

Package: Matrix (via r-universe)

September 24, 2026

Version 1.8-0

VersionNote do also bump src/version.h, inst/include/Matrix/version.h

Date 2026-07-20

Priority recommended

Title Sparse and Dense Matrix Classes and Methods

Description A rich hierarchy of sparse and dense matrix classes, including general, symmetric, triangular, and diagonal matrices with numeric, logical, or pattern entries. Efficient methods for operating on such matrices, often wrapping the 'BLAS', 'LAPACK', and 'SuiteSparse' libraries.

License GPL (>= 2) | file LICENCE

URL <https://codeberg.org/jaganmn/Matrix>

BugReports <https://codeberg.org/jaganmn/Matrix/issues>

Contact Matrix-authors@R-project.org

Depends R (>= 4.5), methods

Imports grDevices, graphics, grid, lattice, stats, utils

Suggests MASS, datasets, sfsmisc, tools

Enhances MatrixModels, SparseM, graph

LazyData no

LazyDataNote not possible, since we use data/*.R and our S4 classes

BuildResaveData no

Encoding UTF-8

Repository <https://jaganmn.r-universe.dev>

Date/Publication 2026-07-24 18:44:51 UTC

RemoteUrl <https://codeberg.org/jaganmn/matrix>

RemoteRef HEAD

RemoteSha f0b626c6ad59226abc6818c8284f8b0537108f31

Contents

Matrix-package	5
abIndex-class	5
abIseq	7
all.equal-methods	8
asUniqueT	8
band-methods	10
bandSparse	11
bdiag	13
bind-methods	15
boolmatmult-methods	17
BunchKaufman-methods	18
CAex	20
chol-methods	21
chol2inv-methods	24
Cholesky-methods	25
coerce-methods	31
coerce-methods-graph	31
coerce-methods-SparseM	32
colSums-methods	33
Complex-methods	35
condest	35
cov2cor-methods	37
CsparseMatrix-class	37
ct-methods	40
denseBunchKaufman-class	43
denseCholesky-class	45
denseLU-class	48
denseMatrix-class	50
denseSchur-class	51
determinant-methods	52
diag-methods	53
Diagonal	54
diagonalMatrix-class	56
diagU2N	57
diff-methods	59
dim-methods	60
dimnames-methods	61
dimScale	63
dMatrix-class	64
dmperm	65
drop-methods	67
drop0	68
expand-methods	69
expm-methods	72
externalFormats	73
facmul-methods	75

fastMisc	76
forceCanonical-methods	80
forceDiagonal-methods	81
forceSymmetric-methods	81
forceTriangular-methods	83
formatSparseM	83
generalMatrix-class	85
head-methods	86
Hilbert	87
image-methods	87
iMatrix-class	90
index-class	91
indexMatrix-class	91
initialize-methods	94
invertPerm	94
is.na-methods	96
is.null.DN	97
isCanonical-methods	98
isDiagonal-methods	100
isSymmetric-methods	101
isTriangular-methods	102
KhatriRao	103
KNex	105
kronecker-methods	106
length-methods	107
lMatrix-class	109
lu-methods	110
mat2triplet	112
Math-methods	114
Math2-methods	114
matmult-methods	115
Matrix	117
Matrix-class	119
MatrixClass	121
MatrixFactorization-class	122
mean-methods	123
nearPD	124
nMatrix-class	127
nnzero-methods	129
norm-methods	130
not-methods	132
Ops-methods	133
pack-methods	134
packedMatrix-class	135
posdefMatrix-class	139
printSpMatrix	140
qr-methods	143
rankMatrix	146

rcond-methods	149
rep-methods	151
rep2abI	153
rleDiff-class	153
rsparsematrix	154
RsparseMatrix-class	156
Schur-methods	158
show-methods	159
skewpart-methods	160
solve-methods	161
sort-methods	164
sparse.model.matrix	165
sparseCholesky-class	167
sparseLU-class	171
sparseMatrix	173
sparseMatrix-class	177
sparseQR-class	178
sparseVector	182
sparseVector-class	183
spMatrix	186
subassign-methods	187
subset-methods	189
Summary-methods	191
symmetricMatrix-class	192
symmpart-methods	194
t-methods	196
toeplitz-methods	198
triangularMatrix-class	199
TsparseMatrix-class	200
unpackedMatrix-class	204
update-methods	207
updown-methods	208
USCounties	210
which-methods	211
wrld_1deg	212
xtfrm-methods	213
zapsmall-methods	214
zMatrix-class	215

Matrix-package	R Package Matrix
----------------	-------------------------

Description

TODO.

Usage

```
Matrix.Version()
```

Examples

```
(Mv <- Matrix.Version())
stopifnot(identical(Mv[["package"]], utils::packageVersion("Matrix")))
```

abIndex-class	Class "abIndex" of Abstract Index Vectors
---------------	---

Description

The "abIndex" [class](#), short for "Abstract Index Vector", is used for dealing with large index vectors more efficiently, than using integer (or [numeric](#)) vectors of the kind `2:1000000` or `c(0:1e5, 1000:1e6)`.

Note that the current implementation details are subject to change, and if you consider working with these classes, please contact the package maintainers (`packageDescription("Matrix")$Maintainer`).

Objects from the Class

Objects can be created by calls of the form `new("abIndex", ...)`, but more easily and typically either by `as(x, "abIndex")` where `x` is an integer (valued) vector, or directly by `abIseq()` and combination `c(...)` of such.

Slots

kind: a [character](#) string, one of `("int32", "double", "rleDiff")`, denoting the internal structure of the abIndex object.

x: Object of class "numLike"; is used (i.e., not of length 0) only iff the object is *not* compressed, i.e., currently exactly when `kind != "rleDiff"`.

rleD: object of class "[rleDiff](#)", used for compression via [rle](#).

Methods

as.numeric, as.integer, as.vector signature(x = "abIndex"): ...

[signature(x = "abIndex", i = "index", j = "ANY", drop = "ANY"): ...

coerce signature(from = "numeric", to = "abIndex"): ...

coerce signature(from = "abIndex", to = "numeric"): ...

coerce signature(from = "abIndex", to = "integer"): ...

length signature(x = "abIndex"): ...

Ops signature(e1 = "numeric", e2 = "abIndex"): These and the following arithmetic and logic operations are **not yet implemented**; see [Ops](#) for a list of these (S4) group methods.

Ops signature(e1 = "abIndex", e2 = "abIndex"): ...

Ops signature(e1 = "abIndex", e2 = "numeric"): ...

Summary signature(x = "abIndex"): ...

show ("abIndex"): simple [show](#) method, building on show(<rleDiff>).

is.na ("abIndex"): works analogously to regular vectors.

is.finite, is.infinite ("abIndex"): ditto.

Note

This is currently experimental and not yet used for our own code. Please contact us (packageDescription("Matrix")\$Maintainer) if you plan to make use of this class.

Partly builds on ideas and code from Jens Oehlschlaegel, as implemented (around 2008, in the GPL'ed part of) package **ff**.

See Also

[rle](#) (**base**) which is used here; [numeric](#)

Examples

```
showClass("abIndex")
ii <- c(-3:40, 20:70)
str(ai <- as(ii, "abIndex"))# note
ai # -> show() method

stopifnot(identical(-3:20,
                    as(abIseq1(-3,20), "vector")))
```

abIseq

*Sequence Generation of "abIndex", Abstract Index Vectors***Description**

Generation of abstract index vectors, i.e., objects of class "[abIndex](#)".

`abIseq()` is designed to work entirely like [seq](#), but producing "[abIndex](#)" vectors.

`abIseq1()` is its basic building block, where `abIseq1(n,m)` corresponds to `n:m`.

`c(x, ...)` will return an "[abIndex](#)" vector, when `x` is one.

Usage

```
abIseq1(from = 1, to = 1)
abIseq (from = 1, to = 1, by = ((to - from)/(length.out - 1)),
       length.out = NULL, along.with = NULL)

## S3 method for class 'abIndex'
c(...)
```

Arguments

<code>from, to</code>	the starting and (maximal) end value of the sequence.
<code>by</code>	number: increment of the sequence.
<code>length.out</code>	desired length of the sequence. A non-negative number, which for <code>seq</code> and <code>seq.int</code> will be rounded up if fractional.
<code>along.with</code>	take the length from the length of this argument.
<code>...</code>	in general an arbitrary number of R objects; here, when the first is an " abIndex " vector, these arguments will be concatenated to a new " abIndex " object.

Value

An abstract index vector, i.e., object of class "[abIndex](#)".

See Also

the class [abIndex](#) documentation; [rep2abI\(\)](#) for another constructor; [rle](#) (**base**).

Examples

```
stopifnot(identical(-3:20,
                    as(abIseq1(-3,20), "vector")))

try( ## (arithmetic) not yet implemented
abIseq(1, 50, by = 3)
)
```

all.equal-methods	<i>Matrix Package Methods for Function all.equal()</i>
-------------------	--

Description

Methods for function `all.equal()` (from R package **base**) are defined for all **Matrix** classes.

Methods

**target = "Matrix", current = "Matrix" **

**target = "ANY", current = "Matrix" **

target = "Matrix", current = "ANY" these three methods are simply using `all.equal.numeric` directly and work via `as.vector()`.

There are more methods, notably also for "`sparseVector`"s, see `showMethods("all.equal")`.

Examples

```
showMethods("all.equal")

(A <- spMatrix(3,3, i= c(1:3,2:1), j=c(3:1,1:2), x = 1:5))
ex <- expand(lu. <- lu(A))
stopifnot( all.equal(as(A[lu.@p + 1L, lu.@q + 1L], "CsparseMatrix"),
                    lu.@L %**% lu.@U),
          with(ex, all.equal(as(P %**% A %**% t(Q), "CsparseMatrix"),
                              L %**% U)),
          with(ex, all.equal(as(A, "CsparseMatrix"),
                              t(P) %**% L %**% U %**% Q)))
```

asUniqueT	<i>Standardize a Sparse Matrix in Triplet Format</i>
-----------	--

Description

Detect or standardize a **TsparseMatrix** with unsorted or duplicated (i, j) pairs.

Usage

```
anyDuplicatedT(x, ...)
isUniqueT(x, byrow = FALSE, isT = is(x, "TsparseMatrix"))
asUniqueT(x, byrow = FALSE, isT = is(x, "TsparseMatrix"))
aggregateT(x)
```

Arguments

x	an R object. anyDuplicatedT and aggregateT require x inheriting from TsparseMatrix . asUniqueT requires x inheriting from Matrix and coerces x to TsparseMatrix if necessary.
...	optional arguments passed to the default method for generic function anyDuplicated .
byrow	a logical indicating if x should be sorted by row then by column.
isT	a logical indicating if x inherits from virtual class TsparseMatrix .

Value

anyDuplicatedT(x) returns the index of the first duplicated (i, j) pair in x (0 if there are no duplicated pairs).

isUniqueT(x) returns TRUE if x is a [TsparseMatrix](#) with sorted, nonduplicated (i, j) pairs and FALSE otherwise.

asUniqueT(x) returns the unique [TsparseMatrix](#) representation of x with sorted, nonduplicated (i, j) pairs. Values corresponding to identical (i, j) pairs are aggregated by addition, where in the logical case “addition” refers to logical OR.

aggregateT(x) aggregates without sorting.

See Also

Virtual class [TsparseMatrix](#).

Examples

```
example("dgTMatrix-class", echo=FALSE)
## -> 'T2' with (i,j,x) slots of length 5 each
T2u <- asUniqueT(T2)
stopifnot(## They "are" the same (and print the same):
  all.equal(T2, T2u, tol=0),
  ## but not internally:
  anyDuplicatedT(T2) == 2,
  anyDuplicatedT(T2u) == 0,
  length(T2 @x) == 5,
  length(T2u@x) == 3)

isUniqueT(T2 ) # FALSE
isUniqueT(T2u) # TRUE

T3 <- T2u
T3[1, c(1,3)] <- 10; T3[2, c(1,5)] <- 20
T3u <- asUniqueT(T3)
str(T3u) # sorted in 'j', and within j, sorted in i
stopifnot(isUniqueT(T3u))

## Logical l.TMatrix and n.TMatrix :
(L2 <- T2 > 0)
validObject(L2u <- asUniqueT(L2))
(N2 <- as(L2, "nMatrix"))
```

```

validObject(N2u <- asUniqueT(N2))
stopifnot(N2u@i == L2u@i, L2u@i == T2u@i, N2@i == L2@i, L2@i == T2@i,
          N2u@j == L2u@j, L2u@j == T2u@j, N2@j == L2@j, L2@j == T2@j)
# now with a nasty NA [partly failed in Matrix 1.1-5]:
L.0N <- L.1N <- L2
L.0N@x[1:2] <- c(FALSE, NA)
L.1N@x[1:2] <- c(TRUE, NA)
validObject(L.0N)
validObject(L.1N)
(m.0N <- as.matrix(L.0N))
(m.1N <- as.matrix(L.1N))
stopifnot(identical(10L, which(is.na(m.0N))), !anyNA(m.1N))
symnum(m.0N)
symnum(m.1N)

```

band-methods

Extract bands of a matrix

Description

Return the matrix obtained by setting to zero elements below a diagonal (`triu`), above a diagonal (`tril`), or outside of a general band (`band`).

Usage

```

band(x, k1, k2, ...)
triu(x, k = 0L, ...)
tril(x, k = 0L, ...)

```

Arguments

<code>x</code>	a matrix-like object
<code>k, k1, k2</code>	integers specifying the diagonals that are not set to zero, $k1 \leq k2$. These are interpreted relative to the main diagonal, which is $k = 0$. Positive and negative values of k indicate diagonals above and below the main diagonal, respectively.
<code>...</code>	optional arguments passed to methods, currently unused by package Matrix .

Details

`triu(x, k)` is equivalent to `band(x, k, dim(x)[2])`. Similarly, `tril(x, k)` is equivalent to `band(x, -dim(x)[1], k)`.

Value

An object of a suitable matrix class, inheriting from `triangularMatrix` where appropriate. It inherits from `sparseMatrix` if and only if `x` does.

Methods

x = "CsparseMatrix" method for compressed, sparse, column-oriented matrices.

x = "RsparseMatrix" method for compressed, sparse, row-oriented matrices.

x = "TsparseMatrix" method for sparse matrices in triplet format.

x = "diagonalMatrix" method for diagonal matrices.

x = "denseMatrix" method for dense matrices in packed or unpacked format.

x = "matrix" method for traditional matrices of implicit class `matrix`.

See Also

`bandSparse` for the *construction* of a banded sparse matrix directly from its non-zero diagonals.

Examples

```
## A random sparse matrix :
set.seed(7)
m <- matrix(0, 5, 5)
m[sample(length(m), size = 14)] <- rep(1:9, length=14)
(mm <- as(m, "CsparseMatrix"))

tril(mm)      # lower triangle
tril(mm, -1)  # strict lower triangle
triu(mm, 1)   # strict upper triangle
band(mm, -1, 2) # general band
(m5 <- Matrix(rnorm(25), ncol = 5))
tril(m5)      # lower triangle
tril(m5, -1)  # strict lower triangle
triu(m5, 1)   # strict upper triangle
band(m5, -1, 2) # general band
(m65 <- Matrix(rnorm(30), ncol = 5)) # not square
triu(m65)     # result not "dtrMatrix" unless square
(sm5 <- crossprod(m65)) # symmetric
  band(sm5, -1, 1) # "dsyMatrix": symmetric band preserves symmetry property
as(band(sm5, -1, 1), "sparseMatrix") # often preferable
(sm <- round(crossprod(triu(mm/2)))) # sparse symmetric ("dsC*")
band(sm, -1,1) # remains "dsC", *however*
band(sm, -2,1) # -> "dgC"
```

bandSparse

Construct Sparse Banded Matrix from (Sup-/Super-) Diagonals

Description

Construct a sparse banded matrix by specifying its non-zero sup- and super-diagonals.

Usage

```
bandSparse(n, m = n, k, diagonals, symmetric = FALSE,
           repr = "C", giveCsparse = (repr == "C"),
           keepInteger = Matrix.keepInteger())
```

Arguments

n, m	the matrix dimension $(n, m) = (nrow, ncol)$.
k	integer vector of “diagonal numbers”, with identical meaning as in <code>band(*, k)</code> , i.e., relative to the main diagonal, which is $k=0$.
diagonals	optional list of sub-/super- diagonals; if missing, the result will be a pattern matrix, i.e., inheriting from class <code>nMatrix</code> . diagonals can also be $n' \times d$ matrix, where $d \leftarrow \text{length}(k)$ and $n' \geq \min(n, m)$. In that case, the sub-/super- diagonals are taken from the columns of diagonals, where only the first several rows will be used (typically) for off-diagonals.
symmetric	logical; if true the result will be symmetric (inheriting from class <code>symmetricMatrix</code>) and only the upper or lower triangle must be specified (via <code>k</code> and <code>diagonals</code>).
repr	<code>character</code> string, one of “C”, “T”, or “R”, specifying the sparse <i>representation</i> to be used for the result, i.e., one from the super classes <code>CsparseMatrix</code> , <code>TsparseMatrix</code> , or <code>RsparseMatrix</code> .
giveCsparse	(deprecated, replaced with repr) : logical indicating if the result should be a <code>CsparseMatrix</code> or a <code>TsparseMatrix</code> , where the default was TRUE, and now is determined from <code>repr</code> ; very often Csparse matrices are more efficient subsequently, but not always.
keepInteger	an optional logical argument introduced in Matrix version 1.8-0. If FALSE, then <code>integer</code> diagonals are treated as <code>double</code> . This behaviour is compatible with earlier versions of Matrix , but it is not so desirable now that matrices with integer data are representable as objects inheriting from virtual class <code>iMatrix</code> . The default value will change to TRUE in the near future; currently, it depends on the value of the environment variable <code>R_MATRIX_KEEP_INTEGER</code> at load time (false if unset), but that can be overridden by <code>options(Matrix.keepInteger=)</code> .

Value

a sparse matrix (of class `CsparseMatrix`) of dimension $n \times m$ with diagonal “bands” as specified.

See Also

`band`, for *extraction* of matrix bands; `bdiag`, `diag`, `sparseMatrix`, `Matrix`.

Examples

```
diags <- list(1:30, 10*(1:20), 100*(1:20))
s1 <- bandSparse(13, k = -c(0:2, 6), diag = c(diags, diags[2]), symm=TRUE)
s1
s2 <- bandSparse(13, k = c(0:2, 6), diag = c(diags, diags[2]), symm=TRUE)
stopifnot(identical(s1, t(s2)), is(s1, "dsCMatrix"))
```

```
## a pattern Matrix of *full* (sub-)diagonals:
bk <- c(0:4, 7,9)
(s3 <- bandSparse(30, k = bk, symm = TRUE))

## If you want a pattern matrix, but with "sparse"-diagonals,
## you currently need to go via logical sparse:
lLis <- lapply(list(rpois(20, 2), rpois(20, 1), rpois(20, 3))[c(1:3, 2:3, 3:2)],
               as.logical)
(s4 <- bandSparse(20, k = bk, symm = TRUE, diag = lLis))
(s4. <- as(drop0(s4), "nsparseMatrix"))

n <- 1e4
bk <- c(0:5, 7,11)
bMat <- matrix(1:8, n, 8, byrow=TRUE)
bLis <- as.data.frame(bMat)
B <- bandSparse(n, k = bk, diag = bLis)
Bs <- bandSparse(n, k = bk, diag = bLis, symmetric=TRUE)
B [1:15, 1:30]
Bs[1:15, 1:30]
## can use a list *or* a matrix for specifying the diagonals:
stopifnot(identical(B, bandSparse(n, k = bk, diag = bMat)),
           identical(Bs, bandSparse(n, k = bk, diag = bMat, symmetric=TRUE))
           , inherits(B, "dtCMatrix") # triangular!
)
```

bdiag

Construct a Block Diagonal Matrix

Description

Build a block diagonal matrix given several building block matrices.

Usage

```
bdiag(..., keepInteger = Matrix.keepInteger())
.bdiag(lst, keepInteger = Matrix.keepInteger())
```

Arguments

...	individual matrices or a list of matrices.
lst	non-empty list of matrices.
keepInteger	an optional logical argument introduced in Matrix version 1.8-0. If FALSE, then integer matrices are treated as double . This behaviour is compatible with earlier versions of Matrix , but it is not so desirable now that matrices with integer data are representable as objects inheriting from virtual class iMatrix . The default value will change to TRUE in the near future; currently, it depends on the value of the environment variable R_MATRIX_KEEP_INTEGER at load time (false if unset), but that can be overridden by options (Matrix.keepInteger=).

Details

For non-trivial argument list, `bdiag()` calls `.bdiag()`. The latter maybe useful to programmers.

Value

A *sparse* matrix obtained by combining the arguments into a block diagonal matrix.

The value of `bdiag()` inherits from class `CsparseMatrix`, whereas `.bdiag()` returns a `TsparseMatrix`.

Note

This function has been written and is efficient for the case of relatively few block matrices which are typically sparse themselves.

It is currently *inefficient* for the case of many small dense block matrices. For the case of *many* dense $k \times k$ matrices, the `bdiag_m()` function in the ‘Examples’ is an order of magnitude faster.

Author(s)

Martin Maechler, built on a version posted by Berton Gunter to R-help; earlier versions have been posted by other authors, notably Scott Chasalow to S-news. Doug Bates’s faster implementation builds on `TsparseMatrix` objects.

See Also

`Diagonal` for constructing matrices of class `diagonalMatrix`, or `kronecker` which also works for “Matrix” inheriting matrices.

`bandSparse` constructs a *banded* sparse matrix from its non-zero sub-/super - diagonals.

Note that other CRAN R packages have own versions of `bdiag()` which return traditional matrices.

Examples

```
bdiag(matrix(1:4, 2), diag(3))
## combine "Matrix" class and traditional matrices:
bdiag(Diagonal(2), matrix(1:3, 3,4), diag(3:2))

mlist <- list(1, 2:3, diag(x=5:3), 27, cbind(1,3:6), 100:101)
bdiag(mlist)
stopifnot(identical(bdiag(mlist),
                    bdiag(lapply(mlist, as.matrix))))

m1 <- c(as(matrix((1:24)%% 11 == 0, 6,4),"nMatrix"),
        rep(list(Diagonal(2, x=TRUE)), 3))
m1n <- c(m1, Diagonal(x = 1:3, keepInteger = TRUE))
sapply(m1n, class) # "ngCMatrix" "ldi*" .. "idiMatrix"
stopifnot(is(bdiag(m1), "lsparseMatrix"),
          is(bdiag(m1n, keepInteger = TRUE),"isparseMatrix") )

## random (diagonal-)block-triangular matrices:
rblockTri <- function(nb, max.ni, lambda = 3) {
  .bdiag(replicate(nb, {
```

```

        n <- sample.int(max.ni, 1)
        tril(Matrix(rpois(n * n, lambda = lambda), n, n)) )))
    }

    (T4 <- rblockTri(4, 10, lambda = 1))
    image(T1 <- rblockTri(12, 20))

##' Fast version of Matrix :: .bdiag() -- for the case of *many* (k x k) matrices:
##' @param lmat list(<mat1>, <mat2>, ....., <mat_N>) where each mat_j is a k x k 'matrix'
##' @return a sparse (N*k x N*k) matrix of class \code{"\linkS4class{dgCMatrix}"}.
bdiag_m <- function(lmat) {
  ## Copyright (C) 2016 Martin Maechler, ETH Zurich
  if(!length(lmat)) return(new("dgCMatrix"))
  stopifnot(is.list(lmat), is.matrix(lmat[[1]]),
            (k <- (d <- dim(lmat[[1]]))[1]) == d[2], # k x k
            all(vapply(lmat, dim, integer(2)) == k)) # all of them
  N <- length(lmat)
  if(N * k > .Machine$integer.max)
    stop("resulting matrix too large; would be M x M, with M=", N*k)
  M <- as.integer(N * k)
  ## result: an M x M matrix
  new("dgCMatrix", Dim = c(M,M),
      ## 'i :' maybe there's a faster way (w/o matrix indexing), but elegant?
      i = as.vector(matrix(0L:(M-1L), nrow=k)[, rep(seq_len(N), each=k)]),
      p = k * 0L:M,
      x = as.double(unlist(lmat, recursive=FALSE, use.names=FALSE)))
}

l12 <- replicate(12, matrix(rpois(16, lambda = 6.4), 4, 4),
                      simplify=FALSE)
dim(T12 <- bdiag_m(l12))# 48 x 48
T12[1:20, 1:20]
```

bind-methods

'cbind()' and 'rbind()' recursively built on cbind2/rbind2

Description

The base functions `cbind` and `rbind` are defined for an arbitrary number of arguments and hence have the first formal argument Now, when S4 objects are found among the arguments, base `cbind()` and `rbind()` internally “dispatch” *recursively*, calling `cbind2` or `rbind2` respectively, where these have methods defined and so should dispatch appropriately.

`cbind2()` and `rbind2()` are from the **methods** package, i.e., standard R, and have been provided for binding together *two* matrices, where in **Matrix**, we have defined methods for these and the 'Matrix' matrices.

Usage

```
## cbind(..., deparse.level = 1)
## rbind(..., deparse.level = 1)

## S4 method for signature 'Matrix,Matrix'
cbind2(x, y, ...)
## S4 method for signature 'Matrix,Matrix'
rbind2(x, y, ...)
```

Arguments

<code>...</code>	for <code>[cr]bind</code> , vector- or matrix-like R objects to be bound together; for <code>[cr]bind2</code> , further arguments passed to or from methods; see cbind and cbind2 .
<code>deparse.level</code>	integer controlling the construction of labels in the case of non-matrix-like arguments; see cbind .
<code>x, y</code>	vector- or matrix-like R objects to be bound together.

Value

typically a ‘matrix-like’ object of a similar [class](#) as the first argument in ...

Note that sometimes by default, the result is a [sparseMatrix](#) if one of the arguments is (even in the case where this is not efficient). In other cases, the result is chosen to be sparse when there are more zero entries than non-zero ones (as the default sparse in [Matrix\(\)](#)).

Author(s)

Martin Maechler

See Also

[cbind](#), [cbind2](#).

Our class definition help pages mentioning `cbind2()` and `rbind2()` methods: ["denseMatrix"](#), ["diagonalMatrix"](#), ["indMatrix"](#).

Examples

```
(a <- matrix(c(2:1,1:2), 2,2))

(M1 <- cbind(0, rbind(a, 7))) # a traditional matrix

D <- Diagonal(2)
(M2 <- cbind(4, a, D, -1, D, 0)) # a sparse Matrix

stopifnot(validObject(M2), inherits(M2, "sparseMatrix"),
           dim(M2) == c(2,9))
```

Description

For boolean or “pattern” matrices, i.e., R objects of class `nMatrix`, it is natural to allow matrix products using boolean instead of numerical arithmetic.

In package **Matrix**, we use the binary operator `%%` (aka “infix”) function) for this and provide methods for all our matrices and the traditional R matrices (see [matrix](#)).

Value

a pattern matrix, i.e., inheriting from “`nMatrix`”, or an “`ldiMatrix`” in case of a diagonal matrix.

Methods

We provide methods for both the “traditional” (R base) matrices and numeric vectors and conceptually all matrices and `sparseVector`s in package **Matrix**.

```
signature(x = "ANY", y = "ANY")
signature(x = "ANY", y = "Matrix")
signature(x = "Matrix", y = "ANY")
signature(x = "nMatrix", y = "nMatrix")
signature(x = "nMatrix", y = "nsparseMatrix")
signature(x = "nsparseMatrix", y = "nMatrix")
signature(x = "nsparseMatrix", y = "nsparseMatrix")
signature(x = "sparseVector", y = "sparseVector")
```

Note

These boolean arithmetic matrix products had been newly introduced for **Matrix** 1.2.0 (March 2015). Its implementation has still not been tested extensively.

Originally, it was left unspecified how non-structural zeros, i.e., 0 ’s as part of the $M \times$ slot should be treated for numeric (“`dMatrix`”) and logical (“`lMatrix`”) sparse matrices. We now specify that boolean matrix products should behave as if applied to `drop0(M)`, i.e., as if dropping such zeros from the matrix before using it.

Equivalently, for all matrices M , boolean arithmetic should work as if applied to $M \neq 0$ (or $M \neq \text{FALSE}$).

The current implementation ends up coercing both x and y to (virtual) class `nsparseMatrix` which may be quite inefficient for dense matrices. A future implementation may well return a matrix with **different** class, but the “same” content, i.e., the same matrix entries m_{ij} .

See Also

`%*%`, `crossprod()`, or `tcrossprod()`, for (regular) matrix product methods.

