as.integer(minbucket), as.integer(lower), as.integer(log))
    }
    if (!PermutedStatistics) ret$PermutedStatistics <- NULL
    ret
}
◇
```

Fragment referenced in 3a.
Uses: `NCOL` 130c.

```
> ### c_max test statistic
> ### no p-value
> doTest(ls1, teststat = "maximum", pvalue = FALSE)
```

```
$TestStatistic
[1] 0.8411982

$p.value
[1] NA

> ### p-value
> doTest(ls1, teststat = "maximum")

$TestStatistic
[1] 0.8411982

$p.value
[1] 0.8852087

> ### log(p)-value
> doTest(ls1, teststat = "maximum", log = TRUE)

$TestStatistic
[1] 0.8411982

$p.value
[1] 0.108822

> ### (1-p)-value
> doTest(ls1, teststat = "maximum", lower = TRUE)

$TestStatistic
[1] 0.8411982

$p.value
[1] 0.1150168

> ### log(1 - p)-value
> doTest(ls1, teststat = "maximum", lower = TRUE, log = TRUE)

$TestStatistic
[1] 0.8411982

$p.value
[1] -2.164164

> ### quadratic
> doTest(ls1, teststat = "quadratic")

$TestStatistic
[1] 1.077484

$p.value
[1] 0.897828
```

Sometimes we are interested in contrasts of linear statistics and their corresponding properties. Examples include linear-by-linear association tests, where we assign numeric scores to each level of a factor. To implement this, we implement `lmult()` so that we can then left-multiply a matrix to an object of class `"LinStatExpCov"`.

⟨ *Contrasts* 14 ⟩ ≡

```
lmult <-
function(x, object)
{
    stopifnot(!object$varonly)
    stopifnot(is.numeric(x))
    if (is.vector(x)) x <- matrix(x, nrow = 1)
    P <- object$dimension[1]
    stopifnot(ncol(x) == P)
    Q <- object$dimension[2]
    ret <- object
    xLS <- x %*% matrix(object$LinearStatistic, nrow = P)
    xExp <- x %*% matrix(object$Expectation, nrow = P)
    xExpX <- x %*% matrix(object$ExpectationX, nrow = P)
    if (Q == 1) {
        xCov <- tcrossprod(x %*% vcov(object), x)
    } else {
        zmat <- matrix(0, nrow = P * Q, ncol = nrow(x))
        mat <- rbind(t(x), zmat)
        mat <- mat[rep.int(seq_len(nrow(mat)), Q - 1),, drop = FALSE]
        mat <- rbind(mat, t(x))
        mat <- matrix(mat, ncol = Q * nrow(x))
        mat <- t(mat)
        xCov <- tcrossprod(mat %*% vcov(object), mat)
    }
    if (!is.matrix(xCov)) xCov <- matrix(xCov)
    if (length(object$PermutedLinearStatistic) > 0) {
        xPS <- apply(object$PermutedLinearStatistic, 2, function(y)
                     as.vector(x %*% matrix(y, nrow = P)))
        if (!is.matrix(xPS)) xPS <- matrix(xPS, nrow = 1)
        ret$PermutedLinearStatistic <- xPS
    }
    ret$LinearStatistic <- as.vector(xLS)
    ret$Expectation <- as.vector(xExp)
    ret$ExpectationX <- as.vector(xExpX)
    ret$Covariance <- as.vector(xCov[lower.tri(xCov, diag = TRUE)])
    ret$Variance <- diag(xCov)
    ret$dimension <- c(NROW(x), Q)
    ret$Xfactor <- FALSE
    if (length(object$StandardisedPermutedLinearStatistic) > 0)
        ret$StandardisedPermutedLinearStatistic <-
            .Call(R_StandardisePermutedLinearStatistic, ret)
    ret
}
◇
```

Fragment referenced in 3a.
Uses: `NROW` 130b, `P` 24a, `Q` 24e, `x` 23d, 24bc, `y` 24df, 25a.

Here is an example for a linear-by-linear association test.

```
> set.seed(29)
> ls1d <- LinStatExpCov(X = model.matrix(~ x - 1), Y = matrix(y, ncol = 1),
+                         nresample = 10, standardise = TRUE)
> set.seed(29)
> ls1s <- LinStatExpCov(X = as.double(1:5)[x], Y = matrix(y, ncol = 1),
+                         nresample = 10, standardise = TRUE)
> ls1c <- lmult(1:5, ls1d)
> stopifnot(isequal(ls1c, ls1s))
> set.seed(29)
> ls1d <- LinStatExpCov(X = model.matrix(~ x - 1), Y = matrix(c(y, y), ncol = 2),
+                         nresample = 10, standardise = TRUE)
```

```
> set.seed(29)
> ls1s <- LinStatExpCov(X = as.double(1:5)[x], Y = matrix(c(y, y), ncol = 2),
+                        nresample = 10, standardise = TRUE)
> ls1c <- lmult(1:5, ls1d)
> stopifnot(isequal(ls1c, ls1s))
```

2.1.4 Tabulations

The tabulation of `ix` and `iy` can be computed without necessarily computing the corresponding linear statistics via `ctabs()`.

⟨ *ctabs Prototype* 15a ⟩ ≡

```
      (ix, iy = integer(0), block = integer(0), weights = integer(0),
       subset = integer(0), checkNAs = TRUE)◇
```

Fragment referenced in 15b, 19.
Uses: `block` 26f, 27b, `subset` 26ade, `weights` 25b.

"ctabs.R" 15b≡

```
      ⟨ R Header 154a ⟩
      ctabs <-
      function⟨ ctabs Prototype 15a ⟩
      {
          stopifnot(is.integer(ix) || is.factor(ix))
          N <- length(ix)

          ⟨ Check ix 9a ⟩

          if (length(iy) > 0) {
              stopifnot(length(iy) == N)
              stopifnot(is.integer(iy) || is.factor(iy))
              ⟨ Check iy 9b ⟩
          }

          ⟨ Check weights, subset, block 5a ⟩

          if (length(iy) == 0)
              if (length(block) == 0)
                  .Call(R_OneTableSums, ix, weights, subset)
              else
                  .Call(R_TwoTableSums, ix, block, weights, subset)[, -1, drop = FALSE]
          else if (length(block) == 0)
              .Call(R_TwoTableSums, ix, iy, weights, subset)
          else
              .Call(R_ThreeTableSums, ix, iy, block, weights, subset)
      }
      ◇
```

Uses: `block` 26f, 27b, `N` 23bc, `R_OneTableSums` 111a, `R_ThreeTableSums` 119b, `R_TwoTableSums` 115a, `subset` 26ade, `weights` 25b, `weights,` 25cd.

```
> t1 <- ctabs(ix = ix, iy = iy)
> t2 <- xtabs(~ ix + iy)
> max(abs(t1[-1, -1] - t2))

[1] 0
```

2.2 Manual Pages

"LinStatExpCov.Rd" 17≡

```
\name{LinStatExpCov}
\alias{LinStatExpCov}
\alias{lmult}
\title{
  Linear Statistics with Expectation and Covariance
}
\description{
  Strasser-Weber type linear statistics and their expectation
  and covariance under the independence hypothesis
}
\usage{
LinStatExpCov⟨ LinStatExpCov Prototype 3b ⟩
lmult(x, object)
}
\arguments{
  \item{X}{numeric matrix of transformations.}
  \item{Y}{numeric matrix of influence functions.}
  \item{ix}{an optional integer vector expanding \code{X}.}
  \item{iy}{an optional integer vector expanding \code{Y}.}
  \item{weights}{an optional integer vector of non-negative case weights.}
  \item{subset}{an optional integer vector defining a subset of observations.}
  \item{block}{an optional factor defining independent blocks of observations.}
  \item{checkNAs}{a logical for switching off missing value checks.  This
    included switching off checks for suitable values of \code{subset}.
    Use at your own risk.}
  \item{varonly}{a logical asking for variances only.}
  \item{nresample}{an integer defining the number of permuted statistics to draw.}
  \item{standardise}{a logical asking to standardise the permuted statistics.}
  \item{tol}{tolerance for zero variances.}
  \item{x}{a contrast matrix to be left-multiplied in case \code{X} was a factor.}
  \item{object}{an object of class \code{"LinStatExpCov"}.}
}
\details{
  The function, after minimal preprocessing, calls the underlying C code
  and computes the linear statistic, its expectation and covariance and,
  optionally, \code{nresample} samples from its permutation distribution.

  When both \code{ix} and \code{iy} are missing, the number of rows of
  \code{X} and \code{Y} is the same, ie the number of observations.

  When \code{X} is missing and \code{ix} a factor, the code proceeds as
  if \code{X} were a dummy matrix of \code{ix} without explicitly
  computing this matrix.

  Both \code{ix} and \code{iy} being present means the code treats them
  as subsetting vectors for \code{X} and \code{Y}.  Note that \code{ix = 0}
  or \code{iy = 0} means that the corresponding observation is missing
  and the first row or \code{X} and \code{Y} must be zero.

  \code{lmult} allows left-multiplication of a contrast matrix when \code{X}
  was (equivalent to) a factor.
}
\value{
  A list.
}
\references{
  Strasser, H. and Weber, C.  (1999).  On the asymptotic theory of permutation
  statistics.  \emph{Mathematical Methods of Statistics} \bold{8}(2), 220--250.
}
\examples{
wilcox.test(Ozone ~ Month, data = airquality, subset = Month \%in\% c(5, 8),
            exact = FALSE, correct = FALSE)
```

```
aq <- subset(airquality, Month \%in\% c(5, 8))
X <- as.double(aq$Month == 5)
Y <- as.double(rank(aq$Ozone, na.last = "keep"))
doTest(LinStatExpCov(X, Y))
```

"doTest.Rd" 18≡

```
\name{doTest}
\alias{doTest}
\title{
  Permutation Test
}
\description{
  Perform permutation test for a linear statistic
}
\usage{
doTest⟨ doTest Prototype 11 ⟩
}
\arguments{
  \item{object}{an object returned by \code{\link{LinStatExpCov}}.}
  \item{teststat}{type of test statistic to use.}
  \item{alternative}{alternative for scalar or maximum-type statistics.}
  \item{pvalue}{a logical indicating if a p-value shall be computed.}
  \item{lower}{a logical indicating if a p-value (\code{lower} is \code{FALSE})
    or 1 - p-value (\code{lower} is \code{TRUE}) shall be returned.}
  \item{log}{a logical, if \code{TRUE} probabilities are log-probabilities.}
  \item{PermutedStatistics}{a logical, return permuted test statistics.}
  \item{minbucket}{minimum weight in either of two groups for maximally selected
    statistics.}
  \item{ordered}{a logical, if \code{TRUE} maximally selected statistics assume
    that the cutpoints are ordered.}
  \item{maxselect}{a logical, if \code{TRUE} maximally selected statistics are
    computed.  This requires that \code{X} was an implicitly defined design
    matrix in \code{\link{LinStatExpCov}}.}
  \item{pargs}{arguments as in \code{\link[mvtnorm:algorithms]{GenzBretz}}.}
}
\details{
  Computes a test statistic, a corresponding p-value and, optionally, cutpoints
  for maximally selected statistics.
}
\value{
  A list.
}
\keyword{htest}
◇
```

"ctabs.Rd" 19≡

```
\name{ctabs}
\alias{ctabs}
\title{
  Cross Tabulation
}
\description{
  Efficient weighted cross tabulation of two factors and a block
}
\usage{
ctabs⟨ ctabs Prototype 15a ⟩
}
\arguments{
  \item{ix}{a integer of positive values with zero indicating a missing.}
  \item{iy}{an optional integer of positive values with zero indicating a
    missing.}
  \item{block}{an optional blocking factor without missings.}
  \item{weights}{an optional vector of case weights, integer or double.}
  \item{subset}{an optional integer vector indicating a subset.}
  \item{checkNAs}{a logical for switching off missing value checks.}
}
\details{
  A faster version of \code{xtabs(weights ~ ix + iy + block, subset)}.
}
\value{
  If \code{block} is present, a three-way table. Otherwise,
  a one- or two-dimensional table.
}
\examples{
ctabs(ix = 1:5, iy = 1:5, weights = 1:5 / 5)
}
\keyword{univar}
◇
```

Uses: `block` 26f, 27b, `subset` 26ade, `weights` 25b, `weights,` 25cd.

Chapter 3

C Code

The main motivation to implement the **libcoin** package comes from the demand to compute high-dimensional linear statistics (with large P and Q) and the corresponding test statistics very often, either for sampling from the permutation null distribution H_0 or for different subsets of the data. Especially the latter task can be performed *without* actually subsetting the data via the `subset` argument very efficiently (in terms of memory consumption and, depending on the circumstances, speed).

We start with the definition of some macros and global variables in the header files.

3.1 Header and Source Files

`"libcoin_internal.h"` 20a≡

⟨ *C Header* 154b ⟩
⟨ *R Includes* 20b ⟩
⟨ *C Macros* 21a ⟩
⟨ *C Global Variables* 21b ⟩
◇

These includes provide some R infrastructure at C level.

⟨ *R Includes* 20b ⟩ ≡

```
#define STRICT_R_HEADERS
#define USE_FC_LEN_T
#include <float.h>        /* for DBL_MIN */
#include <R.h>
#include <Rinternals.h>
#include <Rversion.h>     /* for R_VERSION */
#include <R_ext/Lapack.h> /* for dspev */
#ifndef FCONE
# define FCONE
#endif
```
◇

Fragment referenced in 20a.

We need three macros: `S` computes the element σ_{ij} of a symmetric $n \times n$ matrix when only the lower triangular elements are stored. `LE` implements $\leq$ with some tolerance, `GE` implements $\geq$.

⟨ *C Macros* 21a ⟩ ≡

```
#define S(i, j, n) ((i) >= (j) ? (n) * (j) + (i) - (j) * ((j) + 1) / 2 : (n) * (i) + (j) - (i) * ((i) + 1)
#define LE(x, y, tol) ((x) < (y)) || (fabs((x) - (y)) < (tol))
#define GE(x, y, tol) ((x) > (y)) || (fabs((x) - (y)) < (tol))
```
◇

Fragment referenced in 20a.
Defines: `GE` 51, 54, `LE` 54, `S` 35b, 36a, 44, 45, 57b, 58b, 59b, 62, 64a, 68, 69a, 73a, 76b, 88a, 99, 134, 135a, 137, 143b.
Uses: `x` 23d, 24bc, `y` 24df, 25a.

⟨ *C Global Variables* 21b ⟩ ≡

```
#define ALTERNATIVE_twosided                            1
#define ALTERNATIVE_less                                2
#define ALTERNATIVE_greater                             3

#define TESTSTAT_maximum                                1
#define TESTSTAT_quadratic                              2

#define LinearStatistic_SLOT                            0
#define Expectation_SLOT                                1
#define Covariance_SLOT                                 2
#define Variance_SLOT                                   3
#define ExpectationX_SLOT                               4
#define varonly_SLOT                                    5
#define dim_SLOT                                        6
#define ExpectationInfluence_SLOT                       7
#define CovarianceInfluence_SLOT                        8
#define VarianceInfluence_SLOT                          9
#define Xfactor_SLOT                                    10
#define tol_SLOT                                        11
#define PermutedLinearStatistic_SLOT                    12
#define StandardisedPermutedLinearStatistic_SLOT        13
#define TableBlock_SLOT                                 14
#define Sumweights_SLOT                                 15
#define Table_SLOT                                      16

#define DoSymmetric                                     1
#define DoCenter                                        1
#define DoVarOnly                                       1
#define Power1                                          1
#define Power2                                          2
#define Offset0                                         0
```
◇

Fragment referenced in 20a.
Defines: `CovarianceInfluence_SLOT` 144c, 147b, 148, `Covariance_SLOT` 143bc, 147b, 148, `dim_SLOT` 141c, 142a, 147b, 148, `DoCenter` 77d, 81c, 84a, 85b, 88a, 94b, 106c, `DoSymmetric` 77d, 84a, 88a, `DoVarOnly` 35bcd, 44, `ExpectationInfluence_SLOT` 144b, 147b, 148, `ExpectationX_SLOT` 144a, 147b, 148, `Expectation_SLOT` 143a, 147b, 148, `LinearStatistic_SLOT` 142d, 147b, 148, `Offset0` 33b, 34a, 37, 41a, 43b, 44, 81a, 83a, 84c, 87a, 90a, 94b, 102b, 106c, 111a, 115a, 119b, 123c, 127a, `PermutedLinearStatistic_SLOT` 146bc, 147b, 148, `Power1` 81c, 85b, 106c, `Power2` 84a, 88a, `StandardisedPermutedLinearStatistic_SLOT` 147b, 148, `Sumweights_SLOT` 145b, 146a, 147b, 148, 149b, `TableBlock_SLOT` 34a, 145a, 146a, 147b, 148, 149b, `Table_SLOT` 145cd, 147b, 148, 150, `tol_SLOT` 146d, 147b, 148, `VarianceInfluence_SLOT` 144d, 147b, 148, `Variance_SLOT` 143b, 147b, 148, `varonly_SLOT` 142b, 147b, 148, `Xfactor_SLOT` 142c, 147b, 148.

The corresponding header file contains definitions of functions that can be called via `.Call()` from the **libcoin** package. In addition, packages linking to **libcoin** can access these function at C level (at your own risk, of course!).

"libcoin.h" 22a≡

```
⟨ C Header 154b ⟩
#include "libcoin_internal.h"
⟨ Function Prototypes 22b ⟩
◇
```

⟨ *Function Prototypes* 22b ⟩ ≡

```
extern ⟨ R_ExpectationCovarianceStatistic Prototype 30c ⟩;
extern ⟨ R_PermutedLinearStatistic Prototype 36b ⟩;
extern ⟨ R_StandardisePermutedLinearStatistic Prototype 38b ⟩;
extern ⟨ R_ExpectationCovarianceStatistic_2d Prototype 40a ⟩;
extern ⟨ R_PermutedLinearStatistic_2d Prototype 47a ⟩;
extern ⟨ R_QuadraticTest Prototype 50b ⟩;
extern ⟨ R_MaximumTest Prototype 52b ⟩;
extern ⟨ R_MaximallySelectedTest Prototype 55a ⟩;
extern ⟨ R_ExpectationInfluence Prototype 80b ⟩;
extern ⟨ R_CovarianceInfluence Prototype 82 ⟩;
extern ⟨ R_ExpectationX Prototype 84b ⟩;
extern ⟨ R_CovarianceX Prototype 86 ⟩;
extern ⟨ R_Sums Prototype 89c ⟩;
extern ⟨ R_KronSums Prototype 94a ⟩;
extern ⟨ R_KronSums_Permutation Prototype 102a ⟩;
extern ⟨ R_colSums Prototype 106b ⟩;
extern ⟨ R_OneTableSums Prototype 110b ⟩;
extern ⟨ R_TwoTableSums Prototype 114b ⟩;
extern ⟨ R_ThreeTableSums Prototype 119a ⟩;
extern ⟨ R_order_subset_wrt_block Prototype 123b ⟩;
extern ⟨ R_quadform Prototype 60b ⟩;
extern ⟨ R_kronecker Prototype 132b ⟩;
extern ⟨ R_MPinv_sym Prototype 135b ⟩;
extern ⟨ R_unpack_sym Prototype 138a ⟩;
extern ⟨ R_pack_sym Prototype 140a ⟩;
◇
```

Fragment referenced in 22a.

The C file `libcoin.c` contains all C functions and corresponding R interfaces.

"libcoin.c" 22c≡

```
⟨ C Header 154b ⟩
#include "libcoin_internal.h"
#include <R_ext/stats_stubs.h>  /* for S_rcont2 */
#include <mvtnormAPI.h>         /* for calling mvtnorm */
⟨ Function Definitions 23a ⟩
◇
```

⟨ *Function Definitions* 23a ⟩ ≡

⟨ *MoreUtils* 130a ⟩
⟨ *Memory* 141a ⟩
⟨ *P-Values* 64b ⟩
⟨ *KronSums* 93b ⟩
⟨ *colSums* 106a ⟩
⟨ *SimpleSums* 89b ⟩
⟨ *Tables* 110a ⟩
⟨ *Utils* 123a ⟩
⟨ *LinearStatistics* 77b ⟩
⟨ *Permutations* 127b ⟩
⟨ *ExpectationCovariances* 78a ⟩
⟨ *Test Statistics* 57a ⟩
⟨ *User Interface* 30a ⟩
⟨ *2d User Interface* 39b ⟩
⟨ *Tests* 50a ⟩
◇

Fragment referenced in 22c.

3.2 Variables

N is the number of observations

⟨ *R N Input* 23b ⟩ ≡

```
SEXP N,
```
◇

Fragment referenced in 89c.
Defines: `N` 5ab, 6, 8, 15b, 23c, 33ab, 34ab, 35abcd, 37, 41a, 67, 77d, 81ac, 83a, 84ac, 85b, 87a, 88abc, 90a, 91a, 93a, 94b, 96b, 97a, 99, 101b, 102b, 103b, 104c, 105c, 106c, 107b, 109b, 111a, 112a, 115a, 116a, 119b, 120b, 123c, 124b, 125ab, 126a, 127a, 135a.

which at C level is represented as `R_xlen_t` to allow for $N >$ `INT_MAX`

⟨ *C integer N Input* 23c ⟩ ≡

```
R_xlen_t N
```
◇

Fragment referenced in 24bc, 32, 37, 41a, 77c, 81ab, 83ab, 84c, 85a, 87ab, 90b, 91b, 92abc, 94b, 95c, 102b, 103a, 106c, 111a, 115a, 119b, 123c, 124a, 125ab, 126b.
Defines: `N` 5ab, 6, 8, 15b, 23b, 33ab, 34ab, 35abcd, 37, 41a, 67, 77d, 81ac, 83a, 84ac, 85b, 87a, 88abc, 90a, 91a, 93a, 94b, 96b, 97a, 99, 101b, 102b, 103b, 104c, 105c, 106c, 107b, 109b, 111a, 112a, 115a, 116a, 119b, 120b, 123c, 124b, 125ab, 126a, 127a, 135a.

The regressors $\mathbf{x}_i, i = 1, \dots, N$

⟨ *R x Input* 23d ⟩ ≡

```
SEXP x,
```
◇

Fragment referenced in 30b, 39c, 47a, 77c, 84b, 85a, 86, 87b, 94a, 95c, 102a, 103a, 106b, 110b, 114b, 119a.
Defines: `x` 8, 14, 17, 21a, 24bc, 30d, 31ab, 33ab, 35acd, 36c, 37, 40b, 41ab, 42a, 43b, 44, 47b, 48, 77d, 84c, 85b, 87a, 88a, 94b, 95b, 96b, 97a, 99, 101b, 102b, 103b, 104c, 105c, 106c, 107b, 109b, 111a, 112a, 114a, 115a, 116a, 118b, 119b, 120b, 122b, 130bc, 131a, 135ab, 136ab, 137, 138ab, 139, 140abc.

are either represented as a real matrix with N rows and P columns

⟨ *C integer P Input* 24a ⟩ ≡

```
int P
◇
```

Fragment referenced in 24bc, 32, 77c, 78b, 79, 80a, 85a, 87b, 95c, 103a, 149b, 150.
Defines: `P` 14, 31ab, 33ab, 34a, 35acd, 36a, 37, 41ab, 42a, 43b, 44, 45, 46, 48, 51, 52a, 54, 56, 70, 71, 72, 73a, 74, 75ab, 76ab, 77d, 78b, 79, 80a, 84bc, 85b, 86, 87a, 88a, 94ab, 96b, 97a, 99, 101b, 102ab, 103b, 104c, 105c, 106c, 107b, 109b, 111a, 112a, 114a, 115a, 116a, 118b, 119b, 120b, 122b, 131b, 132a, 135a, 147a, 148.

⟨ *C real x Input* 24b ⟩ ≡

```
double *x,
⟨ C integer N Input 23c ⟩,
⟨ C integer P Input 24a ⟩,
◇
```

Fragment referenced in 95d, 104ab, 107c, 135a.
Defines: `x` 8, 14, 17, 21a, 23d, 24c, 30d, 31ab, 33ab, 35acd, 36c, 37, 40b, 41ab, 42a, 43b, 44, 47b, 48, 77d, 84c, 85b, 87a, 88a, 94b, 95b, 96b, 97a, 99, 101b, 102b, 103b, 104c, 105c, 106c, 107b, 109b, 111a, 112a, 114a, 115a, 116a, 118b, 119b, 120b, 122b, 130bc, 131a, 135ab, 136ab, 137, 138ab, 139, 140abc.

or as a factor (an integer at C level) at P levels

⟨ *C integer x Input* 24c ⟩ ≡

```
int *x,
⟨ C integer N Input 23c ⟩,
⟨ C integer P Input 24a ⟩,
◇
```

Fragment referenced in 100a, 105ab, 112b, 116b, 120c.
Defines: `x` 8, 14, 17, 21a, 23d, 24b, 30d, 31ab, 33ab, 35acd, 36c, 37, 40b, 41ab, 42a, 43b, 44, 47b, 48, 77d, 84c, 85b, 87a, 88a, 94b, 95b, 96b, 97a, 99, 101b, 102b, 103b, 104c, 105c, 106c, 107b, 109b, 111a, 112a, 114a, 115a, 116a, 118b, 119b, 120b, 122b, 130bc, 131a, 135ab, 136ab, 137, 138ab, 139, 140abc.

The influence functions are also either a $N \times Q$ real matrix

⟨ *R y Input* 24d ⟩ ≡

```
SEXP y,
◇
```

Fragment referenced in 30b, 39c, 47a, 80b, 81b, 82, 83b, 94a, 102a, 114b, 119a, 123b.
Defines: `y` 14, 21a, 24f, 25a, 30d, 31ab, 33b, 35ab, 36c, 37, 40b, 41ab, 42a, 43b, 44, 47b, 77d, 81ac, 83a, 84a, 94b, 96b, 97a, 99, 101b, 102b, 103b, 104c, 105c, 115a, 116a, 118b, 119b, 120b, 122b, 123c, 133b, 134.

⟨ *C integer Q Input* 24e ⟩ ≡

```
int Q
◇
```

Fragment referenced in 24f, 25a, 32, 78b, 79, 80a, 81ab, 83ab, 94b, 102b, 149b, 150.
Defines: `Q` 14, 31ab, 33ab, 35abcd, 36a, 37, 41ab, 42a, 43b, 44, 45, 46, 48, 51, 52a, 54, 70, 71, 72, 73abc, 74, 76ab, 77ad, 78b, 79, 80a, 81ac, 83a, 84a, 94b, 96b, 97a, 99, 101b, 102b, 103b, 104c, 105c, 115a, 116a, 118b, 119b, 120b, 122b, 132a, 147a, 148, 149a.

⟨ *C real y Input* 24f ⟩ ≡

```
double *y,
⟨ C integer Q Input 24e ⟩,
◇
```

Fragment referenced in 77c, 95cd, 100a, 103a, 104ab, 105ab.
Defines: `y` 14, 21a, 24d, 25a, 30d, 31ab, 33b, 35ab, 36c, 37, 40b, 41ab, 42a, 43b, 44, 47b, 77d, 81ac, 83a, 84a, 94b, 96b, 97a, 99, 101b, 102b, 103b, 104c, 105c, 115a, 116a, 118b, 119b, 120b, 122b, 123c, 133b, 134.

or a factor at Q levels

⟨ *C integer y Input* 25a ⟩ ≡

```
int *y,
⟨ C integer Q Input 24e ⟩,
◇
```

Fragment referenced in 116b, 120c.
Defines: `y` 14, 21a, 24df, 30d, 31ab, 33b, 35ab, 36c, 37, 40b, 41ab, 42a, 43b, 44, 47b, 77d, 81ac, 83a, 84a, 94b, 96b, 97a, 99, 101b, 102b, 103b, 104c, 105c, 115a, 116a, 118b, 119b, 120b, 122b, 123c, 133b, 134.

The case weights $w_i, i = 1, \dots, N$

⟨ *R weights Input* 25b ⟩ ≡

```
SEXP weights
◇
```

Fragment referenced in 30b, 39c, 77c, 80b, 81b, 82, 83b, 84b, 85a, 86, 87b, 89c, 90b, 94a, 95a, 106b, 107a, 110b, 111b, 114b, 115b, 119a, 120a, 123b, 126b.
Defines: `weights` 3b, 4, 5a, 6, 8, 15ab, 17, 19, 25cd, 30d, 31a, 33b, 34b, 35abcd, 36c, 37, 40b, 41a, 49a, 77d, 81ac, 83a, 84ac, 85b, 87a, 88ab, 90a, 91a, 94b, 96b, 97a, 106c, 107b, 111a, 112a, 115a, 116a, 119b, 120b, 123c, 127a.

can be constant one (`XLENGTH(weights) == 0` or `weights = integer(0)`) or integer-valued, with `HAS_WEIGHTS == 0` in the former case

⟨ *C integer weights Input* 25c ⟩ ≡

```
int *weights,
int HAS_WEIGHTS,
◇
```

Fragment referenced in 92ab, 97c, 98a, 100cd, 108cd, 113ab, 117bc, 121cd.
Defines: `HAS_WEIGHTS` 25d, 93a, 99, 101b, 109b, 114a, 118b, 122b, `weights,` 4, 6, 8, 15b, 19, 25d, 30d, 31a, 33b, 34b, 35abcd, 36c, 37, 40b, 41a, 77d, 81ac, 83a, 84ac, 85b, 87a, 88a, 90a, 94b, 106c, 111a, 115a, 119b, 123c, 127a.
Uses: `weights` 25b.

Case weights larger than `INT_MAX` are stored as double

⟨ *C real weights Input* 25d ⟩ ≡

```
double *weights,
int HAS_WEIGHTS,
◇
```

Fragment referenced in 91b, 92c, 97b, 98b, 100b, 101a, 108b, 109a, 112d, 113c, 117a, 118a, 121b, 122a.
Defines: `HAS_WEIGHTS` 25c, 93a, 99, 101b, 109b, 114a, 118b, 122b, `weights,` 4, 6, 8, 15b, 19, 25c, 30d, 31a, 33b, 34b, 35abcd, 36c, 37, 40b, 41a, 77d, 81ac, 83a, 84ac, 85b, 87a, 88a, 90a, 94b, 106c, 111a, 115a, 119b, 123c, 127a.
Uses: `weights` 25b.

The sum of all case weights is a double

⟨ *C sumweights Input* 25e ⟩ ≡

```
double sumweights
◇
```

Fragment referenced in 79, 80a, 81b, 83b.
Defines: `sumweights` 32, 34ab, 35abcd, 43ab, 44, 46, 48, 49bd, 71, 72, 73b, 77a, 79, 80a, 81ac, 83a, 84a, 127a, 145b.

Subsets $\mathcal{A} \subseteq \{1, \dots, N\}$ are R style indices

⟨ *R subset Input* 26a ⟩ ≡

```
SEXP subset
◇
```

Fragment referenced in 30b, 39c, 77c, 80b, 81b, 82, 83b, 84b, 85a, 86, 87b, 89c, 90b, 94a, 95a, 102a, 103a, 106b, 107a, 110b, 111b, 114b, 115b, 119a, 120a, 123b, 124a, 126ab.
Defines: `subset` 3b, 4, 5ab, 6, 8, 15ab, 17, 19, 26de, 30d, 31a, 32, 33b, 34ab, 36c, 37, 40b, 41a, 43b, 44, 77d, 81ac, 83a, 84ac, 85b, 87a, 88ab, 89a, 90a, 91a, 94b, 96b, 97a, 102b, 103b, 104c, 105c, 106c, 107b, 111a, 112a, 115a, 116a, 119b, 120b, 123c, 124b, 126a, 127a, 128ab, 129ab.

are either not existent (`XLENGTH(subset) == 0`) or of length

⟨ *C integer Nsubset Input* 26b ⟩ ≡

```
R_xlen_t Nsubset
◇
```

Fragment referenced in 26c, 37, 41a, 81a, 83a, 84c, 87a, 90a, 94b, 102b, 106c, 111a, 115a, 119b, 128ab, 129b.
Defines: `Nsubset` 34b, 37, 41a, 77d, 81ac, 83a, 84ac, 85b, 87a, 88abc, 89a, 90a, 91a, 93a, 94b, 96b, 97a, 102b, 103b, 104c, 105c, 106c, 107b, 111a, 112a, 115a, 116a, 119b, 120b, 128ab, 129b.

Optionally, one can specify a subset of the subset via

⟨ *C subset range Input* 26c ⟩ ≡

```
R_xlen_t offset,
⟨ C integer Nsubset Input 26b ⟩
◇
```

Fragment referenced in 26de, 77c, 81b, 83b, 85a, 87b, 90b, 95a, 103a, 107a, 111b, 115b, 120a.
Defines: `offset` 32, 34b, 35abcd, 77d, 81c, 84a, 85b, 88ab, 91a, 96b, 97a, 103b, 104c, 105c, 107b, 112a, 116a, 120b.

where `offset` is a C style index for `subset`.

Subsets are stored either as integer

⟨ *C integer subset Input* 26d ⟩ ≡

```
int *subset,
⟨ C subset range Input 26c ⟩
◇
```

Fragment referenced in 92bc, 98ab, 100d, 101a, 104b, 105b, 108d, 109a, 113bc, 117c, 118a, 121d, 122a.
Defines: `subset` 3b, 4, 5ab, 6, 8, 15ab, 17, 19, 26ae, 30d, 31a, 32, 33b, 34ab, 36c, 37, 40b, 41a, 43b, 44, 77d, 81ac, 83a, 84ac, 85b, 87a, 88ab, 89a, 90a, 91a, 94b, 96b, 97a, 102b, 103b, 104c, 105c, 106c, 107b, 111a, 112a, 115a, 116a, 119b, 120b, 123c, 124b, 126a, 127a, 128ab, 129ab.

or double (to allow for indices larger than `INT_MAX`)

⟨ *C real subset Input* 26e ⟩ ≡

```
double *subset,
⟨ C subset range Input 26c ⟩
◇
```

Fragment referenced in 91b, 92a, 97bc, 100bc, 104a, 105a, 108bc, 112d, 113a, 117ab, 121bc.
Defines: `subset` 3b, 4, 5ab, 6, 8, 15ab, 17, 19, 26ad, 30d, 31a, 32, 33b, 34ab, 36c, 37, 40b, 41a, 43b, 44, 77d, 81ac, 83a, 84ac, 85b, 87a, 88ab, 89a, 90a, 91a, 94b, 96b, 97a, 102b, 103b, 104c, 105c, 106c, 107b, 111a, 112a, 115a, 116a, 119b, 120b, 123c, 124b, 126a, 127a, 128ab, 129ab.

Blocks $\text{block}_i, i = 1, \dots, N$

⟨ *R block Input* 26f ⟩ ≡

```
SEXP block
◇
```

Fragment referenced in 30b, 39c, 47a, 119a, 123b, 124a, 125b, 126a.
Defines: `block` 3b, 4, 5a, 6, 8, 15ab, 17, 19, 27b, 30d, 31ab, 34ab, 36c, 37, 40b, 41ab, 47b, 119b, 120b, 122b, 123c, 124b, 125b, 126a, 145a.

at B levels

⟨ *C integer B Input* 27a ⟩ ≡

```
int B
◇
```

Fragment referenced in 27b, 32, 149b, 150.
Defines: `B` 31ab, 32, 33a, 34a, 37, 41ab, 42b, 45, 46, 48, 49b, 70, 71, 74, 119b, 120b, 122b, 132abc, 133ab, 134, 147a, 148, 149b, 150.

are stored as a factor

⟨ *C integer block Input* 27b ⟩ ≡

```
int *block,
⟨ C integer B Input 27a ⟩,
◇
```

Fragment referenced in 120c.
Defines: `block` 3b, 4, 5a, 6, 8, 15ab, 17, 19, 26f, 30d, 31ab, 34ab, 36c, 37, 40b, 41ab, 47b, 119b, 120b, 122b, 123c, 124b, 125b, 126a, 145a.

The tabulation of block (potentially in subsets) is

⟨ *R blockTable Input* 27c ⟩ ≡

```
SEXP blockTable
◇
```

Fragment referenced in 124a, 125b, 126a.
Defines: `blockTable` 37, 123c, 124b, 125b, 126a.

where the table is of length $B + 1$ and the first element counts the number of missing values (although these are NOT allowed in block).

3.2.1 Example Data and Code

We start with setting-up some toy data sets to be used as test bed. The data over both the 1d and the 2d case, including case weights, subsets and blocks.

```
> N <- 20L
> P <- 3L
> Lx <- 10L
> Ly <- 5L
> Q <- 4L
> B <- 2L
> iX2d <- rbind(0, matrix(runif(Lx * P), nrow = Lx))
> ix <- sample(1:Lx, size = N, replace = TRUE)
> levels(ix) <- 1:Lx
> ixf <- factor(ix, levels = 1:Lx, labels = 1:Lx)
> x <- iX2d[ix + 1,]
> Xfactor <- diag(Lx)[ix,]
> iY2d <- rbind(0, matrix(runif(Ly * Q), nrow = Ly))
> iy <- sample(1:Ly, size = N, replace = TRUE)
> levels(iy) <- 1:Ly
> iyf <- factor(iy, levels = 1:Ly, labels = 1:Ly)
> y <- iY2d[iy + 1,]
> weights <- sample(0:5, size = N, replace = TRUE)
> block <- sample(gl(B, ceiling(N / B))[1:N])
> subset <- sort(sample(1:N, floor(N * 1.5), replace = TRUE))
> subsety <- sample(1:N, floor(N * 1.5), replace = TRUE)
> r1 <- rep(1:ncol(x), ncol(y))
> r1Xfactor <- rep(1:ncol(Xfactor), ncol(y))
> r2 <- rep(1:ncol(y), each = ncol(x))
> r2Xfactor <- rep(1:ncol(y), each = ncol(Xfactor))
```

As a benchmark, we implement linear statistics, their expectation and covariance, taking case weights, subsets and blocks into account, at R level. In a sense, the core of the **libcoin** package is “just” a less memory-hungry and sometimes faster version of this simple function.

```
> LSEC <-
+ function(X, Y, weights = integer(0), subset = integer(0), block = integer(0))
+ {
+     if (length(weights) == 0) weights <- rep.int(1, NROW(X))
+     if (length(subset) == 0) subset <- seq_len(NROW(X))
+
+     X <- X[subset,, drop = FALSE]
+     Y <- Y[subset,, drop = FALSE]
+     weights <- weights[subset]
+
+     if (length(block) == 0) {
+         w. <- sum(weights)
+         wX <- weights * X
+         wY <- weights * Y
+         ExpX <- colSums(wX)
+         ExpY <- colSums(wY) / w.
+         CovX <- crossprod(X, wX)
+         Yc <- t(t(Y) - ExpY)
+         CovY <- crossprod(Yc, weights * Yc) / w.
+         T <- crossprod(X, wY)
+         Exp <- kronecker(ExpY, ExpX)
+         Cov <- w. / (w. - 1) * kronecker(CovY, CovX) -
+                 1 / (w. - 1) * kronecker(CovY, tcrossprod(ExpX))
+
+         list(LinearStatistic = as.vector(T), Expectation = as.vector(Exp),
+               Covariance = Cov, Variance = diag(Cov))
+     } else {
+         block <- block[subset]
+         ret <- list(LinearStatistic = 0, Expectation = 0,
+                     Covariance = 0, Variance = 0)
+         for (b in levels(block)) {
+             tmp <- LSEC(X = X, Y = Y, weights = weights, subset = which(block == b))
+             for (l in names(ret)) ret[[l]] <- ret[[l]] + tmp[[l]]
+         }
+         ret
+     }
+ }
```

```
> cmpr <-
+ function(ret1, ret2)
+ {
+     if (inherits(ret1, "LinStatExpCov")) {
+         if (!ret1$varonly)
+             ret1$Covariance <- vcov(ret1)
+     }
+     ret1 <- ret1[!sapply(ret1, is.null)]
+     ret2 <- ret2[!sapply(ret2, is.null)]
+     nm1 <- names(ret1)
+     nm2 <- names(ret2)
+     nm <- c(nm1, nm2)
+     nm <- names(table(nm))[table(nm) == 2]
+     isequal(ret1[nm], ret2[nm])
+ }
```

We now compute the linear statistic along with corresponding expectation, variance and covariance for later reuse.

```
> LECVxyws <- LinStatExpCov(x, y, weights = weights, subset = subset)
> LEVxyws <- LinStatExpCov(x, y, weights = weights, subset = subset, varonly = TRUE)
```

The following tests compare the high-level R implementation (function `LSEC()`) with the 1d and 2d C level implementations in the two sitations with and without specification of `X` (ie, the dummy matrix in the latter case).

```
> ### with X given
> testit <-
+ function(...)
+ {
+     a <- LinStatExpCov(x, y, ...)
+     b <- LSEC(x, y, ...)
+     d <- LinStatExpCov(X = iX2d, ix = ix, Y = iY2d, iy = iy, ...)
+     cmpr(a, b) && cmpr(d, b)
+ }
> stopifnot(
+     testit() && testit(weights = weights) &&
+     testit(subset = subset) && testit(weights = weights, subset = subset) &&
+     testit(block = block) && testit(weights = weights, block = block) &&
+     testit(subset = subset, block = block) &&
+     testit(weights = weights, subset = subset, block = block)
+ )
> ### without dummy matrix X
> testit <-
+ function(...)
+ {
+     a <- LinStatExpCov(X = ix, y, ...)
+     b <- LSEC(Xfactor, y, ...)
+     d <- LinStatExpCov(X = integer(0), ix = ix, Y = iY2d, iy = iy, ...)
+     cmpr(a, b) && cmpr(d, b)
+ }
> stopifnot(
+     testit() && testit(weights = weights) &&
+     testit(subset = subset) && testit(weights = weights, subset = subset) &&
+     testit(block = block) && testit(weights = weights, block = block) &&
+     testit(subset = subset, block = block) &&
+     testit(weights = weights, subset = subset, block = block)
+ )
```

All three implementations give the same results.

3.3 Conventions

Functions starting with `R_` are C functions callable via `.Call()` from R. That means they all return `SEXP`. These functions allocate memory handled by R.

Functions starting with `RC_` are C functions with `SEXP` or pointer arguments and possibly an `SEXP` return value.

Functions starting with `C_` are C functions with pointer arguments only and return a scalar or nothing.

Return values (arguments modified by a function) are named `ans`, sometimes with dimension (for example: `PQ_ans`).

3.4 C User Interface

3.4.1 One-Dimensional Case ("1d")

⟨ *User Interface* 30a ⟩ ≡

⟨ *RC_ExpectationCovarianceStatistic* 32 ⟩
⟨ *R_ExpectationCovarianceStatistic* 31a ⟩
⟨ *R_PermutedLinearStatistic* 37 ⟩
⟨ *R_StandardisePermutedLinearStatistic* 39a ⟩
◇

Fragment referenced in 23a.

The data are given as $\mathbf{x}_i$ and $\mathbf{y}_i$ for $i = 1, \dots, N$, optionally with case weights, subset and blocks. The latter three variables are ignored when specified as `integer(0)`.

⟨ *User Interface Input* 30b ⟩ ≡

⟨ *R x Input* 23d ⟩
⟨ *R y Input* 24d ⟩
⟨ *R weights Input* 25b ⟩,
⟨ *R subset Input* 26a ⟩,
⟨ *R block Input* 26f ⟩,
◇

Fragment referenced in 30c, 32, 36b.

⟨ *R_ExpectationCovarianceStatistic Prototype* 30c ⟩ ≡

```
SEXP R_ExpectationCovarianceStatistic
(
    ⟨ User Interface Input 30b ⟩
    SEXP varonly,
    SEXP tol
)
◇
```

Fragment referenced in 22b, 31a.
Uses: `R_ExpectationCovarianceStatistic` 31a.

This function can be called from other packages.

`"libcoinAPI.h"` 30d≡

```
⟨ C Header 154b ⟩
#include <R_ext/Rdynload.h>
#include <libcoin.h>

extern SEXP libcoin_R_ExpectationCovarianceStatistic(
    SEXP x, SEXP y, SEXP weights, SEXP subset, SEXP block, SEXP varonly,
    SEXP tol
) {
    static SEXP(*fun)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP))
            R_GetCCallable("libcoin", "R_ExpectationCovarianceStatistic");
    return fun(x, y, weights, subset, block, varonly, tol);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `block` 26f, 27b, `R_ExpectationCovarianceStatistic` 31a, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc,
`y` 24df, 25a.

The C interface essentially sets-up the necessary memory and calls a C level function for the computations.

⟨ *R_ExpectationCovarianceStatistic* 31a ⟩ ≡

```
⟨ R_ExpectationCovarianceStatistic Prototype 30c ⟩
{
    SEXP ans;

    ⟨ Setup Dimensions 31b ⟩

    PROTECT(ans = RC_init_LECV_1d(P, Q, INTEGER(varonly)[0], B, TYPEOF(x) == INTSXP, REAL(tol)[0]));

    RC_ExpectationCovarianceStatistic(x, y, weights, subset, block, ans);

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

Fragment referenced in 30a.
Defines: `R_ExpectationCovarianceStatistic` 6, 30cd, 152c, 153.
Uses: `B` 27a, `block` 26f, 27b, `P` 24a, `Q` 24e, `RC_ExpectationCovarianceStatistic` 32, 45, `RC_init_LECV_1d` 149b,
`subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc, `y` 24df, 25a.

P, Q and B are first extracted from the data. The case where `X` is an implicitly specified dummy matrix, the dimension P is the number of levels of `x`.

⟨ *Setup Dimensions* 31b ⟩ ≡

```
int P, Q, B;

if (TYPEOF(x) == INTSXP) {
    P = NLEVELS(x);
} else {
    P = NCOL(x);
}
Q = NCOL(y);

B = 1;
if (LENGTH(block) > 0)
    B = NLEVELS(block);
◇
```

Fragment referenced in 31a, 37.
Uses: `B` 27a, `block` 26f, 27b, `NCOL` 130c, `NLEVELS` 131a, `P` 24a, `Q` 24e, `x` 23d, 24bc, `y` 24df, 25a.

The core function first computes the linear statistic (as there is no need to pay attention to blocks) and, in a second step, starts a loop over potential blocks.

FIXME: `x` being an integer (`Xfactor`) with some 0 elements is not handled correctly (as `sumweights` doesn't take this information into account; use `subset` to exclude these missings (as done in `LinStatExpCov()`)

⟨ *RC_ExpectationCovarianceStatistic* 32 ⟩ ≡

```
void RC_ExpectationCovarianceStatistic
(
    ⟨ User Interface Input 30b ⟩
    SEXP ans
) {
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer P Input 24a ⟩;
    ⟨ C integer Q Input 24e ⟩;
    ⟨ C integer B Input 27a ⟩;
    double *sumweights, *table;
    double *ExpInf, *VarInf, *CovInf, *ExpX, *ExpXtotal, *VarX, *CovX;
    double *tmpV, *tmpCV;
    SEXP nullvec, subset_block;

    ⟨ Extract Dimensions 33a ⟩

    ⟨ Compute Linear Statistic 33b ⟩

    ⟨ Setup Memory and Subsets in Blocks 34a ⟩

    /* start with subset[0] */
    R_xlen_t offset = (R_xlen_t) table[0];

    for (int b = 0; b < B; b++) {

        ⟨ Compute Sum of Weights in Block 34b ⟩

        /* don't do anything for empty blocks or blocks with weight 1 */
        if (sumweights[b] > 1) {

            ⟨ Compute Expectation Linear Statistic 35a ⟩

            ⟨ Compute Covariance Influence 35b ⟩

            if (C_get_varonly(ans)) {
                ⟨ Compute Variance Linear Statistic 35c ⟩
            } else {
                ⟨ Compute Covariance Linear Statistic 35d ⟩
            }
        }

        /* next iteration starts with subset[cumsum(table[1:(b + 1)])] */
        offset += (R_xlen_t) table[b + 1];
    }

    ⟨ Compute Variance from Covariance 36a ⟩

    R_Free(ExpX); R_Free(VarX); R_Free(CovX);
    UNPROTECT(2);
}
◇
```

Fragment referenced in 30a.
Defines: `RC_ExpectationCovarianceStatistic` 31a.
Uses: `B` 27a, `C_get_varonly` 142b, `offset` 26c, `subset` 26ade, `sumweights` 25e.

The dimensions are available from the return object:

⟨ *Extract Dimensions* 33a ⟩ ≡

```
P = C_get_P(ans);
Q = C_get_Q(ans);
N = NROW(x);
B = C_get_B(ans);
◇
```

Fragment referenced in 32.
Uses: `B` 27a, `C_get_B` 146a, `C_get_P` 141c, `C_get_Q` 142a, `N` 23bc, `NROW` 130b, `P` 24a, `Q` 24e, `x` 23d, 24bc.

The linear statistic $\mathbf{T}(\mathcal{A})$ can be computed without taking blocks into account.

⟨ *Compute Linear Statistic* 33b ⟩ ≡

```
RC_LinearStatistic(x, N, P, REAL(y), Q, weights, subset,
                   Offset0, XLENGTH(subset),
                   C_get_LinearStatistic(ans));
◇
```

Fragment referenced in 32.
Uses: `C_get_LinearStatistic` 142d, `N` 23bc, `Offset0` 21b, `P` 24a, `Q` 24e, `RC_LinearStatistic` 77d, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc, `y` 24df, 25a.

We next extract memory from the return object and allocate some additional memory. The most important step is to tabulate blocks and to order the subset with respect to blocks. In absense of block, this just returns subset.

⟨ *Setup Memory and Subsets in Blocks* 34a ⟩ ≡

```
ExpInf = C_get_ExpectationInfluence(ans);
VarInf = C_get_VarianceInfluence(ans);
CovInf = C_get_CovarianceInfluence(ans);
ExpXtotal = C_get_ExpectationX(ans);
for (int p = 0; p < P; p++) ExpXtotal[p] = 0.0;
ExpX = R_Calloc(P, double);
/* Fix by Joanidis Kristoforos: P > INT_MAX is possible
   for maximally selected statistics (when X is an integer).
   2018-12-13 */
if (C_get_varonly(ans)) {
    VarX = R_Calloc(P, double);
    CovX = R_Calloc(1, double);
} else {
    VarX = R_Calloc(1, double);
    CovX = R_Calloc(PP12(P), double);
}
table = C_get_TableBlock(ans);
sumweights = C_get_Sumweights(ans);
PROTECT(nullvec = allocVector(INTSXP, 0));

if (B == 1) {
    table[0] = 0.0;
    table[1] = RC_Sums(N, nullvec, subset, Offset0, XLENGTH(subset));
} else {
    RC_OneTableSums(INTEGER(block), N, B + 1, nullvec, subset, Offset0,
                    XLENGTH(subset), table);
}
if (table[0] > 0)
    error("No missing values allowed in block");
PROTECT(subset_block = RC_order_subset_wrt_block(N, subset, block,
                                                 VECTOR_ELT(ans, TableBlock_SLOT)));
◇
```

Fragment referenced in 32.
Uses: `B` 27a, `block` 26f, 27b, `C_get_CovarianceInfluence` 144c, `C_get_ExpectationInfluence` 144b, `C_get_ExpectationX` 144a, `C_get_Sumweights` 145b, `C_get_TableBlock` 145a, `C_get_VarianceInfluence` 144d, `C_get_varonly` 142b, `N` 23bc, `Offset0` 21b, `P` 24a, `PP12` 131b, `RC_OneTableSums` 112a, `RC_order_subset_wrt_block` 124b, `RC_Sums` 91a, `subset` 26ade, `sumweights` 25e, `TableBlock_SLOT` 21b.

We compute $\boldsymbol{\mu}(\mathcal{A})$ based on $\mathbb{E}(h \mid S(\mathcal{A}))$ and $\sum_{i \in \mathcal{A}} w_i \mathbf{x}_i$ for the subset given by subset and the bth level of block. The expectation is initialised zero when $b = 0$ and values add-up over blocks.

⟨ *Compute Sum of Weights in Block* 34b ⟩ ≡

```
/* compute sum of case weights in block b of subset */
if (table[b + 1] > 0) {
    sumweights[b] = RC_Sums(N, weights, subset_block,
                            offset, (R_xlen_t) table[b + 1]);
} else {
    /* offset = something and Nsubset = 0 means Nsubset = N in
       RC_Sums; catch empty or zero-weight block levels here */
    sumweights[b] = 0.0;
}
◇
```

Fragment referenced in 32.
Uses: `block` 26f, 27b, `N` 23bc, `Nsubset` 26b, `offset` 26c, `RC_Sums` 91a, `subset` 26ade, `sumweights` 25e, `weights` 25b, `weights,` 25cd.

⟨ *Compute Expectation Linear Statistic* 35a ⟩ ≡

```
RC_ExpectationInfluence(N, y, Q, weights, subset_block, offset,
                        (R_xlen_t) table[b + 1], sumweights[b], ExpInf + b * Q);
RC_ExpectationX(x, N, P, weights, subset_block, offset,
                (R_xlen_t) table[b + 1], ExpX);
for (int p = 0; p < P; p++) ExpXtotal[p] += ExpX[p];
C_ExpectationLinearStatistic(P, Q, ExpInf + b * Q, ExpX, b,
                             C_get_Expectation(ans));
```

◇

Fragment referenced in 32.
Uses: `C_ExpectationLinearStatistic` 78b, `C_get_Expectation` 143a, `N` 23bc, `offset` 26c, `P` 24a, `Q` 24e, `RC_ExpectationInfluence` 81c, `RC_ExpectationX` 85b, `sumweights` 25e, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc, `y` 24df, 25a.

The covariance $\mathbb{V}(h \mid S(\mathcal{A}))$ is now computed for the subset given by subset and the bth level of block. Note that `CovInf` stores the values for each block in the return object (for later reuse).

⟨ *Compute Covariance Influence* 35b ⟩ ≡

```
/* C_ordered_Xfactor and C_unordered_Xfactor need both VarInf and CovInf */
RC_CovarianceInfluence(N, y, Q, weights, subset_block, offset,
                       (R_xlen_t) table[b + 1], ExpInf + b * Q, sumweights[b],
                       !DoVarOnly, CovInf + b * Q * (Q + 1) / 2);
/* extract variance from covariance */
tmpCV = CovInf + b * Q * (Q + 1) / 2;
tmpV = VarInf + b * Q;
for (int q = 0; q < Q; q++) tmpV[q] = tmpCV[S(q, q, Q)];
```

◇

Fragment referenced in 32.
Uses: `C_ordered_Xfactor` 70, `C_unordered_Xfactor` 74, `DoVarOnly` 21b, `N` 23bc, `offset` 26c, `Q` 24e, `RC_CovarianceInfluence` 84a, `S` 21a, `sumweights` 25e, `weights` 25b, `weights,` 25cd, `y` 24df, 25a.

We can now compute the variance or covariance of the linear statistic $\mathbf{\Sigma}(\mathcal{A})$:

⟨ *Compute Variance Linear Statistic* 35c ⟩ ≡

```
RC_CovarianceX(x, N, P, weights, subset_block, offset,
               (R_xlen_t) table[b + 1], ExpX, DoVarOnly, VarX);
C_VarianceLinearStatistic(P, Q, VarInf + b * Q, ExpX, VarX, sumweights[b],
                          b, C_get_Variance(ans));
```

◇

Fragment referenced in 32.
Uses: `C_get_Variance` 143b, `C_VarianceLinearStatistic` 80a, `DoVarOnly` 21b, `N` 23bc, `offset` 26c, `P` 24a, `Q` 24e, `RC_CovarianceX` 88a, `sumweights` 25e, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc.

⟨ *Compute Covariance Linear Statistic* 35d ⟩ ≡

```
RC_CovarianceX(x, N, P, weights, subset_block, offset,
               (R_xlen_t) table[b + 1], ExpX, !DoVarOnly, CovX);
C_CovarianceLinearStatistic(P, Q, CovInf + b * Q * (Q + 1) / 2,
                            ExpX, CovX, sumweights[b], b,
                            C_get_Covariance(ans));
```

◇

Fragment referenced in 32.
Uses: `C_CovarianceLinearStatistic` 79, `C_get_Covariance` 143c, `DoVarOnly` 21b, `N` 23bc, `offset` 26c, `P` 24a, `Q` 24e, `RC_CovarianceX` 88a, `sumweights` 25e, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc.

⟨ *Compute Variance from Covariance* 36a ⟩ ≡

```
/* always return variances */
if (!C_get_varonly(ans)) {
    for (int p = 0; p < mPQB(P, Q, 1); p++)
        C_get_Variance(ans)[p] = C_get_Covariance(ans)[S(p, p, mPQB(P, Q, 1))];
}
◇
```

Fragment referenced in 32.
Uses: `C_get_Covariance` 143c, `C_get_Variance` 143b, `C_get_varonly` 142b, `mPQB` 132a, `P` 24a, `Q` 24e, `S` 21a.

The computation of permuted linear statistics is done outside this general function. The user interface is the same, except for an additional number of permutations to be specified.

⟨ *R_PermutedLinearStatistic Prototype* 36b ⟩ ≡

```
SEXP R_PermutedLinearStatistic
(
    ⟨ User Interface Input 30b ⟩
    SEXP nresample
)
◇
```

Fragment referenced in 22b, 37.
Uses: `R_PermutedLinearStatistic` 37.

This function can be called from other packages.

`"libcoinAPI.h"` 36c≡

```
extern SEXP libcoin_R_PermutedLinearStatistic(
    SEXP x, SEXP y, SEXP weights, SEXP subset, SEXP block, SEXP nresample
) {
    static SEXP(*fun)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP))
            R_GetCCallable("libcoin", "R_PermutedLinearStatistic");
    return fun(x, y, weights, subset, block, nresample);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `block` 26f, 27b, `R_PermutedLinearStatistic` 37, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc, `y` 24df, 25a.

The dimensions are extracted from the data in the same ways as above. The function differentiates between the absense and presense of blocks. Case weights are removed by expanding subset accordingly. Once within-block permutations were set-up the Kronecker product of `X` and `Y` is computed. Note that this function returns the matrix of permuted linear statistics; the R interface assigns this matrix to the corresponding element of the `LinStatExpCov` object (because we are not allowed to modify existing R objects at C level).

⟨ *R_PermutedLinearStatistic* 37 ⟩ ≡

```
⟨ R_PermutedLinearStatistic Prototype 36b ⟩
{
    SEXP ans, expand_subset, block_subset, perm, tmp, blockTable;
    double *linstat;
    int PQ;
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer Nsubset Input 26b ⟩;
    R_xlen_t inresample;

    ⟨ Setup Dimensions 31b ⟩
    PQ = mPQB(P, Q, 1);
    N = NROW(y);
    inresample = (R_xlen_t) REAL(nresample)[0];

    PROTECT(ans = allocMatrix(REALSXP, PQ, inresample));
    PROTECT(expand_subset = RC_setup_subset(N, weights, subset));
    Nsubset = XLENGTH(expand_subset);
    PROTECT(tmp = allocVector(REALSXP, Nsubset));
    PROTECT(perm = allocVector(REALSXP, Nsubset));

    GetRNGstate();
    if (B == 1) {
        for (R_xlen_t np = 0; np < inresample; np++) {
            ⟨ Setup Linear Statistic 38a ⟩
            C_doPermute(REAL(expand_subset), Nsubset, REAL(tmp), REAL(perm));
            RC_KronSums_Permutation(x, NROW(x), P, REAL(y), Q, expand_subset,
                                    Offset0, Nsubset, perm, linstat);
        }
    } else {
        PROTECT(blockTable = allocVector(REALSXP, B + 1));
        /* same as RC_OneTableSums(block, noweights, expand_subset) */
        RC_OneTableSums(INTEGER(block), XLENGTH(block), B + 1, weights, subset, Offset0,
                        XLENGTH(subset), REAL(blockTable));
        PROTECT(block_subset = RC_order_subset_wrt_block(XLENGTH(block), expand_subset,
                                                         block, blockTable));

        for (R_xlen_t np = 0; np < inresample; np++) {
            ⟨ Setup Linear Statistic 38a ⟩
            C_doPermuteBlock(REAL(block_subset), Nsubset, REAL(blockTable),
                             B + 1, REAL(tmp), REAL(perm));
            RC_KronSums_Permutation(x, NROW(x), P, REAL(y), Q, block_subset,
                                    Offset0, Nsubset, perm, linstat);
        }
        UNPROTECT(2);
    }
    PutRNGstate();

    UNPROTECT(4);
    return(ans);
}
◇
```

Fragment referenced in 30a.

Defines: `R_PermutedLinearStatistic` 6, 36bc, 152c, 153.

Uses: `B` 27a, `block` 26f, 27b, `blockTable` 27c, `C_doPermute` 128b, `C_doPermuteBlock` 129b, `mPQB` 132a, `N` 23bc, `NROW` 130b, `Nsubset` 26b, `Offset0` 21b, `P` 24a, `Q` 24e, `RC_KronSums_Permutation` 103b, `RC_OneTableSums` 112a, `RC_order_subset_wrt_block` 124b, `RC_setup_subset` 127a, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *Setup Linear Statistic* 38a ⟩ ≡

```
if (np % 256 == 0) R_CheckUserInterrupt();
linstat = REAL(ans) + PQ * np;
for (int p = 0; p < PQ; p++)
    linstat[p] = 0.0;
◇
```

Fragment referenced in 37, 48.

This small function takes an object containing permuted linear statistics and returns the matrix of standardised linear statistics.

⟨ *R_StandardisePermutedLinearStatistic Prototype* 38b ⟩ ≡

```
SEXP R_StandardisePermutedLinearStatistic
(
    SEXP LECV
)
◇
```

Fragment referenced in 22b, 39a.
Uses: `LECV` 141b.

This function can be called from other packages.

`"libcoinAPI.h"` 38c≡

```
extern SEXP libcoin_R_StandardisePermutedLinearStatistic(
    SEXP LECV
) {
    static SEXP(*fun)(SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP))
            R_GetCCallable("libcoin", "R_StandardisePermutedLinearStatistic");
    return fun(LECV);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `LECV` 141b.

⟨ *R_StandardisePermutedLinearStatistic* 39a ⟩ ≡

```
⟨ R_StandardisePermutedLinearStatistic Prototype 38b ⟩
{
    SEXP ans;
    R_xlen_t nresample = C_get_nresample(LECV);
    double *ls;
    if (!nresample) return(R_NilValue);
    int PQ = C_get_P(LECV) * C_get_Q(LECV);

    PROTECT(ans = allocMatrix(REALSXP, PQ, nresample));

    for (R_xlen_t np = 0; np < nresample; np++) {
        ls = REAL(ans) + PQ * np;
        /* copy first; standarisation is in place */
        for (int p = 0; p < PQ; p++)
            ls[p] = C_get_PermutedLinearStatistic(LECV)[p + PQ * np];
        if (C_get_varonly(LECV)) {
            C_standardise(PQ, ls, C_get_Expectation(LECV),
                          C_get_Variance(LECV), 1, C_get_tol(LECV));
        } else {
            C_standardise(PQ, ls, C_get_Expectation(LECV),
                          C_get_Covariance(LECV), 0, C_get_tol(LECV));
        }
    }
    UNPROTECT(1);
    return(ans);
}
◇
```

Fragment referenced in 30a.
Uses: `C_get_Covariance` 143c, `C_get_Expectation` 143a, `C_get_nresample` 146b, `C_get_P` 141c, `C_get_PermutedLinearStatistic` 146c, `C_get_Q` 142a, `C_get_tol` 146d, `C_get_Variance` 143b, `C_get_varonly` 142b, `C_standardise` 64a, `LECV` 141b.

3.4.2 Two-Dimensional Case (“2d”)

⟨ *2d User Interface* 39b ⟩ ≡

```
⟨ RC_ExpectationCovarianceStatistic_2d 45 ⟩
⟨ R_ExpectationCovarianceStatistic_2d 41a ⟩
⟨ R_PermutedLinearStatistic_2d 48 ⟩
◇
```

Fragment referenced in 23a.

⟨ *2d User Interface Input* 39c ⟩ ≡

```
⟨ R x Input 23d ⟩
SEXP ix,
⟨ R y Input 24d ⟩
SEXP iy,
⟨ R weights Input 25b ⟩,
⟨ R subset Input 26a ⟩,
⟨ R block Input 26f ⟩,
◇
```

Fragment referenced in 40a, 45.

⟨ *R_ExpectationCovarianceStatistic_2d Prototype* 40a ⟩ ≡

```
SEXP R_ExpectationCovarianceStatistic_2d
(
    ⟨ 2d User Interface Input 39c ⟩
    SEXP varonly,
    SEXP tol
)
◇
```

Fragment referenced in 22b, 41a.
Uses: `R_ExpectationCovarianceStatistic_2d` 41a.

This function can be called from other packages.

`"libcoinAPI.h"` 40b≡

```
extern SEXP libcoin_R_ExpectationCovarianceStatistic_2d(
    SEXP x, SEXP ix, SEXP y, SEXP iy, SEXP weights, SEXP subset, SEXP block,
    SEXP varonly, SEXP tol
) {
    static SEXP(*fun)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP))
            R_GetCCallable("libcoin", "R_ExpectationCovarianceStatistic_2d");
    return fun(x, ix, y, iy, weights, subset, block, varonly, tol);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `block` 26f, 27b, `R_ExpectationCovarianceStatistic_2d` 41a, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *R_ExpectationCovarianceStatistic_2d* 41a ⟩ ≡

```
⟨ R_ExpectationCovarianceStatistic_2d Prototype 40a ⟩
{
    SEXP ans;
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer Nsubset Input 26b ⟩;
    int Xfactor;

    N = XLENGTH(ix);
    Nsubset = XLENGTH(subset);
    Xfactor = XLENGTH(x) == 0;

    ⟨ Setup Dimensions 2d 41b ⟩

    PROTECT(ans = RC_init_LECV_2d(P, Q, INTEGER(varonly)[0],
                                Lx, Ly, B, Xfactor, REAL(tol)[0]));

    if (B == 1) {
        RC_TwoTableSums(INTEGER(ix), N, Lx + 1, INTEGER(iy), Ly + 1,
                        weights, subset, Offset0, Nsubset,
                        C_get_Table(ans));
    } else {
        RC_ThreeTableSums(INTEGER(ix), N, Lx + 1, INTEGER(iy), Ly + 1,
                          INTEGER(block), B, weights, subset, Offset0, Nsubset,
                          C_get_Table(ans));
    }
    RC_ExpectationCovarianceStatistic_2d(x, ix, y, iy, weights,
                                         subset, block, ans);

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

Fragment referenced in 39b.
Defines: `R_ExpectationCovarianceStatistic_2d` 8, 40ab, 152c, 153.
Uses: `B` 27a, `block` 26f, 27b, `C_get_Table` 145c, `N` 23bc, `Nsubset` 26b, `Offset0` 21b, `P` 24a, `Q` 24e, `RC_init_LECV_2d` 150, `RC_ThreeTableSums` 120b, `RC_TwoTableSums` 116a, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *Setup Dimensions 2d* 41b ⟩ ≡

```
int P, Q, B, Lx, Ly;

if (XLENGTH(x) == 0) {
    P = NLEVELS(ix);
} else {
    P = NCOL(x);
}
Q = NCOL(y);

B = 1;
if (XLENGTH(block) > 0)
    B = NLEVELS(block);

Lx = NLEVELS(ix);
Ly = NLEVELS(iy);
◇
```

Fragment referenced in 41a, 48.
Uses: `B` 27a, `block` 26f, 27b, `NCOL` 130c, `NLEVELS` 131a, `P` 24a, `Q` 24e, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *Linear Statistic 2d* 42a ⟩ ≡

```
    if (Xfactor) {
        for (int j = 1; j < Lyp1; j++) {     /* j = 0 means NA */
            for (int i = 1; i < Lxp1; i++) { /* i = 0 means NA */
                for (int q = 0; q < Q; q++)
                    linstat[q * (Lxp1 - 1) + (i - 1)] +=
                        btab[j * Lxp1 + i] * REAL(y)[q * Lyp1 + j];
            }
        }
    } else {
        for (int p = 0; p < P; p++) {
            for (int q = 0; q < Q; q++) {
                int qPp = q * P + p;
                int qLy = q * Lyp1;
                for (int i = 0; i < Lxp1; i++) {
                    int pLxi = p * Lxp1 + i;
                    for (int j = 0; j < Lyp1; j++)
                        linstat[qPp] += REAL(y)[qLy + j] * REAL(x)[pLxi] * btab[j * Lxp1 + i];
                }
            }
        }
    }
    ◇
```

Fragment referenced in 45, 49d.
Uses: `P` 24a, `Q` 24e, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *2d Total Table* 42b ⟩ ≡