Examples

```

set.seed(7)
L <- Matrix(rnorm(20) > 1, 4,5)
(N <- as(L, "nMatrix"))
L. <- L; L.[1:2,1] <- TRUE; L.@x[1:2] <- FALSE; L. # has "zeros" to drop0()
D <- Matrix(round(rnorm(30)), 5,6) # -> values in -1:1 (for this seed)
L %%% D
stopifnot(identical(L %%% D, N %%% D),
           all(L %%% D == as((L %%% abs(D)) > 0, "sparseMatrix")))

## cross products , possibly with boolArith = TRUE :
crossprod(N) # -> sparse patter'n' (TRUE/FALSE : boolean arithmetic)
crossprod(N +0) # -> numeric Matrix (with same "pattern")
stopifnot(all(crossprod(N) == t(N) %%% N),
           identical(crossprod(N), crossprod(N +0, boolArith=TRUE)),
           identical(crossprod(L), crossprod(N , boolArith=FALSE)))
crossprod(D, boolArith = TRUE) # pattern: "nsCMatrix"
crossprod(L, boolArith = TRUE) # ditto
crossprod(L, boolArith = FALSE) # numeric: "dsCMatrix"

```

BunchKaufman-methods *Methods for Bunch-Kaufman Factorization*

Description

Computes the Bunch-Kaufman factorization of an $n \times n$ real, symmetric matrix A , which has the general form

$$A = U D_U U' = L D_L L'$$

where D_U and D_L are symmetric, block diagonal matrices composed of b_U and b_L 1×1 or 2×2 diagonal blocks; $U = \prod_{k=1}^{b_U} P_k U_k$ is the product of b_U row-permuted unit upper triangular matrices, each having nonzero entries above the diagonal in 1 or 2 columns; and $L = \prod_{k=1}^{b_L} P_k L_k$ is the product of b_L row-permuted unit lower triangular matrices, each having nonzero entries below the diagonal in 1 or 2 columns.

Methods are built on LAPACK routines dsytrf and dsptrf.

Usage

```

BunchKaufman(x, ...)
## S4 method for signature 'denseMatrix'
BunchKaufman(x, warnSing = TRUE, uplo = "U", trans = "C", ...)
## S4 method for signature 'matrix'
BunchKaufman(x, ...)

```

Arguments

x	a finite symmetric matrix or Matrix to be factorized. If x is square but not symmetric, then it will be <i>treated</i> as symmetric; see uplo.
warnSing	a logical indicating if a warning should be signaled for singular x.
uplo	a string, either "U" or "L", indicating which triangle of x should be used to compute the factorization.
trans	.
...	further arguments passed to or from methods.

Value

An object representing the factorization, inheriting from virtual class BunchKaufmanFactorization. The specific class is BunchKaufman unless x inherits from virtual class [packedMatrix](#), in which case it is pBunchKaufman.

References

The LAPACK source code, including documentation; see <https://netlib.org/lapack/double/dsytrf.f> and <https://netlib.org/lapack/double/dsptrf.f>.

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

Classes BunchKaufman and pBunchKaufman and their methods.

Classes [dsyMatrix](#) and [dspMatrix](#).

Generic functions [expand1](#) and [expand2](#), for constructing matrix factors from the result.

Generic functions [Cholesky](#), [Schur](#), [lu](#), and [qr](#), for computing other factorizations.

Examples

```
showMethods("BunchKaufman", inherited = FALSE)
set.seed(0)

data(CAex, package = "Matrix")
class(CAex) # dgCMatrix
isSymmetric(CAex) # symmetric, but not formally

A <- as(CAex, "symmetricMatrix")
class(A) # dsCMatrix

## Have methods for denseMatrix (unpacked and packed),
## but not yet sparseMatrix ...
## Not run:
(bk.A <- BunchKaufman(A))

## End(Not run)
(bk.A <- BunchKaufman(as(A, "unpackedMatrix")))
```

```
## A ~ U D U U' in floating point
str(e.bk.A <- expand2(bk.A), max.level = 2L)
stopifnot(all.equal(as(A, "matrix"), as(Reduce(`%*%`, e.bk.A), "matrix")))
```

CAex

Albers' example Matrix with "Difficult" Eigen Factorization

Description

An example of a sparse matrix for which `eigen()` seemed to be difficult, an unscaled version of this has been posted to the web, accompanying an E-mail to R-help (<https://stat.ethz.ch/mailman/listinfo/r-help>), by Casper J Albers, Open University, UK.

Usage

```
data(CAex)
```

Format

This is a 72×72 symmetric matrix with 216 non-zero entries in five bands, stored as sparse matrix of class `dgCMatrix`.

Details

Historical note (2006-03-30): In earlier versions of R, `eigen(CAex)` fell into an infinite loop whereas `eigen(CAex, EISPACK=TRUE)` had been okay.

Examples

```
data(CAex, package = "Matrix")
str(CAex) # of class "dgCMatrix"

image(CAex)# -> it's a simple band matrix with 5 bands
## and the eigen values are basically 1 (42 times) and 0 (30 x):
zapsmall(ev <- eigen(CAex, only.values=TRUE)$values)
## i.e., the matrix is symmetric, hence
sCA <- as(CAex, "symmetricMatrix")
## and
stopifnot(class(sCA) == "dsCMatrix",
          as(sCA, "matrix") == as(CAex, "matrix"))
```

Description

Computes the upper triangular Cholesky factor of an $n \times n$ real, symmetric, positive semidefinite matrix A , optionally after pivoting. That is the factor L' in

$$P_1 A P_1' = L L'$$

or (equivalently)

$$A = P_1' L L' P_1$$

where P_1 is a permutation matrix.

Methods for [denseMatrix](#) are built on LAPACK routines `dpstrf`, `dpotrf`, and `dpptrf`. The latter two do not permute rows or columns, so that P_1 is an identity matrix.

Methods for [sparseMatrix](#) are built on CHOLMOD routines `cholmod_analyze` and `cholmod_factorize_p`.

Usage

```
chol(x, ...)
## S4 method for signature 'denseMatrix'
chol(x, pivot = FALSE, tol = -1, ...)
## S4 method for signature 'CsparseMatrix'
chol(x, pivot = FALSE, ...)
```

Arguments

<code>x</code>	a finite , symmetric, positive semidefinite matrix or Matrix to be factorized. If x is square but not symmetric, then it will be <i>treated</i> as symmetric; see <code>uplo</code> . Methods for dense x require positive definiteness when <code>pivot = FALSE</code> . Methods for sparse (but not diagonal) x require positive definiteness unconditionally.
<code>pivot</code>	a logical indicating if the rows and columns of x should be pivoted. Methods for sparse x employ the approximate minimum degree (AMD) algorithm in order to reduce fill-in, i.e., without regard for numerical stability.
<code>tol</code>	a finite numeric tolerance, used only if <code>pivot = TRUE</code> . The factorization algorithm stops if the pivot is less than or equal to <code>tol</code> . Negative <code>tol</code> is equivalent to <code>nrow(x) * .Machine\$double.eps * max(diag(x))</code> .
<code>...</code>	further arguments passed to or from methods.

Details

For x inheriting from [diagonalMatrix](#), the diagonal result is computed directly and without pivoting, i.e., bypassing CHOLMOD.

For all other x , `chol(x, pivot = value)` calls `Cholesky(x, perm = value, ...)` under the hood. If you must know the permutation P_1 in addition to the Cholesky factor L' , then call [Cholesky](#) directly, as the result of `chol(x, pivot = TRUE)` specifies L' but not P_1 .

Value

A matrix, `triangularMatrix`, or `diagonalMatrix` representing the upper triangular Cholesky factor L' . The result is a traditional matrix if `x` is a traditional matrix, dense if `x` is dense, and sparse if `x` is sparse.

References

The LAPACK source code, including documentation; see <https://netlib.org/lapack/double/dpstrf.f>, <https://netlib.org/lapack/double/dpotrf.f>, and <https://netlib.org/lapack/double/dpptrf.f>.

The CHOLMOD source code; see <https://github.com/DrTimothyAldenDavis/SuiteSparse>, notably the header file 'CHOLMOD/Include/cholmod.h' defining `cholmod_factor_struct`.

Chen, Y., Davis, T. A., Hager, W. W., & Rajamanickam, S. (2008). Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software*, 35(3), Article 22, 1-14. doi:10.1145/1391989.1391995

Amestoy, P. R., Davis, T. A., & Duff, I. S. (2004). Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 17(4), 886-905. doi:10.1145/1024074.1024081

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

The default method from `base`, `chol`, called for traditional matrices `x`.

Generic function `Cholesky`, for more flexibility notably when computing the Cholesky factorization and not only the factor L' .

Examples

```
showMethods("chol", inherited = FALSE)
set.seed(0)

## ---- Dense -----

## chol(x, pivot = value) wrapping Cholesky(x, perm = value)
selectMethod("chol", "dsyMatrix")

## Except in packed cases where pivoting is not yet available
selectMethod("chol", "dspMatrix")

## .... Positive definite .....

(A1 <- new("dsyMatrix", Dim = c(2L, 2L), x = c(1, 2, 2, 5)))
(R1.nopivot <- chol(A1))
(R1 <- chol(A1, pivot = TRUE))

## In 2-by-2 cases, we know that the permutation is 1:2 or 2:1,
## even if in general 'chol' does not say ...
```

```

stopifnot(exprs = {
  all.equal( A1 , as(crossprod(R1.nopivot), "dsyMatrix"))
  all.equal(t(A1[2:1, 2:1]), as(crossprod(R1 , "dsyMatrix"))
  identical(Cholesky(A1)@perm, 2:1) # because 5 > 1
})

## .... Positive semidefinite but not positive definite .....

(A2 <- new("dpoMatrix", Dim = c(2L, 2L), x = c(1, 2, 2, 4)))
try(R2.nopivot <- chol(A2)) # fails as not positive definite
(R2 <- chol(A2, pivot = TRUE)) # returns, with a warning and ...

stopifnot(exprs = {
  all.equal(t(A2[2:1, 2:1]), as(crossprod(R2), "dsyMatrix"))
  identical(Cholesky(A2)@perm, 2:1) # because 4 > 1
})

## .... Not positive semidefinite .....

(A3 <- new("dsyMatrix", Dim = c(2L, 2L), x = c(1, 2, 2, 3)))
try(R3.nopivot <- chol(A3)) # fails as not positive definite
(R3 <- chol(A3, pivot = TRUE)) # returns, with a warning and ...

## _Not_ equal: see details and examples in help("Cholesky")
all.equal(t(A3[2:1, 2:1]), as(crossprod(R3), "dsyMatrix"))

## ---- Sparse -----

## chol(x, pivot = value) wrapping
## Cholesky(x, perm = value, LDL = FALSE, super = FALSE)
selectMethod("chol", "dsCMatrix")

## Except in diagonal cases which are handled "directly"
selectMethod("chol", "ddiMatrix")

(A4 <- toeplitz(as(c(10, 0, 1, 0, 3), "sparseVector"))))
(ch.A4.nopivot <- Cholesky(A4, perm = FALSE, LDL = FALSE, super = FALSE))
(ch.A4 <- Cholesky(A4, perm = TRUE, LDL = FALSE, super = FALSE))
(R4.nopivot <- chol(A4))
(R4 <- chol(A4, pivot = TRUE))

det4 <- det(A4)
b4 <- rnorm(5L)
x4 <- solve(A4, b4)

stopifnot(exprs = {
  identical(R4.nopivot, expand1(ch.A4.nopivot, "L."))
  identical(R4, expand1(ch.A4, "L."))
  all.equal(A4,
    as(crossprod(R4.nopivot), "symmetricMatrix"))
  all.equal(A4[ch.A4@perm + 1L, ch.A4@perm + 1L],
    as(crossprod(R4 , "symmetricMatrix"))
  all.equal(diag(R4.nopivot), sqrt(diag(ch.A4.nopivot)))
})

```

```

    all.equal(diag(R4), sqrt(diag(ch.A4)))
    all.equal(sqrt(det4), det(R4.nopivot))
    all.equal(sqrt(det4), det(R4))
    all.equal(det4, det(ch.A4.nopivot, sqrt = FALSE))
    all.equal(det4, det(ch.A4, sqrt = FALSE))
    all.equal(x4, solve(R4.nopivot, solve(t(R4.nopivot), b4)))
    all.equal(x4, solve(ch.A4.nopivot, b4))
    all.equal(x4, solve(ch.A4, b4))
  })

```

chol2inv-methods

Inverse from Cholesky Factor

Description

Given *formally* upper and lower triangular matrices U and L , compute $(U'U)^{-1}$ and $(LL')^{-1}$, respectively.

This function can be seen as way to compute the inverse of a symmetric positive definite matrix given its Cholesky factor. Equivalently, it can be seen as a way to compute $(X'X)^{-1}$ given the R part of the QR factorization of X , if R is constrained to have positive diagonal entries.

Usage

```

chol2inv(x, ...)
## S4 method for signature 'denseMatrix'
chol2inv(x, uplo = "U", ...)
## S4 method for signature 'sparseMatrix'
chol2inv(x, uplo = "U", ...)

```

Arguments

<code>x</code>	a square matrix or Matrix , typically the result of a call to chol . If <code>x</code> is square but not (formally) triangular, then only the upper or lower triangle is considered, depending on optional argument <code>uplo</code> if <code>x</code> is a Matrix .
<code>uplo</code>	a string, either "U" or "L", indicating which triangle of <code>x</code> contains the Cholesky factor. The default is "U", to be consistent with chol2inv from base .
<code>...</code>	further arguments passed to or from methods.

Value

A matrix, [symmetricMatrix](#), or [diagonalMatrix](#) representing the inverse of the positive definite matrix whose Cholesky factor is `x`. The result is a traditional matrix if `x` is a traditional matrix, dense if `x` is dense, and sparse if `x` is sparse.

See Also

The default method from **base**, `chol2inv`, called for traditional matrices `x`.

Generic function `chol`, for computing the upper triangular Cholesky factor L' of a symmetric positive semidefinite matrix.

Generic function `solve`, for solving linear systems and (as a corollary) for computing inverses more generally.

Examples

```
(A <- Matrix(cbind(c(1, 1, 1), c(1, 2, 4), c(1, 4, 16))))
(R <- chol(A))
(L <- t(R))
(R2i <- chol2inv(R))
(L2i <- chol2inv(R))
stopifnot(exprs = {
  all.equal(R2i, tcrossprod(solve(R)))
  all.equal(L2i, crossprod(solve(L)))
  all.equal(as(R2i %*% A, "matrix"), diag(3L)) # the identity
  all.equal(as(L2i %*% A, "matrix"), diag(3L)) # ditto
})
```

Cholesky-methods

*Methods for Cholesky Factorization***Description**

Computes the pivoted Cholesky factorization of an $n \times n$ real, symmetric matrix A , which has the general form

$$P_1 A P_1' = L_1 D L_1' \stackrel{D_{jj} \geq 0}{=} L L'$$

or (equivalently)

$$A = P_1' L_1 D L_1' P_1 \stackrel{D_{jj} \geq 0}{=} P_1' L L' P_1$$

where P_1 is a permutation matrix, L_1 is a unit lower triangular matrix, D is a diagonal matrix, and $L = L_1 \sqrt{D}$. The second equalities hold only for positive semidefinite A , for which the diagonal entries of D are non-negative and $\sqrt{D}$ is well-defined.

Methods for `denseMatrix` are built on LAPACK routines `dpstrf`, `dpotrf`, and `dpptrf`. The latter two do not permute rows or columns, so that P_1 is an identity matrix.

Methods for `sparseMatrix` are built on CHOLMOD routines `cholmod_analyze` and `cholmod_factorize_p`.

Usage

```
Cholesky(A, ...)
## S4 method for signature 'denseMatrix'
Cholesky(A, perm = TRUE, tol = -1, uplo = "U", ...)
## S4 method for signature 'CsparseMatrix'
Cholesky(A, perm = TRUE, LDL = !super, super = FALSE,
```

```

        Imult = 0, force = TRUE, uplo = "U", ...)
## S4 method for signature 'matrix'
Cholesky(A, ...)

```

Arguments

A	a finite , symmetric matrix or Matrix to be factorized. If A is square but not symmetric, then it will be <i>treated</i> as symmetric; see uplo. Methods for dense A require positive definiteness when perm = FALSE and positive semidefiniteness when perm = TRUE. Methods for sparse A require positive definiteness when LDL = TRUE and nonzero leading principal minors (after pivoting) when LDL = FALSE. Methods for sparse, <i>diagonal</i> A are an exception, requiring positive semidefiniteness unconditionally.
perm	a logical indicating if the rows and columns of A should be pivoted. Methods for sparse A employ the approximate minimum degree (AMD) algorithm in order to reduce fill-in, i.e., without regard for numerical stability. Pivoting for sparsity may introduce nonpositive leading principal minors, causing the factorization to fail, in which case it may be necessary to set perm = FALSE.
tol	a finite numeric tolerance, used only if perm = TRUE. The factorization algorithm stops if the pivot is less than or equal to tol. Negative tol is equivalent to $\text{nrow}(A) * .\text{Machine}\$double.eps * \max(\text{diag}(A))$.
LDL	a logical indicating if the simplicial factorization should be computed as $P_1' L_1 D L_1' P_1$, such that the result stores the lower triangular entries of $L_1 - I + D$. The alternative is $P_1' L L' P_1$, such that the result stores the lower triangular entries of $L = L_1 \sqrt{D}$. This argument is ignored if super = TRUE (or if super = NA and the supernodal algorithm is chosen), as the supernodal code does not yet support the LDL = TRUE variant.
super	a logical indicating if the factorization should use the supernodal algorithm. The alternative is the simplicial algorithm. Setting super = NA leaves the choice to a CHOLMOD-internal heuristic.
Imult	a finite number. The matrix that is factorized is $A + \text{Imult} * \text{diag}(\text{nrow}(A))$, i.e., A plus Imult times the identity matrix. This argument is useful for symmetric, indefinite A, as $\text{Imult} > \max(\text{rowSums}(\text{abs}(A)) - \text{diag}(\text{abs}(A)))$ ensures that $A + \text{Imult} * \text{diag}(\text{nrow}(A))$ is diagonally dominant. (Symmetric, diagonally dominant matrices are positive definite.)
force	.
uplo	a string, either "U" or "L", indicating which triangle of A should be used to compute the factorization. The default is "U", even for lower triangular A, to be consistent with chol from base .
...	further arguments passed to or from methods.

Details

Note that the result of a call to Cholesky inherits from CholeskyFactorization but not [Matrix](#). Users who just want a matrix should consider using [chol](#), whose methods are simple wrappers around Cholesky returning just the upper triangular Cholesky factor L' , typically as a [triangularMatrix](#).

However, a more principled approach would be to construct factors as needed from the `CholeskyFactorization` object, e.g., with `expand1(x, "L")`, if `x` is the object.

The behaviour of `Cholesky(A, perm = TRUE)` for dense `A` is somewhat exceptional, in that it expects *without* checking that `A` is positive semidefinite. By construction, if `A` is positive semidefinite and the exact algorithm encounters a zero pivot, then the unfactorized trailing submatrix is the zero matrix, and there is nothing left to do. Hence when the finite precision algorithm encounters a pivot less than `tol`, it signals a warning instead of an error and zeros the trailing submatrix in order to guarantee that $P'LL'P$ is positive semidefinite even if `A` is not. It follows that one way to test for positive semidefiniteness of `A` in the event of a warning is to analyze the error

$$\frac{\|A - P'LL'P\|}{\|A\|}.$$

See the examples and LAPACK Working Note (“LAWN”) 161 for details.

Value

An object representing the factorization, inheriting from virtual class `CholeskyFactorization`. For a traditional matrix `A`, the specific class is `Cholesky`. For `A` inheriting from `unpackedMatrix`, `packedMatrix`, and `sparseMatrix`, the specific class is `Cholesky`, `pCholesky`, and `dCHMsimpl` or `dCHMsuper`, respectively.

References

The LAPACK source code, including documentation; see <https://netlib.org/lapack/double/dpstrf.f>, <https://netlib.org/lapack/double/dpotrf.f>, and <https://netlib.org/lapack/double/dpptrf.f>.

The CHOLMOD source code; see <https://github.com/DrTimothyAldenDavis/SuiteSparse>, notably the header file ‘CHOLMOD/Include/cholmod.h’ defining `cholmod_factor_struct`.

Lucas, C. (2004). *LAPACK-style codes for level 2 and 3 pivoted Cholesky factorizations*. LAPACK Working Note, Number 161. <https://www.netlib.org/lapack/lawnsdpdf/lawn161.pdf>

Chen, Y., Davis, T. A., Hager, W. W., & Rajamanickam, S. (2008). Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software*, 35(3), Article 22, 1-14. doi:10.1145/1391989.1391995

Amestoy, P. R., Davis, T. A., & Duff, I. S. (2004). Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 17(4), 886-905. doi:10.1145/1024074.1024081

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

Classes `Cholesky`, `pCholesky`, `dCHMsimpl` and `dCHMsuper` and their methods.

Classes `dpoMatrix`, `dppMatrix`, and `dsCMatrix`.

Generic function `chol`, for obtaining the upper triangular Cholesky factor L' as a matrix or `Matrix`.

Generic functions `expand1` and `expand2`, for constructing matrix factors from the result.

Generic functions `BunchKaufman`, `Schur`, `lu`, and `qr`, for computing other factorizations.

Examples

```

showMethods("Cholesky", inherited = FALSE)
set.seed(0)

## ---- Dense -----

## .... Positive definite .....

n <- 6L
(A1 <- crossprod(Matrix(rnorm(n * n), n, n)))
(ch.A1.nopivot <- Cholesky(A1, perm = FALSE))
(ch.A1 <- Cholesky(A1))
stopifnot(exprs = {
  length(ch.A1@perm) == ncol(A1)
  isPerm(ch.A1@perm)
  is.unsorted(ch.A1@perm) # typically not the identity permutation
  length(ch.A1.nopivot@perm) == 0L
})

##  $A \sim P_1' L D L' P_1 \sim P_1' L L' P_1$  in floating point
str(e.ch.A1 <- expand2(ch.A1, LDL = TRUE), max.level = 2L)
str(E.ch.A1 <- expand2(ch.A1, LDL = FALSE), max.level = 2L)
stopifnot(exprs = {
  all.equal(as(A1, "matrix"), as(Reduce(`%*%`, e.ch.A1), "matrix"))
  all.equal(as(A1, "matrix"), as(Reduce(`%*%`, E.ch.A1), "matrix"))
})

## .... Positive semidefinite but not positive definite .....

A2 <- A1
A2[1L, ] <- A2[, 1L] <- 0
A2
try(Cholesky(A2, perm = FALSE)) # fails as not positive definite
ch.A2 <- Cholesky(A2) # returns, with a warning and ...
A2.hat <- Reduce(`%*%`, expand2(ch.A2, LDL = FALSE))
norm(A2 - A2.hat, "2") / norm(A2, "2") # 7.670858e-17

## .... Not positive semidefinite .....

A3 <- A1
A3[1L, ] <- A3[, 1L] <- -1
A3
try(Cholesky(A3, perm = FALSE)) # fails as not positive definite
ch.A3 <- Cholesky(A3) # returns, with a warning and ...
A3.hat <- Reduce(`%*%`, expand2(ch.A3, LDL = FALSE))
norm(A3 - A3.hat, "2") / norm(A3, "2") # 1.781568

## Indeed, 'A3' is not positive semidefinite, but 'A3.hat' _is_
ch.A3.hat <- Cholesky(A3.hat)
A3.hat.hat <- Reduce(`%*%`, expand2(ch.A3.hat, LDL = FALSE))
norm(A3.hat - A3.hat.hat, "2") / norm(A3.hat, "2") # 1.777944e-16

```

```
## ---- Sparse -----

## Really just three cases modulo permutation :
##
##      type      factorization  minors of P1 A P1'
##  1  simplicial P1 A P1' = L1 D L1'      nonzero
##  2  simplicial P1 A P1' = L   L  '      positive
##  3  supernodal P1 A P2' = L   L  '      positive

data(KNex, package = "Matrix")
A4 <- crossprod(KNex[["mm"]])

ch.A4 <-
list(pivoted =
  list(simpl1 = Cholesky(A4, perm = TRUE, super = FALSE, LDL = TRUE),
       simpl0 = Cholesky(A4, perm = TRUE, super = FALSE, LDL = FALSE),
       super0 = Cholesky(A4, perm = TRUE, super = TRUE)),
  unpivoted =
  list(simpl1 = Cholesky(A4, perm = FALSE, super = FALSE, LDL = TRUE),
       simpl0 = Cholesky(A4, perm = FALSE, super = FALSE, LDL = FALSE),
       super0 = Cholesky(A4, perm = FALSE, super = TRUE)))
ch.A4

s <- simplify2array
rapply2 <- function(object, f, ...) rapply(object, f, , , how = "list", ...)

s(rapply2(ch.A4, isLDL))
s(m.ch.A4 <- rapply2(ch.A4, expand1, "L")) # giving L = L1 sqrt(D)

## By design, the pivoted and simplicial factorizations
## are more sparse than the unpivoted and supernodal ones ...
s(rapply2(m.ch.A4, object.size))

## Which is nicely visualized by lattice-based methods for 'image'
inm <- c("pivoted", "unpivoted")
jnm <- c("simpl1", "simpl0", "super0")
for(i in 1:2)
  for(j in 1:3)
    print(image(m.ch.A4[[c(i, j)]], main = paste(inm[i], jnm[j])),
          split = c(j, i, 3L, 2L), more = i * j < 6L)

simpl1 <- ch.A4[[c("pivoted", "simpl1")]]
stopifnot(exprs = {
  length(simpl1@perm) == ncol(A4)
  isPerm(simpl1@perm, 0L)
  is.unsorted(simpl1@perm) # typically not the identity permutation
})

## One can expand with and without D regardless of isLDL(.),
## but "without" requires L = L1 sqrt(D), which is conditional
## on min(diag(D)) >= 0, hence "with" is the default
isLDL(simpl1)
stopifnot(min(diag(simpl1)) >= 0)
```

```

str(e.ch.A4 <- expand2(simpl1, LDL = TRUE), max.level = 2L) # default
str(E.ch.A4 <- expand2(simpl1, LDL = FALSE), max.level = 2L)
stopifnot(exprs = {
  all.equal(E.ch.A4[["L" ]], e.ch.A4[["L1" ]]) %% sqrt(e.ch.A4[["D"]])
  all.equal(E.ch.A4[["L." ]], sqrt(e.ch.A4[["D"]]) %% e.ch.A4[["L1." ]])
  all.equal(A4, as(Reduce(`%%`, e.ch.A4), "posdefMatrix"))
  all.equal(A4, as(Reduce(`%%`, E.ch.A4), "posdefMatrix"))
})

## The "same" permutation matrix with "alternate" representation
## [i, perm[i]] {margin=1} <-> [invertPerm(perm)[j], j] {margin=2}
alt <- function(P) {
  P@margin <- 1L + !(P@margin - 1L) # 1 <-> 2
  P@perm <- invertPerm(P@perm)
  P
}

## Expansions are elegant but inefficient (transposes are redundant)
## hence programmers should consider methods for 'expand1' and 'diag'
stopifnot(exprs = {
  identical(expand1(simpl1, "P1"), alt(e.ch.A4[["P1"]]))
  identical(expand1(simpl1, "L"), E.ch.A4[["L"]])
  identical(Diagonal(x = diag(simpl1)), e.ch.A4[["D"]])
})

## chol(A, pivot = value) is a simple wrapper around
## Cholesky(A, perm = value, LDL = FALSE, super = FALSE),
## returning L' = sqrt(D) L1' _but_ giving no information
## about the permutation P1
selectMethod("chol", "dsCMatrix")
stopifnot(all.equal(chol(A4, pivot = TRUE), E.ch.A4[["L." ]]))

## Now a symmetric matrix with positive _and_ negative eigenvalues,
## hence _not_ positive semidefinite
A5 <- new("dsCMatrix",
  Dim = c(7L, 7L),
  p = c(0:1, 3L, 6:7, 10:11, 15L),
  i = c(0L, 0:1, 0:3, 2:5, 3:6),
  x = c(1, 6, 38, 10, 60, 103, -4, 6, -32, -247, -2, -16, -128, -2, -67))
(ev <- eigen(A5, only.values = TRUE)$values)
(t.ev <- table(factor(sign(ev), -1:1))) # the matrix "inertia"

ch.A5 <- Cholesky(A5)
isLDL(ch.A5)
(d.A5 <- diag(ch.A5)) # diag(D) is partly negative

## Sylvester's law of inertia holds here, but not in general
## in finite precision arithmetic
stopifnot(identical(table(factor(sign(d.A5), -1:1)), t.ev))

try(expand1(ch.A5, "L")) # unable to compute L = L1 sqrt(D)
try(expand2(ch.A5, LDL = FALSE)) # ditto
try(chol(A5, pivot = TRUE)) # ditto

```

```
## The default expansion is "square root free" and still works here
str(e.ch.A5 <- expand2(ch.A5, LDL = TRUE), max.level = 2L)
stopifnot(all.equal(A5, as(Reduce(`%*%`, e.ch.A5), "symmetricMatrix")))

## Version of the SuiteSparse library, which includes CHOLMOD
Mv <- Matrix.Version()
Mv[["suitesparse"]]
```

coerce-methods	<i>Coercions in Package Matrix</i>
----------------	---

Description

TODO.

coerce-methods-graph	<i>Conversions "graph" <-> (sparse) Matrix</i>
----------------------	--

Description

Since 2005, package **Matrix** has supported coercions to and from class **graph** from package **igraph**. Since 2013, this functionality has been exposed via functions `T2graph` and `graph2T`, which, unlike methods for `as(from, "<Class>")`, support optional arguments.