```
    for (int i = 0; i < Lxp1 * Lyp1; i++)
        table2d[i] = 0.0;
    for (int b = 0; b < B; b++) {
        for (int i = 0; i < Lxp1; i++) {
            for (int j = 0; j < Lyp1; j++)
                table2d[j * Lxp1 + i] += table[b * Lxp1 * Lyp1 + j * Lxp1 + i];
        }
    }
    ◇
```

Fragment referenced in 45.
Uses: `B` 27a.

⟨ *Col Row Total Sums* 43a ⟩ ≡

```
/* Remember: first row / column count NAs */
/* column sums */
for (int q = 1; q < Lyp1; q++) {
    csum[q] = 0;
    for (int p = 1; p < Lxp1; p++)
        csum[q] += btab[q * Lxp1 + p];
}
csum[0] = 0; /* NA */
/* row sums */
for (int p = 1; p < Lxp1; p++) {
    rsum[p] = 0;
    for (int q = 1; q < Lyp1; q++)
        rsum[p] += btab[q * Lxp1 + p];
}
rsum[0] = 0; /* NA */
/* total sum */
sumweights[b] = 0;
for (int i = 1; i < Lxp1; i++) sumweights[b] += rsum[i];
◇
```

Fragment referenced in 45, 48.
Uses: `sumweights` 25e.

⟨ *2d Expectation* 43b ⟩ ≡

```
RC_ExpectationInfluence(NROW(y), y, Q, Rcsum, subset, Offset0, 0, sumweights[b], ExpInf);

if (LENGTH(x) == 0) {
    for (int p = 0; p < P; p++)
        ExpX[p] = rsum[p + 1];
    } else {
        RC_ExpectationX(x, NROW(x), P, Rrsum, subset, Offset0, 0, ExpX);
}

C_ExpectationLinearStatistic(P, Q, ExpInf, ExpX, b, C_get_Expectation(ans));
◇
```

Fragment referenced in 45.
Uses: `C_ExpectationLinearStatistic` 78b, `C_get_Expectation` 143a, `NROW` 130b, `Offset0` 21b, `P` 24a, `Q` 24e, `RC_ExpectationInfluence` 81c, `RC_ExpectationX` 85b, `subset` 26ade, `sumweights` 25e, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *2d Covariance* 44 ⟩ ≡

```
/* C_ordered_Xfactor needs both VarInf and CovInf */
RC_CovarianceInfluence(NROW(y), y, Q, Rcsum, subset, Offset0, 0, ExpInf, sumweights[b],
                       !DoVarOnly, C_get_CovarianceInfluence(ans));
for (int q = 0; q < Q; q++)
    C_get_VarianceInfluence(ans)[q] = C_get_CovarianceInfluence(ans)[S(q, q, Q)];

if (C_get_varonly(ans)) {
    if (LENGTH(x) == 0) {
        for (int p = 0; p < P; p++) CovX[p] = ExpX[p];
    } else {
        RC_CovarianceX(x, NROW(x), P, Rrsum, subset, Offset0, 0, ExpX, DoVarOnly, CovX);
    }
    C_VarianceLinearStatistic(P, Q, C_get_VarianceInfluence(ans),
                              ExpX, CovX, sumweights[b], b,
                              C_get_Variance(ans));
} else {
    if (LENGTH(x) == 0) {
        for (int p = 0; p < PP12(P); p++) CovX[p] = 0.0;
        for (int p = 0; p < P; p++) CovX[S(p, p, P)] = ExpX[p];
    } else {
        RC_CovarianceX(x, NROW(x), P, Rrsum, subset, Offset0, 0, ExpX, !DoVarOnly, CovX);
    }
    C_CovarianceLinearStatistic(P, Q, C_get_CovarianceInfluence(ans),
                                ExpX, CovX, sumweights[b], b,
                                C_get_Covariance(ans));
}
◇
```

Fragment referenced in 45.
Uses: `C_CovarianceLinearStatistic` 79, `C_get_Covariance` 143c, `C_get_CovarianceInfluence` 144c, `C_get_Variance` 143b, `C_get_VarianceInfluence` 144d, `C_get_varonly` 142b, `C_ordered_Xfactor` 70, `C_VarianceLinearStatistic` 80a, `DoVarOnly` 21b, `NROW` 130b, `Offset0` 21b, `P` 24a, `PP12` 131b, `Q` 24e, `RC_CovarianceInfluence` 84a, `RC_CovarianceX` 88a, `S` 21a, `subset` 26ade, `sumweights` 25e, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *RC_ExpectationCovarianceStatistic_2d* 45 ⟩ ≡

```
void RC_ExpectationCovarianceStatistic_2d
(
    ⟨ 2d User Interface Input 39c ⟩
    SEXP ans
) {
    ⟨ 2d Memory 46 ⟩

    ⟨ 2d Total Table 42b ⟩

    linstat = C_get_LinearStatistic(ans);
    for (int p = 0; p < mPQB(P, Q, 1); p++)
        linstat[p] = 0.0;

    for (int b = 0; b < B; b++) {
        btab = table + Lxp1 * Lyp1 * b;

        ⟨ Linear Statistic 2d 42a ⟩

        ⟨ Col Row Total Sums 43a ⟩

        ⟨ 2d Expectation 43b ⟩

        ⟨ 2d Covariance 44 ⟩
    }

    /* always return variances */
    if (!C_get_varonly(ans)) {
        for (int p = 0; p < mPQB(P, Q, 1); p++)
            C_get_Variance(ans)[p] = C_get_Covariance(ans)[S(p, p, mPQB(P, Q, 1))];
    }

    R_Free(CovX);
    R_Free(table2d);
    UNPROTECT(2);
}
◇
```

Fragment referenced in 39b.
Defines: `RC_ExpectationCovarianceStatistic` 31a, 32.
Uses: `B` 27a, `C_get_Covariance` 143c, `C_get_LinearStatistic` 142d, `C_get_Variance` 143b, `C_get_varonly` 142b, `mPQB` 132a, `P` 24a, `Q` 24e, `S` 21a.

⟨ *2d Memory* 46 ⟩ ≡

```
SEXP Rcsum, Rrsum;
int P, Q, Lxp1, Lyp1, B, Xfactor;
double *ExpInf, *ExpX, *CovX;
double *table, *table2d, *csum, *rsum, *sumweights, *btab, *linstat;

P = C_get_P(ans);
Q = C_get_Q(ans);

ExpInf = C_get_ExpectationInfluence(ans);
ExpX = C_get_ExpectationX(ans);
table = C_get_Table(ans);
sumweights = C_get_Sumweights(ans);

Lxp1 = C_get_dimTable(ans)[0];
Lyp1 = C_get_dimTable(ans)[1];
B = C_get_B(ans);
Xfactor = C_get_Xfactor(ans);

if (C_get_varonly(ans)) {
    CovX = R_Calloc(P, double);
} else {
    CovX = R_Calloc(PP12(P), double);
}

table2d = R_Calloc(Lxp1 * Lyp1, double);
PROTECT(Rcsum = allocVector(REALSXP, Lyp1));
csum = REAL(Rcsum);
PROTECT(Rrsum = allocVector(REALSXP, Lxp1));
rsum = REAL(Rrsum);
◇
```

Fragment referenced in 45.
Uses: `B` 27a, `C_get_B` 146a, `C_get_dimTable` 145d, `C_get_ExpectationInfluence` 144b, `C_get_ExpectationX` 144a, `C_get_P` 141c, `C_get_Q` 142a, `C_get_Sumweights` 145b, `C_get_Table` 145c, `C_get_varonly` 142b, `C_get_Xfactor` 142c, `P` 24a, `PP12` 131b, `Q` 24e, `sumweights` 25e.

```
> LinStatExpCov(X = iX2d, ix = ix, Y = iY2d, iy = iy,
+               weights = weights, subset = subset, nresample = 10)$PermutedLinearStatistic

           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
 [1,] 15.486226 15.432786 15.474636 15.434733 15.515989 15.421226 15.523577
 [2,]  7.864472  7.948006  8.105228  8.390763  8.212044  8.493673  8.415919
 [3,] 12.398382 12.290212 11.905712 12.506639 12.143855 12.549147 12.590900
 [4,] 31.244688 31.476627 31.257920 31.264541 31.096744 31.405390 31.259421
 [5,] 17.500047 17.688850 17.055915 15.065147 16.586396 15.315949 16.382058
 [6,] 24.466142 24.647923 25.464893 24.239801 25.488434 24.296588 23.694321
 [7,] 43.423057 43.421097 43.330443 43.612924 43.424099 43.430492 43.309780
 [8,] 24.311651 23.474319 22.844531 23.490053 23.541204 22.749540 22.388328
 [9,] 34.180046 34.430423 35.397534 33.988759 34.366957 33.293748 33.389741
[10,] 13.461330 13.490553 13.492064 13.434007 13.447127 13.491634 13.476994
[11,]  6.973432  7.169633  7.259611  6.943487  7.017767  7.094398  7.241183
[12,] 10.672723 10.658055 10.574382 10.675107 10.743783 10.837748 10.788257
           [,8]      [,9]     [,10]
 [1,] 15.434192 15.491818 15.398248
 [2,]  7.834800  8.223809  7.925817
 [3,] 12.362877 11.997518 12.169918
 [4,] 31.510352 31.284964 31.576643
 [5,] 18.211173 16.969768 17.197270
 [6,] 23.773081 25.183959 24.742788
 [7,] 43.375471 43.374905 43.496870
 [8,] 23.445718 22.372659 23.729797
 [9,] 34.264857 35.341197 34.487409
[10,] 13.498456 13.473376 13.482370
[11,]  7.221054  7.329256  7.097392
[12,] 10.669965 10.540119 10.702889
```

⟨ *R_PermutedLinearStatistic_2d Prototype* 47a ⟩ ≡

```
SEXP R_PermutedLinearStatistic_2d
(
    ⟨ R x Input 23d ⟩
    SEXP ix,
    ⟨ R y Input 24d ⟩
    SEXP iy,
    ⟨ R block Input 26f ⟩,
    SEXP nresample,
    SEXP itable
)
◇
```

Fragment referenced in 22b, 48.
Uses: `R_PermutedLinearStatistic_2d` 48.

This function can be called from other packages.

`"libcoinAPI.h"` 47b≡

```
extern SEXP libcoin_R_PermutedLinearStatistic_2d(
    SEXP x, SEXP ix, SEXP y, SEXP iy, SEXP block, SEXP nresample,
    SEXP itable
) {
    static SEXP(*fun)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP))
            R_GetCCallable("libcoin", "R_PermutedLinearStatistic_2d");
    return fun(x, ix, y, iy, block, nresample, itable);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `block` 26f, 27b, `R_PermutedLinearStatistic_2d` 48, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *R_PermutedLinearStatistic_2d* 48 ⟩ ≡

```
⟨ R_PermutedLinearStatistic_2d Prototype 47a ⟩
{
    SEXP ans, Ritable;
    int *csum, *rsum, *sumweights, *jwork, *table, *rtable2, maxn = 0, Lxp1, Lyp1, *btab, PQ, Xfactor;
    R_xlen_t inresample;
    double *fact, *linstat;

    ⟨ Setup Dimensions 2d 41b ⟩

    PQ = mPQB(P, Q, 1);
    Xfactor = XLENGTH(x) == 0;
    Lxp1 = Lx + 1;
    Lyp1 = Ly + 1;
    inresample = (R_xlen_t) REAL(nresample)[0];

    PROTECT(ans = allocMatrix(REALSXP, PQ, inresample));

    ⟨ Setup Working Memory 49b ⟩

    ⟨ Convert Table to Integer 49a ⟩

    for (int b = 0; b < B; b++) {
        btab = INTEGER(Ritable) + Lxp1 * Lyp1 * b;
        ⟨ Col Row Total Sums 43a ⟩
        if (sumweights[b] > maxn) maxn = sumweights[b];
    }

    ⟨ Setup Log-Factorials 49c ⟩

    GetRNGstate();

    for (R_xlen_t np = 0; np < inresample; np++) {
        ⟨ Setup Linear Statistic 38a ⟩

        for (int p = 0; p < Lxp1 * Lyp1; p++)
            table[p] = 0;

        for (int b = 0; b < B; b++) {
            ⟨ Compute Permuted Linear Statistic 2d 49d ⟩
        }
    }

    PutRNGstate();

    R_Free(csum); R_Free(rsum); R_Free(sumweights); R_Free(rtable2);
    R_Free(jwork); R_Free(fact); R_Free(table);
    UNPROTECT(2);
    return(ans);
}
◇
```

Fragment referenced in 39b.
Defines: `R_PermutedLinearStatistic_2d` 8, 47ab, 49a, 152c, 153.
Uses: `B` 27a, `mPQB` 132a, `P` 24a, `Q` 24e, `sumweights` 25e, `x` 23d, 24bc.

⟨ *Convert Table to Integer* 49a ⟩ ≡

```
PROTECT(Ritable = allocVector(INTSXP, LENGTH(itable)));
for (int i = 0; i < LENGTH(itable); i++) {
    if (REAL(itable)[i] > INT_MAX)
        error("cannot deal with weights larger INT_MAX in R_PermutedLinearStatistic_2d");
    INTEGER(Ritable)[i] = (int) REAL(itable)[i];
}
◇
```

Fragment referenced in 48.
Uses: `R_PermutedLinearStatistic_2d` 48, `weights` 25b.

⟨ *Setup Working Memory* 49b ⟩ ≡

```
csum = R_Calloc(Lyp1 * B, int);
rsum = R_Calloc(Lxp1 * B, int);
sumweights = R_Calloc(B, int);
table = R_Calloc(Lxp1 * Lyp1, int);
rtable2 = R_Calloc(Lx * Ly , int);
jwork = R_Calloc(Lyp1, int);
◇
```

Fragment referenced in 48.
Uses: `B` 27a, `sumweights` 25e.

⟨ *Setup Log-Factorials* 49c ⟩ ≡

```
fact = R_Calloc(maxn + 1, double);
/* Calculate log-factorials.  fact[i] = lgamma(i+1) */
fact[0] = fact[1] = 0.;
for (int j = 2; j <= maxn; j++)
    fact[j] = fact[j - 1] + log(j);
◇
```

Fragment referenced in 48.

Note: the interface to `S_rcont2` changed in R-4.1.0.

⟨ *Compute Permuted Linear Statistic 2d* 49d ⟩ ≡

```
#if defined(R_VERSION) && R_VERSION >= R_Version(4, 1, 0)
            S_rcont2(Lx, Ly,
                     rsum + Lxp1 * b + 1,
                     csum + Lyp1 * b + 1,
                     sumweights[b], fact, jwork, rtable2);
#else
            S_rcont2(&Lx, &Ly,
                     rsum + Lxp1 * b + 1,
                     csum + Lyp1 * b + 1,
                     sumweights + b, fact, jwork, rtable2);
#endif

for (int j1 = 1; j1 <= Lx; j1++) {
    for (int j2 = 1; j2 <= Ly; j2++)
        table[j2 * Lxp1 + j1] = rtable2[(j2 - 1) * Lx + (j1 - 1)];
}
btab = table;
⟨ Linear Statistic 2d 42a ⟩
◇
```

Fragment referenced in 48.
Uses: `sumweights` 25e.

3.5 Tests

⟨ *Tests* 50a ⟩ ≡

```
⟨ R_QuadraticTest 51 ⟩
⟨ R_MaximumTest 54 ⟩
⟨ R_MaximallySelectedTest 56 ⟩
◇
```

Fragment referenced in 23a.

⟨ *R_QuadraticTest Prototype* 50b ⟩ ≡

```
SEXP R_QuadraticTest
(
    ⟨ R LECV Input 141b ⟩,
    SEXP pvalue,
    SEXP lower,
    SEXP give_log,
    SEXP PermutedStatistics
)
◇
```

Fragment referenced in 22b, 51.

This function can be called from other packages.

`"libcoinAPI.h"` 50c≡

```
extern SEXP libcoin_R_QuadraticTest(
    SEXP LECV, SEXP pvalue, SEXP lower, SEXP give_log, SEXP PermutedStatistics
) {
    static SEXP(*fun)(SEXP, SEXP, SEXP, SEXP, SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP, SEXP, SEXP, SEXP, SEXP))
            R_GetCCallable("libcoin", "R_QuadraticTest");
    return fun(LECV, pvalue, lower, give_log, PermutedStatistics);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `LECV` 141b.

⟨ *R_QuadraticTest* 51 ⟩ ≡

```
⟨ R_QuadraticTest Prototype 50b ⟩
{
    SEXP ans, stat, pval, names, permstat;
    double *MPinv, *ls, st, pst, *ex;
    int rank, P, Q, PQ, greater = 0;
    R_xlen_t nresample;

    ⟨ Setup Test Memory 52a ⟩

    MPinv = R_Calloc(PP12(PQ), double); /* was: C_get_MPinv(LECV); */
    C_MPinv_sym(C_get_Covariance(LECV), PQ, C_get_tol(LECV), MPinv, &rank);

    REAL(stat)[0] = C_quadform(PQ, C_get_LinearStatistic(LECV),
                               C_get_Expectation(LECV), MPinv);

    if (!PVALUE) {
        UNPROTECT(2);
        R_Free(MPinv);
        return(ans);
    }

    if (C_get_nresample(LECV) == 0) {
        REAL(pval)[0] = C_chisq_pvalue(REAL(stat)[0], rank, LOWER, GIVELOG);
    } else {
        nresample = C_get_nresample(LECV);
        ls = C_get_PermutedLinearStatistic(LECV);
        st = REAL(stat)[0];
        ex = C_get_Expectation(LECV);
        greater = 0;
        for (R_xlen_t np = 0; np < nresample; np++) {
            pst = C_quadform(PQ, ls + PQ * np, ex, MPinv);
            if (GE(pst, st, C_get_tol(LECV)))
                greater++;
            if (PSTAT) REAL(permstat)[np] = pst;
        }
        REAL(pval)[0] = C_perm_pvalue(greater, nresample, LOWER, GIVELOG);
    }

    UNPROTECT(2);
    R_Free(MPinv);
    return(ans);
}
◇
```

Fragment referenced in 50a.
Uses: `C_chisq_pvalue` 64c, `C_get_Covariance` 143c, `C_get_Expectation` 143a, `C_get_LinearStatistic` 142d, `C_get_nresample` 146b, `C_get_PermutedLinearStatistic` 146c, `C_get_tol` 146d, `C_perm_pvalue` 65, `C_quadform` 62, `GE` 21a, `LECV` 141b, `P` 24a, `PP12` 131b, `Q` 24e.

⟨ *Setup Test Memory* 52a ⟩ ≡

```
P = C_get_P(LECV);
Q = C_get_Q(LECV);
PQ = mPQB(P, Q, 1);

if (C_get_varonly(LECV) && PQ > 1)
        error("cannot compute adjusted p-value based on variances only");
/*  if (C_get_nresample(LECV) > 0 && INTEGER(PermutedStatistics)[0]) { */
    PROTECT(ans = allocVector(VECSXP, 3));
    PROTECT(names = allocVector(STRSXP, 3));
    SET_VECTOR_ELT(ans, 2, permstat = allocVector(REALSXP, C_get_nresample(LECV)));
    SET_STRING_ELT(names, 2, mkChar("PermutedStatistics"));
/*  } else {
    PROTECT(ans = allocVector(VECSXP, 2));
    PROTECT(names = allocVector(STRSXP, 2));
}
*/
SET_VECTOR_ELT(ans, 0, stat = allocVector(REALSXP, 1));
SET_STRING_ELT(names, 0, mkChar("TestStatistic"));
SET_VECTOR_ELT(ans, 1, pval = allocVector(REALSXP, 1));
SET_STRING_ELT(names, 1, mkChar("p.value"));
namesgets(ans, names);
REAL(pval)[0] = NA_REAL;
int LOWER = INTEGER(lower)[0];
int GIVELOG = INTEGER(give_log)[0];
int PVALUE = INTEGER(pvalue)[0];
int PSTAT = INTEGER(PermutedStatistics)[0];
◇
```

Fragment referenced in 51, 54.
Uses: `C_get_nresample` 146b, `C_get_P` 141c, `C_get_Q` 142a, `C_get_varonly` 142b, `LECV` 141b, `mPQB` 132a, `P` 24a, `Q` 24e.

⟨ *R_MaximumTest Prototype* 52b ⟩ ≡

```
SEXP R_MaximumTest
(
    ⟨ R LECV Input 141b ⟩,
    SEXP alternative,
    SEXP pvalue,
    SEXP lower,
    SEXP give_log,
    SEXP PermutedStatistics,
    SEXP maxpts,
    SEXP releps,
    SEXP abseps
)
◇
```

Fragment referenced in 22b, 54.

This function can be called from other packages.

"libcoinAPI.h" 53≡

```
extern SEXP libcoin_R_MaximumTest(
    SEXP LECV, SEXP alternative, SEXP pvalue, SEXP lower, SEXP give_log,
    SEXP PermutedStatistics, SEXP maxpts, SEXP releps, SEXP abseps
) {
  static SEXP(*fun)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP))
            R_GetCCallable("libcoin", "R_MaximumTest");
    return fun(LECV, alternative, pvalue, lower, give_log, PermutedStatistics, maxpts, releps,
               abseps);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `LECV` 141b.

⟨ *R_MaximumTest* 54 ⟩ ≡

```
⟨ R_MaximumTest Prototype 52b ⟩
{
    SEXP ans, stat, pval, names, permstat;
    double st, pst, *ex, *cv, *ls, tl;
    int P, Q, PQ, vo, alt, greater;
    R_xlen_t nresample;

    ⟨ Setup Test Memory 52a ⟩

    if (C_get_varonly(LECV)) {
        cv = C_get_Variance(LECV);
    } else {
        cv = C_get_Covariance(LECV);
    }
    REAL(stat)[0] = C_maxtype(PQ, C_get_LinearStatistic(LECV),
        C_get_Expectation(LECV), cv, C_get_varonly(LECV), C_get_tol(LECV),
        INTEGER(alternative)[0]);
    if (!PVALUE) {
        UNPROTECT(2);
        return(ans);
    }

    if (C_get_nresample(LECV) == 0) {
        if (C_get_varonly(LECV) && PQ > 1) {
            REAL(pval)[0] = NA_REAL;
            UNPROTECT(2);
            return(ans);
        }
        REAL(pval)[0] = C_maxtype_pvalue(REAL(stat)[0], cv,
            PQ, INTEGER(alternative)[0], LOWER, GIVELOG, INTEGER(maxpts)[0],
            REAL(releps)[0], REAL(abseps)[0], C_get_tol(LECV));
    } else {
        nresample = C_get_nresample(LECV);
        ls = C_get_PermutedLinearStatistic(LECV);
        ex = C_get_Expectation(LECV);
        vo = C_get_varonly(LECV);
        alt = INTEGER(alternative)[0];
        st = REAL(stat)[0];
        tl = C_get_tol(LECV);
        greater = 0;
        for (R_xlen_t np = 0; np < nresample; np++) {
            pst = C_maxtype(PQ, ls + PQ * np, ex, cv, vo, tl, alt);
            if (alt == ALTERNATIVE_less) {
                if (LE(pst, st, tl)) greater++;
            } else {
                if (GE(pst, st, tl)) greater++;
            }
            if (PSTAT) REAL(permstat)[np] = pst;
        }
        REAL(pval)[0] = C_perm_pvalue(greater, nresample, LOWER, GIVELOG);
    }
    UNPROTECT(2);
    return(ans);
}
◇
```

Fragment referenced in 50a.
Uses: `C_get_Covariance` 143c, `C_get_Expectation` 143a, `C_get_LinearStatistic` 142d, `C_get_nresample` 146b, `C_get_tol` 146d, `C_get_Variance` 143b, `C_get_varonly` 142b, `C_maxtype` 63, `C_maxtype_pvalue` 67, `C_perm_pvalue` 65, `GE` 21a, `LE` 21a, `LECV` 141b, `P` 24a, `Q` 24e.

⟨ *R_MaximallySelectedTest Prototype* 55a ⟩ ≡

```
SEXP R_MaximallySelectedTest
(
    SEXP LECV,
    SEXP ordered,
    SEXP teststat,
    SEXP minbucket,
    SEXP lower,
    SEXP give_log
)
◇
```

Fragment referenced in 22b, 56.
Uses: `LECV` 141b.

This function can be called from other packages.

`"libcoinAPI.h"` 55b≡

```
extern SEXP libcoin_R_MaximallySelectedTest(
    SEXP LECV, SEXP ordered, SEXP teststat, SEXP minbucket, SEXP lower, SEXP give_log
) {
    static SEXP(*fun)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP, SEXP, SEXP, SEXP, SEXP, SEXP))
            R_GetCCallable("libcoin", "R_MaximallySelectedTest");
    return fun(LECV, ordered, teststat, minbucket, lower, give_log);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `LECV` 141b.

⟨ *R_MaximallySelectedTest* 56 ⟩ ≡

```
⟨ R_MaximallySelectedTest Prototype 55a ⟩
{
    SEXP ans, index, stat, pval, names, permstat;
    int P, mb;

    P = C_get_P(LECV);
    mb = INTEGER(minbucket)[0];

    PROTECT(ans = allocVector(VECSXP, 4));
    PROTECT(names = allocVector(STRSXP, 4));
    SET_VECTOR_ELT(ans, 0, stat = allocVector(REALSXP, 1));
    SET_STRING_ELT(names, 0, mkChar("TestStatistic"));
    SET_VECTOR_ELT(ans, 1, pval = allocVector(REALSXP, 1));
    SET_STRING_ELT(names, 1, mkChar("p.value"));
    SET_VECTOR_ELT(ans, 3, permstat = allocVector(REALSXP, C_get_nresample(LECV)));
    SET_STRING_ELT(names, 3, mkChar("PermutedStatistics"));
    REAL(pval)[0] = NA_REAL;

    if (INTEGER(ordered)[0]) {
        SET_VECTOR_ELT(ans, 2, index = allocVector(INTSXP, 1));
        C_ordered_Xfactor(LECV, mb, INTEGER(teststat)[0],
                          INTEGER(index), REAL(stat), REAL(permstat),
                          REAL(pval), INTEGER(lower)[0],
                          INTEGER(give_log)[0]);
        if (REAL(stat)[0] > 0)
            INTEGER(index)[0]++; /* R style indexing */
    } else {
        SET_VECTOR_ELT(ans, 2, index = allocVector(INTSXP, P));
        C_unordered_Xfactor(LECV, mb, INTEGER(teststat)[0],
                            INTEGER(index), REAL(stat), REAL(permstat),
                            REAL(pval), INTEGER(lower)[0],
                            INTEGER(give_log)[0]);
    }

    SET_STRING_ELT(names, 2, mkChar("index"));
    namesgets(ans, names);

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

Fragment referenced in 50a.
Uses: `C_get_nresample` 146b, `C_get_P` 141c, `C_ordered_Xfactor` 70, `C_unordered_Xfactor` 74, `LECV` 141b, `P` 24a.

3.6 Test Statistics

⟨ *Test Statistics* 57a ⟩ ≡

⟨ *C_maxstand_Covariance* 57b ⟩
⟨ *C_maxstand_Variance* 58a ⟩
⟨ *C_minstand_Covariance* 58b ⟩
⟨ *C_minstand_Variance* 59a ⟩
⟨ *C_maxabsstand_Covariance* 59b ⟩
⟨ *C_maxabsstand_Variance* 60a ⟩
⟨ *C_quadform* 62 ⟩
⟨ *R_quadform* 61b ⟩
⟨ *C_maxtype* 63 ⟩
⟨ *C_standardise* 64a ⟩
⟨ *C_ordered_Xfactor* 70 ⟩
⟨ *C_unordered_Xfactor* 74 ⟩
◇

Fragment referenced in 23a.

⟨ *C_maxstand_Covariance* 57b ⟩ ≡

```
double C_maxstand_Covariance
(
    const int PQ,
    const double *linstat,
    const double *expect,
    const double *covar_sym,
    const double tol
) {
    double ans = R_NegInf, tmp = 0.0;

    for (int p = 0; p < PQ; p++) {
        tmp = 0.0;
        if (covar_sym[S(p, p, PQ)] > tol)
            tmp = (linstat[p] - expect[p]) / sqrt(covar_sym[S(p, p, PQ)]);
        if (tmp > ans) ans = tmp;
    }
    return(ans);
}
```
◇

Fragment referenced in 57a.
Defines: `C_maxstand_Covariance` 63.
Uses: `S` 21a.

⟨ *C_maxstand_Variance* 58a ⟩ ≡

```
double C_maxstand_Variance
(
    const int PQ,
    const double *linstat,
    const double *expect,
    const double *var,
    const double tol
) {
    double ans = R_NegInf, tmp = 0.0;

    for (int p = 0; p < PQ; p++) {
        tmp = 0.0;
        if (var[p] > tol)
            tmp = (linstat[p] - expect[p]) / sqrt(var[p]);
        if (tmp > ans) ans = tmp;
    }
    return(ans);
}
◇
```

Fragment referenced in 57a.
Defines: `C_maxstand_Variance` 63.

⟨ *C_minstand_Covariance* 58b ⟩ ≡

```
double C_minstand_Covariance
(
    const int PQ,
    const double *linstat,
    const double *expect,
    const double *covar_sym,
    const double tol
) {
    double ans = R_PosInf, tmp = 0.0;

    for (int p = 0; p < PQ; p++) {
        tmp = 0.0;
        if (covar_sym[S(p, p, PQ)] > tol)
            tmp = (linstat[p] - expect[p]) / sqrt(covar_sym[S(p, p, PQ)]);
        if (tmp < ans) ans = tmp;
    }
    return(ans);
}
◇
```

Fragment referenced in 57a.
Defines: `C_minstand_Covariance` 63.
Uses: `S` 21a.

⟨ *C_minstand_Variance* 59a ⟩ ≡

```
double C_minstand_Variance
(
    const int PQ,
    const double *linstat,
    const double *expect,
    const double *var,
    const double tol
) {
    double ans = R_PosInf, tmp = 0.0;

    for (int p = 0; p < PQ; p++) {
        tmp = 0.0;
        if (var[p] > tol)
            tmp = (linstat[p] - expect[p]) / sqrt(var[p]);
        if (tmp < ans) ans = tmp;
    }
    return(ans);
}
◇
```

Fragment referenced in 57a.
Defines: `C_minstand_Variance` 63.

⟨ *C_maxabsstand_Covariance* 59b ⟩ ≡

```
double C_maxabsstand_Covariance
(
    const int PQ,
    const double *linstat,
    const double *expect,
    const double *covar_sym,
    const double tol
) {
    double ans = R_NegInf, tmp = 0.0;

    for (int p = 0; p < PQ; p++) {
        tmp = 0.0;
        if (covar_sym[S(p, p, PQ)] > tol)
            tmp = fabs((linstat[p] - expect[p]) /
                  sqrt(covar_sym[S(p, p, PQ)]));
        if (tmp > ans) ans = tmp;
    }
    return(ans);
}
◇
```

Fragment referenced in 57a.
Defines: `C_maxabsstand_Covariance` 63.
Uses: `S` 21a.

⟨ *C_maxabsstand_Variance* 60a ⟩ ≡

```
double C_maxabsstand_Variance
(
    const int PQ,
    const double *linstat,
    const double *expect,
    const double *var,
    const double tol
) {
    double ans = R_NegInf, tmp = 0.0;

    for (int p = 0; p < PQ; p++) {
        tmp = 0.0;
        if (var[p] > tol)
            tmp = fabs((linstat[p] - expect[p]) / sqrt(var[p]));
        if (tmp > ans) ans = tmp;
    }
    return(ans);
}
◇
```

Fragment referenced in 57a.
Defines: `C_maxabsstand_Variance` 63.

```
> MPinverse <-
+ function(x, tol = sqrt(.Machine$double.eps))
+ {
+     SVD <- svd(x)
+     pos <- SVD$d > max(tol * SVD$d[1L], 0)
+     inv <- SVD$v[, pos, drop = FALSE] %*%
+              ((1/SVD$d[pos]) * t(SVD$u[, pos, drop = FALSE]))
+     list(MPinv = inv, rank = sum(pos))
+ }
> quadform <-
+ function(linstat, expect, MPinv)
+ {
+     censtat <- linstat - expect
+     censtat %*% MPinv %*% censtat
+ }
> linstat <- ls1$LinearStatistic
> expect <- ls1$Expectation
> MPinv <- MPinverse(vcov(ls1))$MPinv
> MPinv_sym <- MPinv[lower.tri(MPinv, diag = TRUE)]
> qf1 <- quadform(linstat, expect, MPinv)
> qf2 <- .Call(libcoin:::R_quadform, linstat, expect, MPinv_sym)
> stopifnot(isequal(qf1, qf2))
```

⟨ *R_quadform Prototype* 60b ⟩ ≡

```
SEXP R_quadform
(
    SEXP linstat,
    SEXP expect,
    SEXP MPinv_sym
)
◇
```

Fragment referenced in 22b, 61b.
Uses: `R_quadform` 61b.

This function can be called from other packages.

"libcoinAPI.h" 61a≡

```
extern SEXP libcoin_R_quadform(
    SEXP linstat, SEXP expect, SEXP MPinv_sym
) {
    static SEXP(*fun)(SEXP, SEXP, SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP, SEXP, SEXP))
            R_GetCCallable("libcoin", "R_quadform");
    return fun(linstat, expect, MPinv_sym);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `R_quadform` 61b.

⟨ *R_quadform* 61b ⟩ ≡

```
⟨ R_quadform Prototype 60b ⟩
{
    SEXP ans;
    int n, PQ;
    double *dlinstat, *dexpect, *dMPinv_sym, *dans;

    n = NCOL(linstat);
    PQ = NROW(linstat);
    dlinstat = REAL(linstat);
    dexpect = REAL(expect);
    dMPinv_sym = REAL(MPinv_sym);

    PROTECT(ans = allocVector(REALSXP, n));
    dans = REAL(ans);
    for (int i = 0; i < n; i++)
      dans[i] = C_quadform(PQ, dlinstat + PQ * i, dexpect, dMPinv_sym);

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

Fragment referenced in 57a.
Defines: `R_quadform` 60b, 61a, 152c, 153.
Uses: `C_quadform` 62, `NCOL` 130c, `NROW` 130b.