Usage

```
graph2T(from, use.weights = )
T2graph(from, need.uniq = !isUniqueT(from), edgemode = NULL)
```

Arguments

from	for <code>graph2T()</code> , an R object of class "graph"; for <code>T2graph()</code> , a sparse matrix inheriting from " TsparseMatrix ".
use.weights	logical indicating if weights should be used, i.e., equivalently the result will be numeric, i.e. of class dgTMatrix ; otherwise the result will be ngTMatrix or nsTMatrix , the latter if the graph is undirected. The default looks if there are weights in the graph, and if any differ from 1, weights are used.
need.uniq	a logical indicating if from may need to be internally "uniqified"; do not set this and hence rather use the default, unless you know what you are doing!
edgemode	one of <code>NULL</code> , <code>"directed"</code> , or <code>"undirected"</code> . The default <code>NULL</code> looks if the matrix is symmetric and assumes <code>"undirected"</code> in that case.

Value

For `graph2T()`, a sparse matrix inheriting from `"TsparseMatrix"`.

For `T2graph()` an R object of class `"graph"`.

See Also

Package **igraph**, which provides similar coercions to and from its class `igraph` via functions `graph_from_adjacency_matrix` and `as_adjacency_matrix`.

Examples

```
if(requireNamespace("graph")) {
  n4 <- LETTERS[1:4]; dns <- list(n4,n4)
  show(a1 <- sparseMatrix(i= c(1:4), j=c(2:4,1), x = 2, dimnames=dns))
  show(g1 <- as(a1, "graph")) # directed
  unlist(graph::edgeWeights(g1)) # all '2'

  show(a2 <- sparseMatrix(i= c(1:4,4), j=c(2:4,1:2), x = TRUE, dimnames=dns))
  show(g2 <- as(a2, "graph")) # directed
  # now if you want it undirected:
  show(g3 <- T2graph(as(a2,"TsparseMatrix"), edgemode="undirected"))
  show(m3 <- as(g3,"Matrix"))
  show( graph2T(g3) ) # a "pattern Matrix" (nsTMatrix)

  a. <- sparseMatrix(i=4:1, j=1:4, dimnames=list(n4, n4), repr="T") # no 'x'
  show(a.) # "ngTMatrix"
  show(g. <- as(a., "graph"))

}
```

coerce-methods-SparseM

*Sparse Matrix Coercion from and to those from package **SparseM***

Description

Methods for coercion from and to sparse matrices from package **SparseM** are provided here, for ease of porting functionality to the **Matrix** package, and comparing functionality of the two packages. All these work via the usual `as(., "<class>")` coercion,

```
as(from, Class)
```

Methods

from = `"matrix.csr"`, **to** = `"dgRMatrix"` ...

from = `"matrix.csc"`, **to** = `"dgCMatrix"` ...

from = `"matrix.coo"`, **to** = `"dgTMatrix"` ...

```

from = "dgRMatrix", to = "matrix.csr" ...
from = "dgCMatrix", to = "matrix.csc" ...
from = "dgTMatrix", to = "matrix.coo" ...
from = "Matrix", to = "matrix.csr" ...
from = "matrix.csr", to = "dgCMatrix" ...
from = "matrix.coo", to = "dgCMatrix" ...
from = "matrix.csr", to = "Matrix" ...
from = "matrix.csc", to = "Matrix" ...
from = "matrix.coo", to = "Matrix" ...

```

See Also

The documentation in CRAN package **SparseM**, such as [SparseM.ontology](#), and one important class, [matrix.csr](#).

colSums-methods

Form Row and Column Sums and Means

Description

Form row and column sums and means for objects, for [sparseMatrix](#) the result may optionally be sparse ([sparseVector](#)), too. Row or column names are kept respectively as for **base** matrices and [colSums](#) methods, when the result is [numeric](#) vector.

Usage

```

colSums(x, na.rm = FALSE, dims = 1L, ...)
rowSums(x, na.rm = FALSE, dims = 1L, ...)
colMeans(x, na.rm = FALSE, dims = 1L, ...)
rowMeans(x, na.rm = FALSE, dims = 1L, ...)

## S4 method for signature 'CsparseMatrix'
colSums(x, na.rm = FALSE, dims = 1L,
        sparseResult = FALSE, ...)
## S4 method for signature 'CsparseMatrix'
rowSums(x, na.rm = FALSE, dims = 1L,
        sparseResult = FALSE, ...)
## S4 method for signature 'CsparseMatrix'
colMeans(x, na.rm = FALSE, dims = 1L,
        sparseResult = FALSE, ...)
## S4 method for signature 'CsparseMatrix'
rowMeans(x, na.rm = FALSE, dims = 1L,
        sparseResult = FALSE, ...)

```

Arguments

<code>x</code>	a Matrix, i.e., inheriting from <code>Matrix</code> .
<code>na.rm</code>	logical. Should missing values (including NaN) be omitted from the calculations?
<code>dims</code>	completely ignored by the <code>Matrix</code> methods.
<code>...</code>	potentially further arguments, for method <code><-></code> generic compatibility.
<code>sparseResult</code>	logical indicating if the result should be sparse, i.e., inheriting from class <code>sparseVector</code> . Only applicable when <code>x</code> is inheriting from a <code>sparseMatrix</code> class.

Value

returns a numeric vector if `sparseResult` is FALSE as per default. Otherwise, returns a `sparseVector`.
`dimnames(x)` are only kept (as `names(v)`) when the resulting `v` is `numeric`, since `sparseVectors` do not have names.

See Also

`colSums` and the `sparseVector` classes.

Examples

```
(M <- bdiag(Diagonal(2), matrix(1:3, 3,4), diag(3:2))) # 7 x 8
colSums(M)
d <- Diagonal(10, c(0,0,10,0,2,rep(0,5)))
MM <- kronecker(d, M)
dim(MM) # 70 80
length(MM@x) # 160, but many are '0' ; drop those:
MM <- drop0(MM)
length(MM@x) # 32
cm <- colSums(MM)
(scm <- colSums(MM, sparseResult = TRUE))
stopifnot(is(scm, "sparseVector"),
           identical(cm, as.numeric(scm)))
rowSums(MM, sparseResult = TRUE) # 14 of 70 are not zero
colMeans(MM, sparseResult = TRUE) # 16 of 80 are not zero
## Since we have no 'NA's, these two are equivalent :
stopifnot(identical(rowMeans(MM, sparseResult = TRUE),
                    rowMeans(MM, sparseResult = TRUE, na.rm = TRUE)),
           rowMeans(Diagonal(16)) == 1/16,
           colSums(Diagonal(7)) == 1)

## dimnames(x) --> names( <value> ) :
dimnames(M) <- list(paste0("r", 1:7), paste0("V",1:8))
M
colSums(M)
rowMeans(M)
## Assertions :
stopifnot(exprs = {
  all.equal(colSums(M),
            structure(c(1,1,6,6,6,6,3,2), names = colnames(M)))
  all.equal(rowMeans(M),
```

```
        structure(c(1,1,4,8,12,3,2)/8, names = paste0("r", 1:7)))
    })
```

Complex-methods	<i>Unary Operations on Complex Numbers</i>
-----------------	--

Description

TODO.

Usage

```
Complex(z)
```

Arguments

z .

condest	<i>Compute Approximate CONDition number and 1-Norm of (Large) Matrices</i>
---------	--

Description

“Estimate”, i.e. compute approximately the CONDition number of a (potentially large, often sparse) matrix A. It works by apply a fast *randomized* approximation of the 1-norm, `norm(A, "1")`, through `onenormest(.)`.

Usage

```
condest(A, t = min(n, 5), normA = norm(A, "1"),
        silent = FALSE, quiet = TRUE)

onenormest(A, t = min(n, 5), A.x, At.x, n,
           silent = FALSE, quiet = silent,
           iter.max = 10, eps = 4 * .Machine$double.eps)
```

Arguments

- A a square matrix, optional for `onenormest()`, where instead of A, A.x and At.x can be specified, see there.
- t number of columns to use in the iterations.
- normA number; (an estimate of) the 1-norm of A, by default `norm(A, "1")`; may be replaced by an estimate.

<code>silent</code>	logical indicating if warning and (by default) convergence messages should be displayed.
<code>quiet</code>	logical indicating if convergence messages should be displayed.
<code>A.x, At.x</code>	when A is missing, these two must be given as functions which compute $A \% x$, or $t(A) \% x$, respectively.
<code>n</code>	<code>== nrow(A)</code> , only needed when A is not specified.
<code>iter.max</code>	maximal number of iterations for the 1-norm estimator.
<code>eps</code>	the relative change that is deemed irrelevant.

Details

`condest()` calls `lu(A)`, and subsequently `onenormest(A.x = , At.x = )` to compute an approximate norm of the *inverse* of A , A^{-1} , in a way which keeps using sparse matrices efficiently when A is sparse.

Note that `onenormest()` uses random vectors and hence *both* functions' results are random, i.e., depend on the random seed, see, e.g., `set.seed()`.

Value

Both functions return a `list`; `condest()` with components,

<code>est</code>	a number > 0 , the estimated (1-norm) condition number $\hat{\kappa}$; when $r := rcond(A)$, $1/\hat{\kappa} \approx r$.
<code>v</code>	the maximal Ax column, scaled to $norm(v) = 1$. Consequently, $norm(Av) = norm(A)/est$; when <code>est</code> is large, v is an approximate null vector.

The function `onenormest()` returns a list with components,

<code>est</code>	a number > 0 , the estimated $norm(A, "1")$.
<code>v</code>	0-1 integer vector length n , with an 1 at the index j with maximal column $A[, j]$ in A .
<code>w</code>	numeric vector, the largest Ax found.
<code>iter</code>	the number of iterations used.

Author(s)

This is based on octave's `condest()` and `onenormest()` implementations with original author Jason Riedy, U Berkeley; translation to R and adaption by Martin Maechler.

References

- Nicholas J. Higham and Françoise Tisseur (2000). A Block Algorithm for Matrix 1-Norm Estimation, with an Application to 1-Norm Pseudospectra. *SIAM J. Matrix Anal. Appl.* **21**, 4, 1185–1201.
- William W. Hager (1984). Condition Estimates. *SIAM J. Sci. Stat. Comput.* **5**, 311–316.

See Also

`norm`, `rcond`.

Examples

```
data(KNex, package = "Matrix")
mtm <- with(KNex, crossprod(mm))
system.time(ce <- condest(mtm))
sum(abs(ce$v)) ## || v ||_1 == 1
## Prove that || A v || = || A || / est (as ||v|| = 1):
stopifnot(all.equal(norm(mtm %*% ce$v),
                     norm(mtm) / ce$est))

## reciprocal
1 / ce$est
system.time(rc <- rcond(mtm)) # takes ca 3 x longer
rc
all.equal(rc, 1/ce$est) # TRUE -- the approximation was good

one <- onenormest(mtm)
str(one) ## est = 12.3
## the maximal column:
which(one$v == 1) # mostly 4, rarely 1, depending on random seed
```

cov2cor-methods	<i>Scale a Covariance Matrix</i>
-----------------	----------------------------------

Description

TODO.

Usage

```
cov2cor(V)
```

Arguments

V .

CsparseMatrix-class	<i>Virtual Class ‘CsparseMatrix’ of Compressed Sparse Column Format Matrices</i>
---------------------	--

Description

CsparseMatrix is a virtual subclass of [sparseMatrix](#) representing compressed sparse column (CSC) format matrices. To represent an m-by-n matrix A, this format stores the positions of m*n or fewer (often $O(\max\{m, n\})$) entries of A, ordering these by column then by row. Authors of subclasses of CsparseMatrix are free to declare how this information is used to specify A. See ‘Subclasses’ for subclasses defined in package **Matrix**.

Slots

`Dim`, `Dimnames` inherited from virtual superclass [Matrix](#).

`p` an integer vector of length `Dim[2]+1` corresponding to a subset V of the set of entries of A . `p[j+1]` is the number of elements of V belonging to one of the first j columns of A , implying

$$0 = p[1] \leq p[j] \leq p[j+1] \leq p[\text{length}(p)] = |V|$$

where $|V|$ denotes the size of V . The number of elements of V belonging to column j of A is given by `p[j+1]-p[j]` or, equivalently, `diff(p)[j]`.

`i` an integer vector of length `p[length(p)]` storing the 0-based row indices of the elements of V ordered by column then by row. Hence, to be valid, `i` must satisfy $0 \leq i < \text{Dim}[1]$ and be increasing within the contiguous segments that partition it by column of A . The segment corresponding to column j can be accessed as `i[(p[j]+1):p[j+1]]` if `p[j] < p[j+1]` and by `i[seq.int(from=p[j], length.out=p[j+1]-p[j])]` more generally. Experimentally, `length(i)` is allowed to exceed `p[length(p)]`; if it does, then the excess elements of `i` are unreferenced.

Subclasses

The nonvirtual subclasses of `CsparseMatrix` defined in package **Matrix** are listed below with links to virtual superclasses defining their data type and structure.

Nonvirtual subclass	Data type	Structure
<code>ngCMatrix</code>	nMatrix	generalMatrix
<code>lgCMatrix</code>	lMatrix	generalMatrix
<code>igCMatrix</code>	iMatrix	generalMatrix
<code>dgCMatrix</code>	dMatrix	generalMatrix
<code>zgCMatrix</code>	zMatrix	generalMatrix
<code>nsCMatrix</code>	nMatrix	symmetricMatrix
<code>lsCMatrix</code>	lMatrix	symmetricMatrix
<code>isCMatrix</code>	iMatrix	symmetricMatrix
<code>dsCMatrix</code>	dMatrix	symmetricMatrix
<code>zsCMatrix</code>	zMatrix	symmetricMatrix
<code>dpCMatrix</code>	dMatrix	posdefMatrix
<code>zpCMatrix</code>	zMatrix	posdefMatrix
<code>ntCMatrix</code>	nMatrix	triangularMatrix
<code>ltCMatrix</code>	lMatrix	triangularMatrix
<code>itCMatrix</code>	iMatrix	triangularMatrix
<code>dtCMatrix</code>	dMatrix	triangularMatrix
<code>ztCMatrix</code>	zMatrix	triangularMatrix

The subclasses of `iMatrix`, `zMatrix`, and `posdefMatrix` were not defined until **Matrix** version 1.8-0.

Classes `.gCMatrix` represent general (unstructured, possibly nonsquare) A . Objects have `p` and `i` slots specifying a subset V of the set of entries of A , as described in ‘Slots’. For `ngCMatrix`, whose data type is boolean, the entries in V have value 1, and the entries not in V have value 0. For other `.gCMatrix`, the entries in V have values taken from an additional `x` slot (a logical, integer, double,

or complex vector), and the entries not in V have value 0. The i and x slots are entirely parallel, both using the ordering of V by column then by row.

Classes `.sCMatrix` represent Hermitian or symmetric A . Compared to `.gCMatrix`, objects have an additional `uplo` slot indicating a restriction on V . If `uplo="U"`, then V contains only upper triangular entries of A . If `uplo="L"`, then V contains only lower triangular entries of A . The entries in V have value 1 (`nsCMatrix`) or values taken from an x slot (other `.sCMatrix`). The entries opposite the main diagonal, namely $A[j, i]$ for all $A[i, j]$ in V , have value $\text{op}(A[i, j])$, where $\text{op}=\text{Conj}$ for Hermitian A and $\text{op}=\text{identity}$ for symmetric A . All other entries of A have value 0. Objects of class `zsCMatrix` have an additional `trans` slot that distinguishes Hermitian and symmetric A , which use `trans="C"` and `"T"`, respectively. If `trans="C"` and V contains diagonal entries of A , then the imaginary parts of the corresponding elements of x are not referenced. (The diagonal entries of Hermitian matrices are real by definition.)

Classes `dpCMatrix` and `zpcMatrix` extend `dsCMatrix` and `zsCMatrix` and represent positive semidefinite A . Positive semidefiniteness of A induces a constraint on the x slot. However, this constraint is not strictly enforced by the validity methods, which test only that the diagonal entries of A are nonnegative (a necessary, not sufficient condition). For `zpcMatrix`, only `trans="C"` is valid.

Classes `.tCMatrix` represent triangular A . Compared to `.gCMatrix`, objects have additional `uplo` and `diag` slots indicating restrictions on V . If `uplo="U"`, then A is upper triangular and V contains only upper triangular entries. If `uplo="L"`, then A is lower triangular and V contains only lower triangular entries. If `diag="U"` (instead of `"N"`), then A is unit triangular and V contains only *strictly* upper or lower triangular entries. The entries in V have value 1 (`ntCMatrix`) or values taken from an x slot (other `.tCMatrix`). With the exception of diagonal entries when `diag="U"`, which have value 1, entries not in V have value 0.

See Also

“Sister” classes [RsparseMatrix](#) and [TsparseMatrix](#) for compressed sparse row and triplet format matrices. Generic functions [cbind2](#), [colSums](#), [\[](#) (when called as `x[, j]`), etc., whose methods for `CsparseMatrix` are particularly efficient.

Examples

```
getClass("CsparseMatrix")

## The common validity check function (based on C code):
getValidity(getClass("CsparseMatrix"))

(m <- Matrix(c(0,0,2:0), 3,5))
str(m)
m[,1]

mm <- Matrix(toeplitz(c(10, 0, 1, 0, 3)), sparse = TRUE)
mm # automatically dsCMatrix
str(mm)
mT <- as(as(mm, "generalMatrix"), "TsparseMatrix")

## Either
(symM <- as(mT, "symmetricMatrix")) # dsT
```

```

(symC <- as(symM, "CsparseMatrix")) # dsC
## or
sT <- Matrix(mT, sparse=TRUE, forceCheck=TRUE) # dsT

sym2 <- as(symC, "TsparseMatrix")
## --> the same as 'symM', a "dsTMatrix"

showClass("dtCMatrix")
showClass("dtTMatrix")
t1 <- new("dtTMatrix", x= c(3,7), i= 0:1, j=3:2, Dim= as.integer(c(4,4)))
t1
## from 0-diagonal to unit-diagonal {low-level step}:
tu <- t1 ; tu@diag <- "U"
tu
(cu <- as(tu, "CsparseMatrix"))
str(cu)# only two entries in @i and @x
stopifnot(cu@i == 1:0,
           all(2 * symmpart(cu) == Diagonal(4) + forceSymmetric(cu)))

t1[1,2:3] <- -1:-2
diag(t1) <- 10*c(1:2,3:2)
t1 # still triangular
(it1 <- solve(t1))
t1. <- solve(it1)
all(abs(t1 - t1.) < 10 * .Machine$double.eps)

## 2nd example
U5 <- new("dtCMatrix", i= c(1L, 0:3), p=c(0L,0L,0:2, 5L), Dim = c(5L, 5L),
         x = rep(1, 5), diag = "U")
U5
(iu <- solve(U5)) # contains one '0'
validObject(iu2 <- solve(U5, Diagonal(5)))# failed in earlier versions

I5 <- iu %*% U5 # should equal the identity matrix
i5 <- iu2 %*% U5
m53 <- matrix(1:15, 5,3, dimnames=list(NULL,letters[1:3]))
asDiag <- function(M) as(drop0(M), "diagonalMatrix")
stopifnot(
  all.equal(Diagonal(5), asDiag(I5), tolerance=1e-14) ,
  all.equal(Diagonal(5), asDiag(i5), tolerance=1e-14) ,
  identical(list(NULL, dimnames(m53)[[2]]), dimnames(solve(U5, m53)))
)

```

Description

Returns the conjugate transpose of a matrix or data frame, defined as `Conj(t(x))` if `x` has complex data and as `t(x)` otherwise.

Usage

```
ct(x)

## S4 method for signature 'CsparseMatrix'
ct(x)
## S4 method for signature 'RsparseMatrix'
ct(x)
## S4 method for signature 'TsparseMatrix'
ct(x)
## S4 method for signature 'denseMatrix'
ct(x)
## S4 method for signature 'diagonalMatrix'
ct(x)
## S4 method for signature 'indMatrix'
ct(x)
## S4 method for signature 'pMatrix'
ct(x)
## S4 method for signature 'sparseVector'
ct(x)
## S4 method for signature 'vector'
ct(x)

.ctCRT(x, lazy = TRUE)
```

Arguments

<code>x</code>	an R object representing a matrix or a vector. Package Matrix provides methods for Matrix objects, sparseVector objects, and traditional matrices and vectors. Vectors must have length in the range of type integer ; they are handled as one-column matrices.
	For <code>.ctCRT</code> , <code>x</code> must inherit from one of <code>[CRT]sparseMatrix</code> .
<code>lazy</code>	a logical indicating if transposition can change the storage format.

Details

For consistency, methods for `ct` preserve the storage format of matrix arguments, even if that is not most efficient. In particular, if `x` is a [CsparseMatrix](#) object, then `ct(x)` is a [CsparseMatrix](#) object, even though the [RsparseMatrix](#) representation of the conjugate transpose is obtained more cheaply. For “lazy” transposing of compressed sparse column or row format matrices, interchanging the column and row formats, use `.ctCRT`.

If `x` is formally Hermitian (see [symmetricMatrix](#)), then `x` and `ct(x)` represent the same matrix with opposite uplo slots, such that one stores only the upper triangle and the other stores only the

lower triangle.

Value

An R object representing the conjugate transpose of `x`.

Methods provided by package **Matrix** return a `Matrix` object. The class is a superclass of the class of `x` if `x` is a `Matrix` object.

See Also

`t` and `.tCRT` for matrix transpose without conjugation. `crossprod` for matrix products involving transposes.

Examples

```
ct(c(1+1i, 2-2i)) # as Conj(t(.)) for complex data

typeof(x2 <- c(1L, 2L))
typeof(print(t(x2))) # "integer"
typeof(print(ct(x2))) # "integer"
typeof(print(Conj(t(x2)))) # "double" as 'Conj' coerces

stopifnot(identical(ct(1:2), t(1:2)), # keeping non-complex data type
           identical(Conj(t(1:2)), 0 + t(1:2)))

rcplx <-
function (n)
complex(modulus = rlnorm(n), argument = runif(n, 0, 2 * pi))

set.seed(14L)
(X <- new("zgRMatrix",
  Dim = c(3L, 5L),
  Dimnames = list(A = paste0("a", 1:3), B = paste0("b", 1:5)),
  p = c(0L, 1L, 2L, 5L), j = c(1L, 0L, 0L, 1L, 4L),
  x = round(rcplx(5L), 2L)))
(ctX <- ct(X)) # compressed sparse row
(.ctX <- .ctCRT(X)) # compressed sparse column

stopifnot(identical(dim(ctX), dim(X)[2:1]),
           identical(dimnames(ctX), dimnames(X)[2:1]),
           identical(X, ct(ctX)),
           identical(X, .ctCRT(.ctX)),
           identical(ctX, as(.ctX, "RsparseMatrix")),
           identical(.ctX, as(ctX, "CsparseMatrix")),
           identical(ctX, .ctCRT(X, lazy = FALSE)))

(S <- crossprod(X, trans = "C")) # formally Hermitian
(ctS <- ct(S)) # *output* is identical but stored triangle is different
stopifnot(is(S, "zpRMatrix"),
           all.equal(S, ctS, tolerance = 0),
           S@uplo == "U", ctS@uplo == "L",
           identical(S, ct(ctS)))
```

denseBunchKaufman-class

*Dense Bunch-Kaufman Factorizations***Description**

Classes BunchKaufman and pBunchKaufman represent Bunch-Kaufman factorizations of $n \times n$ real, symmetric matrices A , having the general form

$$A = U D_U U' = L D_L L'$$

where D_U and D_L are symmetric, block diagonal matrices composed of b_U and b_L 1×1 or 2×2 diagonal blocks; $U = \prod_{k=1}^{b_U} P_k U_k$ is the product of b_U row-permuted unit upper triangular matrices, each having nonzero entries above the diagonal in 1 or 2 columns; and $L = \prod_{k=1}^{b_L} P_k L_k$ is the product of b_L row-permuted unit lower triangular matrices, each having nonzero entries below the diagonal in 1 or 2 columns.

These classes store the nonzero entries of the $2b_U + 1$ or $2b_L + 1$ factors, which are individually sparse, in a dense format as a vector of length nn (BunchKaufman) or $n(n+1)/2$ (pBunchKaufman), the latter giving the “packed” representation.

Slots

Dim, Dimnames inherited from virtual class [MatrixFactorization](#).

uplo a string, either "U" or "L", indicating which triangle (upper or lower) of the factorized symmetric matrix was used to compute the factorization and in turn how the x slot is partitioned.

x a numeric vector of length $n*n$ (BunchKaufman) or $n*(n+1)/2$ (pBunchKaufman), where $n=Dim[1]$. The details of the representation are specified by the manual for LAPACK routines dsytrf and dsptrf.

perm an integer vector of length $n=Dim[1]$ specifying row and column interchanges as described in the manual for LAPACK routines dsytrf and dsptrf.

Extends

Class BunchKaufmanFactorization, directly. Class [MatrixFactorization](#), by class BunchKaufmanFactorization, distance 2.

Instantiation

Objects can be generated directly by calls of the form `new("BunchKaufman", ...)` or `new("pBunchKaufman", ...)`, but they are more typically obtained as the value of `BunchKaufman(x)` for x inheriting from [dsyMatrix](#) or [dspMatrix](#).

Methods

`coerce signature(from = "BunchKaufman", to = "dtrMatrix")`: returns a [dtrMatrix](#), useful for inspecting the internal representation of the factorization; see ‘Note’.

`coerce signature(from = "pBunchKaufman", to = "dtpMatrix")`: returns a [dtpMatrix](#), useful for inspecting the internal representation of the factorization; see ‘Note’.

`determinant signature(x = "p?BunchKaufman", logarithm = "logical")`: computes the determinant of the factorized matrix *A* or its logarithm.

`expand1 signature(x = "p?BunchKaufman")`: see [expand1-methods](#).

`expand2 signature(x = "p?BunchKaufman")`: see [expand2-methods](#).

`solve signature(a = "p?BunchKaufman", b = .)`: see [solve-methods](#).

Note

In **Matrix** < 1.6-0, class `BunchKaufman` extended [dtrMatrix](#) and class `pBunchKaufman` extended [dtpMatrix](#), reflecting the fact that the internal representation of the factorization is fundamentally triangular: there are $n(n+1)/2$ “parameters”, and these can be arranged systematically to form an $n \times n$ triangular matrix. **Matrix** 1.6-0 removed these extensions so that methods would no longer be inherited from `dtrMatrix` and `dtpMatrix`. The availability of such methods gave the wrong impression that `BunchKaufman` and `pBunchKaufman` represent a (singular) matrix, when in fact they represent an ordered set of matrix factors.

The coercions `as(., "dtrMatrix")` and `as(., "dtpMatrix")` are provided for users who understand the caveats.

References

The LAPACK source code, including documentation; see <https://netlib.org/lapack/double/dsytrf.f> and <https://netlib.org/lapack/double/dsptrf.f>.

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

Class [dsyMatrix](#) and its packed counterpart.

Generic functions [BunchKaufman](#), [expand1](#), and [expand2](#).

Examples

```
showClass("denseBunchKaufman")
set.seed(1)

n <- 6L
(A <- forceSymmetric(Matrix(rnorm(n * n), n, n)))

## With dimnames, to see that they are propagated :
dimnames(A) <- rep.int(list(paste0("x", seq_len(n))), 2L)

(bk.A <- BunchKaufman(A))
```

```

str(e.bk.A <- expand2(bk.A, complete = FALSE), max.level = 2L)
str(E.bk.A <- expand2(bk.A, complete = TRUE), max.level = 2L)

## Underlying LAPACK representation
(m.bk.A <- as(bk.A, "dtrMatrix"))
stopifnot(identical(as(m.bk.A, "matrix"), `dim<-`(bk.A@x, bk.A@Dim)))

## Number of factors is 2*b+1, b <= n, which can be nontrivial ...
(b <- (length(E.bk.A) - 1L) %/% 2L)

ae1 <- function(a, b, ...) all.equal(as(a, "matrix"), as(b, "matrix"), ...)
ae2 <- function(a, b, ...) ae1(unname(a), unname(b), ...)

## A ~ U DU U', U := prod(Pk Uk) in floating point
stopifnot(exprs = {
  identical(names(e.bk.A), c("U", "DU", "U."))
  identical(e.bk.A[["U" ]], Reduce(`%*%`, E.bk.A[seq_len(b)]))
  identical(e.bk.A[["U." ]], t(e.bk.A[["U"]]))
  ae1(A, with(e.bk.A, U %*% DU %*% U.))
})

## Factorization handled as factorized matrix
b <- rnorm(n)
stopifnot(identical(det(A), det(bk.A)),
  identical(solve(A, b), solve(bk.A, b)))

```

denseCholesky-class *Dense Cholesky Factorizations*

Description

Classes Cholesky and pCholesky represent dense, pivoted Cholesky factorizations of $n \times n$ real, symmetric, positive semidefinite matrices A , having the general form

$$P_1 A P_1' = L_1 D L_1' = L L'$$

or (equivalently)

$$A = P_1' L_1 D L_1' P_1 = P_1' L L' P_1$$

where P_1 is a permutation matrix, L_1 is a unit lower triangular matrix, D is a non-negative diagonal matrix, and $L = L_1 \sqrt{D}$.

These classes store the entries of the Cholesky factor L or its transpose L' in a dense format as a vector of length nn (Cholesky) or $n(n+1)/2$ (pCholesky), the latter giving the “packed” representation.

Slots

Dim, Dimnames inherited from virtual class [MatrixFactorization](#).

uplo a string, either “U” or “L”, indicating which triangle (upper or lower) of the factorized symmetric matrix was used to compute the factorization and in turn whether x stores L' or L .

- `x` a numeric vector of length $n \times n$ (Cholesky) or $n \times (n+1)/2$ (pCholesky), where $n = \text{Dim}[1]$, listing the entries of the Cholesky factor L or its transpose L' in column-major order.
- `perm` a 1-based integer vector of length $\text{Dim}[1]$ specifying the permutation applied to the rows and columns of the factorized matrix. `perm` of length 0 is valid and equivalent to the identity permutation, implying no pivoting.

Extends

Class `CholeskyFactorization`, directly. Class `MatrixFactorization`, by class `CholeskyFactorization`, distance 2.

Instantiation

Objects can be generated directly by calls of the form `new("Cholesky", ...)` or `new("pCholesky", ...)`, but they are more typically obtained as the value of `Cholesky(x)` for `x` inheriting from `dsyMatrix` or `dspMatrix` (often the subclasses of those reserved for positive semidefinite matrices, namely `dpoMatrix` and `dppMatrix`).

Methods

- `coerce` signature(`from` = "Cholesky", `to` = "dtrMatrix"): returns a `dtrMatrix` representing the Cholesky factor L or its transpose L' ; see 'Note'.
- `coerce` signature(`from` = "pCholesky", `to` = "dtpMatrix"): returns a `dtpMatrix` representing the Cholesky factor L or its transpose L' ; see 'Note'.
- `determinant` signature(`x` = "p?Cholesky", `logarithm` = "logical"): computes the determinant of the factorized matrix A or its logarithm.
- `diag` signature(`x` = "p?Cholesky"): returns a numeric vector of length n containing the diagonal elements of D , which are the squared diagonal elements of L .
- `expand1` signature(`x` = "p?Cholesky"): see [expand1-methods](#).
- `expand2` signature(`x` = "p?Cholesky"): see [expand2-methods](#).
- `solve` signature(`a` = "p?Cholesky", `b` = .): see [solve-methods](#).

Note

In **Matrix** < 1.6-0, class `Cholesky` extended `dtrMatrix` and class `pCholesky` extended `dtpMatrix`, reflecting the fact that the factor L is indeed a triangular matrix. **Matrix** 1.6-0 removed these extensions so that methods would no longer be inherited from `dtrMatrix` and `dtpMatrix`. The availability of such methods gave the wrong impression that `Cholesky` and `pCholesky` represent a (singular) matrix, when in fact they represent an ordered set of matrix factors.

The coercions `as(., "dtrMatrix")` and `as(., "dtpMatrix")` are provided for users who understand the caveats.

References

The LAPACK source code, including documentation; see <https://netlib.org/lapack/double/dpstrf.f>, <https://netlib.org/lapack/double/dpotrf.f>, and <https://netlib.org/lapack/double/dpptrf.f>.

Lucas, C. (2004). *LAPACK-style codes for level 2 and 3 pivoted Cholesky factorizations*. LAPACK Working Note, Number 161. <https://www.netlib.org/lapack/lawnspdf/lawn161.pdf>

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

Class CHMfactor for sparse Cholesky factorizations.

Classes [dpoMatrix](#) and [dppMatrix](#).

Generic functions [Cholesky](#), [expand1](#) and [expand2](#).

Examples

```
showClass("denseCholesky")
set.seed(1)

m <- 30L
n <- 6L
(A <- crossprod(Matrix(rnorm(m * n), m, n)))

## With dimnames, to see that they are propagated :
dimnames(A) <- dn <- rep.int(list(paste0("x", seq_len(n))), 2L)

(ch.A <- Cholesky(A)) # pivoted, by default
str(e.ch.A <- expand2(ch.A, LDL = TRUE), max.level = 2L)
str(E.ch.A <- expand2(ch.A, LDL = FALSE), max.level = 2L)

## Underlying LAPACK representation
(m.ch.A <- as(ch.A, "dtrMatrix")) # which is L', not L, because
A@uplo == "U"
stopifnot(identical(as(m.ch.A, "matrix"), `dim<-`(ch.A@x, ch.A@Dim)))

ae1 <- function(a, b, ...) all.equal(as(a, "matrix"), as(b, "matrix"), ...)
ae2 <- function(a, b, ...) ae1(unname(a), unname(b), ...)

## A ~ P1' L1 D L1' P1 ~ P1' L L' P1 in floating point
stopifnot(exprs = {
  identical(names(e.ch.A), c("P1.", "L1", "D", "L1.", "P1"))
  identical(names(E.ch.A), c("P1.", "L", "L.", "P1"))
  identical(e.ch.A[["P1"]],
    new("pMatrix", Dim = c(n, n), Dimnames = c(list(NULL), dn[2L]),
      margin = 2L, perm = invertPerm(ch.A@perm)))
  identical(e.ch.A[["P1."]], t(e.ch.A[["P1"]]))
  identical(e.ch.A[["L1."]], t(e.ch.A[["L1"]]))
  identical(E.ch.A[["L." ]], t(E.ch.A[["L" ]]))
  identical(e.ch.A[["D"]], Diagonal(x = diag(ch.A)))
  all.equal(E.ch.A[["L"]], with(e.ch.A, L1 %*% sqrt(D)))
  ae1(A, with(e.ch.A, P1. %*% L1 %*% D %*% L1. %*% P1))
  ae1(A, with(E.ch.A, P1. %*% L %*% L. %*% P1))
  ae2(A[ch.A@perm, ch.A@perm], with(e.ch.A, L1 %*% D %*% L1.))
  ae2(A[ch.A@perm, ch.A@perm], with(E.ch.A, L %*% L. ))
})
```

```

})

## Factorization handled as factorized matrix
b <- rnorm(n)
all.equal(det(A), det(ch.A), tolerance = 0)
all.equal(solve(A, b), solve(ch.A, b), tolerance = 0)

## For identical results, we need the _unpivoted_ factorization
## computed by det(A) and solve(A, b)
(ch.A.nopivot <- Cholesky(A, perm = FALSE))
stopifnot(identical(det(A), det(ch.A.nopivot)),
           identical(solve(A, b), solve(ch.A.nopivot, b)))

```

denseLU-class

Dense LU Factorizations

Description

`denseLU` is the class of dense, row-pivoted LU factorizations of $m \times n$ real matrices A , having the general form

$$P_1 A = LU$$

or (equivalently)

$$A = P_1' LU$$

where P_1 is an $m \times m$ permutation matrix, L is an $m \times \min(m, n)$ unit lower trapezoidal matrix, and U is a $\min(m, n) \times n$ upper trapezoidal matrix. If $m = n$, then the factors L and U are triangular.

Slots

`Dim`, `Dimnames` inherited from virtual class [MatrixFactorization](#).

`x` a numeric vector of length `prod(Dim)` storing the triangular L and U factors together in a packed format. The details of the representation are specified by the manual for LAPACK routine `dgetrf`.

`perm` an integer vector of length `min(Dim)` specifying the permutation P_1 as a product of transpositions. The corresponding permutation vector can be obtained as [asPerm](#)(`perm`).

Extends

Class [LU](#), directly. Class [MatrixFactorization](#), by class [LU](#), distance 2.

Instantiation

Objects can be generated directly by calls of the form `new("denseLU", ...)`, but they are more typically obtained as the value of `lu(x)` for `x` inheriting from [denseMatrix](#) (often [dgeMatrix](#)).

Methods

`coerce` signature(from = "denseLU", to = "dgeMatrix"): returns a `dgeMatrix` with the dimensions of the factorized matrix A , equal to L below the diagonal and equal to U on and above the diagonal.

`determinant` signature(x = "denseLU", logarithm = "logical"): computes the determinant of the factorized matrix A or its logarithm.

`expand` signature(x = "denseLU"): see [expand-methods](#).

`expand1` signature(x = "denseLU"): see [expand1-methods](#).

`expand2` signature(x = "denseLU"): see [expand2-methods](#).

`solve` signature(a = "denseLU", b = "missing"): see [solve-methods](#).

References

The LAPACK source code, including documentation; see <https://netlib.org/lapack/double/dgetrf.f>.

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

Class `sparseLU` for sparse LU factorizations.

Class `dgeMatrix`.

Generic functions `lu`, `expand1` and `expand2`.

Examples

```
showClass("denseLU")
set.seed(1)

n <- 3L
(A <- Matrix(round(rnorm(n * n), 2L), n, n))

## With dimnames, to see that they are propagated :
dimnames(A) <- dn <- list(paste0("r", seq_len(n)),
                          paste0("c", seq_len(n)))

(lu.A <- lu(A))
str(e.lu.A <- expand2(lu.A), max.level = 2L)

## Underlying LAPACK representation
(m.lu.A <- as(lu.A, "dgeMatrix")) # which is L and U interlaced
stopifnot(identical(as(m.lu.A, "matrix"), `dim<-`(lu.A@x, lu.A@Dim)))

ae1 <- function(a, b, ...) all.equal(as(a, "matrix"), as(b, "matrix"), ...)
ae2 <- function(a, b, ...) ae1(unname(a), unname(b), ...)

## A ~ P1' L U in floating point
stopifnot(exprs = {
```

```

identical(names(e.lu.A), c("P1.", "L", "U"))
identical(e.lu.A[["P1."]],
          new( "pMatrix", Dim = c(n, n), Dimnames = c(dn[1L], list(NULL)),
              margin = 1L, perm = invertPerm(asPerm(lu.A@perm))))
identical(e.lu.A[["L"]],
          new("dtrMatrix", Dim = c(n, n), Dimnames = list(NULL, NULL),
              uplo = "L", diag = "U", x = lu.A@x))
identical(e.lu.A[["U"]],
          new("dtrMatrix", Dim = c(n, n), Dimnames = c(list(NULL), dn[2L]),
              uplo = "U", diag = "N", x = lu.A@x))
ae1(A, with(e.lu.A, P1. %%% L %%% U))
ae2(A[asPerm(lu.A@perm), ], with(e.lu.A, L %%% U))
})

## Factorization handled as factorized matrix
b <- rnorm(n)
stopifnot(identical(det(A), det(lu.A)),
          identical(solve(A, b), solve(lu.A, b)))

```

denseMatrix-class	<i>Virtual Class ‘denseMatrix’ of Dense Format Matrices</i>
-------------------	---

Description

`denseMatrix` is a virtual subclass of `Matrix` representing dense format matrices. Formally, a dense format is a storage format in which the space required to represent an $m \times n$ matrix with p nonzero entries is not $O(\max\{m, n, p\})$.

Slots

`Dim`, `Dimnames` inherited from virtual superclass `Matrix`.

Subclasses

Package **Matrix** defines two virtual subclasses of `denseMatrix`: `unpackedMatrix` and `packedMatrix`. These correspond to the conventional and packed storage formats defined by LAPACK: <https://netlib.org/lapack/lug/node121.html>. See those help topics for implementation details and (recursively) further subclasses.

See Also

“Complementary” class `sparseMatrix` representing sparse (meaning *not* dense) format matrices.

Examples

```
showClass("denseMatrix")
```

denseSchur-class *Dense Schur Factorizations*

Description

Schur is the class of Schur factorizations of $n \times n$ real matrices A , having the general form

$$A = QTQ'$$

where Q is an orthogonal matrix and T is a block upper triangular matrix with 1×1 or 2×2 diagonal blocks specifying the real and complex conjugate eigenvalues of A . The column vectors of Q are the Schur vectors of A , and T is the Schur form of A .

The Schur factorization generalizes the spectral decomposition of normal matrices A , whose Schur form is block diagonal, to arbitrary square matrices.

Details

The matrix A and its Schur form T are *similar* and thus have the same spectrum. The eigenvalues are computed trivially as the eigenvalues of the diagonal blocks of T .

Slots

Dim, Dimnames inherited from virtual class [MatrixFactorization](#).

x a numeric vector of length prod(Dim) storing the entries of T in column-major order. x of length 0 is value and indicates that T is diagonal with diagonal entries given by values.

vectors a numeric vector of length prod(Dim) storing the entries of Q in column-major order. vectors of length 0 is valid and indicates that the factorization was computed without the Schur vectors.

values a numeric or complex vector of length Dim[1] storing the eigenvalues of the diagonal blocks of T , which are the eigenvalues of T and consequently of the factorized matrix A .

Extends

Class SchurFactorization, directly. Class [MatrixFactorization](#), by class SchurFactorization, distance 2.

Instantiation

Objects can be generated directly by calls of the form `new("Schur", ...)`, but they are more typically obtained as the value of `Schur(x)` for x inheriting from [Matrix](#) (often `dgeMatrix`).

Methods

determinant signature(x = "Schur", logarithm = "logical"): computes the determinant of the factorized matrix A or its logarithm.

expand1 signature(x = "Schur"): see [expand1-methods](#).

expand2 signature(x = "Schur"): see [expand2-methods](#).

solve signature(a = "Schur", b = .): see [solve-methods](#).

References

The LAPACK source code, including documentation; see <https://netlib.org/lapack/double/dgees.f>.

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

Class [dgeMatrix](#).

Generic functions [Schur](#), [expand1](#) and [expand2](#).

Examples

```
showClass("denseSchur")
set.seed(0)

n <- 4L
(A <- Matrix(rnorm(n * n), n, n))

## With dimnames, to see that they are propagated :
dimnames(A) <- list(paste0("r", seq_len(n)),
                    paste0("c", seq_len(n)))

(sch.A <- Schur(A))
str(e.sch.A <- expand2(sch.A), max.level = 2L)

## A ~ Q T Q' in floating point
stopifnot(exprs = {
  identical(names(e.sch.A), c("Q", "T", "Q."))
  all.equal(A, with(e.sch.A, Q %*% T %*% Q.))
})

## Factorization handled as factorized matrix
b <- rnorm(n)
stopifnot(all.equal(det(A), det(sch.A)),
          all.equal(solve(A, b), solve(sch.A, b)))
```

determinant-methods *Matrix Determinant*

Description

TODO.

Usage

```
det(x, ...)
determinant(x, logarithm = TRUE, ...)
```

Arguments

x .
logarithm .
... .

Examples

```
(M <- Matrix(c(0, 1, 0, 0), 6, 4))  
cm <- M[1:4, ] + 10 * Diagonal(4)  
stopifnot(all.equal(det(cm),  
                     determinant(as(cm, "matrix"), log = FALSE)$modulus,  
                     check.attributes = FALSE))
```

diag-methods	<i>Matrix Diagonals</i>
--------------	-------------------------

Description

TODO.

Usage

```
diag(x = 1, nrow, ncol, names = TRUE)  
diag(x) <- value
```

Arguments

x .
nrow .
ncol .
names .
value .

Examples

```
(M <- Matrix(c(0, 1, 0, 0), 6, 4))  
diag(M)
```

Diagonal

*Construct a Diagonal Matrix***Description**

Construct a formally diagonal [Matrix](#), i.e., an object inheriting from virtual class [diagonalMatrix](#) (or, if desired, a *mathematically* diagonal [CsparseMatrix](#)).

Usage

```
Diagonal(n, x = NULL, names = FALSE,
         keepInteger = Matrix::keepInteger())

.sparseDiagonal(n, x = NULL, uplo = "U", shape = "t", unitri = TRUE, kind, cols,
               keepInteger = Matrix::keepInteger())
.trDiagonal(n, x = NULL, uplo = "U", unitri = TRUE, kind)
.symDiagonal(n, x = NULL, uplo = "U", kind)
```

Arguments

n	integer indicating the dimension of the (square) matrix. If missing, then <code>length(x)</code> is used.
x	numeric or logical vector listing values for the diagonal entries, to be recycled as necessary. If <code>NULL</code> (the default), then the result is a unit diagonal matrix. <code>.sparseDiagonal()</code> and friends ignore non- <code>NULL</code> <code>x</code> when <code>kind = "n"</code> .
names	either logical <code>TRUE</code> or <code>FALSE</code> or then a character vector of <code>length</code> <code>n</code> . If true <i>and</i> <code>names(x)</code> is not <code>NULL</code> , use that as both row and column names for the resulting matrix. When a character vector, use it for both dimnames.
uplo	one of <code>c("U", "L")</code> , specifying the uplo slot of the result if the result is formally triangular or symmetric.
shape	one of <code>c("t", "s", "g")</code> , indicating if the result should be formally triangular, symmetric, or “general”. The result will inherit from virtual class triangularMatrix , symmetricMatrix , or generalMatrix , respectively.
unitri	logical indicating if a formally triangular result with ones on the diagonal should be formally <i>unit</i> triangular, i.e., with diag slot equal to “U” rather than “N”.
kind	one of <code>c("d", "l", "n")</code> , indicating the “mode” of the result: numeric, logical, or pattern. The result will inherit from virtual class dsparseMatrix , lsparseMatrix , or nsparseMatrix , respectively. Values other than “n” are ignored when <code>x</code> is non- <code>NULL</code> ; in that case the mode is determined by <code>typeof(x)</code> .
cols	optional integer vector with values in $0:(n-1)$, indexing columns of the specified diagonal matrix. If specified, then the result is (mathematically) $D[, \text{cols}+1]$ rather than D , where $D = \text{Diagonal}(n, x)$, and it is always “general” (i.e., shape is ignored).

`keepInteger` an optional logical argument introduced in **Matrix** version 1.8-0. If `FALSE`, then `integer` `x` are treated as `double`. This behaviour is compatible with earlier versions of **Matrix**, but it is not so desirable now that matrices with integer data are representable as objects inheriting from virtual class `iMatrix`. The default value will change to `TRUE` in the near future; currently, it depends on the value of the environment variable `R_MATRIX_KEEP_INTEGER` at load time (false if unset), but that can be overridden by `options(Matrix.keepInteger=)`.

Value

`Diagonal()` returns an object inheriting from virtual class `diagonalMatrix`.

`.sparseDiagonal()` returns a `CsparseMatrix` representation of `Diagonal(n, x)` or, if `cols` is given, of `Diagonal(n, x)[, cols+1]`. The precise class of the result depends on shape and kind.

`.trDiagonal()` and `.symDiagonal()` are simple wrappers, for `.sparseDiagonal(shape = "t")` and `.sparseDiagonal(shape = "s")`, respectively.

`.sparseDiagonal()` exists primarily to leverage efficient C-level methods available for `CsparseMatrix`.

Author(s)

Martin Maechler

See Also

the generic function `diag` for *extraction* of the diagonal from a matrix works for all “Matrices”.

`bandSparse` constructs a *banded* sparse matrix from its non-zero sub-/super - diagonals. `band(A)` returns a band matrix containing some sub-/super - diagonals of `A`.

`Matrix` for general matrix construction; further, class `diagonalMatrix`.

Examples

```
Diagonal(3)
Diagonal(x = 10^(3:1))
Diagonal(x = (1:4) >= 2)#-> "ldiMatrix"

## Use Diagonal() + kronecker() for "repeated-block" matrices:
M1 <- Matrix(0+0:5, 2,3)
(M <- kronecker(Diagonal(3), M1))

(S <- crossprod(Matrix(rbinom(60, size=1, prob=0.1), 10,6)))
(SI <- S + 10*.symDiagonal(6)) # sparse symmetric still
stopifnot(is(SI, "dsCMatrix"))
(I4 <- .sparseDiagonal(4, shape="t"))# now (2012-10) unitriangular
stopifnot(I4@diag == "U", all(I4 == diag(4)))
```

diagonalMatrix-class *Virtual Class 'diagonalMatrix' of Diagonal Matrices*

Description

diagonalMatrix is a virtual subclass of [triangularMatrix](#) and of [indexMatrix](#) representing diagonal matrices.

Slots

Dim, Dimnames inherited from virtual superclass [Matrix](#). Dim[1] and Dim[2] must be equal.

All direct, nonvirtual subclasses of diagonalMatrix defined in package **Matrix** have two additional slots:

diag a character string, either "N" or "U". "N" indicates that x stores the diagonal entries of the represented matrix. "U" indicates that the represented matrix is unit diagonal and that x is empty.

x a vector of length n=Dim[1]=Dim[2] or 0, conditionally (depending on diag) storing the diagonal entries of the represented matrix.

Subclasses

The direct, nonvirtual subclasses of diagonalMatrix defined in package **Matrix** are listed below with links to virtual superclasses defining their data type.

Nonvirtual subclass	Data type
ndiMatrix	nMatrix
ldiMatrix	lMatrix
idiMatrix	iMatrix
ddiMatrix	dMatrix
zdiMatrix	zMatrix

Class ndiMatrix was not defined until **Matrix** version 1.6-2.

See Also

Related classes [generalMatrix](#), [symmetricMatrix](#), [posdefMatrix](#), [triangularMatrix](#), and [indexMatrix](#) representing general (unstructured, possibly nonsquare), Hermitian or symmetric, positive semidefinite, triangular, and index matrices. Generic function [isDiagonal](#) for testing for testing if a matrix is diagonal. Generic function [forceDiagonal](#) for inducing diagonal structure. Function [Diagonal](#), a constructor for diagonalMatrix objects.

Examples

```
I5 <- Diagonal(5)
D5 <- Diagonal(x = 10*(1:5))
## trivial (but explicitly defined) methods:
stopifnot(identical(crossprod(I5), I5),
           identical(tcrossprod(I5), I5),
           identical(crossprod(I5, D5), D5),
           identical(tcrossprod(D5, I5), D5),
           identical(solve(D5), solve(D5, I5)),
           all.equal(D5, solve(solve(D5)), tolerance = 1e-12)
)
solve(D5)# efficient as is diagonal

# an unusual way to construct a band matrix:
rbind2(cbind2(I5, D5),
       cbind2(D5, I5))

(lM <- Diagonal(x = c(TRUE,FALSE,FALSE)))
str(lM)#> gory details (slots)

crossprod(lM) # numeric
(nM <- as(lM, "nMatrix"))
crossprod(nM) # pattern sparse

(d2 <- Diagonal(x = c(10,1)))
str(d2)
## slightly larger in internal size:
str(as(d2, "sparseMatrix"))

M <- Matrix(cbind(1,2:4))
M %**% d2 #> `fast' multiplication

chol(d2) # trivial
stopifnot(is(cd2 <- chol(d2), "ddiMatrix"),
          all.equal(cd2@x, c(sqrt(10),1)))
```

diagU2N	<i>Transform Triangular/Diagonal Matrices from Unit to General Triangular/Diagonal and Back</i>
---------	---

Description

Transform a triangular or diagonal matrix x , i.e., of `class triangularMatrix` or `diagonalMatrix`, from (internally!) unit triangular (“unitriangular”) to “general” triangular or diagonal, respectively, (`diagU2N(x)`) or back (`diagN2U(x)`). Note that the latter, `diagN2U(x)`, also sets the diagonal to one in cases where `diag(x)` was not all one.

`.diagU2N(x)` and `.diagN2U(x)` assume *without* checking that x is a `triangularMatrix` with suitable `diag` slot (“U” and “N”, respectively), hence they should be used with care.

Usage

```
diagU2N(x, cl = getClassDef(class(x)), checkDense = TRUE)
diagN2U(x, cl = getClassDef(class(x)), checkDense = TRUE)

.diagU2N(x, cl = getClassDef(class(x)), checkDense = FALSE)
.diagN2U(x, cl = getClassDef(class(x)), checkDense = FALSE)
```

Arguments

x	a triangularMatrix , often sparse.
cl	(optional, for speedup only:) class (definition) of x.
checkDense	logical indicating if dense (see denseMatrix) matrices should be considered at all. I.e., when false, the result will be sparse even when x is dense.

Details

The concept of unit triangular matrices with a diag slot of "U" stems from LAPACK.

Value

a triangular matrix of the same [class](#) but with a different diag slot. For diagU2N (semantically) with identical entries as x, whereas in diagN2U(x), the off-diagonal entries are unchanged and the diagonal is set to all 1 even if it was not previously.

Note

Such internal storage details should rarely be of relevance to the user. Hence, these functions really are rather *internal* utilities.

See Also

["diagonalMatrix"](#), ["triangularMatrix"](#), ["dtCMatrix"](#).

Examples

```
D <- Diagonal(7)
Dn <- diagU2N(D)
(iD <- Diagonal(7, 1L, keepInteger = TRUE))
iDn <- diagU2N(iD)
stopifnot(D @diag == "U", identical( D @x, numeric()),
          Dn@diag == "N", identical( Dn@x, rep(1, 7)),
          iDn@diag == "N", identical(iDn@x, rep(1L, 7)),
          identical( D, diagN2U( Dn))
          , identical(iD, diagN2U(iDn))
          )

set.seed(64); (Tr <- Matrix(rpois(49, 1/4), 7, 7)) # sparse enough
(T <- D + triu(Tr, k = 1)) # sparse triangular
(uT <- diagN2U(T)) # "unitriangular"
(t.u <- diagN2U(10*T))# changes the diagonal!
```

```

stopifnot(all(T == uT), diag(t.u) == 1, T@x == 1,
          identical(T, diagU2N(uT)))
T[upper.tri(T)] <- 5 # still "dtC"
T <- diagN2U(as(T,"triangularMatrix"))
inherits(T, "sparseMatrix") # TRUE
## checkDense = T / F :
dT <- as(T, "denseMatrix") # (unitriangular)
sT.n <- diagU2N(dT, checkDense = TRUE) # default
dT.n <- .diagU2N(dT, checkDense = FALSE) # default
stopifnot(is(dT.n, "denseMatrix"), is(sT.n, "sparseMatrix"),
          dT@diag == "U", dT.n@diag == "N", sT.n@diag == "N",
          all(dT == dT.n), all(dT == sT.n))

```

diff-methods

Difference a Vector

Description

TODO.

Usage

```

diff(x, ...)
## S4 method for signature 'sparseVector'
diff(x, lag = 1L, differences = 1L, ...)

```

Arguments

x	.
lag	.
differences	.
...	.

Examples

```

(M <- Matrix(c(0, 1, 0, 0), 6, 4))
diff(M)

```

Description

S4 methods defined in package **Matrix** for S4 generic functions `dim` and `dim<-`.

Usage

```
## S4 method for signature 'Matrix'
dim(x)
## S4 method for signature 'MatrixFactorization'
dim(x)

## S4 replacement method for signature 'denseMatrix,NULL'
dim(x) <- value
## S4 replacement method for signature 'denseMatrix,numeric'
dim(x) <- value
## S4 replacement method for signature 'sparseMatrix,NULL'
dim(x) <- value
## S4 replacement method for signature 'sparseMatrix,numeric'
dim(x) <- value
## S4 replacement method for signature 'sparseVector,numeric'
dim(x) <- value
```

Arguments

<code>x</code>	a Matrix object, a MatrixFactorization object, or a sparseVector object.
<code>value</code>	either <code>NULL</code> or a numeric vector of length 2. If it is numeric, then truncation towards zero by <code>as.integer</code> provides the new dimensions, which must be non-negative with product equal to the length of <code>x</code> .

Value

For `x` inheriting from virtual class **Matrix** or virtual class **MatrixFactorization**, `dim` returns an integer vector of length 2 storing the dimensions of the represented or factorized matrix. Replacement by numeric is a no-op if the old and new dimensions are equal. It otherwise adjusts the dimensions by initializing a new **Matrix** object with the new dimensions. The new object preserves the data type and the storage format (dense or sparse) of `x`, but not its structure nor its dimnames. Replacement by `NULL` gives the result of coercing `x` to vector or **sparseVector** (according to its storage format).

For `x` inheriting from virtual class **sparseVector**, `dim` returns `NULL`. Replacement by numeric constructs from `x` a **CsparseMatrix** object with the given dimensions. Replacement by `NULL` is a no-op.