⟨ *C_quadform* 62 ⟩ ≡

```
double C_quadform
(
    const int PQ,
    const double *linstat,
    const double *expect,
    const double *MPinv_sym
) {
    double ans = 0.0, tmp = 0.0;

    for (int q = 0; q < PQ; q++) {
        tmp = 0.0;
        for (int p = 0; p < PQ; p++)
            tmp += (linstat[p] - expect[p]) * MPinv_sym[S(p, q, PQ)];
        ans += tmp * (linstat[q] - expect[q]);
    }

    return(ans);
}
◇
```

Fragment referenced in 57a.
Defines: `C_quadform` 51, 61b, 73c.
Uses: `S` 21a.

⟨ *C_maxtype* 63 ⟩ ≡

```
double C_maxtype
(
    const int PQ,
    const double *linstat,
    const double *expect,
    const double *covar,
    const int varonly,
    const double tol,
    const int alternative
) {
    double ret = 0.0;

    if (varonly) {
        if (alternative == ALTERNATIVE_twosided) {
            ret = C_maxabsstand_Variance(PQ, linstat, expect, covar, tol);
        } else if (alternative == ALTERNATIVE_less) {
            ret = C_minstand_Variance(PQ, linstat, expect, covar, tol);
        } else if (alternative == ALTERNATIVE_greater) {
            ret = C_maxstand_Variance(PQ, linstat, expect, covar, tol);
        }
    } else {
        if (alternative == ALTERNATIVE_twosided) {
            ret = C_maxabsstand_Covariance(PQ, linstat, expect, covar, tol);
        } else if (alternative == ALTERNATIVE_less) {
            ret = C_minstand_Covariance(PQ, linstat, expect, covar, tol);
        } else if (alternative == ALTERNATIVE_greater) {
            ret = C_maxstand_Covariance(PQ, linstat, expect, covar, tol);
        }
    }
    return(ret);
}
```

◇

Fragment referenced in 57a.
Defines: `C_maxtype` 54, 73c.
Uses: `C_maxabsstand_Covariance` 59b, `C_maxabsstand_Variance` 60a, `C_maxstand_Covariance` 57b, `C_maxstand_Variance` 58a, `C_minstand_Covariance` 58b, `C_minstand_Variance` 59a.

⟨ *C_standardise* 64a ⟩ ≡

```
void C_standardise
(
    const int PQ,
    double *linstat,      /* in place standardisation */
    const double *expect,
    const double *covar,
    const int varonly,
    const double tol
) {
    double var;

    for (int p = 0; p < PQ; p++) {
        if (varonly) {
            var = covar[p];
        } else {
            var = covar[S(p, p, PQ)];
        }
        if (var > tol) {
            linstat[p] = (linstat[p] - expect[p]) / sqrt(var);
        } else {
            linstat[p] = NAN;
        }
    }
}
```

◇

Fragment referenced in 57a.
Defines: `C_standardise` 39a.
Uses: `S` 21a.

⟨ *P-Values* 64b ⟩ ≡

⟨ *C_chisq_pvalue* 64c ⟩
⟨ *C_perm_pvalue* 65 ⟩
⟨ *C_norm_pvalue* 66 ⟩
⟨ *C_maxtype_pvalue* 67 ⟩
◇

Fragment referenced in 23a.

⟨ *C_chisq_pvalue* 64c ⟩ ≡

```
/* lower = 1 means p-value, lower = 0 means 1 - p-value */
double C_chisq_pvalue
(
    const double stat,
    const int df,
    const int lower,
    const int give_log
) {
    return(pchisq(stat, (double) df, lower, give_log));
}
```

◇

Fragment referenced in 64b.
Defines: `C_chisq_pvalue` 51.

⟨ *C_perm_pvalue* 65 ⟩ ≡

```
double C_perm_pvalue
(
    const int greater,
    const double nresample,
    const int lower,
    const int give_log
) {
    double ret;

    if (give_log) {
         if (lower) {
             ret = log1p(- (double) greater / nresample);
         } else {
             ret = log(greater) - log(nresample);
         }
    } else {
        if (lower) {
            ret = 1.0 - (double) greater / nresample;
        } else {
            ret = (double) greater / nresample;
        }
    }
    return(ret);
}
◇
```

Fragment referenced in 64b.
Defines: `C_perm_pvalue` 51, 54, 73d.

⟨ *C_norm_pvalue* 66 ⟩ ≡

```
double C_norm_pvalue
(
    const double stat,
    const int alternative,
    const int lower,
    const int give_log
) {
    double ret;

    if (alternative == ALTERNATIVE_less) {
        return(pnorm(stat, 0.0, 1.0, 1 - lower, give_log));
    } else if (alternative == ALTERNATIVE_greater) {
        return(pnorm(stat, 0.0, 1.0, lower, give_log));
    } else if (alternative == ALTERNATIVE_twosided) {
        if (lower) {
            ret = pnorm(fabs(stat)*-1.0, 0.0, 1.0, 1, 0);
            if (give_log) {
                return(log1p(- 2 * ret));
            } else {
                return(1 - 2 * ret);
            }
        } else {
            ret = pnorm(fabs(stat)*-1.0, 0.0, 1.0, 1, give_log);
            if (give_log) {
                return(ret + log(2));
            } else {
                return(2 * ret);
            }
        }
    }
    return(NA_REAL);
}
◇
```

Fragment referenced in 64b.

⟨ *C_maxtype_pvalue* 67 ⟩ ≡

```
double C_maxtype_pvalue
(
    const double stat,
    const double *Covariance,
    const int n,
    const int alternative,
    const int lower,
    const int give_log,
    int maxpts, /* const? */
    double releps,
    double abseps,
    double tol
) {
    int nu = 0, inform, i, j, sub, nonzero, *infin, *index, rnd = 0;
    double ans, myerror, *lowerbnd, *upperbnd, *delta, *corr, *sd;

    /* univariate problem */
    if (n == 1)
        return(C_norm_pvalue(stat, alternative, lower, give_log));

    ⟨ Setup mvtnorm Memory 68 ⟩

    ⟨ Setup mvtnorm Correlation 69a ⟩

    /* call mvtnorm's mvtdst C function defined in mvtnorm/include/mvtnormAPI.h */
    mvtnorm_C_mvtdst(&nonzero, &nu, lowerbnd, upperbnd, infin, corr, delta,
                     &maxpts, &abseps, &releps, &myerror, &ans, &inform, &rnd);

    /* inform == 0 means: everything is OK */
    switch (inform) {
        case 0: break;
        case 1: warning("cmvnorm: completion with ERROR > EPS"); break;
        case 2: warning("cmvnorm: N > 1000 or N < 1");
                ans = 0.0;
                break;
        case 3: warning("cmvnorm: correlation matrix not positive semi-definite");
                ans = 0.0;
                break;
        default: warning("cmvnorm: unknown problem in MVTDST");
                ans = 0.0;
    }
    R_Free(corr); R_Free(sd); R_Free(lowerbnd); R_Free(upperbnd);
    R_Free(infin); R_Free(delta); R_Free(index);

    /* ans = 1 - p-value */
    if (lower) {
        if (give_log)
            return(log(ans));   /* log(1 - p-value) */
        return(ans);            /* 1 - p-value */
    } else {
        if (give_log)
            return(log1p(ans)); /* log(p-value) */
        return(1 - ans);        /* p-value */
    }
}
◇
```

Fragment referenced in 64b.
Defines: `C_maxtype_pvalue` 54.
Uses: `N` 23bc.

⟨ *Setup mvtnorm Memory* 68 ⟩ ≡

```
if (n == 2)
    corr = R_Calloc(1, double);
else
    corr = R_Calloc(n + ((n - 2) * (n - 1))/2, double);

sd = R_Calloc(n, double);
lowerbnd = R_Calloc(n, double);
upperbnd = R_Calloc(n, double);
infin = R_Calloc(n, int);
delta = R_Calloc(n, double);
index = R_Calloc(n, int);

/* determine elements with non-zero variance */

nonzero = 0;
for (i = 0; i < n; i++) {
    if (Covariance[S(i, i, n)] > tol) {
        index[nonzero] = i;
        nonzero++;
    }
}
◇
```

Fragment referenced in 67.
Uses: `S` 21a.

`mvtdst` assumes the unique elements of the triangular covariance matrix to be passed as argument `CORREL`

⟨ *Setup mvtnorm Correlation* 69a ⟩ ≡

```
    for (int nz = 0; nz < nonzero; nz++) {
        /* handle elements with non-zero variance only */
        i = index[nz];

        /* standard deviations */
        sd[i] = sqrt(Covariance[S(i, i, n)]);

        if (alternative == ALTERNATIVE_less) {
            lowerbnd[nz] = stat;
            upperbnd[nz] = R_PosInf;
            infin[nz] = 1;
        } else if (alternative == ALTERNATIVE_greater) {
            lowerbnd[nz] = R_NegInf;
            upperbnd[nz] = stat;
            infin[nz] = 0;
        } else if (alternative == ALTERNATIVE_twosided) {
            lowerbnd[nz] = fabs(stat) * -1.0;
            upperbnd[nz] = fabs(stat);
            infin[nz] = 2;
        }

        delta[nz] = 0.0;

        /* set up vector of correlations, i.e., the upper
           triangular part of the covariance matrix) */
        for (int jz = 0; jz < nz; jz++) {
            j = index[jz];
            sub = (int) (jz + 1) + (double) ((nz - 1) * nz) / 2 - 1;
            if (sd[i] == 0.0 || sd[j] == 0.0)
                corr[sub] = 0.0;
            else
                corr[sub] = Covariance[S(i, j, n)] / (sd[i] * sd[j]);
        }
    }
    ◇
```

Fragment referenced in 67.
Uses: `S` 21a.

⟨ *maxstat Xfactor Variables* 69b ⟩ ≡

```
    SEXP LECV,
    const int minbucket,
    const int teststat,
    int *wmax,
    double *maxstat,
    double *bmaxstat,
    double *pval,
    const int lower,
    const int give_log
    ◇
```

Fragment referenced in 70, 74.
Uses: `LECV` 141b.

⟨ *C_ordered_Xfactor* 70 ⟩ ≡

```
void C_ordered_Xfactor
(
    ⟨ maxstat Xfactor Variables 69b ⟩
) {
    ⟨ Setup maxstat Variables 71 ⟩

    ⟨ Setup maxstat Memory 72 ⟩

    wmax[0] = NA_INTEGER;

    for (int p = 0; p < P; p++) {
        sumleft += ExpX[p];
        sumright -= ExpX[p];

        for (int q = 0; q < Q; q++) {
            mlinstat[q] += linstat[q * P + p];
            for (R_xlen_t np = 0; np < nresample; np++)
                mblinstat[q + np * Q] += blinstat[q * P + p + np * PQ];
            mexpect[q] += expect[q * P + p];
            if (B == 1) {
                ⟨ Compute maxstat Variance / Covariance Directly 73b ⟩
            } else {
                ⟨ Compute maxstat Variance / Covariance from Total Covariance 73a ⟩
            }
        }

        if ((sumleft >= minbucket) && (sumright >= minbucket) && (ExpX[p] > 0)) {
            ls = mlinstat;
            /* compute MPinv only once */
            if (teststat != TESTSTAT_maximum)
                C_MPinv_sym(mcovar, Q, tol, mMPinv, &rank);
            ⟨ Compute maxstat Test Statistic 73c ⟩
            if (tmp > maxstat[0]) {
                wmax[0] = p;
                maxstat[0] = tmp;
            }

            for (R_xlen_t np = 0; np < nresample; np++) {
                ls = mblinstat + np * Q;
                ⟨ Compute maxstat Test Statistic 73c ⟩
                if (tmp > bmaxstat[np])
                    bmaxstat[np] = tmp;
            }
        }
    }
    ⟨ Compute maxstat Permutation P-Value 73d ⟩
    R_Free(mlinstat); R_Free(mexpect); R_Free(mblinstat);
    R_Free(mvar); R_Free(mcovar); R_Free(mMPinv);
    if (nresample == 0) R_Free(blinstat);
}
◇
```

Fragment referenced in 57a.
Defines: `C_ordered_Xfactor` 35b, 44, 56.
Uses: `B` 27a, `P` 24a, `Q` 24e.

⟨ *Setup maxstat Variables* 71 ⟩ ≡

```
double *linstat, *expect, *covar, *varinf, *covinf, *ExpX, *blinstat, tol, *ls;
int P, Q, B;
R_xlen_t nresample;

double *mlinstat, *mblinstat, *mexpect, *mvar, *mcovar, *mMPinv,
       tmp, sumleft, sumright, sumweights;
int rank, PQ, greater;

Q = C_get_Q(LECV);
P = C_get_P(LECV);
PQ = mPQB(P, Q, 1);
B = C_get_B(LECV);
if (B > 1) {
    if (C_get_varonly(LECV))
        error("need covariance for maximally statistics with blocks");
    covar = C_get_Covariance(LECV);
} else {
    covar = C_get_Variance(LECV); /* make -Wall happy */
}
linstat = C_get_LinearStatistic(LECV);
expect = C_get_Expectation(LECV);
ExpX = C_get_ExpectationX(LECV);
/* both need to be there */
varinf = C_get_VarianceInfluence(LECV);
covinf = C_get_CovarianceInfluence(LECV);
nresample = C_get_nresample(LECV);
if (nresample > 0)
    blinstat = C_get_PermutedLinearStatistic(LECV);
tol = C_get_tol(LECV);
◇
```

Fragment referenced in 70, 74.

Uses: `B` 27a, `C_get_B` 146a, `C_get_Covariance` 143c, `C_get_CovarianceInfluence` 144c, `C_get_Expectation` 143a, `C_get_ExpectationX` 144a, `C_get_LinearStatistic` 142d, `C_get_nresample` 146b, `C_get_P` 141c, `C_get_PermutedLinearStatistic` 146c, `C_get_Q` 142a, `C_get_tol` 146d, `C_get_Variance` 143b, `C_get_VarianceInfluence` 144d, `C_get_varonly` 142b, `LECV` 141b, `mPQB` 132a, `P` 24a, `Q` 24e, `sumweights` 25e.

⟨ *Setup maxstat Memory* 72 ⟩ ≡

```
mlinstat = R_Calloc(Q, double);
mexpect = R_Calloc(Q, double);
if (teststat == TESTSTAT_maximum) {
    mvar = R_Calloc(Q, double);
    /* not needed, but allocate anyway to make -Wmaybe-uninitialized happy */
    mcovar = R_Calloc(1, double);
    mMPinv = R_Calloc(1, double);
} else {
    mcovar = R_Calloc(Q * (Q + 1) / 2, double);
    mMPinv = R_Calloc(Q * (Q + 1) / 2, double);
    /* not needed, but allocate anyway to make -Wmaybe-uninitialized happy */
    mvar = R_Calloc(1, double);
}
if (nresample > 0) {
    mblinstat = R_Calloc(Q * nresample, double);
} else { /* not needed, but allocate anyway to make -Wmaybe-uninitialized happy */
    mblinstat = R_Calloc(1, double);
    blinstat = R_Calloc(1, double);
}

maxstat[0] = 0.0;

for (int q = 0; q < Q; q++) {
    mlinstat[q] = 0.0;
    mexpect[q] = 0.0;
    if (teststat == TESTSTAT_maximum)
        mvar[q] = 0.0;
    for (R_xlen_t np = 0; np < nresample; np++) {
        mblinstat[q + np * Q] = 0.0;
        bmaxstat[np] = 0.0;
    }
}
if (teststat == TESTSTAT_quadratic) {
    for (int q = 0; q < Q * (Q + 1) / 2; q++)
        mcovar[q] = 0.0;
}

sumleft = 0.0;
sumright = 0.0;
for (int p = 0; p < P; p++)
    sumright += ExpX[p];
sumweights = sumright;
◇
```

Fragment referenced in 70, 74.
Uses: `P` 24a, `Q` 24e, `sumweights` 25e.

⟨ *Compute maxstat Variance / Covariance from Total Covariance* 73a ⟩ ≡

```
if (teststat == TESTSTAT_maximum) {
    for (int pp = 0; pp < p; pp++)
        mvar[q] += 2 * covar[S(pp + q * P, p + P * q, mPQB(P, Q, 1))];
    mvar[q] += covar[S(p + q * P, p + P * q, mPQB(P, Q, 1))];
} else {
    for (int qq = 0; qq <= q; qq++) {
        for (int pp = 0; pp < p; pp++)
            mcovar[S(q, qq, Q)] += 2 * covar[S(pp + q * P, p + P * qq, mPQB(P, Q, 1))];
        mcovar[S(q, qq, Q)] += covar[S(p + q * P, p + P * qq, mPQB(P, Q, 1))];
    }
}
◇
```

Fragment referenced in 70.
Uses: `mPQB` 132a, `P` 24a, `Q` 24e, `S` 21a.

⟨ *Compute maxstat Variance / Covariance Directly* 73b ⟩ ≡

```
/* does not work with blocks! */
if (teststat == TESTSTAT_maximum) {
    C_VarianceLinearStatistic(1, Q, varinf, &sumleft, &sumleft,
                              sumweights, 0, mvar);
} else {
    C_CovarianceLinearStatistic(1, Q, covinf, &sumleft, &sumleft,
                                sumweights, 0, mcovar);
}
◇
```

Fragment referenced in 70.
Uses: `C_CovarianceLinearStatistic` 79, `C_VarianceLinearStatistic` 80a, `Q` 24e, `sumweights` 25e.

⟨ *Compute maxstat Test Statistic* 73c ⟩ ≡

```
if (teststat == TESTSTAT_maximum) {
    tmp = C_maxtype(Q, ls, mexpect, mvar, 1, tol,
                    ALTERNATIVE_twosided);
} else {
    tmp = C_quadform(Q, ls, mexpect, mMPinv);
}
◇
```

Fragment referenced in 70, 74.
Uses: `C_maxtype` 63, `C_quadform` 62, `Q` 24e.

⟨ *Compute maxstat Permutation P-Value* 73d ⟩ ≡

```
if (nresample > 0) {
    greater = 0;
    for (R_xlen_t np = 0; np < nresample; np++) {
        if (bmaxstat[np] > maxstat[0]) greater++;
    }
    pval[0] = C_perm_pvalue(greater, nresample, lower, give_log);
}
◇
```

Fragment referenced in 70, 74.
Uses: `C_perm_pvalue` 65.

⟨ *C_unordered_Xfactor* 74 ⟩ ≡

```
void C_unordered_Xfactor
(
    ⟨ maxstat Xfactor Variables 69b ⟩
) {
    double *mtmp;
    int qPp, nc, *levels, Pnonzero, *indl, *contrast;

    ⟨ Setup maxstat Variables 71 ⟩

    ⟨ Setup maxstat Memory 72 ⟩
    mtmp = R_Calloc(P, double);

    for (int p = 0; p < P; p++) wmax[p] = NA_INTEGER;

    ⟨ Count Levels 75a ⟩

    for (int j = 1; j < mi; j++) { /* go though all splits */

        ⟨ Setup unordered maxstat Contrasts 75b ⟩

        ⟨ Compute unordered maxstat Linear Statistic and Expectation 76a ⟩

        if (B == 1) {
            ⟨ Compute unordered maxstat Variance / Covariance Directly 77a ⟩
        } else {
            ⟨ Compute unordered maxstat Variance / Covariance from Total Covariance 76b ⟩
        }

        if ((sumleft >= minbucket) && (sumright >= minbucket)) {
            ls = mlinstat;
            /* compute MPinv only once */
            if (teststat != TESTSTAT_maximum)
                C_MPinv_sym(mcovar, Q, tol, mMPinv, &rank);
            ⟨ Compute maxstat Test Statistic 73c ⟩
            if (tmp > maxstat[0]) {
                for (int p = 0; p < Pnonzero; p++)
                    wmax[levels[p]] = contrast[levels[p]];
                maxstat[0] = tmp;
            }

            for (R_xlen_t np = 0; np < nresample; np++) {
                ls = mblinstat + np * Q;
                ⟨ Compute maxstat Test Statistic 73c ⟩
                if (tmp > bmaxstat[np])
                    bmaxstat[np] = tmp;
            }
        }
    }

    ⟨ Compute maxstat Permutation P-Value 73d ⟩

    R_Free(mlinstat); R_Free(mexpect); R_Free(levels); R_Free(contrast); R_Free(indl); R_Free(mtmp);
    R_Free(mblinstat); R_Free(mvar); R_Free(mcovar); R_Free(mMPinv);
    if (nresample == 0) R_Free(blinstat);
}
◇
```

Fragment referenced in 57a.
Defines: `C_unordered_Xfactor` 35b, 56.
Uses: `B` 27a, `P` 24a, `Q` 24e.

⟨ *Count Levels* 75a ⟩ ≡

```
contrast = R_Calloc(P, int);
Pnonzero = 0;
for (int p = 0; p < P; p++) {
    if (ExpX[p] > 0) Pnonzero++;
}
levels = R_Calloc(Pnonzero, int);
nc = 0;
for (int p = 0; p < P; p++) {
    if (ExpX[p] > 0) {
        levels[nc] = p;
        nc++;
    }
}

if (Pnonzero >= 31)
    error("cannot search for unordered splits in >= 31 levels");

int mi = 1;
for (int l = 1; l < Pnonzero; l++) mi *= 2;
indl = R_Calloc(Pnonzero, int);
for (int p = 0; p < Pnonzero; p++) indl[p] = 0;
◇
```

Fragment referenced in 74.
Uses: `P` 24a.

⟨ *Setup unordered maxstat Contrasts* 75b ⟩ ≡

```
/* indl determines if level p is left or right */
int jj = j;
for (int l = 1; l < Pnonzero; l++) {
    indl[l] = (jj%2);
    jj /= 2;
}

sumleft = 0.0;
sumright = 0.0;
for (int p = 0; p < P; p++) contrast[p] = 0;
for (int p = 0; p < Pnonzero; p++) {
    sumleft += indl[p] * ExpX[levels[p]];
    sumright += (1 - indl[p]) * ExpX[levels[p]];
    contrast[levels[p]] = indl[p];
}
◇
```

Fragment referenced in 74.
Uses: `P` 24a.

⟨ *Compute unordered maxstat Linear Statistic and Expectation* 76a ⟩ ≡

```
for (int q = 0; q < Q; q++) {
    mlinstat[q] = 0.0;
    mexpect[q] = 0.0;
    for (R_xlen_t np = 0; np < nresample; np++)
        mblinstat[q + np * Q] = 0.0;
    for (int p = 0; p < P; p++) {
        qPp = q * P + p;
        mlinstat[q] += contrast[p] * linstat[qPp];
        mexpect[q] += contrast[p] * expect[qPp];
        for (R_xlen_t np = 0; np < nresample; np++)
            mblinstat[q + np * Q] += contrast[p] * blinstat[q * P + p + np * PQ];
    }
}
◇
```

Fragment referenced in 74.
Uses: `P` 24a, `Q` 24e.

⟨ *Compute unordered maxstat Variance / Covariance from Total Covariance* 76b ⟩ ≡

```
if (teststat == TESTSTAT_maximum) {
    for (int q = 0; q < Q; q++) {
        mvar[q] = 0.0;
        for (int p = 0; p < P; p++) {
            qPp = q * P + p;
            mtmp[p] = 0.0;
            for (int pp = 0; pp < P; pp++)
                mtmp[p] += contrast[pp] * covar[S(pp + q * P, qPp, PQ)];
        }
        for (int p = 0; p < P; p++)
            mvar[q] += contrast[p] * mtmp[p];
    }
} else {
    for (int q = 0; q < Q; q++) {
        for (int qq = 0; qq <= q; qq++)
            mcovar[S(q, qq, Q)] = 0.0;
        for (int qq = 0; qq <= q; qq++) {
            for (int p = 0; p < P; p++) {
                mtmp[p] = 0.0;
                for (int pp = 0; pp < P; pp++)
                    mtmp[p] += contrast[pp] * covar[S(pp + q * P, p + P * qq,
                                                  mPQB(P, Q, 1))];
            }
            for (int p = 0; p < P; p++)
                mcovar[S(q, qq, Q)] += contrast[p] * mtmp[p];
        }
    }
}
◇
```

Fragment referenced in 74.
Uses: `mPQB` 132a, `P` 24a, `Q` 24e, `S` 21a.

⟨ *Compute unordered maxstat Variance / Covariance Directly* 77a ⟩ ≡

```
if (teststat == TESTSTAT_maximum) {
    C_VarianceLinearStatistic(1, Q, varinf, &sumleft, &sumleft,
                              sumweights, 0, mvar);
} else {
    C_CovarianceLinearStatistic(1, Q, covinf, &sumleft, &sumleft,
                                sumweights, 0, mcovar);
}
◇
```

Fragment referenced in 74.
Uses: `C_CovarianceLinearStatistic` 79, `C_VarianceLinearStatistic` 80a, `Q` 24e, `sumweights` 25e.

3.7 Linear Statistics

⟨ *LinearStatistics* 77b ⟩ ≡

```
⟨ RC_LinearStatistic 77d ⟩
◇
```

Fragment referenced in 23a.

⟨ *RC_LinearStatistic Prototype* 77c ⟩ ≡

```
void RC_LinearStatistic
(
    ⟨ R x Input 23d ⟩
    ⟨ C integer N Input 23c ⟩,
    ⟨ C integer P Input 24a ⟩,
    ⟨ C real y Input 24f ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩,
    ⟨ C KronSums Answer 96a ⟩
)
◇
```

Fragment referenced in 77d.
Uses: `RC_LinearStatistic` 77d.

⟨ *RC_LinearStatistic* 77d ⟩ ≡

```
⟨ RC_LinearStatistic Prototype 77c ⟩
{
    double center;

    RC_KronSums(x, N, P, y, Q, !DoSymmetric, &center, &center, !DoCenter, weights,
                subset, offset, Nsubset, PQ_ans);
}
◇
```

Fragment referenced in 77b.
Defines: `RC_LinearStatistic` 33b, 77c.
Uses: `DoCenter` 21b, `DoSymmetric` 21b, `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `Q` 24e, `RC_KronSums` 95b, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc, `y` 24df, 25a.

3.8 Expectation and Covariance

⟨ *ExpectationCovariances* 78a ⟩ ≡

⟨ *RC_ExpectationInfluence* 81c ⟩
⟨ *R_ExpectationInfluence* 81a ⟩
⟨ *RC_CovarianceInfluence* 84a ⟩
⟨ *R_CovarianceInfluence* 83a ⟩
⟨ *RC_ExpectationX* 85b ⟩
⟨ *R_ExpectationX* 84c ⟩
⟨ *RC_CovarianceX* 88a ⟩
⟨ *R_CovarianceX* 87a ⟩
⟨ *C_ExpectationLinearStatistic* 78b ⟩
⟨ *C_CovarianceLinearStatistic* 79 ⟩
⟨ *C_VarianceLinearStatistic* 80a ⟩
◇

Fragment referenced in 23a.

3.8.1 Linear Statistic

⟨ *C_ExpectationLinearStatistic* 78b ⟩ ≡

```
void C_ExpectationLinearStatistic
(
    ⟨ C integer P Input 24a ⟩,
    ⟨ C integer Q Input 24e ⟩,
    double *ExpInf,
    double *ExpX,
    const int add,
    double *PQ_ans
) {
    if (!add)
        for (int p = 0; p < mPQB(P, Q, 1); p++) PQ_ans[p] = 0.0;

    for (int p = 0; p < P; p++) {
        for (int q = 0; q < Q; q++)
            PQ_ans[q * P + p] += ExpX[p] * ExpInf[q];
    }
}
```
◇

Fragment referenced in 78a.
Defines: `C_ExpectationLinearStatistic` 35a, 43b.
Uses: `mPQB` 132a, `P` 24a, `Q` 24e.

⟨ *C_CovarianceLinearStatistic* 79 ⟩ ≡

```
void C_CovarianceLinearStatistic
(
    ⟨ C integer P Input 24a ⟩,
    ⟨ C integer Q Input 24e ⟩,
    double *CovInf,
    double *ExpX,
    double *CovX,
    ⟨ C sumweights Input 25e ⟩,
    const int add,
    double *PQPQ_sym_ans
) {
    double f1 = sumweights / (sumweights - 1);
    double f2 = 1.0 / (sumweights - 1);
    double tmp, *PP_sym_tmp;

    if (mPQB(P, Q, 1) == 1) {
        tmp = f1 * CovInf[0] * CovX[0];
        tmp -= f2 * CovInf[0] * ExpX[0] * ExpX[0];
        if (add) {
            PQPQ_sym_ans[0] += tmp;
        } else {
            PQPQ_sym_ans[0] = tmp;
        }
    } else {
        PP_sym_tmp = R_Calloc(PP12(P), double);
        C_KronSums_sym_(ExpX, 1, P,
                        PP_sym_tmp);
        for (int p = 0; p < PP12(P); p++)
            PP_sym_tmp[p] = f1 * CovX[p] - f2 * PP_sym_tmp[p];
        C_kronecker_sym(CovInf, Q, PP_sym_tmp, P, 1 - (add >= 1),
                        PQPQ_sym_ans);
        R_Free(PP_sym_tmp);
    }
}
◇
```

Fragment referenced in 78a.
Defines: `C_CovarianceLinearStatistic` 35d, 44, 73b, 77a, 80a.
Uses: `C_kronecker_sym` 134, `mPQB` 132a, `P` 24a, `PP12` 131b, `Q` 24e, `sumweights` 25e.

⟨ *C_VarianceLinearStatistic* 80a ⟩ ≡

```
void C_VarianceLinearStatistic
(
    ⟨ C integer P Input 24a ⟩,
    ⟨ C integer Q Input 24e ⟩,
    double *VarInf,
    double *ExpX,
    double *VarX,
    ⟨ C sumweights Input 25e ⟩,
    const int add,
    double *PQ_ans
) {
    if (mPQB(P, Q, 1) == 1) {
        C_CovarianceLinearStatistic(P, Q, VarInf, ExpX, VarX,
                                    sumweights, (add >= 1),
                                    PQ_ans);
    } else {
        double *P_tmp;
        P_tmp = R_Calloc(P, double);
        double f1 = sumweights / (sumweights - 1);
        double f2 = 1.0 / (sumweights - 1);
        for (int p = 0; p < P; p++)
            P_tmp[p] = f1 * VarX[p] - f2 * ExpX[p] * ExpX[p];
        C_kronecker(VarInf, 1, Q, P_tmp, 1, P, 1 - (add >= 1),
                    PQ_ans);
        R_Free(P_tmp);
    }
}
◇
```

Fragment referenced in 78a.
Defines: `C_VarianceLinearStatistic` 35c, 44, 73b, 77a.
Uses: `C_CovarianceLinearStatistic` 79, `C_kronecker` 133b, `mPQB` 132a, `P` 24a, `Q` 24e, `sumweights` 25e.

3.8.2 Influence

```
> sumweights <- sum(weights[subset])
> expecty <- colSums(y[subset, ] * weights[subset]) / sumweights
> a0 <- expecty
> a1 <- .Call(libcoin:::R_ExpectationInfluence, y, weights, subset)
> a2 <- .Call(libcoin:::R_ExpectationInfluence, y, as.double(weights), as.double(subset))
> a3 <- .Call(libcoin:::R_ExpectationInfluence, y, weights, as.double(subset))
> a4 <- .Call(libcoin:::R_ExpectationInfluence, y, as.double(weights), subset)
> a5 <- LinStatExpCov(x, y, weights = weights, subset = subset)$ExpectationInfluence
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4) &&
+           isequal(a0, a5))
```

⟨ *R_ExpectationInfluence Prototype* 80b ⟩ ≡

```
SEXP R_ExpectationInfluence
(
    ⟨ R y Input 24d ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩
)
◇
```

Fragment referenced in 22b, 81a.
Uses: `R_ExpectationInfluence` 81a.

⟨ *R_ExpectationInfluence* 81a ⟩ ≡

```
⟨ R_ExpectationInfluence Prototype 80b ⟩
{
    SEXP ans;
    ⟨ C integer Q Input 24e ⟩;
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer Nsubset Input 26b ⟩;
    double sumweights;

    Q = NCOL(y);
    N = XLENGTH(y) / Q;
    Nsubset = XLENGTH(subset);

    sumweights = RC_Sums(N, weights, subset, Offset0, Nsubset);

    PROTECT(ans = allocVector(REALSXP, Q));
    RC_ExpectationInfluence(N, y, Q, weights, subset, Offset0, Nsubset, sumweights, REAL(ans));
    UNPROTECT(1);
    return(ans);
}
◇
```

Fragment referenced in 78a.
Defines: `R_ExpectationInfluence` 80b, 83a, 152c, 153.
Uses: `N` 23bc, `NCOL` 130c, `Nsubset` 26b, `Offset0` 21b, `Q` 24e, `RC_ExpectationInfluence` 81c, `RC_Sums` 91a, `subset` 26ade, `sumweights` 25e, `weights` 25b, `weights,` 25cd, `y` 24df, 25a.

⟨ *RC_ExpectationInfluence Prototype* 81b ⟩ ≡

```
void RC_ExpectationInfluence
(
    ⟨ C integer N Input 23c ⟩,
    ⟨ R y Input 24d ⟩
    ⟨ C integer Q Input 24e ⟩,
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩,
    ⟨ C sumweights Input 25e ⟩,
    ⟨ C colSums Answer 108a ⟩
)
◇
```

Fragment referenced in 81c.
Uses: `RC_ExpectationInfluence` 81c.