See Also

`length`, `dimnames`.

Examples

```
x <- new("ddiMatrix", Dim = c(4L, 4L), x = c(-1, 2, -3, 4))
x
dim(x) <- NULL
x
dim(x) <- c(2L, 8L)
x
```

dimnames-methods

Dimnames of an Object

Description

S4 methods defined in package **Matrix** for S4 generic functions [dimnames](#) and [dimnames<-](#).

Usage

```
## S4 method for signature 'Matrix'
dimnames(x)
## S4 method for signature 'MatrixFactorization'
dimnames(x)
## S4 method for signature 'symmetricMatrix'
dimnames(x)

## S4 replacement method for signature 'Matrix,NULL'
dimnames(x) <- value
## S4 replacement method for signature 'Matrix,list'
dimnames(x) <- value
## S4 replacement method for signature 'MatrixFactorization,NULL'
dimnames(x) <- value
## S4 replacement method for signature 'MatrixFactorization,list'
dimnames(x) <- value
## S4 replacement method for signature 'generalMatrix,NULL'
dimnames(x) <- value
## S4 replacement method for signature 'generalMatrix,list'
dimnames(x) <- value
## S4 replacement method for signature 'symmetricMatrix,NULL'
dimnames(x) <- value
## S4 replacement method for signature 'symmetricMatrix,list'
dimnames(x) <- value
```

Arguments

x	a Matrix object or a MatrixFactorization object.
value	either NULL or a list of length 2 to be converted to valid dimnames following the conventions of the internal default method.

Value

For `x` inheriting from virtual class `Matrix` or virtual class `MatrixFactorization`, `dimnames` returns a list of length 2 storing the row and column names of the represented or factorized matrix. The components are `NULL` or character vectors of length equal to the respective dimensions of `x`. Replacement by list adjusts the `dimnames` by setting `x@Dimnames`.

For these objects, `dimnames` does not return `NULL`, and replacement by `NULL` sets the `dimnames` to `list(NULL, NULL)`.

Note

Typically, `dimnames(x)` is exactly `x@Dimnames`. An exception is made for formally symmetric or Hermitian `x`, for which symmetry of `dnx <- dimnames(x)` (in the sense of `identical(dnx[1], dnx[2])`) is required but symmetry of `x@Dimnames` is not. Indeed, for `x` inheriting from virtual class `symmetricMatrix`, `dimnames(x)` is the result of symmetrizing `x@Dimnames`. In the future, the definition of `symmetricMatrix` may be changed to require symmetry of the `Dimnames` slot, so that `dimnames` can be understood simply as an accessor of the slot.

See Also

`dim`, `length`.

Examples

```
mkDN <- function(dim)
  list(A = paste0("a", seq_len(dim[1L])),
       B = paste0("b", seq_len(dim[2L])))
chkDN <- function(x, dn, atdn = dn)
  stopifnot(identical(dimnames(x), dn), identical(x@Dimnames, atdn))

x <- new("dgeMatrix", Dim = c(2L, 3L), x = c(10, 8, 0, 6, 2, 4))
chkDN(x, NN <- list(NULL, NULL))
dimnames(x) <- AB <- mkDN(dim(x))
chkDN(x, AB)
dimnames(x) <- NULL
chkDN(x, NN)

s <- crossprod(x/10)
is(s, "symmetricMatrix") # TRUE
chkDN(s, NN)
dimnames(s) <- AB <- mkDN(dim(s))
chkDN(s, AB[c(2L, 2L)], AB) # dimnames are symmetric though slot is not
dimnames(s) <- NULL
chkDN(s, NN)
```

dimScale

*Scale the Rows and Columns of a Matrix***Description**

dimScale, rowScale, and colScale implement $D1 \%*\% x \%*\% D2$, $D \%*\% x$, and $x \%*\% D$ for diagonal matrices $D1$, $D2$, and D with diagonal entries $d1$, $d2$, and d , respectively. Unlike the explicit products, these functions preserve `dimnames(x)` and symmetry where appropriate.

Usage

```
dimScale(x, d1 = sqrt(1/diag(x, names = FALSE)), d2 = d1)
rowScale(x, d)
colScale(x, d)
```

Arguments

`x` a matrix, possibly inheriting from virtual class [Matrix](#).
`d1, d2, d` numeric vectors giving factors by which to scale the rows or columns of `x`; they are recycled as necessary.

Details

`dimScale(x)` (with `d1` and `d2` unset) is only roughly equivalent to `cov2cor(x)`. `cov2cor` sets the diagonal entries of the result to 1 (exactly); `dimScale` does not.

Value

The result of scaling `x`, currently always inheriting from virtual class [dMatrix](#).

It inherits from [triangularMatrix](#) if and only if `x` does. In the special case of `dimScale(x, d1, d2)` with identical `d1` and `d2`, it inherits from [symmetricMatrix](#) if and only if `x` does.

Author(s)

Mikael Jagan

See Also

[cov2cor](#)

Examples

```
n <- 6L
(x <- forceSymmetric(matrix(1, n, n)))
dimnames(x) <- rep.int(list(letters[seq_len(n)]), 2L)

d <- seq_len(n)
(D <- Diagonal(x = d))
```

```
(scx <- dimScale(x, d)) # symmetry and 'dimnames' kept
(mmx <- D %*% x %*% D) # symmetry and 'dimnames' lost
stopifnot(identical(unname(as(scx, "generalMatrix")), mmx))

rowScale(x, d)
colScale(x, d)
```

dMatrix-class

Virtual Class 'dMatrix' of Double Precision Matrices

Description

dMatrix is a virtual subclass of [Matrix](#) representing double precision matrices, whose data are stored using numeric vectors of type "double". ddenseMatrix and dsparseMatrix are virtual subclasses of dMatrix representing its intersections with [denseMatrix](#) and [sparseMatrix](#).

Slots

Dim, Dimnames inherited from virtual superclass [Matrix](#).

x a [double](#) vector storing matrix entries. Details such as *which* entries are stored vary by storage format.

Subclasses

The nonvirtual subclasses of dMatrix are listed below with links to virtual superclasses defining their structure and storage format.

Nonvirtual subclass	Structure	Storage format
dgeMatrix	generalMatrix	unpackedMatrix
dsyMatrix	symmetricMatrix	unpackedMatrix
dpoMatrix	posdefMatrix	unpackedMatrix
dtrMatrix	triangularMatrix	unpackedMatrix
dspMatrix	symmetricMatrix	packedMatrix
dppMatrix	posdefMatrix	packedMatrix
dtpMatrix	triangularMatrix	packedMatrix
dgCMatrix	generalMatrix	CsparseMatrix
dsCMatrix	symmetricMatrix	CsparseMatrix
dpCMatrix	posdefMatrix	CsparseMatrix
dtCMatrix	triangularMatrix	CsparseMatrix
dgRMatrix	generalMatrix	RsparseMatrix
dsRMatrix	symmetricMatrix	RsparseMatrix
dpRMatrix	posdefMatrix	RsparseMatrix
dtRMatrix	triangularMatrix	RsparseMatrix
dgTMatrix	generalMatrix	TsparseMatrix
dsTMatrix	symmetricMatrix	TsparseMatrix
dpTMatrix	posdefMatrix	TsparseMatrix

```
dtTMatrix   triangularMatrix   TsparseMatrix
ddiMatrix   diagonalMatrix     unnamed
```

Classes dpCMatrix, dpRMatrix, and dpTMatrix were not defined until **Matrix** version 1.8-0.

See Also

“Sister” classes [nMatrix](#), [lMatrix](#), [iMatrix](#), and [zMatrix](#) representing boolean, logical, integer, and complex matrices.

Examples

```
showClass("dMatrix")

set.seed(101)
round(Matrix(rnorm(28), 4, 7), 2)
M <- Matrix(rlnorm(56, sd = 10), 4, 14)
(M. <- zapsmall(M))
table(as.logical(M. == 0))

showClass("ddenseMatrix")
showMethods(class = "ddenseMatrix", where = "package:Matrix")

showClass("dsparseMatrix")
```

dmperm

Dulmage-Mendelsohn Permutation / Decomposition

Description

For any $n \times m$ (typically) sparse matrix x compute the Dulmage-Mendelsohn row and columns permutations which at first splits the n rows and m columns into coarse partitions each; and then a finer one, reordering rows and columns such that the permuted matrix is “as upper triangular” as possible.

Usage

```
dmperm(x, nAns = 6L, seed = 0L)
```

Arguments

<code>x</code>	a typically sparse matrix; internally coerced to either “ dgCMatrix ” or “ dtCMatrix ”.
<code>nAns</code>	an integer specifying the length of the resulting list . Must be 2, 4, or 6.
<code>seed</code>	an integer code in -1,0,1; determining the (initial) permutation; by default, seed = 0, no (or the identity) permutation; seed = -1 uses the “reverse” permutation k:1; for seed = 1, it is a <i>random</i> permutation (using R’s RNG, seed, etc).

Details

See the book section by Tim Davis; page 122–127, in the References.

Value

a named [list](#) with (by default) 6 components,

p	integer vector with the permutation p, of length nrow(x).
q	integer vector with the permutation q, of length ncol(x).
r	integer vector of length nb+1, where block k is rows r[k] to r[k+1]-1 in A[p,q].
s	integer vector of length nb+1, where block k is cols s[k] to s[k+1]-1 in A[p,q].
rr5	integer vector of length 5, defining the coarse row decomposition.
cc5	integer vector of length 5, defining the coarse column decomposition.

Author(s)

Martin Maechler, with a lot of “encouragement” by Mauricio Vargas.

References

Section 7.4 *Dulmage-Mendelsohn decomposition*, pp. 122 ff of
Timothy A. Davis (2006) *Direct Methods for Sparse Linear Systems*, SIAM Series “Fundamentals of Algorithms”.

See Also

[Schur](#), the class of permutation matrices; “[pMatrix](#)”.

Examples

```
set.seed(17)
(S9 <- rsparsematrix(9, 9, nnz = 10, symmetric=TRUE)) # dsCMatrix
str( dm9 <- dmperm(S9) )
(S9p <- with(dm9, S9[p, q]))
## looks good, but *not* quite upper triangular; these, too:
str( dm9.0 <- dmperm(S9, seed=-1)) # non-random too.
str( dm9_1 <- dmperm(S9, seed= 1)) # a random one
## The last two permutations differ, but have the same effect!
(S9p0 <- with(dm9.0, S9[p, q])) # .. hmm ..
stopifnot(all.equal(S9p0, S9p))# same as as default, but different from the random one
```

```
set.seed(11)
(M <- triu(rsparsematrix(9,11, 1/4)))
dM <- dmperm(M); with(dM, M[p, q])
(Mp <- M[sample.int(nrow(M)), sample.int(ncol(M))])
dMp <- dmperm(Mp); with(dMp, Mp[p, q])
```

```

set.seed(7)
(n7 <- rsparsematrix(5, 12, nnz = 10, rand.x = NULL))
str( dm.7 <- dmperm(n7) )
stopifnot(exprs = {
  lengths(dm.7[1:2]) == dim(n7)
  identical(dm.7, dmperm(as(n7, "dMatrix")))
  identical(dm.7[1:4], dmperm(n7, nAns=4))
  identical(dm.7[1:2], dmperm(n7, nAns=2))
})

```

drop-methods

Drop Redundant Extent Information

Description

S4 methods defined in package **Matrix** for S4 generic function `drop`, generalizing `base::drop`.

Usage

```

## S4 method for signature 'Matrix'
drop(x)
## S4 method for signature 'sparseVector'
drop(x)

```

Arguments

`x` a [Matrix](#) object or a [sparseVector](#) object.

Value

The return value is `x` except if `x` is a [Matrix](#) object with exactly one row or exactly one column. In that case, the return value is the traditional vector obtained by coercing `x` to a traditional matrix and dispatching the default method, namely `base::drop`.

See Also

Generic function `[` for subsetting, which might drop redundant dimensions.

Function `drop0` for deleting small (on an absolute scale) entries of sparse format objects.

Examples

```

m <- new("dgRMatrix",
  Dim = c(3L, 4L),
  Dimnames = list(A = paste0("a", 1L:3L), B = paste0("b", 1L:4L)),
  p = c(0L, 2L, 3L, 4L), j = c(1L, 2L, 0L, 3L), x = c(1, 2, 4, 8))
m
v <- as(m, "sparseVector")
v
stopifnot(identical(drop(m), m),

```

```

identical(drop(v), v))

drop(m[1L, , drop = FALSE]) # names are colnames(m)
drop(m[, 1L, drop = FALSE]) # names are rownames(m)
drop(m[1L, 1L, drop = FALSE]) # names are NULL
rownames(m) <- NULL
drop(m[1L, 1L, drop = FALSE]) # names are colnames(m)[1L]

```

drop0

Drop Non-Structural Zeros from a Sparse Matrix

Description

Deletes “non-structural” zeros (i.e., zeros stored explicitly, in memory) from a sparse matrix and returns the result.

Usage

```
drop0(x, tol = 0, is.Csparse = NA, give.Csparse = TRUE)
```

Arguments

x	a Matrix , typically inheriting from virtual class sparseMatrix . denseMatrix and traditional vectors and matrices are coerced to CsparseMatrix , with zeros dropped automatically, hence users passing such x should consider <code>as(x, "CsparseMatrix")</code> instead, notably in the <code>tol = 0</code> case.
tol	a non-negative number. If x is numeric, then entries less than or equal to tol in absolute value are deleted.
is.Csparse	a logical used only if <code>give.Csparse</code> is TRUE, indicating if x already inherits from virtual class CsparseMatrix , in which case coercion is not attempted, permitting some (typically small) speed-up.
give.Csparse	a logical indicating if the result must inherit from virtual class CsparseMatrix . If FALSE and x inherits from RsparseMatrix , TsparseMatrix , or indMatrix , then the result preserves the class of x. The default value is TRUE only for backwards compatibility.

Value

A [sparseMatrix](#), the result of deleting non-structural zeros from x, possibly after coercion.

Note

`drop0` is sometimes called in conjunction with [zapsmall](#), e.g., when dealing with sparse matrix products; see the example.

See Also

Function [sparseMatrix](#), for constructing objects inheriting from virtual class [sparseMatrix](#); [nnzero](#).

Examples

```
(m <- sparseMatrix(i = 1:8, j = 2:9, x = c(0:2, 3:-1),
                  dims = c(10L, 20L)))
drop0(m)

## A larger example:
t5 <- new("dtCMatrix", Dim = c(5L, 5L), uplo = "L",
        x = c(10, 1, 3, 10, 1, 10, 1, 10, 10),
        i = c(0L,2L,4L, 1L, 3L,2L,4L, 3L, 4L),
        p = c(0L, 3L, 5L, 7:9))
TT <- kronecker(t5, kronecker(kronecker(t5, t5), t5))
IT <- solve(TT)
I. <- TT %*% IT ; nnzero(I.) # 697 ( == 625 + 72 )
I.0 <- drop0(zapsmall(I.))
## which actually can be more efficiently achieved by
I.. <- drop0(I., tol = 1e-15)
stopifnot(all(I.0 == Diagonal(625)), nnzero(I..) == 625)
```

expand-methods

Expand Matrix Factorizations

Description

expand1 and expand2 construct matrix factors from objects specifying matrix factorizations. Such objects typically do not store the factors explicitly, employing instead a compact representation to save memory.

Usage

```
expand1(x, which, ...)
expand2(x, ...)

expand (x, ...)
```

Arguments

x	a matrix factorization, typically inheriting from virtual class MatrixFactorization .
which	a character string indicating a matrix factor.
...	further arguments passed to or from methods.

Details

Methods for expand are retained only for backwards compatibility with **Matrix** < 1.6-0. New code should use expand1 and expand2, whose methods provide more control and behave more consistently. Notably, expand2 obeys the rule that the product of the matrix factors in the returned list should reproduce (within some tolerance) the factorized matrix, *including* its dimnames.

Hence if x is a matrix and y is its factorization, then

```
all.equal(as(x, "matrix"), as(Reduce(`*`, expand2(y)), "matrix"))
```

should in most cases return TRUE.

Value

expand1 returns an object inheriting from virtual class **Matrix**, representing the factor indicated by which, always without row and column names.

expand2 returns a list of factors, typically with names using conventional notation, as in `list(L=, U=)`. The first and last factors get the row and column names of the factorized matrix, which are preserved in the `Dimnames` slot of `x`.

Methods

The following table lists methods for `expand1` together with allowed values of argument `which`.

class(x)	which
Schur	c("Q", "T", "Q.")
denseLU	c("P1", "P1.", "L", "U")
sparseLU	c("P1", "P1.", "P2", "P2.", "L", "U")
sparseQR	c("P1", "P1.", "P2", "P2.", "Q", "Q1", "R", "R1")
BunchKaufman, pBunchKaufman	c("U", "DU", "U.", "L", "DL", "L.")
Cholesky, pCholesky	c("P1", "P1.", "L1", "D", "L1.", "L", "L.")
CHMsimpl, CHMsimpl	c("P1", "P1.", "L1", "D", "L1.", "L", "L.")

Methods for `expand2` and `expand` are described below. Factor names and classes apply also to `expand1`.

`expand2 signature(x = "CHMsimpl")`: expands the factorization $A = P_1' L_1 D L_1' P_1 = P_1' L L' P_1$ as `list(P1., L1, D, L1., P1)` (the default) or as `list(P1., L, L., P1)`, depending on optional logical argument `LDL`. `P1` and `P1.` are **pMatrix**, `L1`, `L1.`, `L`, and `L.` are **dtMatrix**, and `D` is a **ddiMatrix**.

`expand2 signature(x = "CHMsupper")`: as `CHMsimpl`, but the triangular factors are stored as **dgMatrix**.

`expand2 signature(x = "p?Cholesky")`: expands the factorization $A = L_1 D L_1' = L L'$ as `list(L1, D, L1.)` (the default) or as `list(L, L.)`, depending on optional logical argument `LDL`. `L1`, `L1.`, `L`, and `L.` are **dtrMatrix** or **dtpMatrix**, and `D` is a **ddiMatrix**.

`expand2 signature(x = "p?BunchKaufman")`: expands the factorization $A = U D_U U' = L D_L L'$ where $U = \prod_{k=1}^{b_U} P_k U_k$ and $L = \prod_{k=1}^{b_L} P_k L_k$ as `list(U, DU, U.)` or `list(L, DL, L.)`, depending on `x@uplo`. If optional argument `complete` is TRUE, then an unnamed list giving the full expansion with $2b_U + 1$ or $2b_L + 1$ matrix factors is returned instead. P_k are represented as **pMatrix**, U_k and L_k are represented as **dtMatrix**, and D_U and D_L are represented as **dsMatrix**.

`expand2 signature(x = "Schur")`: expands the factorization $A = Q T Q'$ as `list(Q, T, Q.)`. `Q` and `Q.` are `x@Q` and `t(x@Q)` modulo `Dimnames`, and `T` is `x@T`.

`expand2 signature(x = "sparseLU")`: expands the factorization $A = P_1' L U P_2'$ as `list(P1., L, U, P2.)`. `P1.` and `P2.` are **pMatrix**, and `L` and `U` are **dtMatrix**.

`expand2 signature(x = "denseLU")`: expands the factorization $A = P_1'LU$ as `list(P1., L, U)`. `P1.` is a `pMatrix`, and `L` and `U` are `dtrMatrix` if square and `dgeMatrix` otherwise.

`expand2 signature(x = "sparseQR")`: expands the factorization $A = P_1'QR P_2' = P_1'Q_1R_1P_2'$ as `list(P1., Q, R, P2.)` or `list(P1., Q1, R1, P2.)`, depending on optional logical argument `complete`. `P1.` and `P2.` are `pMatrix`, `Q` and `Q1` are `dgeMatrix`, `R` is a `dgCMatrix`, and `R1` is a `dtCMatrix`.

`expand signature(x = "CHMfactor")`: as `expand2`, but returning `list(P, L)`. `expand(x)[["P"]]` and `expand2(x)[["P1."]]` represent the same permutation matrix P_1 but have opposite margin slots and inverted perm slots. The components of `expand(x)` do not preserve `x@Dimnames`.

`expand signature(x = "sparseLU")`: as `expand2`, but returning `list(P, L, U, Q)`. `expand(x)[["Q"]]` and `expand2(x)[["P2."]]` represent the same permutation matrix P_2' but have opposite margin slots and inverted perm slots. `expand(x)[["P"]]` represents the permutation matrix P_1 rather than its transpose P_1' ; it is `expand2(x)[["P1."]]` with an inverted perm slot. `expand(x)[["L"]]` and `expand2(x)[["L"]]` represent the same unit lower triangular matrix L , but with diag slot equal to "N" and "U", respectively. `expand(x)[["L"]]` and `expand(x)[["U"]]` store the *permuted* first and second components of `x@Dimnames` in their `Dimnames` slots.

`expand signature(x = "denseLU")`: as `expand2`, but returning `list(L, U, P)`. `expand(x)[["P"]]` and `expand2(x)[["P1."]]` are identical modulo `Dimnames`. The components of `expand(x)` do not preserve `x@Dimnames`.

See Also

The virtual class `MatrixFactorization` of matrix factorizations.

Generic functions `Cholesky`, `BunchKaufman`, `Schur`, `lu`, and `qr` for *computing* factorizations.

Examples

```
showMethods("expand1", inherited = FALSE)
showMethods("expand2", inherited = FALSE)
set.seed(0)

(A <- Matrix(rnorm(9L, 0, 10), 3L, 3L))
(lu.A <- lu(A))
(e.lu.A <- expand2(lu.A))
stopifnot(exprs = {
  is.list(e.lu.A)
  identical(names(e.lu.A), c("P1.", "L", "U"))
  all(sapply(e.lu.A, is, "Matrix"))
  all.equal(as(A, "matrix"), as(Reduce("%*%", e.lu.A), "matrix"))
})

## 'expand1' and 'expand2' give equivalent results modulo
## dimnames and representation of permutation matrices;
## see also function 'alt' in example("Cholesky-methods")
(a1 <- sapply(names(e.lu.A), expand1, x = lu.A, simplify = FALSE))
all.equal(a1, e.lu.A)

## see help("denseLU-class") and others for more examples
```

Description

Compute the exponential of a matrix.

Usage

```
expm(x)
```

Arguments

`x` a matrix, typically inheriting from the `dMatrix` class.

Details

The exponential of a matrix is defined as the infinite Taylor series $\text{expm}(A) = I + A + A^2/2! + A^3/3! + \dots$ (although this is definitely not the way to compute it). The method for the `dgeMatrix` class uses Ward's diagonal Pade' approximation with three step preconditioning, a recommendation from Moler & Van Loan (1978) "Nineteen dubious ways...".

Value

The matrix exponential of `x`.

Author(s)

This is a translation of the implementation of the corresponding Octave function contributed to the Octave project by A. Scottedward Hodel <A.S.Hodel@Eng.Auburn.EDU>. A bug in there has been fixed by Martin Maechler.

References

https://en.wikipedia.org/wiki/Matrix_exponential

Cleve Moler and Charles Van Loan (2003) Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review* **45**, 1, 3–49. doi:10.1137/S00361445024180

for historical reference mostly:

Moler, C. and Van Loan, C. (1978) Nineteen dubious ways to compute the exponential of a matrix. *SIAM Review* **20**, 4, 801–836. doi:10.1137/1020098

Eric W. Weisstein et al. (1999) *Matrix Exponential*. From MathWorld, <https://mathworld.wolfram.com/MatrixExponential.html>

See Also

Package **expm**, which provides newer (in some cases faster, more accurate) algorithms for computing the matrix exponential via its own (non-generic) function `expm()`. **expm** also implements `logm()`, `sqrtm()`, etc.

Generic function [Schur](#).

Examples

```
(m1 <- Matrix(c(1,0,1,1), ncol = 2))
(e1 <- expm(m1)) ; e <- exp(1)
stopifnot(all.equal(e1@x, c(e,0,e,e), tolerance = 1e-15))
(m2 <- Matrix(c(-49, -64, 24, 31), ncol = 2))
(e2 <- expm(m2))
(m3 <- Matrix(cbind(0, rbind(6*diag(3), 0)))) # sparse!
(e3 <- expm(m3)) # upper triangular
```

externalFormats

*Read and write external matrix formats***Description**

Read matrices stored in the Harwell-Boeing or MatrixMarket formats or write `sparseMatrix` objects to one of these formats.

Usage

```
readHB(file)
readMM(file)
writeMM(obj, file, ...)
```

Arguments

<code>obj</code>	a real sparse matrix
<code>file</code>	for <code>writeMM</code> - the name of the file to be written. For <code>readHB</code> and <code>readMM</code> the name of the file to read, as a character scalar. The names of files storing matrices in the Harwell-Boeing format usually end in <code>".rua"</code> or <code>".rsa"</code> . Those storing matrices in the MatrixMarket format usually end in <code>".mtx"</code> . Alternatively, <code>readHB</code> and <code>readMM</code> accept connection objects.
<code>...</code>	optional additional arguments. Currently none are used in any methods.

Value

The `readHB` and `readMM` functions return an object that inherits from the `"Matrix"` class. Methods for the `writeMM` generic functions usually return `NULL` and, as a side effect, the matrix `obj` is written to file in the MatrixMarket format (`writeMM`).

Note

The Harwell-Boeing format is older and less flexible than the MatrixMarket format. The function `writeHB` was deprecated and has now been removed. Please use `writeMM` instead.

Note that these formats do *not* know anything about `dimnames`, hence these are dropped by `writeMM()`.

A very simple way to export small sparse matrices `S`, is to use `summary(S)` which returns a `data.frame` with columns `i`, `j`, and possibly `x`, see summary in `sparseMatrix-class`, and an example below.

References

<https://math.nist.gov/MatrixMarket/>

<https://sparse.tamu.edu/>

Examples

```
str(pores <- readMM(system.file("external/pores_1.mtx", package = "Matrix")))
str(utm <- readHB(system.file("external/utm300.rua", package = "Matrix")))
str(lundA <- readMM(system.file("external/lund_a.mtx", package = "Matrix")))
str(lundA <- readHB(system.file("external/lund_a.rsa", package = "Matrix")))
## https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/counterx/counterx.htm
str(jgl <- readMM(system.file("external/jgl009.mtx", package = "Matrix")))

## NOTE: The following examples take quite some time
## ---- even on a fast internet connection:
if(FALSE) {
  ## The URL has been corrected, but we need an untar step:
  u. <- url("https://www.cise.ufl.edu/research/sparse/RB/Boeing/msc00726.tar.gz")
  str(sm <- readHB(gzcon(u.)))
}

data(KNex, package = "Matrix")
## Store as MatrixMarket (".mtx") file, here inside temporary dir./folder:
(MMfile <- file.path(tempdir(), "mmMM.mtx"))
writeMM(KNex$mm, file=MMfile)
file.info(MMfile)[c("size", "ctime")] # (some confirmation of the file's)

## very simple export - in triplet format - to text file:
data(CAex, package = "Matrix")
s.CA <- summary(CAex)
s.CA # shows (i, j, x) [columns of a data frame]
message("writing to ", outf <- tempfile())
write.table(s.CA, file = outf, row.names=FALSE)
## and read it back -- showing off sparseMatrix():
str(dd <- read.table(outf, header=TRUE))
## has columns (i, j, x) -> we can use via do.call() as arguments to sparseMatrix():
mm <- do.call(sparseMatrix, dd)
stopifnot(all.equal(mm, CAex, tolerance=1e-15))
```

Description

Multiplies a matrix or vector on the left or right by a factor from a matrix factorization or its transpose.

Usage

```
facmul(x, factor, y, trans = FALSE, left = TRUE, ...)
```

Arguments

<code>x</code>	a MatrixFactorization object.
<code>factor</code>	a character string indicating a factor in the factorization represented by <code>x</code> , typically an element of <code>names(expand2(x, ...))</code> .
<code>y</code>	a matrix or vector to be multiplied on the left or right by the factor or its transpose.
<code>trans</code>	a logical indicating if the transpose of the factor should be used, rather than the factor itself.
<code>left</code>	a logical indicating if the <code>y</code> should be multiplied on the left by the factor, rather than on the right.
<code>...</code>	further arguments passed to or from methods.

Details

`facmul` is experimental and currently no methods are exported from **Matrix**.

Value

The value of `op(M) %*% y` or `y %*% op(M)`, depending on `left`, where `M` is the factor (always *without* `dimnames`) and `op(M)` is `M` or `t(M)`, depending on `trans`.