⟨ *RC_ExpectationInfluence* 81c ⟩ ≡

```
⟨ RC_ExpectationInfluence Prototype 81b ⟩
{
    double center;

    RC_colSums(REAL(y), N, Q, Power1, &center, !DoCenter, weights,
               subset, offset, Nsubset, P_ans);
    for (int q = 0; q < Q; q++)
        P_ans[q] = P_ans[q] / sumweights;
}
◇
```

Fragment referenced in 78a.
Defines: `RC_ExpectationInfluence` 35a, 43b, 81ab.
Uses: `DoCenter` 21b, `N` 23bc, `Nsubset` 26b, `offset` 26c, `Power1` 21b, `Q` 24e, `RC_colSums` 107b, `subset` 26ade, `sumweights` 25e, `weights` 25b, `weights,` 25cd, `y` 24df, 25a.

```
> sumweights <- sum(weights[subset])
> yc <- t(t(y) - expecty)
> r1y <- rep(1:ncol(y), ncol(y))
> r2y <- rep(1:ncol(y), each = ncol(y))
> a0 <- colSums(yc[subset, r1y] * yc[subset, r2y] * weights[subset]) / sumweights
> a0 <- matrix(a0, ncol = ncol(y))
> vary <- diag(a0)
> a0 <- a0[lower.tri(a0, diag = TRUE)]
> a1 <- .Call(libcoin:::R_CovarianceInfluence, y, weights, subset, 0L)
> a2 <- .Call(libcoin:::R_CovarianceInfluence, y, as.double(weights), as.double(subset), 0L)
> a3 <- .Call(libcoin:::R_CovarianceInfluence, y, weights, as.double(subset), 0L)
> a4 <- .Call(libcoin:::R_CovarianceInfluence, y, as.double(weights), subset, 0L)
> a5 <- LinStatExpCov(x, y, weights = weights, subset = subset)$CovarianceInfluence
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4) &&
+           isequal(a0, a5))
> a0 <- vary
> a1 <- .Call(libcoin:::R_CovarianceInfluence, y, weights, subset, 1L)
> a2 <- .Call(libcoin:::R_CovarianceInfluence, y, as.double(weights), as.double(subset), 1L)
> a3 <- .Call(libcoin:::R_CovarianceInfluence, y, weights, as.double(subset), 1L)
> a4 <- .Call(libcoin:::R_CovarianceInfluence, y, as.double(weights), subset, 1L)
> a5 <- LinStatExpCov(x, y, weights = weights, subset = subset, varonly = TRUE)$VarianceInfluence
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4) &&
+           isequal(a0, a5))
```

⟨ *R_CovarianceInfluence Prototype* 82 ⟩ ≡

```
SEXP R_CovarianceInfluence
(
    ⟨ R y Input 24d ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    SEXP varonly
)
◇
```

Fragment referenced in 22b, 83a.
Uses: `R_CovarianceInfluence` 83a.

⟨ *R_CovarianceInfluence* 83a ⟩ ≡

```
⟨ R_CovarianceInfluence Prototype 82 ⟩
{
    SEXP ans;
    SEXP ExpInf;
    ⟨ C integer Q Input 24e ⟩;
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer Nsubset Input 26b ⟩;
    double sumweights;

    Q = NCOL(y);
    N = XLENGTH(y) / Q;
    Nsubset = XLENGTH(subset);

    PROTECT(ExpInf = R_ExpectationInfluence(y, weights, subset));

    sumweights = RC_Sums(N, weights, subset, Offset0, Nsubset);

    if (INTEGER(varonly)[0]) {
        PROTECT(ans = allocVector(REALSXP, Q));
    } else {
        PROTECT(ans = allocVector(REALSXP, Q * (Q + 1) / 2));
    }
    RC_CovarianceInfluence(N, y, Q, weights, subset, Offset0, Nsubset, REAL(ExpInf), sumweights,
                           INTEGER(varonly)[0], REAL(ans));
    UNPROTECT(2);
    return(ans);
}
◇
```

Fragment referenced in 78a.
Defines: `R_CovarianceInfluence` 82, 152c, 153.
Uses: `N` 23bc, `NCOL` 130c, `Nsubset` 26b, `Offset0` 21b, `Q` 24e, `RC_CovarianceInfluence` 84a, `RC_Sums` 91a, `R_ExpectationInfluence` 81a, `subset` 26ade, `sumweights` 25e, `weights` 25b, `weights,` 25cd, `y` 24df, 25a.

⟨ *RC_CovarianceInfluence Prototype* 83b ⟩ ≡

```
void RC_CovarianceInfluence
(
    ⟨ C integer N Input 23c ⟩,
    ⟨ R y Input 24d ⟩
    ⟨ C integer Q Input 24e ⟩,
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩,
    double *ExpInf,
    ⟨ C sumweights Input 25e ⟩,
    int VARONLY,
    ⟨ C KronSums Answer 96a ⟩
)
◇
```

Fragment referenced in 84a.
Uses: `RC_CovarianceInfluence` 84a.

⟨ *RC_CovarianceInfluence* 84a ⟩ ≡

```
⟨ RC_CovarianceInfluence Prototype 83b ⟩
{
    if (VARONLY) {
        RC_colSums(REAL(y), N, Q, Power2, ExpInf, DoCenter, weights,
                   subset, offset, Nsubset, PQ_ans);
        for (int q = 0; q < Q; q++)
            PQ_ans[q] = PQ_ans[q] / sumweights;
    } else {
        RC_KronSums(y, N, Q, REAL(y), Q, DoSymmetric, ExpInf, ExpInf, DoCenter, weights,
                    subset, offset, Nsubset, PQ_ans);
        for (int q = 0; q < Q * (Q + 1) / 2; q++)
            PQ_ans[q] = PQ_ans[q] / sumweights;
    }
}
◇
```

Fragment referenced in 78a.
Defines: `RC_CovarianceInfluence` 35b, 44, 83ab.
Uses: `DoCenter` 21b, `DoSymmetric` 21b, `N` 23bc, `Nsubset` 26b, `offset` 26c, `Power2` 21b, `Q` 24e, `RC_colSums` 107b, `RC_KronSums` 95b, `subset` 26ade, `sumweights` 25e, `weights` 25b, `weights,` 25cd, `y` 24df, 25a.

3.8.3 X

⟨ *R_ExpectationX Prototype* 84b ⟩ ≡

```
SEXP R_ExpectationX
(
    ⟨ R x Input 23d ⟩
    SEXP P,
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩
)
◇
```

Fragment referenced in 22b, 84c.
Uses: `P` 24a, `R_ExpectationX` 84c.

⟨ *R_ExpectationX* 84c ⟩ ≡

```
⟨ R_ExpectationX Prototype 84b ⟩
{
    SEXP ans;
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer Nsubset Input 26b ⟩;

    N = XLENGTH(x) / INTEGER(P)[0];
    Nsubset = XLENGTH(subset);

    PROTECT(ans = allocVector(REALSXP, INTEGER(P)[0]));
    RC_ExpectationX(x, N, INTEGER(P)[0], weights, subset,
                    Offset0, Nsubset, REAL(ans));
    UNPROTECT(1);
    return(ans);
}
◇
```

Fragment referenced in 78a.
Defines: `R_ExpectationX` 84b, 87a, 152c, 153.
Uses: `N` 23bc, `Nsubset` 26b, `Offset0` 21b, `P` 24a, `RC_ExpectationX` 85b, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc.

⟨ *RC_ExpectationX Prototype* 85a ⟩ ≡

```
void RC_ExpectationX
(
    ⟨ R x Input 23d ⟩
    ⟨ C integer N Input 23c ⟩,
    ⟨ C integer P Input 24a ⟩,
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩,
    ⟨ C OneTableSums Answer 112c ⟩
)
◇
```

Fragment referenced in 85b.
Uses: `RC_ExpectationX` 85b.

⟨ *RC_ExpectationX* 85b ⟩ ≡

```
⟨ RC_ExpectationX Prototype 85a ⟩
{
    double center;

    if (TYPEOF(x) == INTSXP) {
        double* Pp1tmp = R_Calloc(P + 1, double);
        RC_OneTableSums(INTEGER(x), N, P + 1, weights, subset, offset, Nsubset, Pp1tmp);
        for (int p = 0; p < P; p++) P_ans[p] = Pp1tmp[p + 1];
        R_Free(Pp1tmp);
    } else {
        RC_colSums(REAL(x), N, P, Power1, &center, !DoCenter, weights, subset, offset, Nsubset, P_ans);
    }
}
◇
```

Fragment referenced in 78a.
Defines: `RC_ExpectationX` 35a, 43b, 84c, 85a.
Uses: `DoCenter` 21b, `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `Power1` 21b, `RC_colSums` 107b, `RC_OneTableSums` 112a, `subset` 26ade, `weights` 25b, `weights`, 25cd, `x` 23d, 24bc.

```
> a0 <- colSums(x[subset, ] * weights[subset])
> a1 <- .Call(libcoin:::R_ExpectationX, x, P, weights, subset);
> a2 <- .Call(libcoin:::R_ExpectationX, x, P, as.double(weights), as.double(subset))
> a3 <- .Call(libcoin:::R_ExpectationX, x, P, weights, as.double(subset))
> a4 <- .Call(libcoin:::R_ExpectationX, x, P, as.double(weights), subset)
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4) &&
+           isequal(a0, LECVxyws$ExpectationX))
> a0 <- colSums(x[subset, ]^2 * weights[subset])
> a1 <- .Call(libcoin:::R_CovarianceX, x, P, weights, subset, 1L)
> a2 <- .Call(libcoin:::R_CovarianceX, x, P, as.double(weights), as.double(subset), 1L)
> a3 <- .Call(libcoin:::R_CovarianceX, x, P, weights, as.double(subset), 1L)
> a4 <- .Call(libcoin:::R_CovarianceX, x, P, as.double(weights), subset, 1L)
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4))
> a0 <- as.vector(colSums(Xfactor[subset, ] * weights[subset]))
> a1 <- .Call(libcoin:::R_ExpectationX, ix, Lx, weights, subset)
> a2 <- .Call(libcoin:::R_ExpectationX, ix, Lx, as.double(weights), as.double(subset))
> a3 <- .Call(libcoin:::R_ExpectationX, ix, Lx, weights, as.double(subset))
> a4 <- .Call(libcoin:::R_ExpectationX, ix, Lx, as.double(weights), subset)
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4))
```

```
> a1 <- .Call(libcoin:::R_CovarianceX, ix, Lx, weights, subset, 1L)
> a2 <- .Call(libcoin:::R_CovarianceX, ix, Lx, as.double(weights), as.double(subset), 1L)
> a3 <- .Call(libcoin:::R_CovarianceX, ix, Lx, weights, as.double(subset), 1L)
> a4 <- .Call(libcoin:::R_CovarianceX, ix, Lx, as.double(weights), subset, 1L)
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4))
> r1x <- rep(1:ncol(Xfactor), ncol(Xfactor))
> r2x <- rep(1:ncol(Xfactor), each = ncol(Xfactor))
> a0 <- colSums(Xfactor[subset, r1x] * Xfactor[subset, r2x] * weights[subset])
> a0 <- matrix(a0, ncol = ncol(Xfactor))
> vary <- diag(a0)
> a0 <- a0[lower.tri(a0, diag = TRUE)]
> a1 <- .Call(libcoin:::R_CovarianceX, ix, Lx, weights, subset, 0L)
> a2 <- .Call(libcoin:::R_CovarianceX, ix, Lx, as.double(weights), as.double(subset), 0L)
> a3 <- .Call(libcoin:::R_CovarianceX, ix, Lx, weights, as.double(subset), 0L)
> a4 <- .Call(libcoin:::R_CovarianceX, ix, Lx, as.double(weights), subset, 0L)
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4))
```

⟨ *R_CovarianceX Prototype* 86 ⟩ ≡

```
SEXP R_CovarianceX
(
    ⟨ R x Input 23d ⟩
    SEXP P,
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    SEXP varonly
)
◇
```

Fragment referenced in 22b, 87a.
Uses: `P` 24a, `R_CovarianceX` 87a.

⟨ *R_CovarianceX* 87a ⟩ ≡

```
⟨ R_CovarianceX Prototype 86 ⟩
{
    SEXP ans;
    SEXP ExpX;
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer Nsubset Input 26b ⟩;

    N = XLENGTH(x) / INTEGER(P)[0];
    Nsubset = XLENGTH(subset);

    PROTECT(ExpX = R_ExpectationX(x, P, weights, subset));

    if (INTEGER(varonly)[0]) {
        PROTECT(ans = allocVector(REALSXP, INTEGER(P)[0]));
    } else {
        PROTECT(ans = allocVector(REALSXP, INTEGER(P)[0] * (INTEGER(P)[0] + 1) / 2));
    }
    RC_CovarianceX(x, N, INTEGER(P)[0], weights, subset, Offset0, Nsubset, REAL(ExpX),
                   INTEGER(varonly)[0], REAL(ans));
    UNPROTECT(2);
    return(ans);
}
◇
```

Fragment referenced in 78a.
Defines: `R_CovarianceX` 86, 152c, 153.
Uses: `N` 23bc, `Nsubset` 26b, `Offset0` 21b, `P` 24a, `RC_CovarianceX` 88a, `R_ExpectationX` 84c, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc.

⟨ *RC_CovarianceX Prototype* 87b ⟩ ≡

```
void RC_CovarianceX
(
    ⟨ R x Input 23d ⟩
    ⟨ C integer N Input 23c ⟩,
    ⟨ C integer P Input 24a ⟩,
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩,
    double *ExpX,
    int VARONLY,
    ⟨ C KronSums Answer 96a ⟩
)
◇
```

Fragment referenced in 88a.
Uses: `RC_CovarianceX` 88a.

⟨ *RC_CovarianceX* 88a ⟩ ≡

```
⟨ RC_CovarianceX Prototype 87b ⟩
{
    double center;

    if (TYPEOF(x) == INTSXP) {
        if (VARONLY) {
            for (int p = 0; p < P; p++) PQ_ans[p] = ExpX[p];
        } else {
            for (int p = 0; p < PP12(P); p++)
                PQ_ans[p] = 0.0;
            for (int p = 0; p < P; p++)
                PQ_ans[S(p, p, P)] = ExpX[p];
        }
    } else {
        if (VARONLY) {
            RC_colSums(REAL(x), N, P, Power2, &center, !DoCenter, weights,
                       subset, offset, Nsubset, PQ_ans);
        } else {
            RC_KronSums(x, N, P, REAL(x), P, DoSymmetric, &center, &center, !DoCenter, weights,
                        subset, offset, Nsubset, PQ_ans);
        }
    }
}
◇
```

Fragment referenced in 78a.
Defines: `RC_CovarianceX` 35cd, 44, 87ab.
Uses: `DoCenter` 21b, `DoSymmetric` 21b, `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `Power2` 21b, `PP12` 131b, `RC_colSums` 107b, `RC_KronSums` 95b, `S` 21a, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc.

3.9 Computing Sums

The core concept of all functions in the section is the computation of various sums over observations, case weights, or blocks. We start with an initialisation of the loop over all observations

⟨ *init subset loop* 88b ⟩ ≡

```
R_xlen_t diff = 0;
s = subset + offset;
w = weights;
/* subset is R-style index in 1:N */
if (Nsubset > 0)
    diff = (R_xlen_t) s[0] - 1;
◇
```

Fragment referenced in 93a, 99, 101b, 109b, 114a, 118b, 122b.
Uses: `N` 23bc, `Nsubset` 26b, `offset` 26c, `subset` 26ade, `weights` 25b.

and loop over $i = 1, \dots, N$ when no subset was specified or over the subset of the subset given by `offset` and `Nsubset`, allowing for number of observations larger than `INT_MAX`

⟨ *start subset loop* 88c ⟩ ≡

```
for (R_xlen_t i = 0; i < (Nsubset == 0 ? N : Nsubset) - 1; i++)
◇
```

Fragment referenced in 93a, 99, 101b, 109b, 114a, 118b, 122b.
Uses: `N` 23bc, `Nsubset` 26b.

After computions in the loop, we compute the next element

⟨ *continue subset loop* 89a ⟩ ≡

```
if (Nsubset > 0) {
    /* NB: diff also works with R style index */
    diff = (R_xlen_t) s[1] - s[0];
    if (diff < 0)
        error("subset not sorted");
    s++;
} else {
    diff = 1;
}
◇
```

Fragment referenced in 93a, 99, 101b, 109b, 114a, 118b, 122b.
Uses: `Nsubset` 26b, `subset` 26ade.

3.9.1 Simple Sums

⟨ *SimpleSums* 89b ⟩ ≡

```
⟨ C_Sums_dweights_dsubset 91b ⟩
⟨ C_Sums_iweights_dsubset 92a ⟩
⟨ C_Sums_iweights_isubset 92b ⟩
⟨ C_Sums_dweights_isubset 92c ⟩
⟨ RC_Sums 91a ⟩
⟨ R_Sums 90a ⟩
◇
```

Fragment referenced in 23a.

```
> a0 <- sum(weights[subset])
> a1 <- .Call(libcoin:::R_Sums, N, weights, subset)
> a2 <- .Call(libcoin:::R_Sums, N, as.double(weights), as.double(subset))
> a3 <- .Call(libcoin:::R_Sums, N, weights, as.double(subset))
> a4 <- .Call(libcoin:::R_Sums, N, as.double(weights), subset)
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4))
```

⟨ *R_Sums Prototype* 89c ⟩ ≡

```
SEXP R_Sums
(
    ⟨ R N Input 23b ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩
)
◇
```

Fragment referenced in 22b, 90a.
Uses: `R_Sums` 90a.

⟨ *R_Sums* 90a ⟩ ≡

```
⟨ R_Sums Prototype 89c ⟩
{
    SEXP ans;
    ⟨ C integer Nsubset Input 26b ⟩;

    Nsubset = XLENGTH(subset);

    PROTECT(ans = allocVector(REALSXP, 1));
    REAL(ans)[0] = RC_Sums(INTEGER(N)[0], weights, subset, Offset0, Nsubset);
    UNPROTECT(1);

    return(ans);
}
◇
```

Fragment referenced in 89b.
Defines: `R_Sums` 89c, 152c, 153.
Uses: `N` 23bc, `Nsubset` 26b, `Offset0` 21b, `RC_Sums` 91a, `subset` 26ade, `weights` 25b, `weights,` 25cd.

⟨ *RC_Sums Prototype* 90b ⟩ ≡

```
double RC_Sums
(
    ⟨ C integer N Input 23c ⟩,
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩
)
◇
```

Fragment referenced in 91a.
Uses: `RC_Sums` 91a.

⟨ *RC_Sums* 91a ⟩ ≡

```
⟨ RC_Sums Prototype 90b ⟩
{
    if (XLENGTH(weights) == 0) {
        if (XLENGTH(subset) == 0) {
            return((double) N);
        } else {
            return((double) Nsubset);
        }
    }
    if (TYPEOF(weights) == INTSXP) {
        if (TYPEOF(subset) == INTSXP) {
            return(C_Sums_iweights_isubset(N, INTEGER(weights), XLENGTH(weights),
                                           INTEGER(subset), offset, Nsubset));
        } else {
            return(C_Sums_iweights_dsubset(N, INTEGER(weights), XLENGTH(weights),
                                           REAL(subset), offset, Nsubset));
        }
    } else {
        if (TYPEOF(subset) == INTSXP) {
            return(C_Sums_dweights_isubset(N, REAL(weights), XLENGTH(weights),
                                           INTEGER(subset), offset, Nsubset));
        } else {
            return(C_Sums_dweights_dsubset(N, REAL(weights), XLENGTH(weights),
                                           REAL(subset), offset, Nsubset));
        }
    }
}
◇
```

Fragment referenced in 89b.
Defines: `RC_Sums` 34ab, 81a, 83a, 90ab, 123c, 127a.
Uses: `C_Sums_dweights_dsubset` 91b, `C_Sums_dweights_isubset` 92c, `C_Sums_iweights_dsubset` 92a, `C_Sums_iweights_isubset` 92b, `N` 23bc, `Nsubset` 26b, `offset` 26c, `subset` 26ade, `weights` 25b.

⟨ *C_Sums_dweights_dsubset* 91b ⟩ ≡

```
double C_Sums_dweights_dsubset
(
    ⟨ C integer N Input 23c ⟩,
    ⟨ C real weights Input 25d ⟩
    ⟨ C real subset Input 26e ⟩
) {
    double *s, *w;
    ⟨ Sums Body 93a ⟩
}
◇
```

Fragment referenced in 89b.
Defines: `C_Sums_dweights_dsubset` 91a.

⟨ *C_Sums_iweights_dsubset* 92a ⟩ ≡

```
double C_Sums_iweights_dsubset
(
    ⟨ C integer N Input 23c ⟩,
    ⟨ C integer weights Input 25c ⟩
    ⟨ C real subset Input 26e ⟩
) {
    double *s;
    int *w;
    ⟨ Sums Body 93a ⟩
}
◇
```

Fragment referenced in 89b.
Defines: `C_Sums_iweights_dsubset` 91a.

⟨ *C_Sums_iweights_isubset* 92b ⟩ ≡

```
double C_Sums_iweights_isubset
(
    ⟨ C integer N Input 23c ⟩,
    ⟨ C integer weights Input 25c ⟩
    ⟨ C integer subset Input 26d ⟩
) {
    int *s, *w;
    ⟨ Sums Body 93a ⟩
}
◇
```

Fragment referenced in 89b.
Defines: `C_Sums_iweights_isubset` 91a.

⟨ *C_Sums_dweights_isubset* 92c ⟩ ≡

```
double C_Sums_dweights_isubset
(
    ⟨ C integer N Input 23c ⟩,
    ⟨ C real weights Input 25d ⟩
    ⟨ C integer subset Input 26d ⟩
) {
    int *s;
    double *w;
    ⟨ Sums Body 93a ⟩
}
◇
```

Fragment referenced in 89b.
Defines: `C_Sums_dweights_isubset` 91a.

⟨ *Sums Body* 93a ⟩ ≡

```
double ans = 0.0;

if (Nsubset > 0) {
    if (!HAS_WEIGHTS) return((double) Nsubset);
} else {
    if (!HAS_WEIGHTS) return((double) N);
}

⟨ init subset loop 88b ⟩
⟨ start subset loop 88c ⟩
{
    w = w + diff;
    ans += w[0];
    ⟨ continue subset loop 89a ⟩
}
w = w + diff;
ans += w[0];

return(ans);
◇
```

Fragment referenced in 91b, 92abc.
Uses: `HAS_WEIGHTS` 25cd, `N` 23bc, `Nsubset` 26b.

3.9.2 Kronecker Sums

⟨ *KronSums* 93b ⟩ ≡

```
⟨ C_KronSums_dweights_dsubset 97b ⟩
⟨ C_KronSums_iweights_dsubset 97c ⟩
⟨ C_KronSums_iweights_isubset 98a ⟩
⟨ C_KronSums_dweights_isubset 98b ⟩
⟨ C_XfactorKronSums_dweights_dsubset 100b ⟩
⟨ C_XfactorKronSums_iweights_dsubset 100c ⟩
⟨ C_XfactorKronSums_iweights_isubset 100d ⟩
⟨ C_XfactorKronSums_dweights_isubset 101a ⟩
⟨ RC_KronSums 95b ⟩
⟨ R_KronSums 94b ⟩
⟨ C_KronSums_Permutation_isubset 104b ⟩
⟨ C_KronSums_Permutation_dsubset 104a ⟩
⟨ C_XfactorKronSums_Permutation_isubset 105b ⟩
⟨ C_XfactorKronSums_Permutation_dsubset 105a ⟩
⟨ RC_KronSums_Permutation 103b ⟩
⟨ R_KronSums_Permutation 102b ⟩
◇
```

Fragment referenced in 23a.

```
> r1 <- rep(1:ncol(x), ncol(y))
> r2 <- rep(1:ncol(y), each = ncol(x))
> a0 <- colSums(x[subset, r1] * y[subset, r2] * weights[subset])
> a1 <- .Call(libcoin:::R_KronSums, x, P, y, weights, subset, 0L)
> a2 <- .Call(libcoin:::R_KronSums, x, P, y, as.double(weights), as.double(subset), 0L)
> a3 <- .Call(libcoin:::R_KronSums, x, P, y, weights, as.double(subset), 0L)
> a4 <- .Call(libcoin:::R_KronSums, x, P, y, as.double(weights), subset, 0L)
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4))
> a0 <- as.vector(colSums(Xfactor[subset, r1Xfactor] *
+                         y[subset, r2Xfactor] * weights[subset]))
```

```
> a1 <- .Call(libcoin:::R_KronSums, ix, Lx, y, weights, subset, 0L)
> a2 <- .Call(libcoin:::R_KronSums, ix, Lx, y, as.double(weights), as.double(subset), 0L)
> a3 <- .Call(libcoin:::R_KronSums, ix, Lx, y, weights, as.double(subset), 0L)
> a4 <- .Call(libcoin:::R_KronSums, ix, Lx, y, as.double(weights), subset, 0L)
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4))
```

⟨ *R_KronSums Prototype* 94a ⟩ ≡

```
    SEXP R_KronSums
    (
        ⟨ R x Input 23d ⟩
        SEXP P,
        ⟨ R y Input 24d ⟩
        ⟨ R weights Input 25b ⟩,
        ⟨ R subset Input 26a ⟩,
        SEXP symmetric
    )
    ◇
```

Fragment referenced in 22b, 94b.
Uses: `P` 24a, `R_KronSums` 94b.

⟨ *R_KronSums* 94b ⟩ ≡

```
    ⟨ R_KronSums Prototype 94a ⟩
    {
        SEXP ans;
        ⟨ C integer Q Input 24e ⟩;
        ⟨ C integer N Input 23c ⟩;
        ⟨ C integer Nsubset Input 26b ⟩;

        double center;

        Q = NCOL(y);
        N = XLENGTH(y) / Q;
        Nsubset = XLENGTH(subset);

        if (INTEGER(symmetric)[0]) {
            PROTECT(ans = allocVector(REALSXP, INTEGER(P)[0] * (INTEGER(P)[0] + 1) / 2));
        } else {
            PROTECT(ans = allocVector(REALSXP, INTEGER(P)[0] * Q));
        }
        RC_KronSums(x, N, INTEGER(P)[0], REAL(y), Q, INTEGER(symmetric)[0], &center, &center,
                    !DoCenter, weights, subset, Offset0, Nsubset, REAL(ans));
        UNPROTECT(1);
        return(ans);
    }
    ◇
```

Fragment referenced in 93b.
Defines: `R_KronSums` 94a, 152c, 153.
Uses: `DoCenter` 21b, `N` 23bc, `NCOL` 130c, `Nsubset` 26b, `Offset0` 21b, `P` 24a, `Q` 24e, `RC_KronSums` 95b, `subset` 26ade, `weights` 25b, `weights`, 25cd, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *RC_KronSums Prototype* 95a ⟩ ≡

```
void RC_KronSums
(
    ⟨ RC KronSums Input 95c ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩,
    ⟨ C KronSums Answer 96a ⟩
)
◇
```

Fragment referenced in 95b.
Uses: `RC_KronSums` 95b.

⟨ *RC_KronSums* 95b ⟩ ≡

```
⟨ RC_KronSums Prototype 95a ⟩
{
    if (TYPEOF(x) == INTSXP) {
        ⟨ KronSums Integer x 96b ⟩
    } else {
        ⟨ KronSums Double x 97a ⟩
    }
}
◇
```

Fragment referenced in 93b.
Defines: `RC_KronSums` 77d, 84a, 88a, 94b, 95a.
Uses: `x` 23d, 24bc.

⟨ *RC KronSums Input* 95c ⟩ ≡

```
⟨ R x Input 23d ⟩
⟨ C integer N Input 23c ⟩,
⟨ C integer P Input 24a ⟩,
⟨ C real y Input 24f ⟩
const int SYMMETRIC,
double *centerx,
double *centery,
const int CENTER,
◇
```

Fragment referenced in 95a.

⟨ *C KronSums Input* 95d ⟩ ≡

```
⟨ C real x Input 24b ⟩
⟨ C real y Input 24f ⟩
const int SYMMETRIC,
double *centerx,
double *centery,
const int CENTER,
◇
```

Fragment referenced in 97bc, 98ab.

⟨ *C KronSums Answer* 96a ⟩ ≡

```
double *PQ_ans
◇
```

Fragment referenced in 77c, 83b, 87b, 95a, 97bc, 98ab, 100bcd, 101a, 103a, 104ab, 105ab.

⟨ *KronSums Integer x* 96b ⟩ ≡

```
if (SYMMETRIC) error("not implemented");
if (CENTER) error("not implemented");
if (TYPEOF(weights) == INTSXP) {
    if (TYPEOF(subset) == INTSXP) {
        C_XfactorKronSums_iweights_isubset(INTEGER(x), N, P, y, Q,
            INTEGER(weights), XLENGTH(weights) > 0, INTEGER(subset),
            offset, Nsubset, PQ_ans);
    } else {
        C_XfactorKronSums_iweights_dsubset(INTEGER(x), N, P, y, Q,
            INTEGER(weights), XLENGTH(weights) > 0, REAL(subset),
            offset, Nsubset, PQ_ans);
    }
} else {
    if (TYPEOF(subset) == INTSXP) {
        C_XfactorKronSums_dweights_isubset(INTEGER(x), N, P, y, Q,
            REAL(weights), XLENGTH(weights) > 0, INTEGER(subset),
            offset, Nsubset, PQ_ans);
    } else {
        C_XfactorKronSums_dweights_dsubset(INTEGER(x), N, P, y, Q,
            REAL(weights), XLENGTH(weights) > 0, REAL(subset),
            offset, Nsubset, PQ_ans);
    }
}
◇
```

Fragment referenced in 95b.
Uses: `C_XfactorKronSums_dweights_dsubset` 100b, `C_XfactorKronSums_dweights_isubset` 101a, `C_XfactorKronSums_iweights_dsubset` 100c, `C_XfactorKronSums_iweights_isubset` 100d, `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `Q` 24e, `subset` 26ade, `weights` 25b, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *KronSums Double x* 97a ⟩ ≡

```
    if (TYPEOF(weights) == INTSXP) {
        if (TYPEOF(subset) == INTSXP) {
            C_KronSums_iweights_isubset(REAL(x), N, P, y, Q, SYMMETRIC, centerx, centery, CENTER,
                INTEGER(weights), XLENGTH(weights) > 0, INTEGER(subset),
                offset, Nsubset, PQ_ans);
        } else {
            C_KronSums_iweights_dsubset(REAL(x), N, P, y, Q, SYMMETRIC, centerx, centery, CENTER,
                INTEGER(weights), XLENGTH(weights) > 0, REAL(subset),
                offset, Nsubset, PQ_ans);
        }
    } else {
        if (TYPEOF(subset) == INTSXP) {
            C_KronSums_dweights_isubset(REAL(x), N, P, y, Q, SYMMETRIC, centerx, centery, CENTER,
                REAL(weights), XLENGTH(weights) > 0, INTEGER(subset),
                offset, Nsubset, PQ_ans);
        } else {
            C_KronSums_dweights_dsubset(REAL(x), N, P, y, Q, SYMMETRIC, centerx, centery, CENTER,
                REAL(weights), XLENGTH(weights) > 0, REAL(subset),
                offset, Nsubset, PQ_ans);
        }
    }
    ◇
```

Fragment referenced in 95b.
Uses: `C_KronSums_dweights_dsubset` 97b, `C_KronSums_dweights_isubset` 98b, `C_KronSums_iweights_dsubset` 97c, `C_KronSums_iweights_isubset` 98a, `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `Q` 24e, `subset` 26ade, `weights` 25b, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *C_KronSums_dweights_dsubset* 97b ⟩ ≡

```
    void C_KronSums_dweights_dsubset
    (
        ⟨ C KronSums Input 95d ⟩
        ⟨ C real weights Input 25d ⟩
        ⟨ C real subset Input 26e ⟩,
        ⟨ C KronSums Answer 96a ⟩
    ) {
        double *s, *w;
        ⟨ KronSums Body 99 ⟩
    }
    ◇
```

Fragment referenced in 93b.
Defines: `C_KronSums_dweights_dsubset` 97a.

⟨ *C_KronSums_iweights_dsubset* 97c ⟩ ≡

```
    void C_KronSums_iweights_dsubset
    (
        ⟨ C KronSums Input 95d ⟩
        ⟨ C integer weights Input 25c ⟩
        ⟨ C real subset Input 26e ⟩,
        ⟨ C KronSums Answer 96a ⟩
    ) {
        double *s;
        int *w;
        ⟨ KronSums Body 99 ⟩
    }
    ◇
```

Fragment referenced in 93b.
Defines: `C_KronSums_iweights_dsubset` 97a.

⟨ *C_KronSums_iweights_isubset* 98a ⟩ ≡

```
void C_KronSums_iweights_isubset
(
    ⟨ C KronSums Input 95d ⟩
    ⟨ C integer weights Input 25c ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C KronSums Answer 96a ⟩
) {
    int *s, *w;
    ⟨ KronSums Body 99 ⟩
}
◇
```

Fragment referenced in 93b.
Defines: `C_KronSums_iweights_isubset` 97a.

⟨ *C_KronSums_dweights_isubset* 98b ⟩ ≡

```
void C_KronSums_dweights_isubset
(
    ⟨ C KronSums Input 95d ⟩
    ⟨ C real weights Input 25d ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C KronSums Answer 96a ⟩
) {
    int *s;
    double *w;
    ⟨ KronSums Body 99 ⟩
}
◇
```

Fragment referenced in 93b.
Defines: `C_KronSums_dweights_isubset` 97a.

⟨ *KronSums Body* 99 ⟩ ≡

```
double *xx, *yy, cx = 0.0, cy = 0.0, *thisPQ_ans;
int idx;

for (int p = 0; p < P; p++) {
    for (int q = (SYMMETRIC ? p : 0); q < Q; q++) {
        /* SYMMETRIC is column-wise, default
           is row-wise (maybe need to change this) */
        if (SYMMETRIC) {
            idx = S(p, q, P);
        } else {
            idx = q * P + p;
        }
        PQ_ans[idx] = 0.0;
        thisPQ_ans = PQ_ans + idx;
        yy = y + N * q;
        xx = x + N * p;

        if (CENTER) {
            cx = centerx[p];
            cy = centery[q];
        }
        ⟨ init subset loop 88b ⟩
        ⟨ start subset loop 88c ⟩
        {
            xx = xx + diff;
            yy = yy + diff;
            if (HAS_WEIGHTS) {
                w = w + diff;
                if (CENTER) {
                    thisPQ_ans[0] += (xx[0] - cx) * (yy[0] - cy) * w[0];
                } else {
                    thisPQ_ans[0] += xx[0] * yy[0] * w[0];
                }
            } else {
                if (CENTER) {
                    thisPQ_ans[0] += (xx[0] - cx) * (yy[0] - cy);
                } else {
                    thisPQ_ans[0] += xx[0] * yy[0];
                }
            }
            ⟨ continue subset loop 89a ⟩
        }
        xx = xx + diff;
        yy = yy + diff;
        if (HAS_WEIGHTS) {
            w = w + diff;
            thisPQ_ans[0] += (xx[0] - cx) * (yy[0] - cy) * w[0];
        } else {
            thisPQ_ans[0] += (xx[0] - cx) * (yy[0] - cy);
        }
    }
}
```

◇

Fragment referenced in 97bc, 98ab.
Uses: `HAS_WEIGHTS` 25cd, `N` 23bc, `P` 24a, `Q` 24e, `S` 21a, `x` 23d, 24bc, `y` 24df, 25a.

Xfactor Kronecker Sums

⟨ *C XfactorKronSums Input* 100a ⟩ ≡

```
⟨ C integer x Input 24c ⟩
⟨ C real y Input 24f ⟩
◇
```

Fragment referenced in 100bcd, 101a.

⟨ *C_XfactorKronSums_dweights_dsubset* 100b ⟩ ≡

```
void C_XfactorKronSums_dweights_dsubset
(
    ⟨ C XfactorKronSums Input 100a ⟩
    ⟨ C real weights Input 25d ⟩
    ⟨ C real subset Input 26e ⟩,
    ⟨ C KronSums Answer 96a ⟩
) {
    double *s, *w;
    ⟨ XfactorKronSums Body 101b ⟩
}
◇
```

Fragment referenced in 93b.
Defines: `C_XfactorKronSums_dweights_dsubset` 96b.

⟨ *C_XfactorKronSums_iweights_dsubset* 100c ⟩ ≡

```
void C_XfactorKronSums_iweights_dsubset
(
    ⟨ C XfactorKronSums Input 100a ⟩
    ⟨ C integer weights Input 25c ⟩
    ⟨ C real subset Input 26e ⟩,
    ⟨ C KronSums Answer 96a ⟩
) {
    double *s;
    int *w;
    ⟨ XfactorKronSums Body 101b ⟩
}
◇
```

Fragment referenced in 93b.
Defines: `C_XfactorKronSums_iweights_dsubset` 96b.

⟨ *C_XfactorKronSums_iweights_isubset* 100d ⟩ ≡

```
void C_XfactorKronSums_iweights_isubset
(
    ⟨ C XfactorKronSums Input 100a ⟩
    ⟨ C integer weights Input 25c ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C KronSums Answer 96a ⟩
) {
    int *s, *w;
    ⟨ XfactorKronSums Body 101b ⟩
}
◇
```

Fragment referenced in 93b.
Defines: `C_XfactorKronSums_iweights_isubset` 96b.

⟨ *C_XfactorKronSums_dweights_isubset* 101a ⟩ ≡

```
void C_XfactorKronSums_dweights_isubset
(
    ⟨ C XfactorKronSums Input 100a ⟩
    ⟨ C real weights Input 25d ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C KronSums Answer 96a ⟩
) {
    int *s;
    double *w;
    ⟨ XfactorKronSums Body 101b ⟩
}
◇
```

Fragment referenced in 93b.
Defines: `C_XfactorKronSums_dweights_isubset` 96b.

⟨ *XfactorKronSums Body* 101b ⟩ ≡

```
int *xx, ixi;
double *yy;

for (int p = 0; p < mPQB(P, Q, 1); p++) PQ_ans[p] = 0.0;

for (int q = 0; q < Q; q++) {
    yy = y + N * q;
    xx = x;
    ⟨ init subset loop 88b ⟩
    ⟨ start subset loop 88c ⟩
    {
        xx = xx + diff;
        yy = yy + diff;
        ixi = xx[0] - 1;
        if (HAS_WEIGHTS) {
            w = w + diff;
            if (ixi >= 0)
                PQ_ans[ixi + q * P] += yy[0] * w[0];
        } else {
            if (ixi >= 0)
                PQ_ans[ixi + q * P] += yy[0];
        }
        ⟨ continue subset loop 89a ⟩
    }
    xx = xx + diff;
    yy = yy + diff;
    ixi = xx[0] - 1;
    if (HAS_WEIGHTS) {
        w = w + diff;
        if (ixi >= 0)
            PQ_ans[ixi + q * P] += yy[0] * w[0];
    } else {
        if (ixi >= 0)
            PQ_ans[ixi + q * P] += yy[0];
    }
}
◇
```

Fragment referenced in 100bcd, 101a.
Uses: `HAS_WEIGHTS` 25cd, `mPQB` 132a, `N` 23bc, `P` 24a, `Q` 24e, `x` 23d, 24bc, `y` 24df, 25a.

Permuted Kronecker Sums

```
> a0 <- colSums(x[subset, r1] * y[subsety, r2])
> a1 <- .Call(libcoin:::R_KronSums_Permutation, x, P, y, subset, subsety)
> a2 <- .Call(libcoin:::R_KronSums_Permutation, x, P, y, as.double(subset), as.double(subsety))
> stopifnot(isequal(a0, a1) && isequal(a0, a2))
> a0 <- as.vector(colSums(Xfactor[subset, r1Xfactor] * y[subsety, r2Xfactor]))
> a1 <- .Call(libcoin:::R_KronSums_Permutation, ix, Lx, y, subset, subsety)
> a2 <- .Call(libcoin:::R_KronSums_Permutation, ix, Lx, y, as.double(subset), as.double(subsety))
> stopifnot(isequal(a0, a1) && isequal(a0, a2))
```

⟨ *R_KronSums_Permutation Prototype* 102a ⟩ ≡

```
SEXP R_KronSums_Permutation
(
    ⟨ R x Input 23d ⟩
    SEXP P,
    ⟨ R y Input 24d ⟩
    ⟨ R subset Input 26a ⟩,
    SEXP subsety
)
◇
```

Fragment referenced in 22b, 102b.
Uses: `P` 24a, `R_KronSums_Permutation` 102b.

⟨ *R_KronSums_Permutation* 102b ⟩ ≡

```
⟨ R_KronSums_Permutation Prototype 102a ⟩
{
    SEXP ans;
    ⟨ C integer Q Input 24e ⟩;
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer Nsubset Input 26b ⟩;

    Q = NCOL(y);
    N = XLENGTH(y) / Q;
    Nsubset = XLENGTH(subset);

    PROTECT(ans = allocVector(REALSXP, INTEGER(P)[0] * Q));
    RC_KronSums_Permutation(x, N, INTEGER(P)[0], REAL(y), Q, subset, Offset0, Nsubset,
                            subsety, REAL(ans));
    UNPROTECT(1);
    return(ans);
}
◇
```

Fragment referenced in 93b.
Defines: `R_KronSums_Permutation` 102a, 152c, 153.
Uses: `N` 23bc, `NCOL` 130c, `Nsubset` 26b, `Offset0` 21b, `P` 24a, `Q` 24e, `RC_KronSums_Permutation` 103b, `subset` 26ade, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *RC_KronSums_Permutation Prototype* 103a ⟩ ≡

```
void RC_KronSums_Permutation
(
    ⟨ R x Input 23d ⟩
    ⟨ C integer N Input 23c ⟩,
    ⟨ C integer P Input 24a ⟩,
    ⟨ C real y Input 24f ⟩
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩,
    SEXP subsety,
    ⟨ C KronSums Answer 96a ⟩
)
◇
```

Fragment referenced in 103b.
Uses: `RC_KronSums_Permutation` 103b.

⟨ *RC_KronSums_Permutation* 103b ⟩ ≡

```
⟨ RC_KronSums_Permutation Prototype 103a ⟩
{
    if (TYPEOF(x) == INTSXP) {
        if (TYPEOF(subset) == INTSXP) {
            C_XfactorKronSums_Permutation_isubset(INTEGER(x), N, P, y, Q,
                                                  INTEGER(subset), offset, Nsubset,
                                                  INTEGER(subsety), PQ_ans);
        } else {
            C_XfactorKronSums_Permutation_dsubset(INTEGER(x), N, P, y, Q,
                                                  REAL(subset), offset, Nsubset,
                                                  REAL(subsety), PQ_ans);
        }
    } else {
        if (TYPEOF(subset) == INTSXP) {
            C_KronSums_Permutation_isubset(REAL(x), N, P, y, Q,
                                           INTEGER(subset), offset, Nsubset,
                                           INTEGER(subsety), PQ_ans);
        } else {
            C_KronSums_Permutation_dsubset(REAL(x), N, P, y, Q,
                                           REAL(subset), offset, Nsubset,
                                           REAL(subsety), PQ_ans);
        }
    }
}
◇
```

Fragment referenced in 93b.
Defines: `RC_KronSums_Permutation` 37, 102b, 103a.
Uses: `C_KronSums_Permutation_dsubset` 104a, `C_KronSums_Permutation_isubset` 104b, `C_XfactorKronSums_Permutation_dsubset` 105a, `C_XfactorKronSums_Permutation_isubset` 105b, `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `Q` 24e, `subset` 26ade, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *C_KronSums_Permutation_dsubset* 104a ⟩ ≡

```
void C_KronSums_Permutation_dsubset
(
    ⟨ C real x Input 24b ⟩
    ⟨ C real y Input 24f ⟩
    ⟨ C real subset Input 26e ⟩,
    double *subsety,
    ⟨ C KronSums Answer 96a ⟩
) {
    ⟨ KronSums Permutation Body 104c ⟩
}
◇
```

Fragment referenced in 93b.
Defines: `C_KronSums_Permutation_dsubset` 103b.

⟨ *C_KronSums_Permutation_isubset* 104b ⟩ ≡

```
void C_KronSums_Permutation_isubset
(
    ⟨ C real x Input 24b ⟩
    ⟨ C real y Input 24f ⟩
    ⟨ C integer subset Input 26d ⟩,
    int *subsety,
    ⟨ C KronSums Answer 96a ⟩
) {
    ⟨ KronSums Permutation Body 104c ⟩
}
◇
```

Fragment referenced in 93b.
Defines: `C_KronSums_Permutation_isubset` 103b.

Because `subset` might not be ordered (in the presence of blocks) we have to go through all elements explicitly here.

⟨ *KronSums Permutation Body* 104c ⟩ ≡

```
R_xlen_t qP, qN, pN, qPp;

for (int q = 0; q < Q; q++) {
    qN = q * N;
    qP = q * P;
    for (int p = 0; p < P; p++) {
        qPp = qP + p;
        PQ_ans[qPp] = 0.0;
        pN = p * N;
        for (R_xlen_t i = offset; i < Nsubset; i++)
            PQ_ans[qPp] += y[qN + (R_xlen_t) subsety[i] - 1] *
                           x[pN + (R_xlen_t) subset[i] - 1];
    }
}
◇
```

Fragment referenced in 104ab.
Uses: `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `Q` 24e, `subset` 26ade, `x` 23d, 24bc, `y` 24df, 25a.

Xfactor Permuted Kronecker Sums

⟨ *C_XfactorKronSums_Permutation_dsubset* 105a ⟩ ≡

```
void C_XfactorKronSums_Permutation_dsubset
(
    ⟨ C integer x Input 24c ⟩
    ⟨ C real y Input 24f ⟩
    ⟨ C real subset Input 26e ⟩,
    double *subsety,
    ⟨ C KronSums Answer 96a ⟩
) {
    ⟨ XfactorKronSums Permutation Body 105c ⟩
}
◇
```

Fragment referenced in 93b.
Defines: `C_XfactorKronSums_Permutation_dsubset` 103b.

⟨ *C_XfactorKronSums_Permutation_isubset* 105b ⟩ ≡

```
void C_XfactorKronSums_Permutation_isubset
(
    ⟨ C integer x Input 24c ⟩
    ⟨ C real y Input 24f ⟩
    ⟨ C integer subset Input 26d ⟩,
    int *subsety,
    ⟨ C KronSums Answer 96a ⟩
) {
    ⟨ XfactorKronSums Permutation Body 105c ⟩
}
◇
```

Fragment referenced in 93b.
Defines: `C_XfactorKronSums_Permutation_isubset` 103b.

⟨ *XfactorKronSums Permutation Body* 105c ⟩ ≡

```
R_xlen_t qP, qN;

for (int p = 0; p < mPQB(P, Q, 1); p++) PQ_ans[p] = 0.0;

for (int q = 0; q < Q; q++) {
    qP = q * P;
    qN = q * N;
    for (R_xlen_t i = offset; i < Nsubset; i++)
        PQ_ans[x[(R_xlen_t) subset[i] - 1] - 1 + qP] += y[qN + (R_xlen_t) subsety[i] - 1];
}
◇
```

Fragment referenced in 105ab.
Uses: `mPQB` 132a, `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `Q` 24e, `subset` 26ade, `x` 23d, 24bc, `y` 24df, 25a.

3.9.3 Column Sums

⟨ *colSums* 106a ⟩ ≡

```
⟨ C_colSums_dweights_dsubset 108b ⟩
⟨ C_colSums_iweights_dsubset 108c ⟩
⟨ C_colSums_iweights_isubset 108d ⟩
⟨ C_colSums_dweights_isubset 109a ⟩
⟨ RC_colSums 107b ⟩
⟨ R_colSums 106c ⟩
◇
```

Fragment referenced in 23a.

```
> a0 <- colSums(x[subset, ] * weights[subset])
> a1 <- .Call(libcoin:::R_colSums, x, weights, subset)
> a2 <- .Call(libcoin:::R_colSums, x, as.double(weights), as.double(subset))
> a3 <- .Call(libcoin:::R_colSums, x, weights, as.double(subset))
> a4 <- .Call(libcoin:::R_colSums, x, as.double(weights), subset)
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4))
```

⟨ *R_colSums Prototype* 106b ⟩ ≡

```
SEXP R_colSums
(
    ⟨ R x Input 23d ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩
)
◇
```

Fragment referenced in 22b, 106c.
Uses: `R_colSums` 106c.

⟨ *R_colSums* 106c ⟩ ≡

```
⟨ R_colSums Prototype 106b ⟩
{
    SEXP ans;
    int P;
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer Nsubset Input 26b ⟩;
    double center;

    P = NCOL(x);
    N = XLENGTH(x) / P;
    Nsubset = XLENGTH(subset);

    PROTECT(ans = allocVector(REALSXP, P));
    RC_colSums(REAL(x), N, P, Power1, &center, !DoCenter, weights, subset, Offset0,
               Nsubset, REAL(ans));
    UNPROTECT(1);
    return(ans);
}
◇
```

Fragment referenced in 106a.
Defines: `R_colSums` 106b, 152c, 153.
Uses: `DoCenter` 21b, `N` 23bc, `NCOL` 130c, `Nsubset` 26b, `Offset0` 21b, `P` 24a, `Power1` 21b, `RC_colSums` 107b, `subset` 26ade, `weights` 25b, `weights`, 25cd, `x` 23d, 24bc.

⟨ *RC_colSums Prototype* 107a ⟩ ≡

```
void RC_colSums
(
    ⟨ C colSums Input 107c ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩,
    ⟨ C colSums Answer 108a ⟩
)
◇
```

Fragment referenced in 107b.
Uses: `RC_colSums` 107b.

⟨ *RC_colSums* 107b ⟩ ≡

```
⟨ RC_colSums Prototype 107a ⟩
{
    if (TYPEOF(weights) == INTSXP) {
        if (TYPEOF(subset) == INTSXP) {
            C_colSums_iweights_isubset(x, N, P, power, centerx, CENTER,
                                       INTEGER(weights), XLENGTH(weights) > 0, INTEGER(subset),
                                       offset, Nsubset, P_ans);
        } else {
            C_colSums_iweights_dsubset(x, N, P, power, centerx, CENTER,
                                       INTEGER(weights), XLENGTH(weights) > 0, REAL(subset),
                                       offset, Nsubset, P_ans);
        }
    } else {
        if (TYPEOF(subset) == INTSXP) {
            C_colSums_dweights_isubset(x, N, P, power, centerx, CENTER,
                                       REAL(weights), XLENGTH(weights) > 0, INTEGER(subset),
                                       offset, Nsubset, P_ans);
        } else {
            C_colSums_dweights_dsubset(x, N, P, power, centerx, CENTER,
                                       REAL(weights), XLENGTH(weights) > 0, REAL(subset),
                                       offset, Nsubset, P_ans);
        }
    }
}
◇
```

Fragment referenced in 106a.
Defines: `RC_colSums` 81c, 84a, 85b, 88a, 106c, 107a.
Uses: `C_colSums_dweights_dsubset` 108b, `C_colSums_dweights_isubset` 109a, `C_colSums_iweights_dsubset` 108c, `C_colSums_iweights_isubset` 108d, `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `subset` 26ade, `weights` 25b, `x` 23d, 24bc.

⟨ *C colSums Input* 107c ⟩ ≡

```
⟨ C real x Input 24b ⟩
const int power,
double *centerx,
const int CENTER,
◇
```

Fragment referenced in 107a, 108bcd, 109a.

⟨ *C colSums Answer* 108a ⟩ ≡

```
double *P_ans
◇
```

Fragment referenced in 81b, 107a, 108bcd, 109a.

⟨ *C_colSums_dweights_dsubset* 108b ⟩ ≡

```
void C_colSums_dweights_dsubset
(
    ⟨ C colSums Input 107c ⟩
    ⟨ C real weights Input 25d ⟩
    ⟨ C real subset Input 26e ⟩,
    ⟨ C colSums Answer 108a ⟩
) {
    double *s, *w;
    ⟨ colSums Body 109b ⟩
}
◇
```

Fragment referenced in 106a.
Defines: `C_colSums_dweights_dsubset` 107b.

⟨ *C_colSums_iweights_dsubset* 108c ⟩ ≡

```
void C_colSums_iweights_dsubset
(
    ⟨ C colSums Input 107c ⟩
    ⟨ C integer weights Input 25c ⟩
    ⟨ C real subset Input 26e ⟩,
    ⟨ C colSums Answer 108a ⟩
) {
    double *s;
    int *w;
    ⟨ colSums Body 109b ⟩
}
◇
```

Fragment referenced in 106a.
Defines: `C_colSums_iweights_dsubset` 107b.

⟨ *C_colSums_iweights_isubset* 108d ⟩ ≡

```
void C_colSums_iweights_isubset
(
    ⟨ C colSums Input 107c ⟩
    ⟨ C integer weights Input 25c ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C colSums Answer 108a ⟩
) {
    int *s, *w;
    ⟨ colSums Body 109b ⟩
}
◇
```

Fragment referenced in 106a.
Defines: `C_colSums_iweights_isubset` 107b.

⟨ *C_colSums_dweights_isubset* 109a ⟩ ≡

```
void C_colSums_dweights_isubset
(
    ⟨ C colSums Input 107c ⟩
    ⟨ C real weights Input 25d ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C colSums Answer 108a ⟩
) {
    int *s;
    double *w;
    ⟨ colSums Body 109b ⟩
}
◇
```

Fragment referenced in 106a.
Defines: `C_colSums_dweights_isubset` 107b.

⟨ *colSums Body* 109b ⟩ ≡

```
double *xx, cx = 0.0;

for (int p = 0; p < P; p++) {
    P_ans[0] = 0.0;
    xx = x + N * p;
    if (CENTER) {
        cx = centerx[p];
    }
    ⟨ init subset loop 88b ⟩
    ⟨ start subset loop 88c ⟩
    {
        xx = xx + diff;
        if (HAS_WEIGHTS) {
            w = w + diff;
            P_ans[0] += pow(xx[0] - cx, power) * w[0];
        } else {
            P_ans[0] += pow(xx[0] - cx, power);
        }
        ⟨ continue subset loop 89a ⟩
    }
    xx = xx + diff;
    if (HAS_WEIGHTS) {
        w = w + diff;
        P_ans[0] += pow(xx[0] - cx, power) * w[0];
    } else {
        P_ans[0] += pow(xx[0] - cx, power);
    }
    P_ans++;
}
◇
```

Fragment referenced in 108bcd, 109a.
Uses: `HAS_WEIGHTS` 25cd, `N` 23bc, `P` 24a, `x` 23d, 24bc.

3.9.4 Tables

OneTable Sums

⟨ *Tables* 110a ⟩ ≡

```
⟨ C_OneTableSums_dweights_dsubset 112d ⟩
⟨ C_OneTableSums_iweights_dsubset 113a ⟩
⟨ C_OneTableSums_iweights_isubset 113b ⟩
⟨ C_OneTableSums_dweights_isubset 113c ⟩
⟨ RC_OneTableSums 112a ⟩
⟨ R_OneTableSums 111a ⟩
⟨ C_TwoTableSums_dweights_dsubset 117a ⟩
⟨ C_TwoTableSums_iweights_dsubset 117b ⟩
⟨ C_TwoTableSums_iweights_isubset 117c ⟩
⟨ C_TwoTableSums_dweights_isubset 118a ⟩
⟨ RC_TwoTableSums 116a ⟩
⟨ R_TwoTableSums 115a ⟩
⟨ C_ThreeTableSums_dweights_dsubset 121b ⟩
⟨ C_ThreeTableSums_iweights_dsubset 121c ⟩
⟨ C_ThreeTableSums_iweights_isubset 121d ⟩
⟨ C_ThreeTableSums_dweights_isubset 122a ⟩
⟨ RC_ThreeTableSums 120b ⟩
⟨ R_ThreeTableSums 119b ⟩
◇
```

Fragment referenced in 23a.

```
> a0 <- as.vector(xtabs(weights ~ ixf, subset = subset))
> a1 <- ctabs(ix, weights = weights, subset = subset)[-1]
> a2 <- ctabs(ix, weights = as.double(weights), subset = as.double(subset))[-1]
> a3 <- ctabs(ix, weights = weights, subset = as.double(subset))[-1]
> a4 <- ctabs(ix, weights = as.double(weights), subset = subset)[-1]
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4))
```

⟨ *R_OneTableSums Prototype* 110b ⟩ ≡

```
SEXP R_OneTableSums
(
    ⟨ R x Input 23d ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩
)
◇
```

Fragment referenced in 22b, 111a.
Uses: `R_OneTableSums` 111a.

⟨ *R_OneTableSums* 111a ⟩ ≡

```
⟨ R_OneTableSums Prototype 110b ⟩
{
    SEXP ans;
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer Nsubset Input 26b ⟩;
    int P;

    N = XLENGTH(x);
    Nsubset = XLENGTH(subset);
    P = NLEVELS(x) + 1;

    PROTECT(ans = allocVector(REALSXP, P));
    RC_OneTableSums(INTEGER(x), N, P, weights, subset,
                    Offset0, Nsubset, REAL(ans));
    UNPROTECT(1);
    return(ans);
}
◇
```

Fragment referenced in 110a.
Defines: `R_OneTableSums` 15b, 110b, 123c, 152c, 153.
Uses: `N` 23bc, `NLEVELS` 131a, `Nsubset` 26b, `Offset0` 21b, `P` 24a, `RC_OneTableSums` 112a, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc.

⟨ *RC_OneTableSums Prototype* 111b ⟩ ≡

```
void RC_OneTableSums
(
    ⟨ C OneTableSums Input 112b ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩,
    ⟨ C OneTableSums Answer 112c ⟩
)
◇
```

Fragment referenced in 112a.
Uses: `RC_OneTableSums` 112a.

⟨ *RC_OneTableSums* 112a ⟩ ≡

```
⟨ RC_OneTableSums Prototype 111b ⟩
{
    if (TYPEOF(weights) == INTSXP) {
        if (TYPEOF(subset) == INTSXP) {
            C_OneTableSums_iweights_isubset(x, N, P,
                                            INTEGER(weights), XLENGTH(weights) > 0, INTEGER(subset),
                                            offset, Nsubset, P_ans);
        } else {
            C_OneTableSums_iweights_dsubset(x, N, P,
                                            INTEGER(weights), XLENGTH(weights) > 0, REAL(subset),
                                            offset, Nsubset, P_ans);
        }
    } else {
        if (TYPEOF(subset) == INTSXP) {
            C_OneTableSums_dweights_isubset(x, N, P,
                                            REAL(weights), XLENGTH(weights) > 0, INTEGER(subset),
                                            offset, Nsubset, P_ans);
        } else {
            C_OneTableSums_dweights_dsubset(x, N, P,
                                            REAL(weights), XLENGTH(weights) > 0, REAL(subset),
                                            offset, Nsubset, P_ans);
        }
    }
}
◇
```

Fragment referenced in 110a.
Defines: `RC_OneTableSums` 34a, 37, 85b, 111ab.
Uses: `C_OneTableSums_dweights_dsubset` 112d, `C_OneTableSums_dweights_isubset` 113c, `C_OneTableSums_iweights_dsubset` 113a, `C_OneTableSums_iweights_isubset` 113b, `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `subset` 26ade, `weights` 25b, `x` 23d, 24bc.

⟨ *C OneTableSums Input* 112b ⟩ ≡

```
⟨ C integer x Input 24c ⟩
◇
```

Fragment referenced in 111b, 112d, 113abc.

⟨ *C OneTableSums Answer* 112c ⟩ ≡

```
double *P_ans
◇
```

Fragment referenced in 85a, 111b, 112d, 113abc.

⟨ *C_OneTableSums_dweights_dsubset* 112d ⟩ ≡

```
void C_OneTableSums_dweights_dsubset
(
    ⟨ C OneTableSums Input 112b ⟩
    ⟨ C real weights Input 25d ⟩
    ⟨ C real subset Input 26e ⟩,
    ⟨ C OneTableSums Answer 112c ⟩
) {
    double *s, *w;
    ⟨ OneTableSums Body 114a ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_OneTableSums_dweights_dsubset` 112a.

⟨ *C_OneTableSums_iweights_dsubset* 113a ⟩ ≡

```
void C_OneTableSums_iweights_dsubset
(
    ⟨ C OneTableSums Input 112b ⟩
    ⟨ C integer weights Input 25c ⟩
    ⟨ C real subset Input 26e ⟩,
    ⟨ C OneTableSums Answer 112c ⟩
) {
    double *s;
    int *w;
    ⟨ OneTableSums Body 114a ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_OneTableSums_iweights_dsubset` 112a.

⟨ *C_OneTableSums_iweights_isubset* 113b ⟩ ≡

```
void C_OneTableSums_iweights_isubset
(
    ⟨ C OneTableSums Input 112b ⟩
    ⟨ C integer weights Input 25c ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C OneTableSums Answer 112c ⟩
) {
    int *s, *w;
    ⟨ OneTableSums Body 114a ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_OneTableSums_iweights_isubset` 112a.

⟨ *C_OneTableSums_dweights_isubset* 113c ⟩ ≡

```
void C_OneTableSums_dweights_isubset
(
    ⟨ C OneTableSums Input 112b ⟩
    ⟨ C real weights Input 25d ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C OneTableSums Answer 112c ⟩
) {
    int *s;
    double *w;
    ⟨ OneTableSums Body 114a ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_OneTableSums_dweights_isubset` 112a.

⟨ *OneTableSums Body* 114a ⟩ ≡

```
int *xx;

for (int p = 0; p < P; p++) P_ans[p] = 0.0;

xx = x;
⟨ init subset loop 88b ⟩
⟨ start subset loop 88c ⟩
{
    xx = xx + diff;
    if (HAS_WEIGHTS) {
        w = w + diff;
        P_ans[xx[0]] += (double) w[0];
    } else {
        P_ans[xx[0]]++;
    }
    ⟨ continue subset loop 89a ⟩
}
xx = xx + diff;
if (HAS_WEIGHTS) {
    w = w + diff;
    P_ans[xx[0]] += w[0];
} else {
    P_ans[xx[0]]++;
}
◇
```

Fragment referenced in 112d, 113abc.
Uses: `HAS_WEIGHTS` 25cd, `P` 24a, `x` 23d, 24bc.

TwoTable Sums

```
> a0 <- xtabs(weights ~ ixf + iyf, subset = subset)
> class(a0) <- "matrix"
> dimnames(a0) <- NULL
> attributes(a0)$call <- NULL
> a1 <- ctabs(ix, iy, weights = weights, subset = subset)[-1, -1]
> a2 <- ctabs(ix, iy, weights = as.double(weights),
+             subset = as.double(subset))[-1, -1]
> a3 <- ctabs(ix, iy, weights = weights, subset = as.double(subset))[-1, -1]
> a4 <- ctabs(ix, iy, weights = as.double(weights), subset = subset)[-1, -1]
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4))
```

⟨ *R_TwoTableSums Prototype* 114b ⟩ ≡

```
SEXP R_TwoTableSums
(
    ⟨ R x Input 23d ⟩
    ⟨ R y Input 24d ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩
)
◇
```

Fragment referenced in 22b, 115a.
Uses: `R_TwoTableSums` 115a.

⟨ *R_TwoTableSums* 115a ⟩ ≡

```
⟨ R_TwoTableSums Prototype 114b ⟩
{
    SEXP ans, dim;
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer Nsubset Input 26b ⟩;
    int P, Q;

    N = XLENGTH(x);
    Nsubset = XLENGTH(subset);
    P = NLEVELS(x) + 1;
    Q = NLEVELS(y) + 1;

    PROTECT(ans = allocVector(REALSXP, mPQB(P, Q, 1)));
    PROTECT(dim = allocVector(INTSXP, 2));
    INTEGER(dim)[0] = P;
    INTEGER(dim)[1] = Q;
    dimgets(ans, dim);
    RC_TwoTableSums(INTEGER(x), N, P, INTEGER(y), Q,
                    weights, subset, Offset0, Nsubset, REAL(ans));
    UNPROTECT(2);
    return(ans);
}
◇
```

Fragment referenced in 110a.
Defines: `R_TwoTableSums` 15b, 114b, 152c, 153.
Uses: `mPQB` 132a, `N` 23bc, `NLEVELS` 131a, `Nsubset` 26b, `Offset0` 21b, `P` 24a, `Q` 24e, `RC_TwoTableSums` 116a, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *RC_TwoTableSums Prototype* 115b ⟩ ≡

```
void RC_TwoTableSums
(
    ⟨ C TwoTableSums Input 116b ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩,
    ⟨ C TwoTableSums Answer 116c ⟩
)
◇
```

Fragment referenced in 116a.
Uses: `RC_TwoTableSums` 116a.

⟨ *RC_TwoTableSums* 116a ⟩ ≡

```
⟨ RC_TwoTableSums Prototype 115b ⟩
{
    if (TYPEOF(weights) == INTSXP) {
        if (TYPEOF(subset) == INTSXP) {
            C_TwoTableSums_iweights_isubset(x, N, P, y, Q,
                                            INTEGER(weights), XLENGTH(weights) > 0, INTEGER(subset),
                                            offset, Nsubset, PQ_ans);
        } else {
            C_TwoTableSums_iweights_dsubset(x, N, P, y, Q,
                                            INTEGER(weights), XLENGTH(weights) > 0, REAL(subset),
                                            offset, Nsubset, PQ_ans);
        }
    } else {
        if (TYPEOF(subset) == INTSXP) {
            C_TwoTableSums_dweights_isubset(x, N, P, y, Q,
                                            REAL(weights), XLENGTH(weights) > 0, INTEGER(subset),
                                            offset, Nsubset, PQ_ans);
        } else {
            C_TwoTableSums_dweights_dsubset(x, N, P, y, Q,
                                            REAL(weights), XLENGTH(weights) > 0, REAL(subset),
                                            offset, Nsubset, PQ_ans);
        }
    }
}
◇
```

Fragment referenced in 110a.
Defines: `RC_TwoTableSums` 41a, 115ab.
Uses: `C_TwoTableSums_dweights_dsubset` 117a, `C_TwoTableSums_dweights_isubset` 118a, `C_TwoTableSums_iweights_dsubset` 117b, `C_TwoTableSums_iweights_isubset` 117c, `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `Q` 24e, `subset` 26ade, `weights` 25b, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *C TwoTableSums Input* 116b ⟩ ≡

```
⟨ C integer x Input 24c ⟩
⟨ C integer y Input 25a ⟩
◇
```

Fragment referenced in 115b, 117abc, 118a.

⟨ *C TwoTableSums Answer* 116c ⟩ ≡

```
double *PQ_ans
◇
```

Fragment referenced in 115b, 117abc, 118a.

⟨ *C_TwoTableSums_dweights_dsubset* 117a ⟩ ≡

```
void C_TwoTableSums_dweights_dsubset
(
    ⟨ C TwoTableSums Input 116b ⟩
    ⟨ C real weights Input 25d ⟩
    ⟨ C real subset Input 26e ⟩,
    ⟨ C TwoTableSums Answer 116c ⟩
) {
    double *s, *w;
    ⟨ TwoTableSums Body 118b ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_TwoTableSums_dweights_dsubset` 116a.

⟨ *C_TwoTableSums_iweights_dsubset* 117b ⟩ ≡

```
void C_TwoTableSums_iweights_dsubset
(
    ⟨ C TwoTableSums Input 116b ⟩
    ⟨ C integer weights Input 25c ⟩
    ⟨ C real subset Input 26e ⟩,
    ⟨ C TwoTableSums Answer 116c ⟩
) {
    double *s;
    int *w;
    ⟨ TwoTableSums Body 118b ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_TwoTableSums_iweights_dsubset` 116a.

⟨ *C_TwoTableSums_iweights_isubset* 117c ⟩ ≡

```
void C_TwoTableSums_iweights_isubset
(
    ⟨ C TwoTableSums Input 116b ⟩
    ⟨ C integer weights Input 25c ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C TwoTableSums Answer 116c ⟩
) {
    int *s, *w;
    ⟨ TwoTableSums Body 118b ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_TwoTableSums_iweights_isubset` 116a.

⟨ *C_TwoTableSums_dweights_isubset* 118a ⟩ ≡

```
void C_TwoTableSums_dweights_isubset
(
    ⟨ C TwoTableSums Input 116b ⟩
    ⟨ C real weights Input 25d ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C TwoTableSums Answer 116c ⟩
) {
    int *s;
    double *w;
    ⟨ TwoTableSums Body 118b ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_TwoTableSums_dweights_isubset` 116a.