Examples

```
## Conceptually, methods for 'facmul' _would_ behave as follows ...
## Not run:
n <- 3L
x <- lu(Matrix(rnorm(n * n), n, n))
y <- rnorm(n)
L <- unname(expand2(x)[[nm <- "L"]])
stopifnot(exprs = {
  all.equal(facmul(x, nm, y, trans = FALSE, left = TRUE), L %*% y)
  all.equal(facmul(x, nm, y, trans = FALSE, left = FALSE), y %*% L)
  all.equal(facmul(x, nm, y, trans = TRUE, left = TRUE), crossprod(L, y))
  all.equal(facmul(x, nm, y, trans = TRUE, left = FALSE), tcrossprod(y, L))
})
```

```

}))

## End(Not run)

```

fastMisc

“Low Level” Coercions and Methods

Description

“Semi-API” functions used internally by **Matrix**, often to bypass S4 dispatch and avoid the associated overhead. These are exported to provide this capability to expert users. Typical users should continue to rely on S4 generic functions to dispatch suitable methods, by calling, e.g., `as(., <class>)` for coercions.

Usage

```

.M2kind(from, kind = ".", sparse = NA)

.M2gen(from, kind = ".")
.M2sym(from, uplo = NULL, trans = "C", ...)
.M2tri(from, uplo = NULL)
.M2diag(from)

.M2v(from)
.M2m(from)
.M2unpacked(from)
.M2packed(from)
.M2C(from)
.M2R(from)
.M2T(from)

.M2V(from)
.m2V(from, kind = ".")

.sparse2dense(from, packed = FALSE)
.diag2dense(from, kind = ".", shape = "t", packed = FALSE,
             uplo = "U", trans = "T")
.ind2dense(from, kind = "n")
.m2dense(from, class = ".ge",
          uplo = "U", trans = "C", diag = "N", margin = 2L)

.dense2sparse(from, repr = "C")
.diag2sparse(from, kind = ".", shape = "t", repr = "C",
             uplo = "U", trans = "T")
.ind2sparse(from, kind = "n", repr = ".")
.m2sparse(from, class = ".gC",
          uplo = "U", trans = "C", diag = "N", margin = 2L)

```

```
.diag.dsC(x, Chx = Cholesky(x, LDL = TRUE), res.kind = "diag")

.solve.dgC.lu (a, b, tol = .Machine$double.eps, check = TRUE)
.solve.dgC.qr (a, b, order = 3L, check = TRUE)
.solve.dgC.chol(a, b, check = TRUE)

.updateCHMfactor(object, parent, mult = 0)
```

Arguments

from, x, a, b	a Matrix (for <code>.M2*()</code> et al.), matrix (for <code>.m2*()</code> et al.), or vector.
kind	a string (" <code>.</code> ", " <code>,</code> ", " <code>n</code> ", " <code>l</code> ", or " <code>d</code> ") specifying the “kind” of the result. " <code>.</code> " indicates that the kind of from should be preserved. " <code>,</code> " is equivalent to " <code>z</code> " if from is complex and to " <code>d</code> " otherwise. " <code>n</code> " indicates that the result should inherit from nMatrix or nsparseVector (and so on).
shape	a string (" <code>.</code> ", " <code>g</code> ", " <code>s</code> ", or " <code>t</code> ") specifying the “shape” of the result. " <code>.</code> " indicates that the shape of from should be preserved. " <code>g</code> " indicates that the result should inherit from generalMatrix (and so on).
repr	a string (" <code>.</code> ", " <code>C</code> ", " <code>R</code> ", or " <code>T</code> ") specifying the sparse representation of the result. " <code>.</code> " is accepted only by <code>.ind2sparse</code> and indicates the most efficient representation, which is " <code>C</code> " (" <code>R</code> ") for <code>margin = 2 (1)</code> . " <code>C</code> " indicates that the result should inherit from CsparseMatrix (and so on).
packed	a logical indicating if the result should inherit from packedMatrix rather than from unpackedMatrix . It is ignored for from inheriting from generalMatrix .
sparse	a logical indicating if the result should inherit from sparseMatrix rather than from denseMatrix . If NA, then the result will be formally sparse if and only if from is.
uplo	a string (" <code>U</code> " or " <code>L</code> ") indicating whether the result (if symmetric or triangular) should store the upper or lower triangle of from. The elements of from in the opposite triangle are ignored.
trans	a string (" <code>C</code> " or " <code>T</code> ") indicating whether the result (if symmetric) should formally equal its conjugate transpose or transpose. It matters only in the complex case.
diag	a string (" <code>N</code> " or " <code>U</code> ") indicating whether the result (if triangular) should be formally nonunit or unit triangular. In the unit triangular case, the diagonal elements of from are ignored.
margin	an integer (1 or 2) indicating if the result should be a 1-row matrix or a 1-column matrix in the case where from is a vector but not a matrix.
class	a string whose first three characters specify the class of the result. It should match the pattern " <code>^[.nld](ge sy tr sp tp)"</code> for <code>.m2dense</code> and " <code>^[.nld][gst][CRT]"</code> for <code>.m2sparse</code> , where " <code>.</code> " in the first position is equivalent to " <code>l</code> " for logical arguments and " <code>d</code> " for numeric arguments.
...	optional arguments passed to isSymmetric .
Chx	optionally, the Cholesky (x, ...) factorization of x. If supplied, then x is unused.

res.kind	a string in c("trace", "sumLog", "prod", "min", "max", "range", "diag", "diagBack").
tol	see lu-methods .
order	see qr-methods .
check	a logical indicating if the first argument should be tested for inheritance from dgCMatrix and coerced if necessary. Set to FALSE for speed only if it is known to already inherit from dgCMatrix .
object	a Cholesky factorization inheriting from virtual class CHMfactor, almost always the result of a call to generic function Cholesky .
parent	an object of class dsCMatrix or class dgCMatrix .
mult	a numeric vector of postive length. Only the first element is used, and that must be finite.

Details

Functions with names of the form `.<A>2<B>` implement coercions from virtual class A to the “nearest” non-virtual subclass of virtual class B, where the virtual classes are abbreviated as follows:

M [Matrix](#)
V [sparseVector](#)
m matrix
v vector
dense [denseMatrix](#)
unpacked [unpackedMatrix](#)
packed [packedMatrix](#)
sparse [CsparseMatrix](#), [RsparseMatrix](#), or [TsparseMatrix](#)
C [CsparseMatrix](#)
R [RsparseMatrix](#)
T [TsparseMatrix](#)
gen [generalMatrix](#)
sym [symmetricMatrix](#)
tri [triangularMatrix](#)
diag [diagonalMatrix](#)
ind [indMatrix](#)

Abbreviations should be seen as a guide, rather than as an exact description of behaviour. Notably, `.m2dense`, `.m2sparse`, and `.m2V` accept vectors that are not matrices.

`.diag.dsC(x)`: `.diag.dsC` computes (or uses if `Chx` is supplied) the Cholesky factorization of `x` as LDL' in order to calculate one of several possible statistics from the diagonal entries of D . See `res.kind` under ‘Arguments’.

`.solve.dgC.*(a, b)`: `.solve.dgC.lu(a, b)` needs a square matrix `a`. `.solve.dgC.qr(a, b)` needs a “long” matrix `a`, with `nrow(a) >= ncol(a)`. `.solve.dgC.chol(a, b)` needs a “wide” matrix `a`, with `nrow(a) <= ncol(a)`.

All three may be used to solve sparse linear systems directly. Only `.solve.dgC.qr` and `.solve.dgC.chol` be used to solve sparse *least squares* problems.

`.updateCHMfactor(object, parent, mult)`: `.updateCHMfactor` updates `object` with the result of Cholesky factorizing $F(\text{parent}) + \text{mult}[1] * \text{diag}(\text{nrow}(\text{parent}))$, i.e., $F(\text{parent})$ plus `mult[1]` times the identity matrix, where $F = \text{identity}$ if `parent` is a `dsCMatrix` and $F = \text{tcrossprod}$ if `parent` is a `dgCMatrix`. The nonzero pattern of $F(\text{parent})$ must match that of `S` if `object = Cholesky(S, ...)`.

Examples

```
D. <- diag(x = c(1, 1, 2, 3, 5, 8))
D.0 <- Diagonal(x = c(0, 0, 0, 3, 5, 8))
S. <- toeplitz(as.double(1:6))
C. <- new("dgCMatrix", Dim = c(3L, 4L),
          p = c(0L, 1L, 1L, 1L, 3L), i = c(1L, 0L, 2L), x = c(-8, 2, 3))

stopifnot(exprs = {
  identical(.M2sym(D.), as(D., "symmetricMatrix"))
  identical(.M2tri(D.), as(D., "triangularMatrix"))
  identical(.M2diag(D.), as(D., "diagonalMatrix"))
  identical(.M2kind(C., "l"),
            as(C., "lMatrix"))
  identical(.M2kind(.sparse2dense(C.), "l"),
            as(as(C., "denseMatrix"), "lMatrix"))
  identical(.diag2sparse(D.0, ".", "t", "C"),
            .dense2sparse(.diag2dense(D.0, ".", "t", TRUE), "C"))
  identical(.M2gen(.diag2dense(D.0, ".", "s", FALSE)),
            .sparse2dense(.M2gen(.diag2sparse(D.0, ".", "s", "T"))))
  identical(S.,
            .M2m(.m2sparse(S., ".sR", uplo = "U", trans = "C")))
  identical(S. * lower.tri(S.) + diag(1, 6L),
            .M2m(.m2dense(S., ".tr", uplo = "L", diag = "U")))
  identical(.M2R(C.), .M2R(.M2T(C.)))
  identical(.tCRT(C.), .M2R(t(C.)))
})

A <- tcrossprod(C.)/6 + Diagonal(3, 1/3); A[1,2] <- 3; A
stopifnot(exprs = {
  is.numeric(x. <- c(2.2, 0, -1.2))
  all.equal(x., .solve.dgC.lu(A, c(1,0,0), check=FALSE))
  all.equal(x., .solve.dgC.qr(A, c(1,0,0), check=FALSE))
})

## Solving sparse least squares:

X <- rbind(A, Diagonal(3)) # design matrix X (for L.S.)
Xt <- t(X)                # *transposed* X (for L.S.)
(y <- drop(crossprod(Xt, 1:3)) + c(-1,1)/1000) # small rand.err.
```

```

str(solveCh <- .solve.dgC.chol(Xt, y, check=FALSE)) # Xt *is* dgC..
stopifnot(exprs = {
  all.equal(solveCh$coef, 1:3, tol = 1e-3) # rel.err ~ 1e-4
  all.equal(solveCh$coef, drop(solve(tcrossprod(Xt), Xt %% y)))
  all.equal(solveCh$coef, .solve.dgC.qr(X, y, check=FALSE))
})

```

forceCanonical-methods

Canonicalize an Object

Description

Obtain a “canonical” representation of an R object as defined by corresponding methods for [isCanonical](#).

Usage

```
forceCanonical(x, ...)
```

Arguments

<code>x</code>	an R object.
<code>...</code>	optional arguments passed to methods.

Details

Method writers should ensure that canonicalization is idempotent. In other words, $f(f(x))$ and $f(x)$ should be identical if $f = \text{forceCanonical}$.

Value

A canonical object, typically of class `class(x)` or a superclass, such that `isCanonical(.)` is TRUE.

See Also

[isCanonical](#).

Examples

```

utils::example(isCanonical, package = "Matrix")
stopifnot(identical(unname(obj), lapply(unname(obj.), forceCanonical)))

```

forceDiagonal-methods *Create diagonalMatrix From the Diagonal Part of a Matrix*

Description

From a matrix or [Matrix](#) `x`, use the diagonal part, `diag(x)` to create a corresponding [diagonalMatrix](#).

Usage

```
forceDiagonal(x, ...)

## S4 method for signature 'matrix'
forceDiagonal(x, diag = NULL, ...)
```

Arguments

<code>x</code>	a matrix or Matrix .
<code>diag</code>	a string, "U" or "N" or <code>NULL</code> , determining the <code>diag</code> slot of the result; if <code>NULL</code> , as by default, the code checks for unit diagonal and uses "U" in that case.
<code>...</code>	optional arguments passed from or to other methods.

Value

a [Matrix](#) inheriting from "[diagonalMatrix](#)", even when `x` is not diagonal.

See Also

`as(x, "diagonalMatrix")` does *not* force, but signal an error when `x` is not diagonal, see [isDiagonal](#).

Examples

```
forceDiagonal(diag(3)) # --> <ddiMatrix>
forceDiagonal(matrix(1:4, 2))# a <idiMatrix>
stopifnot(identical(forceDiagonal(Matrix(1:4, 2)),
                     Diagonal(x = c(1, 4))))
```

forceSymmetric-methods

Symmetrize a Square Matrix

Description

Returns the symmetric matrix that results from reflecting one triangle of a square matrix onto the other.

Usage

```
forceSymmetric(x, ...)

## S4 method for signature 'denseMatrix'
forceSymmetric(x, uplo = NULL, trans = "C", ...)
## S4 method for signature 'CsparseMatrix'
forceSymmetric(x, uplo = NULL, trans = "C", ...)
## S4 method for signature 'RsparseMatrix'
forceSymmetric(x, uplo = NULL, trans = "C", ...)
## S4 method for signature 'TsparseMatrix'
forceSymmetric(x, uplo = NULL, trans = "C", ...)
## S4 method for signature 'diagonalMatrix'
forceSymmetric(x, uplo = "U", trans = "C", ...)
## S4 method for signature 'indMatrix'
forceSymmetric(x, ...)
## S4 method for signature 'matrix'
forceSymmetric(x, uplo = "U", trans = "C", ...)
```

Arguments

x	a square matrix or Matrix .
uplo	a character string, "U" or "L", indicating which of the upper and lower triangles should be reflected. Where it is the default, NULL is accepted and equivalent to "U", unless x has an uplo slot, in which case that is used instead.
trans	a character string, "C" or "T", indicating either the conjugate transpose or transpose, i.e., whether the result is formally Hermitian or symmetric. It matters only in the complex case. The default is "C", for consistency with methods for isSymmetric .
...	optional arguments passed from or to other methods.

Details

forceSymmetric(x) can be understood as a coercion to virtual class [symmetricMatrix](#). However, unlike as(x, "symmetricMatrix"), methods for forceSymmetric do *not* require that x is already approximately symmetric.

Value

An object inheriting from virtual class [symmetricMatrix](#), representing a Hermitian or symmetric matrix.

See Also

[isSymmetric](#), [symmpart](#), [skewpart](#).

Examples

```
## Hilbert matrix
i <- 1:6
h6 <- 1/outer(i - 1L, i, "+")
sd <- sqrt(diag(h6))
hh <- t(h6/sd)/sd # theoretically symmetric
isSymmetric(hh, tol=0) # FALSE; hence
try( as(hh, "symmetricMatrix") ) # fails, but this works fine:
H6 <- forceSymmetric(hh)

## result can be pretty surprising:
(M <- Matrix(1:36, 6))
forceSymmetric(M) # symmetric, hence very different in lower triangle
(tm <- tril(M))
forceSymmetric(tm)
```

forceTriangular-methods

Get the Upper or Lower Triangular Part of a Square Matrix

Description

TODO.

Usage

```
forceTriangular(x, ...)
```

Arguments

x a square matrix or [Matrix](#).
... optional arguments passed from or to other methods.

formatSparseM

Formatting Sparse Numeric Matrices Utilities

Description

Utilities for formatting sparse numeric matrices in a flexible way. These functions are used by the [format](#) and print methods for sparse matrices and can be applied as well to standard R matrices. Note that *all* arguments but the first are optional.

formatSparseM() is the main “workhorse” of [formatSpMatrix](#), the format method for sparse matrices.

.formatSparseSimple() is a simple helper function, also dealing with (short/empty) column names construction.

Usage

```
formatSparseM(x, zero.print = ".", align = c("fancy", "right"),
              m = as(x, "matrix"), asLogical=NULL, uniDiag=FALSE,
              digits=NULL, cx, iN0, dn = dimnames(m))

.formatSparseSimple(m, asLogical=FALSE, digits=NULL,
                   col.names, note.dropping.colnames = TRUE,
                   dn=dimnames(m))
```

Arguments

<code>x</code>	an R object inheriting from class sparseMatrix .
<code>zero.print</code>	character which should be used for <i>structural</i> zeroes. The default "." may occasionally be replaced by " " (blank); using "0" would look almost like <code>print()</code> ing of non-sparse matrices.
<code>align</code>	a string specifying how the <code>zero.print</code> codes should be aligned, see formatSpMatrix .
<code>m</code>	(optional) a (standard R) matrix version of <code>x</code> .
<code>asLogical</code>	should the matrix be formatted as a logical matrix (or rather as a numeric one); mostly for <code>formatSparseM()</code> .
<code>uniDiag</code>	logical indicating if the diagonal entries of a sparse unit triangular or unit-diagonal matrix should be formatted as "I" instead of "1" (to emphasize that the 1's are "structural").
<code>digits</code>	significant digits to use for printing, see print.default .
<code>cx</code>	(optional) character matrix; a formatted version of <code>x</code> , still with strings such as "0.00" for the zeros.
<code>iN0</code>	(optional) integer vector, specifying the location of the <i>non</i> -zeroes of <code>x</code> .
<code>col.names, note.dropping.colnames</code>	see formatSpMatrix .
<code>dn</code>	dimnames to be used; a list (of length two) with row and column names (or <code>NULL</code>).

Value

a character matrix like `cx`, where the zeros have been replaced with (padded versions of) `zero.print`. As this is a *dense* matrix, do not use these functions for really large (really) sparse matrices!

Author(s)

Martin Maechler

See Also

[formatSpMatrix](#) which calls `formatSparseM()` and is the [format](#) method for sparse matrices.
[printSpMatrix](#) which is used by the (typically implicitly called) [show](#) and [print](#) methods for sparse matrices.

Examples

```
m <- suppressWarnings(matrix(c(0, 3.2, 0,0, 11,0,0,0,-7,0), 4,9))
fm <- formatSparseM(m)
noquote(fm)
## nice, but this is nicer {with "units" vertically aligned}:
print(fm, quote=FALSE, right=TRUE)
## and "the same" as :
Matrix(m)

## align = "right" is cheaper --> the "." are not aligned:
noquote(f2 <- formatSparseM(m,align="r"))
stopifnot(f2 == fm | m == 0, dim(f2) == dim(m),
          (f2 == ".") == (m == 0))
```

generalMatrix-class *Virtual Class ‘generalMatrix’ of General Matrices*

Description

generalMatrix is a virtual subclass of [Matrix](#) representing general (unstructured, possibly non-square) matrices. Nonvirtual subclasses of generalMatrix impose no constraints on the entries of the represented matrix beyond the constraint imposed by the data type.

Slots

Dim, Dimnames inherited from virtual superclass [Matrix](#).

factors a list of [MatrixFactorization](#) objects caching factorizations of the represented matrix.

It is updated “automagically” by methods for generic functions that compute matrix factorizations, such as [lu](#). The caching mechanism is not idiomatic as it circumvents R’s usual copy semantics; it may be deprecated in a future version of **Matrix**, hence code intended to be forwards compatible will not access the factors slot.

Subclasses

The direct, nonvirtual subclasses of generalMatrix defined in package **Matrix** are listed below with links to virtual superclasses defining their data type and storage format.

Nonvirtual subclass	Data type	Storage format
ngeMatrix	nMatrix	unpackedMatrix
lgeMatrix	lMatrix	unpackedMatrix
igeMatrix	iMatrix	unpackedMatrix
dgeMatrix	dMatrix	unpackedMatrix
zgeMatrix	zMatrix	unpackedMatrix
ngCMatrix	nMatrix	CsparseMatrix
lgCMatrix	lMatrix	CsparseMatrix
igCMatrix	iMatrix	CsparseMatrix
dgCMatrix	dMatrix	CsparseMatrix

zgCMatrix	zMatrix	CsparseMatrix
ngRMatrix	nMatrix	RsparseMatrix
lgRMatrix	lMatrix	RsparseMatrix
igRMatrix	iMatrix	RsparseMatrix
dgRMatrix	dMatrix	RsparseMatrix
zgRMatrix	zMatrix	RsparseMatrix
ngTMatrix	nMatrix	TsparseMatrix
lgTMatrix	lMatrix	TsparseMatrix
igTMatrix	iMatrix	TsparseMatrix
dgTMatrix	dMatrix	TsparseMatrix
zgTMatrix	zMatrix	TsparseMatrix

The subclasses of `iMatrix` and `zMatrix` were not defined until **Matrix** version 1.8-0.

See Also

Related classes [symmetricMatrix](#), [posdefMatrix](#), [triangularMatrix](#), [diagonalMatrix](#), and [indexMatrix](#) representing Hermitian or symmetric, positive semidefinite, triangular, diagonal, and index matrices. Generic functions [Schur](#), [lu](#), and [qr](#) for Schur, LU, and QR factorization of general matrices.

head-methods	<i>Head and Tail of a Matrix or Vector</i>
--------------	--

Description

TODO.

Usage

```
head(x, ...)
## S4 method for signature 'Matrix'
head(x, n = 6L, ...)
## S4 method for signature 'sparseVector'
head(x, n = 6L, ...)

tail(x, ...)
## S4 method for signature 'Matrix'
tail(x, n = 6L, ...)
## S4 method for signature 'sparseVector'
tail(x, n = 6L, ...)
```

Arguments

x	.
n	.
...	.

Hilbert	<i>Generate a Hilbert matrix</i>
---------	----------------------------------

Description

Generate the n by n symmetric Hilbert matrix. Because these matrices are ill-conditioned for moderate to large n , they are often used for testing numerical linear algebra code.

Usage

```
Hilbert(n)
```

Arguments

`n` a non-negative integer.

Value

the n by n symmetric Hilbert matrix as a "dpoMatrix" object.

See Also

the class [dpoMatrix](#)

Examples

```
Hilbert(6)
```

image-methods	<i>Methods for image() in Package 'Matrix'</i>
---------------	--

Description

Methods for function [image](#) in package **Matrix**. An image of a matrix simply color codes all matrix entries and draws the $n \times m$ matrix using an $n \times m$ grid of (colored) rectangles.

The **Matrix** package image methods are based on [levelplot\(\)](#) from package **lattice**; hence these methods return an "object" of class "trellis", producing a graphic when (auto-) [print\(\)](#)ed.

Usage

```
## S4 method for signature 'dgTMatrix'
image(x,
      xlim = c(1, di[2]),
      ylim = c(di[1], 1), aspect = "iso",
      sub = sprintf("Dimensions: %d x %d", di[1], di[2]),
      xlab = "Column", ylab = "Row", cuts = 15,
      useRaster = FALSE,
      useAbs = NULL, colorkey = !useAbs,
      col.regions = NULL,
      lwd = NULL, border.col = NULL, ...)
```

Arguments

<code>x</code>	a Matrix object, i.e., fulfilling <code>is(x, "Matrix")</code> .
<code>xlim, ylim</code>	x- and y-axis limits; may be used to “zoom into” matrix. Note that <i>x, y</i> “feel reversed”: <code>ylim</code> is for the rows (= 1st index) and <code>xlim</code> for the columns (= 2nd index). For convenience, when the limits are integer valued, they are both extended by 0.5; also, <code>ylim</code> is always used decreasingly.
<code>aspect</code>	aspect ratio specified as number (<i>y/x</i>) or string; see <code>levelplot</code> .
<code>sub, xlab, ylab</code>	axis annotation with sensible defaults; see <code>plot.default</code> .
<code>cuts</code>	number of levels the range of matrix values would be divided into.
<code>useRaster</code>	logical indicating if raster graphics should be used (instead of the tradition rectangle vector drawing). If true, <code>panel.levelplot.raster</code> (from lattice package) is used, and the colorkey is also done via rasters, see also <code>levelplot</code> and possibly <code>grid.raster</code> . Note that using raster graphics may often be faster, but can be slower, depending on the matrix dimensions and the graphics device (dimensions).
<code>useAbs</code>	logical indicating if <code>abs(x)</code> should be shown; if TRUE, the former (implicit) default, the default <code>col.regions</code> will be <code>grey</code> colors (and no colorkey drawn). The default is FALSE unless the matrix has no negative entries.
<code>colorkey</code>	logical indicating if a color key aka ‘legend’ should be produced. Default is to draw one, unless <code>useAbs</code> is true. You can also specify a <code>list</code> , see <code>levelplot</code> , such as <code>list(raster=TRUE)</code> in the case of rastering.
<code>col.regions</code>	vector of gradually varying colors; see <code>levelplot</code> .
<code>lwd</code>	(only used when <code>useRaster</code> is false:) non-negative number or NULL (default), specifying the line-width of the rectangles of each non-zero matrix entry (drawn by <code>grid.rect</code>). The default depends on the matrix dimension and the device size.
<code>border.col</code>	color for the border of each rectangle. NA means no border is drawn. When NULL as by default, <code>border.col <- if (lwd < .01) NA else NULL</code> is used. Consider using an opaque color instead of NULL which corresponds to <code>grid::get.gpar("col")</code> .
<code>...</code>	further arguments passed to methods and <code>levelplot</code> , notably at for specifying (possibly non equidistant) cut values for dividing the matrix values (superseding <code>cuts</code> above).

Value

as all **lattice** graphics functions, `image(<Matrix>)` returns a "trellis" object, effectively the result of `levelplot()`.

Methods

All methods currently end up calling the method for the `dgTMatrix` class. Use `showMethods(image)` to list them all.

See Also

`levelplot`, and `print.trellis` from package **lattice**.

Examples

```
showMethods(image)
## And if you want to see the method definitions:
showMethods(image, includeDefs = TRUE, inherited = FALSE)

data(CAex, package = "Matrix")
image(CAex, main = "image(CAex)") -> imgC; imgC
stopifnot(!is.null(leg <- imgC$legend), is.list(leg$right)) # failed for 2 days ..
image(CAex, useAbs=TRUE, main = "image(CAex, useAbs=TRUE)")

cCA <- Cholesky(crossprod(CAex), Imult = .01)
## See ?print.trellis --- place two image() plots side by side:
print(image(cCA, main="Cholesky(crossprod(CAex), Imult = .01)",
  split=c(x=1,y=1,nx=2, ny=1), more=TRUE)
print(image(cCA, useAbs=TRUE,
  split=c(x=2,y=1,nx=2,ny=1))

data(USCounties, package = "Matrix")
image(USCounties)# huge
image(sign(USCounties))## just the pattern
  # how the result looks, may depend heavily on
  # the device, screen resolution, antialiasing etc
  # e.g. x11(type="Xlib") may show very differently than cairo-based

## Drawing borders around each rectangle;
  # again, viewing depends very much on the device:
image(USCounties[1:400,1:200], lwd=.1)
## Using (xlim,ylim) has advantage : matrix dimension and (col/row) indices:
image(USCounties, c(1,200), c(1,400), lwd=.1)
image(USCounties, c(1,300), c(1,200), lwd=.5 )
image(USCounties, c(1,300), c(1,200), lwd=.01)
## These 3 are all equivalent :
(I1 <- image(USCounties, c(1,100), c(1,100), useAbs=FALSE))
I2 <- image(USCounties, c(1,100), c(1,100), useAbs=FALSE, border.col=NA)
I3 <- image(USCounties, c(1,100), c(1,100), useAbs=FALSE, lwd=2, border.col=NA)
stopifnot(all.equal(I1, I2, check.environment=FALSE),
  all.equal(I2, I3, check.environment=FALSE))
## using an opaque border color
```

```

image(USCounties, c(1,100), c(1,100), useAbs=FALSE, lwd=3, border.col = adjustcolor("skyblue", 1/2))

if(interactive() || nzchar(Sys.getenv("R_MATRIX_CHECK_EXTRA"))) {
## Using raster graphics: For PDF this would give a 77 MB file,
## however, for such a large matrix, this is typically considerably
## *slower* (than vector graphics rectangles) in most cases :
if(doPNG <- !dev.interactive())
  png("image-USCounties-raster.png", width=3200, height=3200)
image(USCounties, useRaster = TRUE) # should not suffer from anti-aliasing
if(doPNG)
  dev.off()
  ## and now look at the *.png image in a viewer you can easily zoom in and out
}#only if(doExtras)

```

iMatrix-class

Virtual Class ‘iMatrix’ of Integer Matrices

Description

iMatrix is a virtual subclass of [Matrix](#) representing integer matrices, whose data are stored using numeric vectors of type "integer". idenseMatrix and isparseMatrix are virtual subclasses of iMatrix representing its intersections with [denseMatrix](#) and [sparseMatrix](#).

Slots

Dim, Dimnames inherited from virtual superclass [Matrix](#).

x an [integer](#) vector storing matrix entries. Details such as *which* entries are stored vary by storage format.

Subclasses

The nonvirtual subclasses of iMatrix are listed below with links to virtual superclasses defining their structure and storage format.

Nonvirtual subclass	Structure	Storage format
igeMatrix	generalMatrix	unpackedMatrix
isyMatrix	symmetricMatrix	unpackedMatrix
itrMatrix	triangularMatrix	unpackedMatrix
ispMatrix	symmetricMatrix	packedMatrix
itpMatrix	triangularMatrix	packedMatrix
igCMatrix	generalMatrix	CsparseMatrix
isCMatrix	symmetricMatrix	CsparseMatrix
itCMatrix	triangularMatrix	CsparseMatrix
igRMatrix	generalMatrix	RsparseMatrix
isRMatrix	symmetricMatrix	RsparseMatrix
itRMatrix	triangularMatrix	RsparseMatrix
igTMatrix	generalMatrix	TsparseMatrix
isTMatrix	symmetricMatrix	TsparseMatrix

```
itMatrix  triangularMatrix  TsparseMatrix
idiMatrix  diagonalMatrix    unnamed
```

Subclasses of iMatrix were not defined until **Matrix** version 1.8-0.

See Also

“Sister” classes [nMatrix](#), [lMatrix](#), [dMatrix](#), and [zMatrix](#) representing boolean, logical, double, and complex matrices.

Examples

```
## TODO
```

index-class	<i>Virtual Class “index” of Index Vectors</i>
-------------	---

Description

Class `index` is a virtual class designating index vectors, or “subscripts”, for (possibly named) vectors and arrays. It is typically used in signatures of methods for the subscript and subassignment operators, namely `[]` and `[]<-`. It is implemented as a *union* of the atomic vector classes [numeric](#), [logical](#), and [character](#).

See Also

[\[\]](#), [\[\]-methods](#), and [\[\]<--methods](#).

Examples

```
showClass("index")
```

indexMatrix-class	<i>Virtual Class ‘indexMatrix’ of Generalized Index Matrices</i>
-------------------	--

Description

`indexMatrix` is a virtual subclass of [Matrix](#) representing generalized row or column index matrices. A generalized row index matrix is a matrix with at most one nonzero entry in each row. A generalized column index matrix is a matrix with at most one nonzero entry in each column.

Multiplying a matrix on the left by a generalized row index matrix is equivalent to sampling rows with replacement and scaling the sampled rows. Multiplying a matrix on the right by a generalized column index matrix is equivalent to sampling columns with replacement and scaling the sampled columns.

Standard index matrices are generalized index matrices whose rows or columns are all standard unit vectors. Permutation matrices are standard index matrices whose rows *and* columns are all standard unit vectors. These special cases are represented by nonvirtual subclasses `indMatrix` and `pMatrix`, documented below.

Diagonal matrices are generalized index matrices which are square and whose nonzero entries are situated on the main diagonal. This special case is represented by virtual subclass `diagonalMatrix`.

Slots

`Dim`, `Dimnames` inherited from virtual superclass `Matrix`.

All direct, nonvirtual subclasses of `indexMatrix` defined in package **Matrix** have two additional slots:

`margin` an integer, either 1 or 2, indicating that the represented matrix is a generalized row (1) or column (2) index matrix.

`perm` an integer vector of length `Dim[margin]` with elements sampled with replacement from `seq_len(Dim[-margin])`. The represented matrix is zero except for entries at `[i, perm[i]]` (`margin=1`) or `[perm[j], j]` (`margin=2`), which may be nonzero.

Subclasses

Currently, the only direct, nonvirtual subclass of `indexMatrix` is `indMatrix`, which represents standard index matrices. Objects have `margin` and `perm` slots, as described in ‘Slots’. The represented matrix is equal to 1 in positions `[i, perm[i]]` (`margin=1`) or `[perm[j], j]` (`margin=2`) and otherwise equal to 0. Thus, `indMatrix` extends not only `indexMatrix` but also `nMatrix`, the virtual class of boolean matrices, and `sparseMatrix`, the virtual class of sparse format matrices. It is extended by `pMatrix`, which represents permutation matrices. Objects of class `pMatrix` have no additional slots. However, to be valid, their `perm` slot must be a permutation of `seq_len(n)` for `n=Dim[1]=Dim[2]`.

See Also

Related classes `generalMatrix`, `symmetricMatrix`, `posdefMatrix`, `triangularMatrix`, and `diagonalMatrix` representing general (unstructured, possibly nonsquare), Hermitian or symmetric, positive semidefinite, triangular, and diagonal matrices. Function `isPerm` for testing if an integer vector is a valid permutation vector. Function `invertPerm` for computing the inverse of a permutation vector. `matmult-methods` for matrix products involving `indexMatrix`.

Examples

```
p1 <- as(c(2,3,1), "pMatrix")
(sm1 <- as(rep(c(2,3,1), e=3), "indMatrix"))
stopifnot(all(sm1 == p1[rep(1:3, each=3),]))

## row-indexing of a <pMatrix> turns it into an <indMatrix>:
class(p1[rep(1:3, each=3),])

set.seed(12) # so we know '10' is in sample
## random index matrix for 30 observations and 10 unique values:
(s10 <- as(sample(10, 30, replace=TRUE), "indMatrix"))
```

```

## Sample rows of a numeric matrix :
(mm <- matrix(1:10, nrow=10, ncol=3))
s10 %%% mm

set.seed(27)
IM1 <- as(sample(1:20, 100, replace=TRUE), "indMatrix")
IM2 <- as(sample(1:18, 100, replace=TRUE), "indMatrix")
(c12 <- crossprod(IM1,IM2))
## same as cross-tabulation of the two index vectors:
stopifnot(all(c12 - unclass(table(IM1@perm, IM2@perm)) == 0))

# 3 observations, 4 implied values, first does not occur in sample:
as(2:4, "indMatrix")
# 3 observations, 5 values, first and last do not occur in sample:
as(list(2:4, 5), "indMatrix")

as(sm1, "nMatrix")
s10[1:7, 1:4] # gives an "ngTMatrix" (most economic!)
s10[1:4, ] # preserves "indMatrix"-class

I1 <- as(c(5:1,6:4,7:3), "indMatrix")
I2 <- as(7:1, "pMatrix")
(I12 <- rbind(I1, I2))
stopifnot(is(I12, "indMatrix"),
          identical(I12, rbind(I1, I2)),
          colSums(I12) == c(2L,2:4,4:2))

(pm1 <- as(as.integer(c(2,3,1)), "pMatrix"))
t(pm1) # is the same as
solve(pm1)
pm1 %%% t(pm1) # check that the transpose is the inverse
stopifnot(all(diag(3) == as(pm1 %%% t(pm1), "matrix")),
          is.logical(as(pm1, "matrix"))))

set.seed(11)
## random permutation matrix :
(p10 <- as(sample(10),"pMatrix"))

## Permute rows / columns of a numeric matrix :
(mm <- round(array(rnorm(3 * 3), c(3, 3)), 2))
mm %%% pm1
pm1 %%% mm
try(as(as.integer(c(3,3,1)), "pMatrix"))# Error: not a permutation

as(pm1, "TsparseMatrix")
p10[1:7, 1:4] # gives an "ngTMatrix" (most economic!)

## row-indexing of a <pMatrix> keeps it as an <indMatrix>:
p10[1:3, ]

```

initialize-methods	<i>Initialization</i>
--------------------	-----------------------

Description

TODO.

Usage

```
## S4 method for signature 'Matrix'
initialize(.Object, ...)
## S4 method for signature 'MatrixFactorization'
initialize(.Object, ...)
## S4 method for signature 'sparseVector'
initialize(.Object, i, x, ...)
```

Arguments

.Object	.
i	.
x	.
...	.

invertPerm	<i>Utilities for Permutation Vectors</i>
------------	--

Description

invertPerm and signPerm compute the inverse and sign of a length-n permutation vector. isPerm tests if a length-n integer vector is a valid permutation vector. asPerm coerces a length-m transposition vector to a length-n permutation vector, where $m \leq n$.

Usage

```
invertPerm(p, off = 1L, ioff = 1L)
signPerm(p, off = 1L)
isPerm(p, off = 1L)
asPerm(pivot, off = 1L, ioff = 1L, n = length(pivot))

invPerm(p, zero.p = FALSE, zero.res = FALSE)
```

Arguments

<code>p</code>	an integer vector of length <code>n</code> .
<code>pivot</code>	an integer vector of length <code>m</code> .
<code>off</code>	an integer offset, indicating that <code>p</code> is a permutation of <code>off+0:(n-1)</code> or that <code>pivot</code> contains <code>m</code> values sampled with replacement from <code>off+0:(n-1)</code> .
<code>ioff</code>	an integer offset, indicating that the result should be a permutation of <code>ioff+0:(n-1)</code> .
<code>n</code>	a integer greater than or equal to <code>m</code> , indicating the length of the result. Transpositions are applied to a permutation vector <code>vector</code> initialized as <code>seq_len(n)</code> .
<code>zero.p</code>	a logical. Equivalent to <code>off=0</code> if TRUE and <code>off=1</code> if FALSE.
<code>zero.res</code>	a logical. Equivalent to <code>ioff=0</code> if TRUE and <code>ioff=1</code> if FALSE.

Details

`invertPerm(p, off, ioff=1)` is equivalent to `order(p)` or `sort.list(p)` for all values of `off`. For the default value `off=1`, it returns the value of `p` after `p[p] <- seq_along(p)`.

`invPerm` is a simple wrapper around `invertPerm`, retained for backwards compatibility.

Value

By default, i.e., with `off=1` and `ioff=1`:

`invertPerm(p)` returns an integer vector of length `length(p)` such that `p[invertPerm(p)]` and `invertPerm(p)[p]` are both `seq_along(p)`, i.e., the identity permutation.

`signPerm(p)` returns 1 if `p` is an even permutation and -1 otherwise (i.e., if `p` is odd).

`isPerm(p)` returns TRUE if `p` is a permutation of `seq_along(p)` and FALSE otherwise.

`asPerm(pivot)` returns the result of transposing elements `i` and `pivot[i]` of a permutation vector initialized as `seq_len(n)`, for `i` in `seq_along(pivot)`.

See Also

Class `pMatrix` of permutation matrices.

Examples

```
p <- sample(10L) # a random permutation vector
ip <- invertPerm(p)
s <- signPerm(p)
```

```
## 'p' and 'ip' are indeed inverses:
```

```
stopifnot(exprs = {
  isPerm(p)
  isPerm(ip)
  identical(s, 1L) || identical(s, -1L)
  identical(s, signPerm(ip))
  identical(p[ip], 1:10)
  identical(ip[p], 1:10)
  identical(invertPerm(ip), p)
```

```

}))

## Product of transpositions (1 2)(2 1)(4 3)(6 8)(10 1) = (3 4)(6 8)(1 10)
pivot <- c(2L, 1L, 3L, 3L, 5L, 8L, 7L, 8L, 9L, 1L)
q <- asPerm(pivot)
stopifnot(exprs = {
  identical(q, c(10L, 2L, 4L, 3L, 5L, 8L, 7L, 6L, 9L, 1L))
  identical(q[q], seq_len(10L)) # because the permutation is odd:
  signPerm(q) == -1L
})

invPerm # a less general version of 'invertPerm'

```

is.na-methods

is.na(), is.finite() Methods for 'Matrix' Objects

Description

Methods for generic functions [anyNA\(\)](#), [is.na\(\)](#), [is.nan\(\)](#), [is.infinite\(\)](#), and [is.finite\(\)](#), for objects inheriting from virtual class [Matrix](#) or [sparseVector](#).

Usage

```

## S4 method for signature 'denseMatrix'
is.na(x)
## S4 method for signature 'CsparseMatrix'
is.na(x)
## S4 method for signature 'RsparseMatrix'
is.na(x)
## S4 method for signature 'TsparseMatrix'
is.na(x)
## S4 method for signature 'diagonalMatrix'
is.na(x)
## S4 method for signature 'indMatrix'
is.na(x)
## S4 method for signature 'sparseVector'
is.na(x)

## and likewise for anyNA, is.nan, is.infinite, is.finite

```

Arguments

x an R object, here a sparse or dense matrix or vector.

Value

For [is.*\(\)](#), an [nMatrix](#) or [nsparseVector](#) matching the dimensions of **x** and specifying the positions in **x** of (some subset of) [NA](#), [NaN](#), [Inf](#), and [-Inf](#). For [anyNA\(\)](#), TRUE if **x** contains NA or NaN and FALSE otherwise.

See Also[NA](#), [NaN](#), [Inf](#)**Examples**

```
(M <- Matrix(1:6, nrow = 4, ncol = 3,
            dimnames = list(letters[1:4], LETTERS[1:3])))
stopifnot(!anyNA(M), !any(is.na(M)))

M[2:3, 2] <- NA
(inM <- is.na(M))
stopifnot(anyNA(M), sum(inM) == 2)

(A <- spMatrix(nrow = 10, ncol = 20,
              i = c(1, 3:8), j = c(2, 9, 6:10), x = 7 * (1:7)))
stopifnot(!anyNA(A), !any(is.na(A)))

A[2, 3] <- A[1, 2] <- A[5, 5:9] <- NA
(inA <- is.na(A))
stopifnot(anyNA(A), sum(inA) == 1 + 1 + 5)
```

is.null.DN

*Are the Dimnames dn NULL-like ?***Description**

Are the [dimnames](#) dn [NULL](#)-like?

`is.null.DN(dn)` is less strict than `is.null(dn)`, because it is also true ([TRUE](#)) when the dimnames dn are “like” [NULL](#), or `list(NULL, NULL)`, as they can easily be for the traditional R matrices ([matrix](#)) which have no formal [class](#) definition, and hence much freedom in how their [dimnames](#) look like.

Usage

```
is.null.DN(dn)
```

Arguments

dn [dimnames\(\)](#) of a [matrix](#)-like R object.

Value

[logical](#) [TRUE](#) or [FALSE](#).

Note

This function is really to be used on “traditional” matrices rather than those inheriting from [Matrix](#), as the latter will always have `dimnames list(NULL, NULL)` exactly, in such a case.

Author(s)

Martin Maechler

See Also[is.null](#), [dimnames](#), [matrix](#).**Examples**

```

m1 <- m2 <- m3 <- m4 <- m <-
  matrix(round(100 * rnorm(6)), 2, 3)
dimnames(m1) <- list(NULL, NULL)
dimnames(m2) <- list(NULL, character())
dimnames(m3) <- rev(dimnames(m2))
dimnames(m4) <- rep(list(character()),2)

m4 # prints absolutely identically to m

c.o <- capture.output
cm <- c.o(m)
stopifnot(exprs = {
  m == m1; m == m2; m == m3; m == m4
  identical(cm, c.o(m1)); identical(cm, c.o(m2))
  identical(cm, c.o(m3)); identical(cm, c.o(m4))
})

hasNoDimnames <- function(.) is.null.DN(dimnames(.))
stopifnot(exprs = {
  hasNoDimnames(m)
  hasNoDimnames(m1); hasNoDimnames(m2)
  hasNoDimnames(m3); hasNoDimnames(m4)
  hasNoDimnames(Matrix(m) -> M)
  hasNoDimnames(as(M, "sparseMatrix"))
})

```

isCanonical-methods *Test if an Object is Canonical*

Description

Where the representation of an abstract object by an R object is not unique (because some details are unspecified by the class and unreferenced by methods), test for a “canonical” representation.

Usage

```
isCanonical(object, ...)
```

Arguments

object an R object.
 ... optional arguments passed to methods.

Details

What is meant by “canonical” is up to the method writer. In particular, a canonical representation need not be unique. In other words, canonicalize(x) and canonicalize(y) need not be identical for all sufficiently similar x and y of a given class.

Value

TRUE if x is canonical and FALSE otherwise.

See Also

[forceCanonical](#).

Examples

```
## NA->TRUE for nonzero pattern dense ("ndenseMatrix")
m1 <- m1. <- new("ngeMatrix", Dim = c(2L, 2L),
                x = c(FALSE, NA, TRUE, NA))
m1@x <- c(FALSE, TRUE, TRUE, TRUE)
m1; m1. # same class, same matrix, different representations

## transpose for symmetric, conjugate transpose for Hermitian
m2 <- m2. <- new("zsyMatrix", Dim = c(2L, 2L), uplo = "U", trans = "C",
                x = c(1+1i, 2+2i, 4+4i, 8+8i))
m2@x <- c(1+0i, 4-4i, 4+4i, 8+0i)
m2; m2.

## zero for triangular, setting diagonal if diag="U"
m3 <- m3. <- new("dtrMatrix", Dim = c(2L, 2L), uplo = "L", diag = "U",
                x = c(1, 2, 4, 8))
m3@x <- c(1, 2, 0, 1)
m3; m3.

## integer-valued (and where possible integer-typed) slots
## for sparse vectors
v1 <- v1. <- new("nsparseVector", length = 4.2, i = c(2.1, 4.5))
v1@length <- 4L; v1@i <- c(2L, 4L)
v1; v1.

## similarly
im <- .Machine[["integer.max"]]
v2 <- v2. <- new("nsparseVector", length = im + 1.5, i = im + 0.5)
v2@length <- im + 1; v2@i <- im
## v2; v2. # don't print such long vectors

nms <- paste0(rep(c("m", "v"), c(5L, 2L)), c(1:3, 1:2))
```

```
obj <- mget(nms, inherits = FALSE)
obj. <- mget(paste0(nms, "."), inherits = FALSE)
stopifnot( all(vapply(obj , isCanonical, NA)),
           !any(vapply(obj., isCanonical, NA)))
```

isDiagonal-methods *Test if a Matrix is Diagonal*

Description

isDiagonal tests whether its argument is a diagonal matrix. Unlike the analogous [isSymmetric](#), it is generically from **Matrix** rather than **base**. Hence **Matrix** defines methods for traditional matrices of implicit class ["matrix"](#) or explicit class ["table"](#) in addition to matrices inheriting from virtual class ["Matrix"](#).

By our definition, diagonal matrices are *square*, i.e., they have the same number of rows and columns.

Usage

```
isDiagonal(object, ...)
```

Arguments

object	an R object, typically a matrix.
...	optional arguments passed from or to other methods.

Value

A [logical](#), either TRUE or FALSE (never [NA](#)).

See Also

[isSymmetric](#); virtual class ["diagonalMatrix"](#) and subclasses.

Examples

```
isDiagonal(matrix(c(2,0,0,1), 2,2)) # TRUE
Np <- rpois(100, 1)
(tab <- table(Np, Np+1L))
isDiagonal(tab) # TRUE

## Look at implementations:
showMethods("isDiagonal", includeDefs = TRUE)
```

isSymmetric-methods *Test if a Matrix is Symmetric (Hermitian)*

Description

Tests whether its argument is a symmetric square matrix, by default tolerating some numerical fuzz and requiring symmetric dimnames in addition to symmetry in the mathematical sense. A [complex](#) matrix Z, i.e., inheriting from [zMatrix](#), must be “Hermitian” for `isSymmetric(Z)` to be true.

`isSymmetric` is a generic function in **base**, which has a [method](#) for traditional matrices of implicit class `matrix`. Documented here are methods in package **Matrix** for subclasses of virtual class [Matrix](#).

Usage

```
## S4 method for signature 'denseMatrix'
isSymmetric(object, tol = 100 * .Machine$double.eps, tol1 = 8 * tol,
             trans = "C", checkDN = TRUE, ...)
## S4 method for signature 'CsparseMatrix'
isSymmetric(object, tol = 100 * .Machine$double.eps,
             trans = "C", checkDN = TRUE, ...)
## S4 method for signature 'RsparseMatrix'
isSymmetric(object, tol = 100 * .Machine$double.eps,
             trans = "C", checkDN = TRUE, ...)
## S4 method for signature 'TsparseMatrix'
isSymmetric(object, tol = 100 * .Machine$double.eps,
             trans = "C", checkDN = TRUE, ...)
## S4 method for signature 'diagonalMatrix'
isSymmetric(object, tol = 100 * .Machine$double.eps,
             trans = "C", checkDN = TRUE, ...)
## S4 method for signature 'indMatrix'
isSymmetric(object, checkDN = TRUE, ...)
```

Arguments

<code>object</code>	a Matrix .
<code>tol, tol1</code>	see isSymmetric.matrix .
<code>trans</code>	a character string, "C" or "T", indicating either the conjugate transpose or transpose, i.e., whether to test for Hermitian or symmetric object. It matters only in the complex case. The default is "C", i.e., Hermitian, for consistency with isSymmetric.matrix .
<code>checkDN</code>	a logical indicating whether symmetry of the Dimnames slot of object should be checked.
<code>...</code>	further arguments passed from or to other methods. Notably, for numeric or complex (but not integer, logical, or pattern) object, these are used to set a tolerance for asymmetry. See arguments <code>tol</code> , <code>tol1</code> , and <code>...</code> of isSymmetric.matrix .

Details

The Dimnames [slot](#) of object, say `dn`, is considered to be symmetric if and only if

- `dn[[1]]` and `dn[[2]]` are identical *or* one is NULL; *and*
- `ndn <- names(dn)` is NULL *or* `ndn[1]` and `ndn[2]` are identical *or* one is the empty string `""`.

Hence `list(a=nms, a=nms)` is considered to be symmetric and so too are `list(a=nms, NULL)` and `list(NULL, a=nms)`.

Note that this definition is *looser* than that employed by [isSymmetric.matrix](#), which requires `dn[1]` and `dn[2]` to be identical, where `dn` is the `dimnames` attribute of a traditional matrix.

Value

A logical, either TRUE or FALSE (never NA).

See Also

[forceSymmetric](#), [symmpart](#), [skewpart](#); virtual class [symmetricMatrix](#) and its subclasses.

Examples

```
isSymmetric(Diagonal(4)) # TRUE of course
M <- Matrix(c(1,2,2,1), 2,2)
isSymmetric(M) # TRUE (*and* of formal class "dsyMatrix")
isSymmetric(as(M, "generalMatrix")) # still symmetric, even if not "formally"
isSymmetric(triu(M)) # FALSE

## Look at implementations:
showMethods("isSymmetric", includeDefs = TRUE) # includes S3 generic from base
```

isTriangular-methods *Test if a Matrix is Triangular*

Description

`isTriangular` tests whether its argument is a triangular matrix. Unlike the analogous [isSymmetric](#), it is generically from **Matrix** rather than **base**. Hence **Matrix** defines methods for traditional matrices of implicit [class "matrix"](#) in addition to matrices inheriting from virtual class ["Matrix"](#).

By our definition, triangular matrices are *square*, i.e., they have the same number of rows and columns.

Usage

```
isTriangular(object, upper = NA, ...)
```

Arguments

object	an R object, typically a matrix.
upper	a logical , either TRUE or FALSE, in which case TRUE is returned only for upper or lower triangular object; or otherwise NA (the default), in which case TRUE is returned for any triangular object.
...	optional arguments passed from or to other methods.

Value

A [logical](#), either TRUE or FALSE (never [NA](#)).

If object is triangular and upper is NA, then isTriangular returns TRUE with an [attribute](#) kind, either "U" or "L", indicating that object is **upper** or **lower** triangular, respectively. Users should not rely on how kind is determined for diagonal matrices, which are both upper and lower triangular.

See Also

[isSymmetric](#); virtual class "[triangularMatrix](#)" and subclasses.

Examples

```
isTriangular(Diagonal(4))
## is TRUE: a diagonal matrix is also (both upper and lower) triangular
(M <- Matrix(c(1,2,0,1), 2,2))
isTriangular(M) # TRUE (*and* of formal class "dtrMatrix")
isTriangular(as(M, "generalMatrix")) # still triangular, even if not "formally"
isTriangular(crossprod(M)) # FALSE

## Look at implementations:
showMethods("isTriangular", includeDefs = TRUE)
```

KhatriRao

Khatri-Rao Matrix Product

Description

Computes Khatri-Rao products for any kind of matrices.

The Khatri-Rao product is a column-wise Kronecker product. Originally introduced by Khatri and Rao (1968), it has many different applications, see Liu and Trenkler (2008) for a survey. Notably, it is used in higher-dimensional tensor decompositions, see Bader and Kolda (2008).

Usage

```
KhatriRao(X, Y = X, FUN = "*", sparseY = TRUE, make.dimnames = FALSE)
```

Arguments

X, Y	matrices of with the same number of columns.
FUN	the (name of the) function to be used for the column-wise Kronecker products, see kronecker , defaulting to the usual multiplication.
sparseY	logical specifying if Y should be coerced and treated as sparseMatrix . Set this to FALSE, e.g., to distinguish structural zeros from zero entries.
make.dimnames	logical indicating if the result should inherit dimnames from X and Y in a simple way.

Value

a "[CsparseMatrix](#)", say R, the Khatri-Rao product of X ($n \times k$) and Y ($m \times k$), is of dimension $(n \cdot m) \times k$, where the j-th column, $R[, j]$ is the kronecker product [kronecker](#)($X[, j]$, $Y[, j]$).

Note

The current implementation is efficient for large sparse matrices.

Author(s)

Original by Michael Cysouw, Univ. Marburg; minor tweaks, bug fixes etc, by Martin Maechler.

References

Khatri, C. G., and Rao, C. Radhakrishna (1968) Solutions to Some Functional Equations and Their Applications to Characterization of Probability Distributions. *Sankhya: Indian J. Statistics, Series A* **30**, 167–180.

Bader, Brett W, and Tamara G Kolda (2008) Efficient MATLAB Computations with Sparse and Factored Tensors. *SIAM J. Scientific Computing* **30**, 205–231.

See Also

[kronecker](#).

Examples

```
## Example with very small matrices:
m <- matrix(1:12,3,4)
d <- diag(1:4)
KhatriRao(m,d)
KhatriRao(d,m)
dimnames(m) <- list(LETTERS[1:3], letters[1:4])
KhatriRao(m,d, make.dimnames=TRUE)
KhatriRao(d,m, make.dimnames=TRUE)
dimnames(d) <- list(NULL, paste0("D", 1:4))
KhatriRao(m,d, make.dimnames=TRUE)
KhatriRao(d,m, make.dimnames=TRUE)
dimnames(d) <- list(paste0("d", 10*1:4), paste0("D", 1:4))
(Kmd <- KhatriRao(m,d, make.dimnames=TRUE))
```

```

(Kdm <- KhatriRao(d,m, make.dimnames=TRUE))

nm <- as(m, "nsparseMatrix")
nd <- as(d, "nsparseMatrix")
KhatriRao(nm,nd, make.dimnames=TRUE)
KhatriRao(nd,nm, make.dimnames=TRUE)

stopifnot(dim(KhatriRao(m,d)) == c(nrow(m)*nrow(d), ncol(d)))
## border cases / checks:
zm <- nm; zm[] <- FALSE # all FALSE matrix
stopifnot(all(K1 <- KhatriRao(nd, zm) == 0), identical(dim(K1), c(12L, 4L)),
          all(K2 <- KhatriRao(zm, nd) == 0), identical(dim(K2), c(12L, 4L)))

d0 <- d; d0[] <- 0; m0 <- Matrix(d0[-1,])
stopifnot(all(K3 <- KhatriRao(d0, m) == 0), identical(dim(K3), dim(Kdm)),
          all(K4 <- KhatriRao(m, d0) == 0), identical(dim(K4), dim(Kmd)),
          all(KhatriRao(d0, d0) == 0), all(KhatriRao(m0, d0) == 0),
          all(KhatriRao(d0, m0) == 0), all(KhatriRao(m0, m0) == 0),
          identical(dimnames(KhatriRao(m, d0, make.dimnames=TRUE)), dimnames(Kmd)))

## a matrix with "structural" and non-structural zeros:
m01 <- new("dgCMatrix", i = c(0L, 2L, 0L, 1L), p = c(0L, 0L, 0L, 2L, 4L),
          Dim = 3:4, x = c(1, 0, 1, 0))
D4 <- Diagonal(4, x=1:4) # "as" d
DU <- Diagonal(4)# unit-diagonal: uplo="U"
(K5 <- KhatriRao( d, m01))
K5d <- KhatriRao( d, m01, sparseY=FALSE)
K5Dd <- KhatriRao(D4, m01, sparseY=FALSE)
K5Ud <- KhatriRao(DU, m01, sparseY=FALSE)
(K6 <- KhatriRao(diag(3), t(m01)))
K6D <- KhatriRao(Diagonal(3), t(m01))
K6d <- KhatriRao(diag(3), t(m01), sparseY=FALSE)
K6Dd <- KhatriRao(Diagonal(3), t(m01), sparseY=FALSE)
stopifnot(exprs = {
  all(K5 == K5d)
  identical(cbind(c(7L, 10L), c(3L, 4L)),
            which(K5 != 0, arr.ind = TRUE, useNames=FALSE))
  identical(K5d, K5Dd)
  identical(K6, K6D)
  all(K6 == K6d)
  identical(cbind(3:4, 1L),
            which(K6 != 0, arr.ind = TRUE, useNames=FALSE))
  identical(K6d, K6Dd)
})

```

Description

A model matrix `mm` and corresponding response vector `y` used in an example by Koenker and Ng. The matrix `mm` is a sparse matrix with 1850 rows and 712 columns but only 8758 non-zero entries. It is a "dgCMatrix" object. The vector `y` is just `numeric` of length 1850.