⟨ *TwoTableSums Body* 118b ⟩ ≡

```
int *xx, *yy;

for (int p = 0; p < Q * P; p++) PQ_ans[p] = 0.0;

yy = y;
xx = x;
⟨ init subset loop 88b ⟩
⟨ start subset loop 88c ⟩
{
    xx = xx + diff;
    yy = yy + diff;
    if (HAS_WEIGHTS) {
        w = w + diff;
        PQ_ans[yy[0] * P + xx[0]] += (double) w[0];
    } else {
        PQ_ans[yy[0] * P + xx[0]]++;
    }
    ⟨ continue subset loop 89a ⟩
}
xx = xx + diff;
yy = yy + diff;
if (HAS_WEIGHTS) {
    w = w + diff;
    PQ_ans[yy[0] * P + xx[0]] += w[0];
} else {
    PQ_ans[yy[0] * P + xx[0]]++;
}
◇
```

Fragment referenced in 117abc, 118a.
Uses: `HAS_WEIGHTS` 25cd, `P` 24a, `Q` 24e, `x` 23d, 24bc, `y` 24df, 25a.

ThreeTable Sums

```
> a0 <- xtabs(weights ~ ixf + iyf + block, subset = subset)
> class(a0) <- "array"
> dimnames(a0) <- NULL
> attributes(a0)$call <- NULL
> a1 <- ctabs(ix, iy, block, weights, subset)[-1, -1,]
> a2 <- ctabs(ix, iy, block, as.double(weights), as.double(subset))[-1,-1,]
> a3 <- ctabs(ix, iy, block, weights, as.double(subset))[-1,-1,]
> a4 <- ctabs(ix, iy, block, as.double(weights), subset)[-1,-1,]
> stopifnot(isequal(a0, a1) && isequal(a0, a2) &&
+           isequal(a0, a3) && isequal(a0, a4))
```

⟨ *R_ThreeTableSums Prototype* 119a ⟩ ≡

```
SEXP R_ThreeTableSums
(
    ⟨ R x Input 23d ⟩
    ⟨ R y Input 24d ⟩
    ⟨ R block Input 26f ⟩,
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩
)
◇
```

Fragment referenced in 22b, 119b.
Uses: `R_ThreeTableSums` 119b.

⟨ *R_ThreeTableSums* 119b ⟩ ≡

```
⟨ R_ThreeTableSums Prototype 119a ⟩
{
    SEXP ans, dim;
    ⟨ C integer N Input 23c ⟩;
    ⟨ C integer Nsubset Input 26b ⟩;
    int P, Q, B;

    N = XLENGTH(x);
    Nsubset = XLENGTH(subset);
    P = NLEVELS(x) + 1;
    Q = NLEVELS(y) + 1;
    B = NLEVELS(block);

    PROTECT(ans = allocVector(REALSXP, mPQB(P, Q, B)));
    PROTECT(dim = allocVector(INTSXP, 3));
    INTEGER(dim)[0] = P;
    INTEGER(dim)[1] = Q;
    INTEGER(dim)[2] = B;
    dimgets(ans, dim);
    RC_ThreeTableSums(INTEGER(x), N, P, INTEGER(y), Q,
                      INTEGER(block), B,
                      weights, subset, Offset0, Nsubset, REAL(ans));
    UNPROTECT(2);
    return(ans);
}
◇
```

Fragment referenced in 110a.
Defines: `R_ThreeTableSums` 15b, 119a, 152c, 153.
Uses: `B` 27a, `block` 26f, 27b, `mPQB` 132a, `N` 23bc, `NLEVELS` 131a, `Nsubset` 26b, `Offset0` 21b, `P` 24a, `Q` 24e, `RC_ThreeTableSums` 120b, `subset` 26ade, `weights` 25b, `weights,` 25cd, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *RC_ThreeTableSums Prototype* 120a ⟩ ≡

```
void RC_ThreeTableSums
(
    ⟨ C ThreeTableSums Input 120c ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ C subset range Input 26c ⟩,
    ⟨ C ThreeTableSums Answer 121a ⟩
)
◇
```

Fragment referenced in 120b.
Uses: `RC_ThreeTableSums` 120b.

⟨ *RC_ThreeTableSums* 120b ⟩ ≡

```
⟨ RC_ThreeTableSums Prototype 120a ⟩
{
    if (TYPEOF(weights) == INTSXP) {
        if (TYPEOF(subset) == INTSXP) {
            C_ThreeTableSums_iweights_isubset(x, N, P, y, Q, block, B,
                                              INTEGER(weights), XLENGTH(weights) > 0, INTEGER(subset),
                                              offset, Nsubset, PQL_ans);
        } else {
            C_ThreeTableSums_iweights_dsubset(x, N, P, y, Q, block, B,
                                              INTEGER(weights), XLENGTH(weights) > 0, REAL(subset),
                                              offset, Nsubset, PQL_ans);
        }
    } else {
        if (TYPEOF(subset) == INTSXP) {
            C_ThreeTableSums_dweights_isubset(x, N, P, y, Q, block, B,
                                              REAL(weights), XLENGTH(weights) > 0, INTEGER(subset),
                                              offset, Nsubset, PQL_ans);
        } else {
            C_ThreeTableSums_dweights_dsubset(x, N, P, y, Q, block, B,
                                              REAL(weights), XLENGTH(weights) > 0, REAL(subset),
                                              offset, Nsubset, PQL_ans);
        }
    }
}
◇
```

Fragment referenced in 110a.
Defines: `RC_ThreeTableSums` 41a, 119b, 120a.
Uses: `B` 27a, `block` 26f, 27b, `C_ThreeTableSums_dweights_dsubset` 121b, `C_ThreeTableSums_dweights_isubset` 122a, `C_ThreeTableSums_iweights_dsubset` 121c, `C_ThreeTableSums_iweights_isubset` 121d, `N` 23bc, `Nsubset` 26b, `offset` 26c, `P` 24a, `Q` 24e, `subset` 26ade, `weights` 25b, `x` 23d, 24bc, `y` 24df, 25a.

⟨ *C ThreeTableSums Input* 120c ⟩ ≡

```
⟨ C integer x Input 24c ⟩
⟨ C integer y Input 25a ⟩
⟨ C integer block Input 27b ⟩
◇
```

Fragment referenced in 120a, 121bcd, 122a.

⟨ *C ThreeTableSums Answer* 121a ⟩ ≡

```
double *PQL_ans
◇
```

Fragment referenced in 120a, 121bcd, 122a.

⟨ *C_ThreeTableSums_dweights_dsubset* 121b ⟩ ≡

```
void C_ThreeTableSums_dweights_dsubset
(
    ⟨ C ThreeTableSums Input 120c ⟩
    ⟨ C real weights Input 25d ⟩
    ⟨ C real subset Input 26e ⟩,
    ⟨ C ThreeTableSums Answer 121a ⟩
) {
    double *s, *w;
    ⟨ ThreeTableSums Body 122b ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_ThreeTableSums_dweights_dsubset` 120b.

⟨ *C_ThreeTableSums_iweights_dsubset* 121c ⟩ ≡

```
void C_ThreeTableSums_iweights_dsubset
(
    ⟨ C ThreeTableSums Input 120c ⟩
    ⟨ C integer weights Input 25c ⟩
    ⟨ C real subset Input 26e ⟩,
    ⟨ C ThreeTableSums Answer 121a ⟩
) {
    double *s;
    int *w;
    ⟨ ThreeTableSums Body 122b ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_ThreeTableSums_iweights_dsubset` 120b.

⟨ *C_ThreeTableSums_iweights_isubset* 121d ⟩ ≡

```
void C_ThreeTableSums_iweights_isubset
(
    ⟨ C ThreeTableSums Input 120c ⟩
    ⟨ C integer weights Input 25c ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C ThreeTableSums Answer 121a ⟩
) {
    int *s, *w;
    ⟨ ThreeTableSums Body 122b ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_ThreeTableSums_iweights_isubset` 120b.

⟨ *C_ThreeTableSums_dweights_isubset* 122a ⟩ ≡

```
void C_ThreeTableSums_dweights_isubset
(
    ⟨ C ThreeTableSums Input 120c ⟩
    ⟨ C real weights Input 25d ⟩
    ⟨ C integer subset Input 26d ⟩,
    ⟨ C ThreeTableSums Answer 121a ⟩
) {
    int *s;
    double *w;
    ⟨ ThreeTableSums Body 122b ⟩
}
◇
```

Fragment referenced in 110a.
Defines: `C_ThreeTableSums_dweights_isubset` 120b.

⟨ *ThreeTableSums Body* 122b ⟩ ≡

```
int *xx, *yy, *bb, PQ = mPQB(P, Q, 1);

for (int p = 0; p < PQ * B; p++) PQL_ans[p] = 0.0;

yy = y;
xx = x;
bb = block;
⟨ init subset loop 88b ⟩
⟨ start subset loop 88c ⟩
{
    xx = xx + diff;
    yy = yy + diff;
    bb = bb + diff;
    if (HAS_WEIGHTS) {
        w = w + diff;
        PQL_ans[(bb[0] - 1) * PQ + yy[0] * P + xx[0]] += (double) w[0];
    } else {
        PQL_ans[(bb[0] - 1) * PQ + yy[0] * P + xx[0]]++;
    }
    ⟨ continue subset loop 89a ⟩
}
xx = xx + diff;
yy = yy + diff;
bb = bb + diff;
if (HAS_WEIGHTS) {
    w = w + diff;
    PQL_ans[(bb[0] - 1) * PQ + yy[0] * P + xx[0]] += w[0];
} else {
    PQL_ans[(bb[0] - 1) * PQ + yy[0] * P + xx[0]]++;
}
◇
```

Fragment referenced in 121bcd, 122a.
Uses: `B` 27a, `block` 26f, 27b, `HAS_WEIGHTS` 25cd, `mPQB` 132a, `P` 24a, `Q` 24e, `x` 23d, 24bc, `y` 24df, 25a.

3.10 Utilities

3.10.1 Blocks

```
> sb <- sample(block)
> ns1 <- do.call(c, tapply(subset, sb[subset], function(i) i))
> ns2 <- .Call(libcoin:::R_order_subset_wrt_block, y, integer(0), subset, sb)
> stopifnot(isequal(ns1, ns2))
```

⟨ *Utils* 123a ⟩ ≡

```
⟨ C_setup_subset 125a ⟩
⟨ C_setup_subset_block 125b ⟩
⟨ C_order_subset_wrt_block 126a ⟩
⟨ RC_order_subset_wrt_block 124b ⟩
⟨ R_order_subset_wrt_block 123c ⟩
◇
```

Fragment referenced in 23a.

⟨ *R_order_subset_wrt_block Prototype* 123b ⟩ ≡

```
SEXP R_order_subset_wrt_block
(
    ⟨ R y Input 24d ⟩
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ R block Input 26f ⟩
)
◇
```

Fragment referenced in 22b, 123c.
Uses: `R_order_subset_wrt_block` 123c.

⟨ *R_order_subset_wrt_block* 123c ⟩ ≡

```
⟨ R_order_subset_wrt_block Prototype 123b ⟩
{
    ⟨ C integer N Input 23c ⟩;
    SEXP blockTable, ans;

    N = XLENGTH(y) / NCOL(y);

    if (XLENGTH(weights) > 0)
        error("cannot deal with weights here");

    if (NLEVELS(block) > 1) {
        PROTECT(blockTable = R_OneTableSums(block, weights, subset));
    } else {
        PROTECT(blockTable = allocVector(REALSXP, 2));
        REAL(blockTable)[0] = 0.0;
        REAL(blockTable)[1] = RC_Sums(N, weights, subset, Offset0, XLENGTH(subset));
    }

    PROTECT(ans = RC_order_subset_wrt_block(N, subset, block, blockTable));

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

Fragment referenced in 123a.
Defines: `R_order_subset_wrt_block` 123b, 152c, 153.
Uses: `block` 26f, 27b, `blockTable` 27c, `N` 23bc, `NCOL` 130c, `NLEVELS` 131a, `Offset0` 21b, `RC_order_subset_wrt_block` 124b, `RC_Sums` 91a, `R_OneTableSums` 111a, `subset` 26ade, `weights` 25b, `weights,` 25cd, `y` 24df, 25a.

⟨ *RC_order_subset_wrt_block Prototype* 124a ⟩ ≡

```
SEXP RC_order_subset_wrt_block
(
    ⟨ C integer N Input 23c ⟩,
    ⟨ R subset Input 26a ⟩,
    ⟨ R block Input 26f ⟩,
    ⟨ R blockTable Input 27c ⟩
)
◇
```

Fragment referenced in 124b.
Uses: `RC_order_subset_wrt_block` 124b.

⟨ *RC_order_subset_wrt_block* 124b ⟩ ≡

```
⟨ RC_order_subset_wrt_block Prototype 124a ⟩
{
    SEXP ans;
    int NOBLOCK = (XLENGTH(block) == 0 || XLENGTH(blockTable) == 2);

    if (XLENGTH(subset) > 0) {
        if (NOBLOCK) {
            return(subset);
        } else {
            PROTECT(ans = allocVector(TYPEOF(subset), XLENGTH(subset)));
            C_order_subset_wrt_block(subset, block, blockTable, ans);
            UNPROTECT(1);
            return(ans);
        }
    } else {
        PROTECT(ans = allocVector(TYPEOF(subset), N));
        if (NOBLOCK) {
            C_setup_subset(N, ans);
        } else {
            C_setup_subset_block(N, block, blockTable, ans);
        }
        UNPROTECT(1);
        return(ans);
    }
}
◇
```

Fragment referenced in 123a.
Defines: `RC_order_subset_wrt_block` 34a, 37, 123c, 124a.
Uses: `block` 26f, 27b, `blockTable` 27c, `C_order_subset_wrt_block` 126a, `C_setup_subset` 125a, `C_setup_subset_block` 125b, `N` 23bc, `subset` 26ade.

⟨ *C_setup_subset* 125a ⟩ ≡

```
void C_setup_subset
(
    ⟨ C integer N Input 23c ⟩,
    SEXP ans
) {
    for (R_xlen_t i = 0; i < N; i++) {
        /* ans is R style index in 1:N */
        if (TYPEOF(ans) == INTSXP) {
            INTEGER(ans)[i] = i + 1;
        } else {
            REAL(ans)[i] = (double) i + 1;
        }
    }
}
◇
```

Fragment referenced in 123a.
Defines: `C_setup_subset` 124b, 127a.
Uses: `N` 23bc.

⟨ *C_setup_subset_block* 125b ⟩ ≡

```
void C_setup_subset_block
(
    ⟨ C integer N Input 23c ⟩,
    ⟨ R block Input 26f ⟩,
    ⟨ R blockTable Input 27c ⟩,
    SEXP ans
) {
    double *cumtable;
    int Nlevels = LENGTH(blockTable);

    cumtable = R_Calloc(Nlevels, double);
    for (int k = 0; k < Nlevels; k++) cumtable[k] = 0.0;

    /* table[0] are missings, ie block == 0 ! */
    for (int k = 1; k < Nlevels; k++)
        cumtable[k] = cumtable[k - 1] + REAL(blockTable)[k - 1];

    for (R_xlen_t i = 0; i < N; i++) {
        /* ans is R style index in 1:N */
        if (TYPEOF(ans) == INTSXP) {
            INTEGER(ans)[(int) cumtable[INTEGER(block)[i]]++] = i + 1;
        } else {
            REAL(ans)[(R_xlen_t) cumtable[INTEGER(block)[i]]++] = (double) i + 1;
        }
    }

    R_Free(cumtable);
}
◇
```

Fragment referenced in 123a.
Defines: `C_setup_subset_block` 124b.
Uses: `block` 26f, 27b, `blockTable` 27c, `N` 23bc.

⟨ *C_order_subset_wrt_block* 126a ⟩ ≡

```
void C_order_subset_wrt_block
(
    ⟨ R subset Input 26a ⟩,
    ⟨ R block Input 26f ⟩,
    ⟨ R blockTable Input 27c ⟩,
    SEXP ans
) {
    double *cumtable;
    int Nlevels = LENGTH(blockTable);

    cumtable = R_Calloc(Nlevels, double);
    for (int k = 0; k < Nlevels; k++) cumtable[k] = 0.0;

    /* table[0] are missings, ie block == 0 ! */
    for (int k = 1; k < Nlevels; k++)
        cumtable[k] = cumtable[k - 1] + REAL(blockTable)[k - 1];

    /* subset is R style index in 1:N */
    if (TYPEOF(subset) == INTSXP) {
        for (R_xlen_t i = 0; i < XLENGTH(subset); i++)
            INTEGER(ans)[(int) cumtable[INTEGER(block)[INTEGER(subset)[i] - 1]]++] = INTEGER(subset)[i];
    } else {
        for (R_xlen_t i = 0; i < XLENGTH(subset); i++)
            REAL(ans)[(R_xlen_t) cumtable[INTEGER(block)[(R_xlen_t) REAL(subset)[i] - 1]]++] = REAL(subset
    }

    R_Free(cumtable);
}
◇
```

Fragment referenced in 123a.
Defines: `C_order_subset_wrt_block` 124b.
Uses: `block` 26f, 27b, `blockTable` 27c, `N` 23bc, `subset` 26ade.

⟨ *RC_setup_subset Prototype* 126b ⟩ ≡

```
SEXP RC_setup_subset
(
    ⟨ C integer N Input 23c ⟩,
    ⟨ R weights Input 25b ⟩,
    ⟨ R subset Input 26a ⟩
)
◇
```

Fragment referenced in 127a.
Uses: `RC_setup_subset` 127a.

Because this will only be used when really needed (in Permutations) we can be a little bit more generous with memory here. The return value is always `REALSXP`.

⟨ *RC_setup_subset* 127a ⟩ ≡

```
⟨ RC_setup_subset Prototype 126b ⟩
{
    SEXP ans, mysubset;
    R_xlen_t sumweights;

    if (XLENGTH(subset) == 0) {
        PROTECT(mysubset = allocVector(REALSXP, N));
        C_setup_subset(N, mysubset);
    } else {
        PROTECT(mysubset = coerceVector(subset, REALSXP));
    }

    if (XLENGTH(weights) == 0) {
        UNPROTECT(1);
        return(mysubset);
    }

    sumweights = (R_xlen_t) RC_Sums(N, weights, mysubset, Offset0, XLENGTH(subset));
    PROTECT(ans = allocVector(REALSXP, sumweights));

    R_xlen_t itmp = 0;
    for (R_xlen_t i = 0; i < XLENGTH(mysubset); i++) {
        if (TYPEOF(weights) == REALSXP) {
            for (R_xlen_t j = 0; j < REAL(weights)[(R_xlen_t) REAL(mysubset)[i] - 1]; j++)
                REAL(ans)[itmp++] = REAL(mysubset)[i];
        } else {
            for (R_xlen_t j = 0; j < INTEGER(weights)[(R_xlen_t) REAL(mysubset)[i] - 1]; j++)
                REAL(ans)[itmp++] = REAL(mysubset)[i];
        }
    }
    UNPROTECT(2);
    return(ans);
}
◇
```

Fragment referenced in 127b.
Defines: `RC_setup_subset` 37, 126b.
Uses: `C_setup_subset` 125a, `N` 23bc, `Offset0` 21b, `RC_Sums` 91a, `subset` 26ade, `sumweights` 25e, `weights` 25b, `weights,` 25cd.

3.10.2 Permutation Helpers

⟨ *Permutations* 127b ⟩ ≡

```
⟨ RC_setup_subset 127a ⟩
⟨ C_Permute 128a ⟩
⟨ C_doPermute 128b ⟩
⟨ C_PermuteBlock 129a ⟩
⟨ C_doPermuteBlock 129b ⟩
◇
```

Fragment referenced in 23a.

⟨ *C_Permute* 128a ⟩ ≡

```
void C_Permute
(
    double *subset,
    ⟨ C integer Nsubset Input 26b ⟩,
    double *ans
) {
    R_xlen_t n = Nsubset, j;

    for (R_xlen_t i = 0; i < Nsubset; i++) {
        j = n * unif_rand();
        ans[i] = subset[j];
        subset[j] = subset[--n];
    }
}
◇
```

Fragment referenced in 127b.
Defines: `C_Permute` 128b, 129a.
Uses: `Nsubset` 26b, `subset` 26ade.

⟨ *C_doPermute* 128b ⟩ ≡

```
void C_doPermute
(
    double *subset,
    ⟨ C integer Nsubset Input 26b ⟩,
    double *Nsubset_tmp,
    double *perm
) {
    Memcpy(Nsubset_tmp, subset, Nsubset);
    C_Permute(Nsubset_tmp, Nsubset, perm);
}
◇
```

Fragment referenced in 127b.
Defines: `C_doPermute` 37.
Uses: `C_Permute` 128a, `Nsubset` 26b, `subset` 26ade.

⟨ *C_PermuteBlock* 129a ⟩ ≡

```
void C_PermuteBlock
(
    double *subset,
    double *table,
    int Nlevels,
    double *ans
) {
    double *px, *pans;

    px = subset;
    pans = ans;

    for (R_xlen_t j = 0; j < Nlevels; j++) {
        if (table[j] > 0) {
            C_Permute(px, (R_xlen_t) table[j], pans);
            px += (R_xlen_t) table[j];
            pans += (R_xlen_t) table[j];
        }
    }
}
◇
```

Fragment referenced in 127b.
Defines: `C_PermuteBlock` 129b.
Uses: `C_Permute` 128a, `subset` 26ade.

⟨ *C_doPermuteBlock* 129b ⟩ ≡

```
void C_doPermuteBlock
(
    double *subset,
    ⟨ C integer Nsubset Input 26b ⟩,
    double *table,
    int Nlevels,
    double *Nsubset_tmp,
    double *perm
) {
    Memcpy(Nsubset_tmp, subset, Nsubset);
    C_PermuteBlock(Nsubset_tmp, table, Nlevels, perm);
}
◇
```

Fragment referenced in 127b.
Defines: `C_doPermuteBlock` 37.
Uses: `C_PermuteBlock` 129a, `Nsubset` 26b, `subset` 26ade.

3.10.3 Other Utils

⟨ *MoreUtils* 130a ⟩ ≡

⟨ *NROW* 130b ⟩
⟨ *NCOL* 130c ⟩
⟨ *NLEVELS* 131a ⟩
⟨ *C_kronecker* 133b ⟩
⟨ *R_kronecker* 133a ⟩
⟨ *C_kronecker_sym* 134 ⟩
⟨ *C_KronSums_sym* 135a ⟩
⟨ *C_MPinv_sym* 137 ⟩
⟨ *R_MPinv_sym* 136b ⟩
⟨ *R_unpack_sym* 139 ⟩
⟨ *R_pack_sym* 140c ⟩
◇

Fragment referenced in 23a.

⟨ *NROW* 130b ⟩ ≡

```
int NROW
(
    SEXP x
) {
    SEXP a;
    a = getAttrib(x, R_DimSymbol);
    if (a == R_NilValue) return(XLENGTH(x));
    if (TYPEOF(a) == REALSXP)
        return(REAL(a)[0]);
    return(INTEGER(a)[0]);
}
```
◇

Fragment referenced in 130a.
Defines: `NROW` 6, 8, 9ab, 14, 33a, 37, 43b, 44, 61b, 131a, 133a, 140c.
Uses: `x` 23d, 24bc.

⟨ *NCOL* 130c ⟩ ≡

```
int NCOL
(
    SEXP x
) {
    SEXP a;
    a = getAttrib(x, R_DimSymbol);
    if (a == R_NilValue) return(1);
    if (TYPEOF(a) == REALSXP)
        return(REAL(a)[1]);
    return(INTEGER(a)[1]);
}
```
◇

Fragment referenced in 130a.
Defines: `NCOL` 12, 31b, 41b, 61b, 81a, 83a, 94b, 102b, 106c, 123c, 133a.
Uses: `x` 23d, 24bc.

⟨ *NLEVELS* 131a ⟩ ≡

```
int NLEVELS
(
    SEXP x
) {
    SEXP a;
    int maxlev = 0;

    a = getAttrib(x, R_LevelsSymbol);
    if (a == R_NilValue) {
        if (TYPEOF(x) != INTSXP)
            error("cannot determine number of levels");
        for (R_xlen_t i = 0; i < XLENGTH(x); i++) {
            if (INTEGER(x)[i] > maxlev)
                maxlev = INTEGER(x)[i];
        }
        return(maxlev);
    }
    return(NROW(a));
}
◇
```

Fragment referenced in 130a.
Defines: `NLEVELS` 31b, 41b, 111a, 115a, 119b, 123c.
Uses: `NROW` 130b, `x` 23d, 24bc.

Check for integer overflow when computing $P(P+1)/2$ and PQ.

⟨ *PP12* 131b ⟩ ≡

```
int PP12
(
    int P
) {
    double dP = (double) P;
    double ans;

    ans = dP * (dP + 1) / 2;

    if (ans > INT_MAX)
        error("cannot allocate memory: number of levels too large");

    return((int) ans);
}
◇
```

Fragment referenced in 141a.
Defines: `PP12` 34a, 44, 46, 51, 79, 88a, 148, 149a.
Uses: `P` 24a.

⟨ *mPQB* 132a ⟩ ≡

```
int mPQB
(
    int P,
    int Q,
    int B
) {
    double ans = P * Q * B;

    if (ans > INT_MAX)
        error("cannot allocate memory: number of levels too large");

    return((int) ans);
}
◇
```

Fragment referenced in 141a.
Defines: `mPQB` 36a, 37, 45, 48, 52a, 71, 73a, 76b, 78b, 79, 80a, 101b, 105c, 115a, 119b, 122b, 148.
Uses: `B` 27a, `P` 24a, `Q` 24e.

```
> A <- matrix(runif(12), ncol = 3)
> B <- matrix(runif(10), ncol = 2)
> K1 <- kronecker(A, B)
> K2 <- .Call(libcoin:::R_kronecker, A, B)
> stopifnot(isequal(K1, K2))
```

⟨ *R_kronecker Prototype* 132b ⟩ ≡

```
SEXP R_kronecker
(
    SEXP A,
    SEXP B
)
◇
```

Fragment referenced in 22b, 133a.
Uses: `B` 27a.

This function can be called from other packages.

`"libcoinAPI.h"` 132c≡

```
extern SEXP libcoin_R_kronecker(
    SEXP A, SEXP B
) {
    static SEXP(*fun)(SEXP, SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP, SEXP))
            R_GetCCallable("libcoin", "R_kronecker");
    return fun(A, B);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `B` 27a.

⟨ *R_kronecker* 133a ⟩ ≡

```
⟨ R_kronecker Prototype 132b ⟩
{
    int m, n, r, s;
    SEXP ans;

    if (!isReal(A) || !isReal(B))
        error("R_kronecker: A and / or B are not of type REALSXP");

    m = NROW(A);
    n = NCOL(A);
    r = NROW(B);
    s = NCOL(B);

    PROTECT(ans = allocMatrix(REALSXP, m * n, r * s));
    C_kronecker(REAL(A), m, n, REAL(B), r, s, 1, REAL(ans));
    UNPROTECT(1);
    return(ans);
}
◇
```

Fragment referenced in 130a.
Uses: `B` 27a, `C_kronecker` 133b, `NCOL` 130c, `NROW` 130b.

⟨ *C_kronecker* 133b ⟩ ≡

```
void C_kronecker
(
    const double *A,
    const int m,
    const int n,
    const double *B,
    const int r,
    const int s,
    const int overwrite,
    double *ans
) {
    int mr, js, ir;
    double y;

    if (overwrite) {
        for (int i = 0; i < m * r * n * s; i++) ans[i] = 0.0;
    }

    mr = m * r;
    for (int i = 0; i < m; i++) {
        ir = i * r;
        for (int j = 0; j < n; j++) {
            js = j * s;
            y = A[j*m + i];
            for (int k = 0; k < r; k++) {
                for (int l = 0; l < s; l++)
                    ans[(js + l) * mr + ir + k] += y * B[l * r + k];
            }
        }
    }
}
◇
```

Fragment referenced in 130a.
Defines: `C_kronecker` 80a, 133a.
Uses: `B` 27a, `y` 24df, 25a.

⟨ *C_kronecker_sym* 134 ⟩ ≡

```
void C_kronecker_sym
(
    const double *A,
    const int m,
    const double *B,
    const int r,
    const int overwrite,
    double *ans
) {
    int mr, js, ir, s;
    double y;

    mr = m * r;
    s = r;

    if (overwrite) {
        for (int i = 0; i < mr * (mr + 1) / 2; i++) ans[i] = 0.0;
    }

    for (int i = 0; i < m; i++) {
        ir = i * r;
        for (int j = 0; j <= i; j++) {
            js = j * s;
            y = A[S(i, j, m)];
            for (int k = 0; k < r; k++) {
                for (int l = 0; l < (j < i ? s : k + 1); l++) {
                    ans[S(ir + k, js + l, mr)] += y * B[S(k, l, r)];
                }
            }
        }
    }
}
```

◇

Fragment referenced in 130a.
Defines: `C_kronecker_sym` 79.
Uses: `B` 27a, `S` 21a, `y` 24df, 25a.

⟨ *C_KronSums_sym* 135a ⟩ ≡

```
/* sum_i (t(x[i,]) %*% x[i,]) */
void C_KronSums_sym_
(
    ⟨ C real x Input 24b ⟩
    double *PP_sym_ans
) {
    int pN, qN, SpqP;

    for (int q = 0; q < P; q++) {
        qN = q * N;
        for (int p = 0; p <= q; p++) {
            PP_sym_ans[S(p, q, P)] = 0.0;
            pN = p * N;
            SpqP = S(p, q, P);
            for (int i = 0; i < N; i++)
                 PP_sym_ans[SpqP] += x[qN + i] * x[pN + i];
        }
    }
}
◇
```

Fragment referenced in 130a.
Defines: `C_KronSums_sym` Never used.
Uses: `N` 23bc, `P` 24a, `S` 21a, `x` 23d, 24bc.

```
> covar <- vcov(ls1)
> covar_sym <- ls1$Covariance
> n <- (sqrt(1 + 8 * length(covar_sym)) - 1) / 2
> tol <- sqrt(.Machine$double.eps)
> MP1 <- MPinverse(covar, tol)
> MP2 <- .Call(libcoin:::R_MPinv_sym, covar_sym, as.integer(n), tol)
> lt <- lower.tri(covar, diag = TRUE)
> stopifnot(isequal(MP1$MPinv[lt], MP2$MPinv) &&
+           isequal(MP1$rank, MP2$rank))
```

⟨ *R_MPinv_sym Prototype* 135b ⟩ ≡

```
SEXP R_MPinv_sym
(
    SEXP x,
    SEXP n,
    SEXP tol
)
◇
```

Fragment referenced in 22b, 136b.
Uses: `R_MPinv_sym` 136b, `x` 23d, 24bc.

This function can be called from other packages.

"libcoinAPI.h" 136a≡

```
extern SEXP libcoin_R_MPinv_sym(
    SEXP x, SEXP n, SEXP tol
) {
    static SEXP(*fun)(SEXP, SEXP, SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP, SEXP, SEXP))
            R_GetCCallable("libcoin", "R_MPinv_sym");
    return fun(x, n, tol);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `R_MPinv_sym` 136b, `x` 23d, 24bc.

⟨ *R_MPinv_sym* 136b ⟩ ≡

```
⟨ R_MPinv_sym Prototype 135b ⟩
{
    int m;
    SEXP ans, names, MPinv, rank;

    m = INTEGER(n)[0];
    if (m == 0)
        m = (int) (sqrt(0.25 + 2 * LENGTH(x)) - 0.5);

    PROTECT(ans = allocVector(VECSXP, 2));
    PROTECT(names = allocVector(STRSXP, 2));
    SET_VECTOR_ELT(ans, 0, MPinv = allocVector(REALSXP, LENGTH(x)));
    SET_STRING_ELT(names, 0, mkChar("MPinv"));
    SET_VECTOR_ELT(ans, 1, rank = allocVector(INTSXP, 1));
    SET_STRING_ELT(names, 1, mkChar("rank"));
    namesgets(ans, names);

    C_MPinv_sym(REAL(x), m, REAL(tol)[0], REAL(MPinv), INTEGER(rank));

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

Fragment referenced in 130a.
Defines: `R_MPinv_sym` 135b, 136a, 152c, 153.
Uses: `x` 23d, 24bc.

⟨ *C_MPinv_sym* 137 ⟩ ≡

```
void C_MPinv_sym
(
    const double *x,
    const int n,
    const double tol,
    double *dMP,
    int *rank
) {
    double *val, *vec, dtol, *rx, *work, valinv;
    int valzero = 0, info = 0, kn;

    if (n == 1) {
        if (x[0] > tol) {
            dMP[0] = 1 / x[0];
            rank[0] = 1;
        } else {
            dMP[0] = 0;
            rank[0] = 0;
        }
    } else {
        rx = R_Calloc(n * (n + 1) / 2, double);
        Memcpy(rx, x, n * (n + 1) / 2);
        work = R_Calloc(3 * n, double);
        val = R_Calloc(n, double);
        vec = R_Calloc(n * n, double);

        F77_CALL(dspev)("V", "L", &n, rx, val, vec, &n, work,
                        &info FCONE FCONE);

        dtol = val[n - 1] * tol;

        for (int k = 0; k < n; k++)
            valzero += (val[k] < dtol);
        rank[0] = n - valzero;

        for (int k = 0; k < n * (n + 1) / 2; k++) dMP[k] = 0.0;

        for (int k = valzero; k < n; k++) {
            valinv = 1 / val[k];
            kn = k * n;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j <= i; j++) {
                    /* MP is symmetric */
                    dMP[S(i, j, n)] += valinv * vec[kn + i] * vec[kn + j];
                }
            }
        }
        R_Free(rx); R_Free(work); R_Free(val); R_Free(vec);
    }
}
◇
```

Fragment referenced in 130a.
Uses: `S` 21a, `x` 23d, 24bc.

```
> m <- matrix(c(3, 2, 1,
+               2, 4, 2,
+               1, 2, 5),
+             ncol = 3)
> s <- m[lower.tri(m, diag = TRUE)]
> u1 <- .Call(libcoin:::R_unpack_sym, s, NULL, 0L)
> u2 <- .Call(libcoin:::R_unpack_sym, s, NULL, 1L)
> stopifnot(isequal(m, u1) && isequal(diag(m), u2))
```

⟨ *R_unpack_sym Prototype* 138a ⟩ ≡

```
    SEXP R_unpack_sym
    (
        SEXP x,
        SEXP names,
        SEXP diagonly
    )
    ◇
```

Fragment referenced in 22b, 139.
Uses: `R_unpack_sym` 139, `x` 23d, 24bc.

This function can be called from other packages.

`"libcoinAPI.h"` 138b≡

```
    extern SEXP libcoin_R_unpack_sym(
        SEXP x, SEXP names, SEXP diagonly
    ) {
        static SEXP(*fun)(SEXP, SEXP, SEXP) = NULL;
        if (fun == NULL)
            fun = (SEXP(*)(SEXP, SEXP, SEXP))
                R_GetCCallable("libcoin", "R_unpack_sym");
        return fun(x, names, diagonly);
    }
    ◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `R_unpack_sym` 139, `x` 23d, 24bc.

⟨ *R_unpack_sym* 139 ⟩ ≡

```
    ⟨ R_unpack_sym Prototype 138a ⟩
    {
        R_xlen_t n, k = 0;
        SEXP ans, dimnames;
        double *dx, *dans;

        /* m = n * (n + 1)/2 <=> n^2 + n - 2 * m = 0 */
        n = sqrt(0.25 + 2 * XLENGTH(x)) - 0.5;

        dx = REAL(x);
        if (INTEGER(diagonly)[0]) {
            PROTECT(ans = allocVector(REALSXP, n));
            if (names != R_NilValue) {
                namesgets(ans, names);
            }
            dans = REAL(ans);
            for (R_xlen_t i = 0; i < n; i++) {
                dans[i] = dx[k];
                k += n - i;
            }
        } else {
            PROTECT(ans = allocMatrix(REALSXP, n, n));
            if (names != R_NilValue) {
                PROTECT(dimnames = allocVector(VECSXP, 2));
                SET_VECTOR_ELT(dimnames, 0, names);
                SET_VECTOR_ELT(dimnames, 1, names);
                dimnamesgets(ans, dimnames);
                UNPROTECT(1);
            }
            dans = REAL(ans);
            for (R_xlen_t i = 0; i < n; i++) {
                dans[i * n + i] = dx[k];     /* diagonal */
                k++;
                for (R_xlen_t j = i + 1; j < n; j++) {
                    dans[i * n + j] = dx[k]; /* lower triangular */
                    dans[j * n + i] = dx[k]; /* upper triangular */
                    k++;
                }
            }
        }

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

Fragment referenced in 130a.
Defines: `R_unpack_sym` 10, 138ab, 152c, 153.
Uses: `x` 23d, 24bc.

```
> m <- matrix(c(4, 3, 2, 1,
+               3, 5, 4, 2,
+               2, 4, 6, 5,
+               1, 2, 5, 7),
+             ncol = 4)
> s <- m[lower.tri(m, diag = TRUE)]
> p <- .Call(libcoin:::R_pack_sym, m)
> stopifnot(isequal(s, p))
```

⟨ *R_pack_sym Prototype* 140a ⟩ ≡

```
SEXP R_pack_sym
(
    SEXP x
)
◇
```

Fragment referenced in 22b, 140c.
Uses: `R_pack_sym` 140c, `x` 23d, 24bc.

This function can be called from other packages.

`"libcoinAPI.h"` 140b≡

```
extern SEXP libcoin_R_pack_sym(
    SEXP x
) {
    static SEXP(*fun)(SEXP) = NULL;
    if (fun == NULL)
        fun = (SEXP(*)(SEXP))
            R_GetCCallable("libcoin", "R_pack_sym");
    return fun(x);
}
◇
```

File defined by 30d, 36c, 38c, 40b, 47b, 50c, 53, 55b, 61a, 132c, 136a, 138b, 140b.
Uses: `R_pack_sym` 140c, `x` 23d, 24bc.

⟨ *R_pack_sym* 140c ⟩ ≡

```
⟨ R_pack_sym Prototype 140a ⟩
{
    R_xlen_t n, k = 0;
    SEXP ans;
    double *dx, *dans;

    n = NROW(x);
    dx = REAL(x);
    PROTECT(ans = allocVector(REALSXP, n * (n + 1) / 2));
    dans = REAL(ans);

    for (R_xlen_t i = 0; i < n; i++) {
        for (R_xlen_t j = i; j < n; j++) {
          dans[k] = dx[i * n + j];
          k++;
        }
    }

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

Fragment referenced in 130a.
Defines: `R_pack_sym` 140ab, 152c, 153.
Uses: `NROW` 130b, `x` 23d, 24bc.

3.11 Memory

⟨ *Memory* 141a ⟩ ≡

```
⟨ C_get_P 141c ⟩
⟨ C_get_Q 142a ⟩
⟨ PP12 131b ⟩
⟨ mPQB 132a ⟩
⟨ C_get_varonly 142b ⟩
⟨ C_get_Xfactor 142c ⟩
⟨ C_get_LinearStatistic 142d ⟩
⟨ C_get_Expectation 143a ⟩
⟨ C_get_Variance 143b ⟩
⟨ C_get_Covariance 143c ⟩
⟨ C_get_ExpectationX 144a ⟩
⟨ C_get_ExpectationInfluence 144b ⟩
⟨ C_get_CovarianceInfluence 144c ⟩
⟨ C_get_VarianceInfluence 144d ⟩
⟨ C_get_TableBlock 145a ⟩
⟨ C_get_Sumweights 145b ⟩
⟨ C_get_Table 145c ⟩
⟨ C_get_dimTable 145d ⟩
⟨ C_get_B 146a ⟩
⟨ C_get_nresample 146b ⟩
⟨ C_get_PermutedLinearStatistic 146c ⟩
⟨ C_get_tol 146d ⟩
⟨ RC_init_LECV_1d 149b ⟩
⟨ RC_init_LECV_2d 150 ⟩
◇
```

Fragment referenced in 23a.

⟨ *R LECV Input* 141b ⟩ ≡

```
SEXP LECV
◇
```

Fragment referenced in 50b, 52b, 141c, 142abcd, 143abc, 144abcd, 145abcd, 146abcd.
Defines: `LECV` 38bc, 39a, 50c, 51, 52a, 53, 54, 55ab, 56, 69b, 71, 141c, 142abcd, 143abc, 144abcd, 145abcd, 146abcd.

⟨ *C_get_P* 141c ⟩ ≡

```
int C_get_P
(
    ⟨ R LECV Input 141b ⟩
) {
    return(INTEGER(VECTOR_ELT(LECV, dim_SLOT))[0]);
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_P` 33a, 39a, 46, 52a, 56, 71, 143bc, 146b.
Uses: `dim_SLOT` 21b, `LECV` 141b.

⟨ *C_get_Q* 142a ⟩ ≡

```
int C_get_Q
(
    ⟨ R LECV Input 141b ⟩
) {
    return(INTEGER(VECTOR_ELT(LECV, dim_SLOT))[1]);
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_Q` 33a, 39a, 46, 52a, 71, 143bc, 146b.
Uses: `dim_SLOT` 21b, `LECV` 141b.

⟨ *C_get_varonly* 142b ⟩ ≡

```
int C_get_varonly
(
    ⟨ R LECV Input 141b ⟩
) {
    return(INTEGER(VECTOR_ELT(LECV, varonly_SLOT))[0]);
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_varonly` 32, 34a, 36a, 39a, 44, 45, 46, 52a, 54, 71, 143c.
Uses: `LECV` 141b, `varonly_SLOT` 21b.

⟨ *C_get_Xfactor* 142c ⟩ ≡

```
int C_get_Xfactor
(
    ⟨ R LECV Input 141b ⟩
) {
    return(INTEGER(VECTOR_ELT(LECV, Xfactor_SLOT))[0]);
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_Xfactor` 46.
Uses: `LECV` 141b, `Xfactor_SLOT` 21b.

⟨ *C_get_LinearStatistic* 142d ⟩ ≡

```
double* C_get_LinearStatistic
(
    ⟨ R LECV Input 141b ⟩
) {
    return(REAL(VECTOR_ELT(LECV, LinearStatistic_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_LinearStatistic` 33b, 45, 51, 54, 71, 149a.
Uses: `LECV` 141b, `LinearStatistic_SLOT` 21b.

⟨ *C_get_Expectation* 143a ⟩ ≡

```
double* C_get_Expectation
(
    ⟨ R LECV Input 141b ⟩
) {
    return(REAL(VECTOR_ELT(LECV, Expectation_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_Expectation` 35a, 39a, 43b, 51, 54, 71, 149a.
Uses: `Expectation_SLOT` 21b, `LECV` 141b.

⟨ *C_get_Variance* 143b ⟩ ≡

```
double* C_get_Variance
(
    ⟨ R LECV Input 141b ⟩
) {
    int PQ = C_get_P(LECV) * C_get_Q(LECV);
    double *var, *covar;

    if (isNull(VECTOR_ELT(LECV, Variance_SLOT))) {
        SET_VECTOR_ELT(LECV, Variance_SLOT,
                       allocVector(REALSXP, PQ));
        if (!isNull(VECTOR_ELT(LECV, Covariance_SLOT))) {
            covar = REAL(VECTOR_ELT(LECV, Covariance_SLOT));
            var = REAL(VECTOR_ELT(LECV, Variance_SLOT));
            for (int p = 0; p < PQ; p++)
                var[p] = covar[S(p, p, PQ)];
        }
    }
    return(REAL(VECTOR_ELT(LECV, Variance_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_Variance` 35c, 36a, 39a, 44, 45, 54, 71, 143c, 149a.
Uses: `Covariance_SLOT` 21b, `C_get_P` 141c, `C_get_Q` 142a, `LECV` 141b, `S` 21a, `Variance_SLOT` 21b.

⟨ *C_get_Covariance* 143c ⟩ ≡

```
double* C_get_Covariance
(
    ⟨ R LECV Input 141b ⟩
) {
    int PQ = C_get_P(LECV) * C_get_Q(LECV);
    if (C_get_varonly(LECV) && PQ > 1)
        error("Cannot extract covariance from variance only object");
    if (C_get_varonly(LECV) && PQ == 1)
        return(C_get_Variance(LECV));
    return(REAL(VECTOR_ELT(LECV, Covariance_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_Covariance` 35d, 36a, 39a, 44, 45, 51, 54, 71, 149a.
Uses: `Covariance_SLOT` 21b, `C_get_P` 141c, `C_get_Q` 142a, `C_get_Variance` 143b, `C_get_varonly` 142b, `LECV` 141b.

⟨ *C_get_ExpectationX* 144a ⟩ ≡

```
double* C_get_ExpectationX
(
    ⟨ R LECV Input 141b ⟩
) {
    return(REAL(VECTOR_ELT(LECV, ExpectationX_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_ExpectationX` 34a, 46, 71.
Uses: `ExpectationX_SLOT` 21b, `LECV` 141b.

⟨ *C_get_ExpectationInfluence* 144b ⟩ ≡

```
double* C_get_ExpectationInfluence
(
    ⟨ R LECV Input 141b ⟩
) {
    return(REAL(VECTOR_ELT(LECV, ExpectationInfluence_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_ExpectationInfluence` 34a, 46, 149a.
Uses: `ExpectationInfluence_SLOT` 21b, `LECV` 141b.

⟨ *C_get_CovarianceInfluence* 144c ⟩ ≡

```
double* C_get_CovarianceInfluence
(
    ⟨ R LECV Input 141b ⟩
) {
    return(REAL(VECTOR_ELT(LECV, CovarianceInfluence_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_CovarianceInfluence` 34a, 44, 71, 149a.
Uses: `CovarianceInfluence_SLOT` 21b, `LECV` 141b.

⟨ *C_get_VarianceInfluence* 144d ⟩ ≡

```
double* C_get_VarianceInfluence
(
    ⟨ R LECV Input 141b ⟩
) {
    return(REAL(VECTOR_ELT(LECV, VarianceInfluence_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_VarianceInfluence` 34a, 44, 71, 149a.
Uses: `LECV` 141b, `VarianceInfluence_SLOT` 21b.

⟨ *C_get_TableBlock* 145a ⟩ ≡

```
double* C_get_TableBlock
(
    ⟨ R LECV Input 141b ⟩
) {
    if (VECTOR_ELT(LECV, TableBlock_SLOT) == R_NilValue)
        error("object does not contain table block slot");
    return(REAL(VECTOR_ELT(LECV, TableBlock_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_TableBlock` 34a.
Uses: `block` 26f, 27b, `LECV` 141b, `TableBlock_SLOT` 21b.

⟨ *C_get_Sumweights* 145b ⟩ ≡

```
double* C_get_Sumweights
(
    ⟨ R LECV Input 141b ⟩
) {
    if (VECTOR_ELT(LECV, Sumweights_SLOT) == R_NilValue)
        error("object does not contain sumweights slot");
    return(REAL(VECTOR_ELT(LECV, Sumweights_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_Sumweights` 34a, 46.
Uses: `LECV` 141b, `sumweights` 25e, `Sumweights_SLOT` 21b.

⟨ *C_get_Table* 145c ⟩ ≡

```
double* C_get_Table
(
    ⟨ R LECV Input 141b ⟩
) {
    if (LENGTH(LECV) <= Table_SLOT)
        error("Cannot extract table from object");
    return(REAL(VECTOR_ELT(LECV, Table_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_Table` 41a, 46.
Uses: `LECV` 141b, `Table_SLOT` 21b.

⟨ *C_get_dimTable* 145d ⟩ ≡

```
int* C_get_dimTable
(
    ⟨ R LECV Input 141b ⟩
) {
    if (LENGTH(LECV) <= Table_SLOT)
        error("Cannot extract table from object");
    return(INTEGER(getAttrib(VECTOR_ELT(LECV, Table_SLOT),
                             R_DimSymbol)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_dimTable` 46, 146a.
Uses: `LECV` 141b, `Table_SLOT` 21b.

⟨ *C_get_B* 146a ⟩ ≡

```
int C_get_B
(
    ⟨ R LECV Input 141b ⟩
) {
    if (VECTOR_ELT(LECV, TableBlock_SLOT) != R_NilValue)
        return(LENGTH(VECTOR_ELT(LECV, Sumweights_SLOT)));
    return(C_get_dimTable(LECV)[2]);
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_B` 33a, 46, 71.
Uses: `C_get_dimTable` 145d, `LECV` 141b, `Sumweights_SLOT` 21b, `TableBlock_SLOT` 21b.

⟨ *C_get_nresample* 146b ⟩ ≡

```
R_xlen_t C_get_nresample
(
    ⟨ R LECV Input 141b ⟩
) {
    int PQ = C_get_P(LECV) * C_get_Q(LECV);
    return(XLENGTH(VECTOR_ELT(LECV, PermutedLinearStatistic_SLOT)) / PQ);
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_nresample` 39a, 51, 52a, 54, 56, 71.
Uses: `C_get_P` 141c, `C_get_Q` 142a, `LECV` 141b, `PermutedLinearStatistic_SLOT` 21b.

⟨ *C_get_PermutedLinearStatistic* 146c ⟩ ≡

```
double* C_get_PermutedLinearStatistic
(
    ⟨ R LECV Input 141b ⟩
) {
    return(REAL(VECTOR_ELT(LECV, PermutedLinearStatistic_SLOT)));
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_PermutedLinearStatistic` 39a, 51, 71.
Uses: `LECV` 141b, `PermutedLinearStatistic_SLOT` 21b.

⟨ *C_get_tol* 146d ⟩ ≡

```
double C_get_tol
(
    ⟨ R LECV Input 141b ⟩
) {
    return(REAL(VECTOR_ELT(LECV, tol_SLOT))[0]);
}
◇
```

Fragment referenced in 141a.
Defines: `C_get_tol` 39a, 51, 54, 71.
Uses: `LECV` 141b, `tol_SLOT` 21b.

⟨ *Memory Input Checks* 147a ⟩ ≡

```
if (P <= 0)
    error("P is not positive");

if (Q <= 0)
    error("Q is not positive");

if (B <= 0)
    error("B is not positive");

if (varonly < 0 || varonly > 1)
    error("varonly is not 0 or 1");

if (Xfactor < 0 || Xfactor > 1)
    error("Xfactor is not 0 or 1");

if (tol <= DBL_MIN)
    error("tol is not positive");
◇
```

Fragment referenced in 148.
Uses: `B` 27a, `P` 24a, `Q` 24e.

⟨ *Memory Names* 147b ⟩ ≡

```
PROTECT(names = allocVector(STRSXP, Table_SLOT + 1));
SET_STRING_ELT(names, LinearStatistic_SLOT, mkChar("LinearStatistic"));
SET_STRING_ELT(names, Expectation_SLOT, mkChar("Expectation"));
SET_STRING_ELT(names, varonly_SLOT, mkChar("varonly"));
SET_STRING_ELT(names, Variance_SLOT, mkChar("Variance"));
SET_STRING_ELT(names, Covariance_SLOT, mkChar("Covariance"));
SET_STRING_ELT(names, ExpectationX_SLOT, mkChar("ExpectationX"));
SET_STRING_ELT(names, dim_SLOT, mkChar("dimension"));
SET_STRING_ELT(names, ExpectationInfluence_SLOT,
               mkChar("ExpectationInfluence"));
SET_STRING_ELT(names, Xfactor_SLOT, mkChar("Xfactor"));
SET_STRING_ELT(names, CovarianceInfluence_SLOT,
               mkChar("CovarianceInfluence"));
SET_STRING_ELT(names, VarianceInfluence_SLOT,
               mkChar("VarianceInfluence"));
SET_STRING_ELT(names, TableBlock_SLOT, mkChar("TableBlock"));
SET_STRING_ELT(names, Sumweights_SLOT, mkChar("Sumweights"));
SET_STRING_ELT(names, PermutedLinearStatistic_SLOT,
               mkChar("PermutedLinearStatistic"));
SET_STRING_ELT(names, StandardisedPermutedLinearStatistic_SLOT,
               mkChar("StandardisedPermutedLinearStatistic"));
SET_STRING_ELT(names, tol_SLOT, mkChar("tol"));
SET_STRING_ELT(names, Table_SLOT, mkChar("Table"));
◇
```

Fragment referenced in 148.
Uses: `CovarianceInfluence_SLOT` 21b, `Covariance_SLOT` 21b, `dim_SLOT` 21b, `ExpectationInfluence_SLOT` 21b, `ExpectationX_SLOT` 21b, `Expectation_SLOT` 21b, `LinearStatistic_SLOT` 21b, `PermutedLinearStatistic_SLOT` 21b, `StandardisedPermutedLinearStatistic_SLOT` 21b, `Sumweights_SLOT` 21b, `TableBlock_SLOT` 21b, `Table_SLOT` 21b, `tol_SLOT` 21b, `VarianceInfluence_SLOT` 21b, `Variance_SLOT` 21b, `varonly_SLOT` 21b, `Xfactor_SLOT` 21b.

⟨ *R_init_LECV* 148 ⟩ ≡

```
SEXP vo, d, names, tolerance, tmp;
int PQ;

⟨ Memory Input Checks 147a ⟩
PQ = mPQB(P, Q, 1);
⟨ Memory Names 147b ⟩

/* Table_SLOT is always last and only used in 2d case, ie omitted here */
PROTECT(ans = allocVector(VECSXP, Table_SLOT + 1));
SET_VECTOR_ELT(ans, LinearStatistic_SLOT, allocVector(REALSXP, PQ));
SET_VECTOR_ELT(ans, Expectation_SLOT, allocVector(REALSXP, PQ));
SET_VECTOR_ELT(ans, varonly_SLOT, vo = allocVector(INTSXP, 1));
INTEGER(vo)[0] = varonly;
if (varonly) {
    SET_VECTOR_ELT(ans, Variance_SLOT, tmp = allocVector(REALSXP, PQ));
} else {
    /* always return variance */
    SET_VECTOR_ELT(ans, Variance_SLOT, tmp = allocVector(REALSXP, PQ));
    SET_VECTOR_ELT(ans, Covariance_SLOT,
                   tmp = allocVector(REALSXP, PP12(PQ)));
}
SET_VECTOR_ELT(ans, ExpectationX_SLOT, allocVector(REALSXP, P));
SET_VECTOR_ELT(ans, dim_SLOT, d = allocVector(INTSXP, 2));
INTEGER(d)[0] = P;
INTEGER(d)[1] = Q;
SET_VECTOR_ELT(ans, ExpectationInfluence_SLOT,
               tmp = allocVector(REALSXP, B * Q));
for (int q = 0; q < B * Q; q++) REAL(tmp)[q] = 0.0;

/* should always _both_ be there */
SET_VECTOR_ELT(ans, VarianceInfluence_SLOT,
               tmp = allocVector(REALSXP, B * Q));
for (int q = 0; q < B * Q; q++) REAL(tmp)[q] = 0.0;

SET_VECTOR_ELT(ans, CovarianceInfluence_SLOT,
               tmp = allocVector(REALSXP, B * Q * (Q + 1) / 2));
for (int q = 0; q < B * Q * (Q + 1) / 2; q++) REAL(tmp)[q] = 0.0;

SET_VECTOR_ELT(ans, Xfactor_SLOT, allocVector(INTSXP, 1));
INTEGER(VECTOR_ELT(ans, Xfactor_SLOT))[0] = Xfactor;
SET_VECTOR_ELT(ans, TableBlock_SLOT, tmp = allocVector(REALSXP, B + 1));
for (int q = 0; q < B + 1; q++) REAL(tmp)[q] = 0.0;
SET_VECTOR_ELT(ans, Sumweights_SLOT, allocVector(REALSXP, B));
SET_VECTOR_ELT(ans, PermutedLinearStatistic_SLOT,
               allocMatrix(REALSXP, 0, 0));
SET_VECTOR_ELT(ans, StandardisedPermutedLinearStatistic_SLOT,
               allocMatrix(REALSXP, 0, 0));
SET_VECTOR_ELT(ans, tol_SLOT, tolerance = allocVector(REALSXP, 1));
REAL(tolerance)[0] = tol;
namesgets(ans, names);

⟨ Initialise Zero 149a ⟩
◇
```

Fragment referenced in 149b, 150.
Uses: `B` 27a, `CovarianceInfluence_SLOT` 21b, `Covariance_SLOT` 21b, `dim_SLOT` 21b, `ExpectationInfluence_SLOT` 21b, `ExpectationX_SLOT` 21b, `Expectation_SLOT` 21b, `LinearStatistic_SLOT` 21b, `mPQB` 132a, `P` 24a, `PermutedLinearStatistic_SLOT` 21b, `PP12` 131b, `Q` 24e, `StandardisedPermutedLinearStatistic_SLOT` 21b, `Sumweights_SLOT` 21b, `TableBlock_SLOT` 21b, `Table_SLOT` 21b, `tol_SLOT` 21b, `VarianceInfluence_SLOT` 21b, `Variance_SLOT` 21b, `varonly_SLOT` 21b, `Xfactor_SLOT` 21b.

⟨ *Initialise Zero* 149a ⟩ ≡

```
/* set inital zeros */
for (int p = 0; p < PQ; p++) {
    C_get_LinearStatistic(ans)[p] = 0.0;
    C_get_Expectation(ans)[p] = 0.0;
    if (varonly)
        C_get_Variance(ans)[p] = 0.0;
}
if (!varonly) {
    for (int p = 0; p < PP12(PQ); p++)
        C_get_Covariance(ans)[p] = 0.0;
}
for (int q = 0; q < Q; q++) {
    C_get_ExpectationInfluence(ans)[q] = 0.0;
    C_get_VarianceInfluence(ans)[q] = 0.0;
}
for (int q = 0; q < Q * (Q + 1) / 2; q++)
    C_get_CovarianceInfluence(ans)[q] = 0.0;
◇
```

Fragment referenced in 148.
Uses: `C_get_Covariance` 143c, `C_get_CovarianceInfluence` 144c, `C_get_Expectation` 143a, `C_get_ExpectationInfluence` 144b, `C_get_LinearStatistic` 142d, `C_get_Variance` 143b, `C_get_VarianceInfluence` 144d, `PP12` 131b, `Q` 24e.

⟨ *RC_init_LECV_1d* 149b ⟩ ≡

```
SEXP RC_init_LECV_1d
(
    ⟨ C integer P Input 24a ⟩,
    ⟨ C integer Q Input 24e ⟩,
    int varonly,
    ⟨ C integer B Input 27a ⟩,
    int Xfactor,
    double tol
) {
    SEXP ans;

    ⟨ R_init_LECV 148 ⟩

    SET_VECTOR_ELT(ans, TableBlock_SLOT,
                   allocVector(REALSXP, B + 1));

    SET_VECTOR_ELT(ans, Sumweights_SLOT,
                   allocVector(REALSXP, B));

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

Fragment referenced in 141a.
Defines: `RC_init_LECV_1d` 31a.
Uses: `B` 27a, `Sumweights_SLOT` 21b, `TableBlock_SLOT` 21b.

⟨ *RC_init_LECV_2d* 150 ⟩ ≡

```
SEXP RC_init_LECV_2d
(
    ⟨ C integer P Input 24a ⟩,
    ⟨ C integer Q Input 24e ⟩,
    int varonly,
    int Lx,
    int Ly,
    ⟨ C integer B Input 27a ⟩,
    int Xfactor,
    double tol
) {
    SEXP ans, tabdim, tab;

    if (Lx <= 0)
        error("Lx is not positive");

    if (Ly <= 0)
        error("Ly is not positive");

    ⟨ R_init_LECV 148 ⟩

    PROTECT(tabdim = allocVector(INTSXP, 3));
    INTEGER(tabdim)[0] = Lx + 1;
    INTEGER(tabdim)[1] = Ly + 1;
    INTEGER(tabdim)[2] = B;
    SET_VECTOR_ELT(ans, Table_SLOT,
                   tab = allocVector(REALSXP,
                       INTEGER(tabdim)[0] *
                       INTEGER(tabdim)[1] *
                       INTEGER(tabdim)[2]));
    dimgets(tab, tabdim);

    UNPROTECT(3);
    return(ans);
}
◇
```

Fragment referenced in 141a.
Defines: `RC_init_LECV_2d` 41a.
Uses: `B` 27a, `Table_SLOT` 21b.

Chapter 4

Package Infrastructure

`"AAA.R"` 151a≡

```
⟨ R Header 154a ⟩
.onUnload <-
function(libpath)
    library.dynam.unload("libcoin", libpath)
◇
```

`"DESCRIPTION"` 151b≡

```
Package: libcoin
Title: Linear Test Statistics for Permutation Inference
Date: 2023-09-26
Version: 1.0-10
Authors@R: person("Torsten", "Hothorn", role = c("aut", "cre"),
                  email = "Torsten.Hothorn@R-project.org")
Description: Basic infrastructure for linear test statistics and permutation
  inference in the framework of Strasser and Weber (1999) <https://epub.wu.ac.at/102/>.
  This package must not be used by end-users. CRAN package 'coin' implements all
  user interfaces and is ready to be used by anyone.
Depends: R (>= 3.4.0)
Suggests: coin
Imports: stats, mvtnorm
LinkingTo: mvtnorm
NeedsCompilation: yes
License: GPL-2
◇
```

`"NAMESPACE"` 151c≡

```
useDynLib(libcoin, .registration = TRUE)

importFrom("stats", complete.cases, vcov)
importFrom("mvtnorm", GenzBretz)

export(LinStatExpCov, doTest, ctabs, lmult)

S3method(vcov, LinStatExpCov)
◇
```

Add flag `-g` to `PKG_CFLAGS` for `operf` profiling (this is not portable).

"Makevars" 152a≡

```
PKG_CFLAGS=$(C_VISIBILITY)
PKG_LIBS = $(LAPACK_LIBS) $(BLAS_LIBS) $(FLIBS)
◇
```

"libcoin-win.def" 152b≡

```
LIBRARY libcoin.dll
EXPORTS
  R_init_libcoin
◇
```

Other packages can link against **libcoin**. A small example package is contained in `libcoin/inst/C_API_example`.

"libcoin-init.c" 152c≡

```
⟨ C Header 154b ⟩
#include "libcoin.h"
#include <R_ext/Rdynload.h>
#include <R_ext/Visibility.h>

#define CALLDEF(name, n) {#name, (DL_FUNC) &name, n}
#define REGCALL(name) R_RegisterCCallable("libcoin", #name, (DL_FUNC) &name)

static const R_CallMethodDef callMethods[] = {
    CALLDEF(R_ExpectationCovarianceStatistic, 7),
    CALLDEF(R_PermutedLinearStatistic, 6),
    CALLDEF(R_StandardisePermutedLinearStatistic, 1),
    CALLDEF(R_ExpectationCovarianceStatistic_2d, 9),
    CALLDEF(R_PermutedLinearStatistic_2d, 7),
    CALLDEF(R_QuadraticTest, 5),
    CALLDEF(R_MaximumTest, 9),
    CALLDEF(R_MaximallySelectedTest, 6),
    CALLDEF(R_ExpectationInfluence, 3),
    CALLDEF(R_CovarianceInfluence, 4),
    CALLDEF(R_ExpectationX, 4),
    CALLDEF(R_CovarianceX, 5),
    CALLDEF(R_Sums, 3),
    CALLDEF(R_KronSums, 6),
    CALLDEF(R_KronSums_Permutation, 5),
    CALLDEF(R_colSums, 3),
    CALLDEF(R_OneTableSums, 3),
    CALLDEF(R_TwoTableSums, 4),
    CALLDEF(R_ThreeTableSums, 5),
    CALLDEF(R_order_subset_wrt_block, 4),
    CALLDEF(R_quadform, 3),
    CALLDEF(R_kronecker, 2),
    CALLDEF(R_MPinv_sym, 3),
    CALLDEF(R_unpack_sym, 3),
    CALLDEF(R_pack_sym, 1),
    {NULL, NULL, 0}
};
◇
```

File defined by 152c, 153.

Uses: `R_colSums` 106c, `R_CovarianceInfluence` 83a, `R_CovarianceX` 87a, `R_ExpectationCovarianceStatistic` 31a, `R_ExpectationCovarianceStatistic_2d` 41a, `R_ExpectationInfluence` 81a, `R_ExpectationX` 84c, `R_KronSums` 94b, `R_KronSums_Permutation` 102b, `R_MPinv_sym` 136b, `R_OneTableSums` 111a, `R_order_subset_wrt_block` 123c, `R_pack_sym` 140c, `R_PermutedLinearStatistic` 37, `R_PermutedLinearStatistic_2d` 48, `R_quadform` 61b, `R_Sums` 90a, `R_ThreeTableSums` 119b, `R_TwoTableSums` 115a, `R_unpack_sym` 139.

"libcoin-init.c" 153≡

```
void attribute_visible R_init_libcoin
(
    DllInfo *dll
) {
    R_registerRoutines(dll, NULL, callMethods, NULL, NULL);
    R_useDynamicSymbols(dll, FALSE);
    R_forceSymbols(dll, TRUE);
    REGCALL(R_ExpectationCovarianceStatistic);
    REGCALL(R_PermutedLinearStatistic);
    REGCALL(R_StandardisePermutedLinearStatistic);
    REGCALL(R_ExpectationCovarianceStatistic_2d);
    REGCALL(R_PermutedLinearStatistic_2d);
    REGCALL(R_QuadraticTest);
    REGCALL(R_MaximumTest);
    REGCALL(R_MaximallySelectedTest);
    REGCALL(R_ExpectationInfluence);
    REGCALL(R_CovarianceInfluence);
    REGCALL(R_ExpectationX);
    REGCALL(R_CovarianceX);
    REGCALL(R_Sums);
    REGCALL(R_KronSums);
    REGCALL(R_KronSums_Permutation);
    REGCALL(R_colSums);
    REGCALL(R_OneTableSums);
    REGCALL(R_TwoTableSums);
    REGCALL(R_ThreeTableSums);
    REGCALL(R_order_subset_wrt_block);
    REGCALL(R_quadform);
    REGCALL(R_kronecker);
    REGCALL(R_MPinv_sym);
    REGCALL(R_unpack_sym);
    REGCALL(R_pack_sym);
}
◇
```

File defined by 152c, 153.
Uses: `R_colSums` 106c, `R_CovarianceInfluence` 83a, `R_CovarianceX` 87a, `R_ExpectationCovarianceStatistic` 31a, `R_ExpectationCovarianceStatistic_2d` 41a, `R_ExpectationInfluence` 81a, `R_ExpectationX` 84c, `R_KronSums` 94b, `R_KronSums_Permutation` 102b, `R_MPinv_sym` 136b, `R_OneTableSums` 111a, `R_order_subset_wrt_block` 123c, `R_pack_sym` 140c, `R_PermutedLinearStatistic` 37, `R_PermutedLinearStatistic_2d` 48, `R_quadform` 61b, `R_Sums` 90a, `R_ThreeTableSums` 119b, `R_TwoTableSums` 115a, `R_unpack_sym` 139.

⟨ *R Header* 154a ⟩ ≡

```
###    Copyright (C) 2017-2023 Torsten Hothorn
###
###    This file is part of the 'libcoin' R add-on package.
###
###    'libcoin' 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.
###
###    'libcoin' 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 'libcoin'.  If not, see <http://www.gnu.org/licenses/>.
###
###
###    DO NOT EDIT THIS FILE
###
###    Edit 'libcoin.w' and run 'nuweb -r libcoin.w'
◇
```

Fragment referenced in 3a, 15b, 151a.

⟨ *C Header* 154b ⟩ ≡

```
/*
    Copyright (C) 2017-2023 Torsten Hothorn

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

    'libcoin' 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.

    'libcoin' 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 'libcoin'.  If not, see <http://www.gnu.org/licenses/>.


    DO NOT EDIT THIS FILE

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

Fragment referenced in 20a, 22ac, 30d, 152c.

Index

Files

Fragments

Identifiers

Bibliography

Helmut Strasser and Christian Weber. On the asymptotic theory of permutation statistics. *Mathematical Methods of Statistics*, 8:220–250, 1999. Preprint available from `https://epub.wu.ac.at/102`. 1