Usage

```
data(KNex)
```

References

Roger Koenker and Pin Ng (2003). SparseM: A sparse matrix package for R; *J. of Statistical Software*, **8** (6), [doi:10.18637/jss.v008.i06](https://doi.org/10.18637/jss.v008.i06)

Examples

```
data(KNex, package = "Matrix")
class(KNex$mm)
dim(KNex$mm)
image(KNex$mm)
str(KNex)

system.time( # a fraction of a second
  sparse.sol <- with(KNex, solve(crossprod(mm), crossprod(mm, y))))

head(round(sparse.sol,3))

## Compare with QR-based solution ("more accurate, but slightly slower"):
system.time(
  sp.sol2 <- with(KNex, qr.coef(qr(mm), y) ))

all.equal(as.vector(sparse.sol), sp.sol2, tolerance = 1e-13) # TRUE
```

kronecker-methods

Methods for Function 'kronecker()' in Package 'Matrix'

Description

Computes Kronecker products for objects inheriting from "`Matrix`".

In order to preserve sparseness, we treat $0 \times \text{NA}$ as 0 , not as `NA` as usually in R (and as used for the `base` function `kronecker`).

Methods

```
kronecker signature(X = "Matrix", Y = "ANY") .....
kronecker signature(X = "ANY", Y = "Matrix") .....
kronecker signature(X = "diagonalMatrix", Y = "ANY") .....
```

```

kronecker signature(X = "sparseMatrix", Y = "ANY") .....
kronecker signature(X = "TsparseMatrix", Y = "TsparseMatrix") .....
kronecker signature(X = "dgTMatrix", Y = "dgTMatrix") .....
kronecker signature(X = "dtTMatrix", Y = "dtTMatrix") .....
kronecker signature(X = "indMatrix", Y = "indMatrix") .....

```

Examples

```

(t1 <- spMatrix(5,4, x= c(3,2,-7,11), i= 1:4, j=4:1)) # 5 x 4
(t2 <- kronecker(Diagonal(3, 2:4), t1))                # 15 x 12

## should also work with special-cased logical matrices
l3 <- upper.tri(matrix(,3,3))
M <- Matrix(l3)
(N <- as(M, "nsparseMatrix")) # "ntCMatrix" (upper triangular)
N2 <- as(N, "generalMatrix")  # (lost "t"riangularity)
MM <- kronecker(M,M)
NN <- kronecker(N,N) # "dtTMatrix" i.e. did keep
NN2 <- kronecker(N2,N2)
stopifnot(identical(NN,MM),
           is(NN2, "sparseMatrix"), all(NN2 == NN),
           is(NN, "triangularMatrix"))

```

length-methods

Length of an Object

Description

S4 methods defined in package **Matrix** for S4 generic functions [length](#) and [length<-](#).

Usage

```

## S4 method for signature 'Matrix'
length(x)
## S4 method for signature 'MatrixFactorization'
length(x)
## S4 method for signature 'sparseVector'
length(x)

## S4 replacement method for signature 'denseMatrix,numeric'
length(x) <- value
## S4 replacement method for signature 'sparseMatrix,numeric'
length(x) <- value
## S4 replacement method for signature 'sparseVector,numeric'
length(x) <- value

```

Arguments

<code>x</code>	a Matrix object, a MatrixFactorization object, or a sparseVector object.
<code>value</code>	a numeric vector of length 1. Truncation towards zero by trunc provides the new length, which must be nonnegative.

Value

For `x` inheriting from virtual class [Matrix](#) or virtual class [MatrixFactorization](#), `length` returns a numeric vector of length 1 storing the product of the dimensions of the represented or factorized matrix. The type is double if and only if the product is not in the range of integer. If the product is not in the range of double, then it is rounded in the direction of zero to the nearest representable integer, and the offset is preserved as an attribute `off`, such that the product is the exact result of adding `length(x)` and `off`. Replacement by numeric is a no-op if the old and new lengths are equal. It otherwise coerces `x` to vector or [sparseVector](#) (according to its storage format) and dispatches.

For `x` inheriting from virtual class [sparseVector](#), `length` returns a numeric vector of length 1 storing the length of the represented vector. The type is double if and only if the length is not in the range of integer. Replacement by numeric is a no-op if the old and new lengths are equal. It otherwise shortens `x` by deleting trailing elements or lengthens `x` by adding trailing [NA](#).

See Also

[dim](#), [dimnames](#).

Examples

```
M <- function (Dim) new("lgTMatrix", Dim = Dim)
V <- function (length) new("lsparseVector", length = length)
im <- .Machine$integer.max

stopifnot(identical(length(M(c(im, 1L))), im),
          identical(length(M(c(im, 2L))), im * 2),
          identical(length(M(c(im, im))),
                    structure(0x1p+62 - 0x1p+32, off = 1)),
          identical(length(V(      im )), im),
          identical(length(V(as.double(im))), im))

(vv <- V(4L))
length(vv) <- 2L
vv # shortened
length(vv) <- 4L
vv # lengthened with NA

## Lengthening by adding trailing zeros
vv <- new("lsparseVector", length = length(vv) + 4L,
        i = vv@i, x = vv@x)
vv
## Shortening by deleting trailing zeros
vv <- new("lsparseVector", length = max(0L, vv@i),
        i = vv@i, x = vv@x)
vv
```

lMatrix-class	Virtual Class ‘lMatrix’ of Logical Matrices
---------------	---

Description

lMatrix is a virtual subclass of [Matrix](#) representing logical matrices, whose entries are FALSE, TRUE, or NA. ldenseMatrix and lsparseMatrix are virtual subclasses of lMatrix representing its intersections with [denseMatrix](#) and [sparseMatrix](#).

lMatrix objects tend to arise as the return value of `op(A)`, where A is a Matrix object and op is a function that preserves matrix dimensions and returns FALSE, TRUE, or NA by element (e.g., `op(A) := A == 0` or `!A` or `A | B` for some conformable B).

Slots

Dim, Dimnames inherited from virtual superclass [Matrix](#).

x a [logical](#) vector storing matrix entries. Details such as *which* entries are stored vary by storage format.

Subclasses

The nonvirtual subclasses of lMatrix are listed below with links to virtual superclasses defining their structure and storage format.

Nonvirtual subclass	Structure	Storage format
lgeMatrix	generalMatrix	unpackedMatrix
lsyMatrix	symmetricMatrix	unpackedMatrix
ltrMatrix	triangularMatrix	unpackedMatrix
lspMatrix	symmetricMatrix	packedMatrix
ltpMatrix	triangularMatrix	packedMatrix
lgCMatrix	generalMatrix	CsparseMatrix
lsCMatrix	symmetricMatrix	CsparseMatrix
ltCMatrix	triangularMatrix	CsparseMatrix
lgRMatrix	generalMatrix	RsparseMatrix
lsRMatrix	symmetricMatrix	RsparseMatrix
ltRMatrix	triangularMatrix	RsparseMatrix
lgTMatrix	generalMatrix	TsparseMatrix
lsTMatrix	symmetricMatrix	TsparseMatrix
ltTMatrix	triangularMatrix	TsparseMatrix
ldiMatrix	diagonalMatrix	<i>unnamed</i>

See Also

“Sister” classes [nMatrix](#), [iMatrix](#), [dMatrix](#), and [zMatrix](#) representing boolean, integer, double, and complex matrices.

Examples

```
showClass("ldenseMatrix")

as(diag(3) > 0, "ldenseMatrix")

(m <- Matrix(c(0,0,2:0), 3,5, dimnames=list(LETTERS[1:3],NULL)))
(lm <- (m > 1)) # lgC
!lm           # no longer sparse
stopifnot(is(lm,"lsparseMatrix"),
           identical(!lm, m <= 1))

data(KNex, package = "Matrix")
str(mmG.1 <- (KNex $ mm) > 0.1)# "lgC..."
table(mmG.1@x)# however with many ``non-structural zeros''
## from logical to nz_pattern -- okay when there are no NA's :
nmG.1 <- as(mmG.1, "nMatrix") # <<< has "TRUE" also where mmG.1 had FALSE
## from logical to "double"
dmG.1 <- as(mmG.1, "dMatrix") # has '0' and back:
lmG.1 <- as(dmG.1, "lMatrix")
stopifnot(identical(nmG.1, as((KNex $ mm) != 0,"nMatrix")),
           validObject(lmG.1),
           identical(lmG.1, mmG.1))

class(xnx <- crossprod(nmG.1))# "nsC.."
class(xlx <- crossprod(mmG.1))# "dsC.." : numeric
is0 <- (xlx == 0)
mean(as.vector(is0))# 99.3% zeros: quite sparse, but
table(xlx@x == 0)# more than half of the entries are (non-structural!) 0
stopifnot(isSymmetric(xlx), isSymmetric(xnx),
           ## compare xnx and xlx : have the *same* non-structural 0s :
           sapply(slotNames(xnx),
                   function(n) identical(slot(xnx, n), slot(xlx, n)))))
```

lu-methods

Methods for LU Factorization

Description

Computes the pivoted LU factorization of an $m \times n$ real matrix A , which has the general form

$$P_1 A P_2 = LU$$

or (equivalently)

$$A = P_1' L U P_2'$$

where P_1 is an $m \times m$ permutation matrix, P_2 is an $n \times n$ permutation matrix, L is an $m \times \min(m, n)$ unit lower trapezoidal matrix, and U is a $\min(m, n) \times n$ upper trapezoidal matrix.

Methods for [denseMatrix](#) are built on LAPACK routine `dgetrf`, which does not permute columns, so that P_2 is an identity matrix.

Methods for [sparseMatrix](#) are built on CXSparse routine `cs_lu`, which requires $m = n$, so that L and U are triangular matrices.

Usage

```

lu(x, ...)
## S4 method for signature 'denseMatrix'
lu(x, warnSing = TRUE, direct = FALSE, ...)
## S4 method for signature 'CsparseMatrix'
lu(x, errSing = TRUE, order = NA_integer_,
    tol = 1, direct = FALSE, ...)
## S4 method for signature 'matrix'
lu(x, ...)

```

Arguments

<code>x</code>	a finite matrix or Matrix to be factorized, which must be square if sparse.
<code>warnSing</code>	a logical indicating if a warning should be signaled for singular <code>x</code> . Used only by methods for dense matrices.
<code>errSing</code>	a logical indicating if an error should be signaled for singular <code>x</code> . Used only by methods for sparse matrices.
<code>order</code>	an integer in $0:3$ passed to CXSparse routine <code>cs_sqr</code> , indicating a strategy for choosing the column permutation P_2 . 0 means no column permutation. 1, 2, and 3 indicate a fill-reducing ordering of $A + A'$, $\tilde{A}'\tilde{A}$, and $A'A$, where $\tilde{A}$ is A with “dense” rows removed. NA (the default) is equivalent to 2 if <code>tol == 1</code> and 1 otherwise. Do not set to 0 unless you know that the column order of A is already sensible.
<code>tol</code>	a number. The original pivot element is used if its absolute value exceeds <code>tol * a</code> , where <code>a</code> is the maximum in absolute value of the other possible pivot elements. Set <code>tol < 1</code> only if you know what you are doing.
<code>direct</code>	a logical indicating if the result should be constructed “directly” where <code>x</code> is already (formally) upper triangular or unit lower triangular. In this case, methods do not permute or test for singular <code>x</code> .
<code>...</code>	further arguments passed to or from methods.

Details

What happens when `x` is determined to be near-singular differs by method. The method for class [dgeMatrix](#) completes the factorization, warning if `warnSing = TRUE` and in any case returning a valid [denseLU](#) object. Users of this method can detect singular `x` with a suitable warning handler; see [tryCatch](#). In contrast, the method for class [dgCMatrix](#) abandons further computation, throwing an error if `errSing = TRUE` and otherwise returning NA. Users of this method can detect singular `x` with an error handler or by setting `errSing = FALSE` and testing for a formal result with `is(, "sparseLU")`.

Value

An object representing the factorization, inheriting from virtual class [LU](#). The specific class is [denseLU](#) unless `x` inherits from virtual class [sparseMatrix](#), in which case it is [sparseLU](#).

References

The LAPACK source code, including documentation; see <https://netlib.org/lapack/double/dgetrf.f>.

Davis, T. A. (2006). *Direct methods for sparse linear systems*. Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898718881

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

Classes [denseLU](#) and [sparseLU](#) and their methods.

Classes [dgeMatrix](#) and [dgCMatrix](#).

Generic functions [expand1](#) and [expand2](#), for constructing matrix factors from the result.

Generic functions [Cholesky](#), [BunchKaufman](#), [Schur](#), and [qr](#), for computing other factorizations.

Examples

```
showMethods("lu", inherited = FALSE)
set.seed(0)

## ---- Dense -----

(A1 <- Matrix(rnorm(9L), 3L, 3L))
(lu.A1 <- lu(A1))

(A2 <- round(10 * A1[, -3L]))
(lu.A2 <- lu(A2))

## A ~ P1' L U in floating point
str(e.lu.A2 <- expand2(lu.A2), max.level = 2L)
stopifnot(all.equal(A2, Reduce(`%*%`, e.lu.A2)))

## ---- Sparse -----

A3 <- as(readMM(system.file("external/pores_1.mtx", package = "Matrix")),
         "CsparseMatrix")
(lu.A3 <- lu(A3))

## A ~ P1' L U P2' in floating point
str(e.lu.A3 <- expand2(lu.A3), max.level = 2L)
stopifnot(all.equal(A3, Reduce(`%*%`, e.lu.A3)))
```

Description

From an R object coercible to "[TsparseMatrix](#)", typically a (sparse) matrix, produce its triplet representation which may collapse to a "Duplet" in the case of binary aka pattern, such as "[nMatrix](#)" objects.

Usage

```
mat2triplet(x, uniqT = FALSE)
```

Arguments

x	any R object for which <code>as(x, "TsparseMatrix")</code> works; typically a matrix of one of the Matrix package matrices.
uniqT	logical indicating if the triplet representation should be 'unique' in the sense of asUniqueT (byrow=FALSE).

Value

A [list](#), typically with three components,

i	vector of row indices for all non-zero entries of x
j	vector of columns indices for all non-zero entries of x
x	vector of all non-zero entries of x; exists only when <code>as(x, "TsparseMatrix")</code> is not a " nsparseMatrix ".

Note that the [order](#) of the entries is determined by the coercion to "[TsparseMatrix](#)" and hence typically with increasing j (and increasing i within ties of j).

Note

The `mat2triplet()` utility was created to be a more efficient and more predictable substitute for `summary(<sparseMatrix>)`. UseRs have wrongly expected the latter to return a data frame with columns i and j which however is wrong for a "[diagonalMatrix](#)".

See Also

The `summary()` method for "[sparseMatrix](#)", [summary, sparseMatrix-method](#).

`mat2triplet()` is conceptually the *inverse* function of [spMatrix](#) and (one case of) [sparseMatrix](#).

Examples

```
mat2triplet # simple definition

i <- c(1,3:8); j <- c(2,9,6:10); x <- 7 * (1:7)
(Ax <- sparseMatrix(i, j, x = x)) ## 8 x 10 "dgCMatrix"
str(trA <- mat2triplet(Ax))
stopifnot(i == sort(trA$i), sort(j) == trA$j, x == sort(trA$x))
```

```
D <- Diagonal(x=4:2)
summary(D)
str(mat2triplet(D))
```

Math-methods	<i>Mathematical Functions</i>
--------------	-------------------------------

Description

TODO.

Usage

```
Math(x)
## log(x, \dots)
```

Arguments

x .

Math2-methods	<i>Rounding of Numbers</i>
---------------	----------------------------

Description

TODO.

Usage

```
Math2(x, digits)
```

Arguments

x .
digits .

Description

The basic matrix product, `%*%` is implemented for all our [Matrix](#) and also for [sparseVector](#) classes, fully analogously to R's base `matrix` and `vector` objects.

The functions [crossprod](#) and [tcrossprod](#) are matrix products or “cross products”, ideally implemented efficiently without computing `t(.)`'s unnecessarily. They also return [symmetricMatrix](#) classed matrices when easily detectable, e.g., in `crossprod(m)`, the one argument case.

`tcrossprod()` takes the cross-product of the transpose of a matrix. `tcrossprod(x)` is formally equivalent to, but faster than, the call `x %*% t(x)`, and so is `tcrossprod(x, y)` instead of `x %*% t(y)`.

Boolean matrix products are computed via either `%&%` or `boolArith = TRUE`.

Usage

```
## S4 method for signature 'CsparseMatrix,diagonalMatrix'
x %*% y

## S4 method for signature 'CsparseMatrix,diagonalMatrix'
crossprod(x, y = NULL, trans = "T", boolArith = NA, ...)
## .... and for many more signatures

## S4 method for signature 'TsparseMatrix,missing'
tcrossprod(x, y = NULL, trans = "T", boolArith = NA, ...)
## .... and for many more signatures
```

Arguments

<code>x</code>	a matrix-like object
<code>y</code>	a matrix-like object, or for <code>[t]crossprod()</code> <code>NULL</code> (by default); the latter case is formally equivalent to <code>y = x</code> .
<code>trans</code>	TODO.
<code>boolArith</code>	logical , i.e., <code>NA</code> , <code>TRUE</code> , or <code>FALSE</code> . If true the result is (coerced to) a pattern matrix, i.e., "nMatrix" , unless there are <code>NA</code> entries and the result will be a "lMatrix" . If false the result is (coerced to) numeric. When <code>NA</code> , currently the default, the result is a pattern matrix when <code>x</code> and <code>y</code> are "nsparseMatrix" and numeric otherwise.
<code>...</code>	potentially more arguments passed to and from methods.

Details

For some classes in the *Matrix* package, such as `dgCMatrix`, it is much faster to calculate the cross-product of the transpose directly instead of calculating the transpose first and then its cross-product.

`boolArith = TRUE` for regular (“non cross”) matrix products, `%%` cannot be specified. Instead, we provide the `%%` operator for *boolean* matrix products.

Value

A *Matrix* object, in the one argument case of an appropriate *symmetric* matrix class, i.e., inheriting from `symmetricMatrix`.

Methods

`%%` signature(`x = "dgeMatrix"`, `y = "dgeMatrix"`): Matrix multiplication; ditto for several other signature combinations, see `showMethods("%%", class = "dgeMatrix")`.

`%%` signature(`x = "dtrMatrix"`, `y = "matrix"`) and other signatures (use `showMethods("%%", class = "dtrMatrix")`): matrix multiplication. Multiplication of (matching) triangular matrices now should remain triangular (in the sense of class `triangularMatrix`).

crossprod signature(`x = "dgeMatrix"`, `y = "dgeMatrix"`): ditto for several other signatures, use `showMethods("crossprod", class = "dgeMatrix")`, matrix crossproduct, an efficient version of `t(x) %*% y`.

crossprod signature(`x = "CsparseMatrix"`, `y = "missing"`) returns `t(x) %*% x` as an `dsCMatrix` object.

crossprod signature(`x = "TsparseMatrix"`, `y = "missing"`) returns `t(x) %*% x` as an `dsCMatrix` object.

crossprod, tcrossprod signature(`x = "dtrMatrix"`, `y = "matrix"`) and other signatures, see “`%%`” above.

Note

`boolArith = TRUE`, `FALSE` or `NA` has been newly introduced for **Matrix** 1.2.0 (March 2015). Its implementation has still not been tested extensively. Notably the behaviour for sparse matrices with `x` slots containing extra zeros had not been documented previously, see the `%%` help page.

Currently, `boolArith = TRUE` is implemented via `CsparseMatrix` coercions which may be quite inefficient for dense matrices. Contributions for efficiency improvements are welcome.

See Also

`tcrossprod` in R’s base, and `crossprod` and `%*%`. **Matrix** package `%%` for boolean matrix product methods.

Examples

```
## A random sparse "incidence" matrix :
m <- matrix(0, 400, 500)
set.seed(12)
m[runif(314, 0, length(m))] <- 1
```

```

mm <- as(m, "CsparseMatrix")
object.size(m) / object.size(mm) # smaller by a factor of > 200

## tcrossprod() is very fast:
system.time(tCmm <- tcrossprod(mm))# 0 (PIII, 933 MHz)
system.time(cm <- crossprod(t(m))) # 0.16
system.time(cm. <- tcrossprod(m)) # 0.02

stopifnot(cm == as(tCmm, "matrix"))

## show sparse sub matrix
tCmm[1:16, 1:30]
```

Matrix

Construct a Classed Matrix

Description

Construct a Matrix of a class inheriting from Matrix, i.e., a “**Matrix**-package Matrix object”, using a syntax known from base R’s `matrix()`.

Usage

```
Matrix(data=NA, nrow=1, ncol=1, byrow=FALSE, dimnames=NULL,
       sparse = NULL, doDiag = TRUE, forceCheck = FALSE,
       keepInteger = Matrix.keepInteger())
```

Arguments

<code>data</code>	an optional “numeric-alike” data vector, <code>matrix</code> (or also “ <code>Matrix</code> ”).
<code>nrow, ncol</code>	when data is not a matrix, the desired number of rows and columns, respectively.
<code>byrow</code>	logical. If FALSE (the default) the matrix is filled by columns, otherwise the matrix is filled by rows.
<code>dimnames</code>	a <code>dimnames</code> attribute for the matrix: a list of two character components. They are set if not <code>NULL</code> (as per default).
<code>sparse</code>	logical or <code>NULL</code> , specifying if the result should be sparse or not. By default, it is made sparse when more than half of the entries are 0.
<code>doDiag</code>	logical indicating if a <code>diagonalMatrix</code> object should be returned when the resulting matrix is diagonal (<i>mathematically</i>). As class <code>diagonalMatrix</code> extends <code>sparseMatrix</code> , this is a natural default for all values of <code>sparse</code> . Otherwise, if <code>doDiag</code> is false, a dense or sparse (depending on <code>sparse</code>) <i>symmetric</i> matrix will be returned.
<code>forceCheck</code>	logical indicating if the checks for structure should even happen when data is already a “Matrix” object.

`keepInteger` an optional logical argument introduced in **Matrix** version 1.8-0. If `FALSE`, then `integer` data are treated as `double`. This behaviour is compatible with earlier versions of **Matrix**, but it is not so desirable now that matrices with integer data are representable as objects inheriting from virtual class `iMatrix`. The default value will change to `TRUE` in the near future; currently, it depends on the value of the environment variable `R_MATRIX_KEEP_INTEGER` at load time (false if unset), but that can be overridden by `options(Matrix.keepInteger=)`.

Details

If either of `nrow` or `ncol` is not given, an attempt is made to infer it from the length of data and the other parameter (analogously to how `matrix()` works). Further, `Matrix()` makes efforts to keep `logical` matrices logical, i.e., inheriting from class `lMatrix`, and analogously for integer or complex data. `raw` data is currently coerced to `integer` and treated as such. `Matrix()` also determines specially structured matrices such as symmetric, triangular or diagonal ones. Note that a *symmetric* matrix also needs symmetric `dimnames`, e.g., by specifying `dimnames = list(NULL, NULL)`, see the examples.

Most of the time, the function works via a traditional (*full*) `matrix`. However, `Matrix(0, nrow, ncol)` directly constructs an “empty” `sparseMatrix`, as does `Matrix(FALSE, *)`.

Although it is sometime possible to mix unclassed matrices (created with `matrix`) with ones of class “`Matrix`”, it is much safer to always use carefully constructed ones of class “`Matrix`”.

Value

Returns matrix of a class that inherits from “`Matrix`”. Only if data is not a `matrix` and does not already inherit from class `Matrix` are the arguments `nrow`, `ncol` and `byrow` made use of.

See Also

The classes `Matrix`, `symmetricMatrix`, `triangularMatrix`, and `diagonalMatrix`; further, `matrix`. Special matrices can be constructed, e.g., via `sparseMatrix` (sparse), `bdiag` (block-diagonal), `bandSparse` (banded sparse), or `Diagonal`.

Examples

```
Matrix(0, 3, 2)           # 3 by 2 matrix of zeros -> sparse
Matrix(0, 3, 2, sparse=FALSE)# -> 'dense'

## 4 cases - 3 different results :
Matrix(0, 2, 2)           # diagonal !
Matrix(0, 2, 2, sparse=FALSE)# (ditto)
Matrix(0, 2, 2,          doDiag=FALSE)# -> sparse symm. "dsCMatrix"
Matrix(0, 2, 2, sparse=FALSE, doDiag=FALSE)# -> dense symm. "dsyMatrix"

Matrix(1:6, 3, 2)         # a 3 by 2 matrix (+ integer warning)
Matrix(1:6 + 1, nrow=3)

## logical ones:
Matrix(diag(4) > 0) # -> "ldiMatrix" with diag = "U"
Matrix(diag(4) > 0, sparse=TRUE) # (ditto)
```

```

Matrix(diag(4) >= 0) # -> "lsyMatrix" (of all 'TRUE')
## triangular
l3 <- upper.tri(matrix(,3,3))
(M <- Matrix(l3))    # -> "ltCMatrix"
Matrix(! l3)         # -> "ltrMatrix"
as(l3, "CsparseMatrix")# "lgCMatrix"

Matrix(1:9, nrow=3,
      dimnames = list(c("a", "b", "c"), c("A", "B", "C")))
(I3 <- Matrix(diag(3)))# identity, i.e., unit "diagonalMatrix"
str(I3) # note 'diag = "U"' and the empty 'x' slot

(A <- cbind(a=c(2,1), b=1:2))# symmetric *apart* from dimnames
Matrix(A)                  # hence 'dgeMatrix'
(As <- Matrix(A, dimnames = list(NULL,NULL)))# -> symmetric
forceSymmetric(A) # also symmetric, w/ symm. dimnames
stopifnot(is(As, "symmetricMatrix"),
          is(Matrix(0, 3,3), "sparseMatrix"),
          is(Matrix(FALSE, 1,1), "sparseMatrix"))

```

Matrix-class

Virtual Class ‘Matrix’ of Matrices

Description

Matrix is a virtual class representing matrices. All classes of matrix defined in package **Matrix** are extensions of Matrix. Nonvirtual class `matrix` defined in package **methods**, from which traditional matrices inherit *implicitly*, does *not* extend Matrix, but methods are defined so that Matrix objects and matrix objects are interoperable.

Slots

Dim an integer vector of length 2 giving the dimensions of the matrix. It is typically accessed using `dim`; see [dim-methods](#).

Dimnames a list of length 2. Each element must be NULL or a character vector of length equal to the corresponding element of Dim. It is typically accessed using `dimnames`; see [dimnames-methods](#).

Subclasses

Objects inheriting from virtual class Matrix can be categorized in three ways: by data type, by structure, and by storage format. Accordingly, it can be helpful to think of nonvirtual subclasses of Matrix as triples

(<data type>, <structure>, <storage format>)

and to think of virtual subclasses of Matrix as sets of triples satisfying constraints.

The data type of a Matrix object refers to the set of possible values of entries in the represented matrix. Package **Matrix** supports five data types: boolean (or “nonzero pattern”), logical, integer,

double, and complex. How boolean data are stored varies by format, whereas logical, integer, double, and complex data are always stored in atomic vectors of the corresponding type. Virtual classes `nMatrix`, `lMatrix`, `iMatrix`, `dMatrix`, and `zMatrix` designate Matrix objects whose data type is one of the supported five (n=nonzero pattern, l=logical, i=integer, d=double, z=complex, following the LAPACK naming scheme).

The structure of a Matrix object refers to the mathematical properties of the represented matrix. Virtual classes `generalMatrix`, `symmetricMatrix`, `triangularMatrix`, `diagonalMatrix`, and `indexMatrix` represent general (not having any specific properties), Hermitian or symmetric, triangular, diagonal, and index matrices. `posdefMatrix` is a virtual subclass of `symmetricMatrix` representing positive semidefinite matrices.

The storage format of a Matrix object describes the relation between what is stored in memory and entries in the represented matrix. A matrix can be represented in multiple formats, some more efficient than others depending on, e.g., the sparsity of the matrix and the order in which entries are accessed in computation. Virtual class `sparseMatrix` designates Matrix objects whose format is such that the space required to represent an $m \times n$ matrix with p nonzero entries is $O(\max\{m, n, p\})$. Its “complement” is virtual class `denseMatrix`. `CsparseMatrix`, `RsparseMatrix`, and `TsparseMatrix` are virtual subclasses of `sparseMatrix` corresponding to the compressed sparse column, compressed sparse row, and triplet formats. `unpackedMatrix` and `packedMatrix` are virtual subclasses of `denseMatrix` corresponding to the conventional and packed formats defined by LAPACK, storing data of length mn and $n(n+1)/2$ and listing matrix entries in column-major order: <https://netlib.org/lapack/lug/node121.html>

A canonical example of a nonvirtual subclass of Matrix is `dgCMatrix`. Objects of this class have double precision data, general structure, and compressed sparse column format. Accordingly, `dgCMatrix` extends virtual classes `dMatrix`, `generalMatrix`, and `CsparseMatrix`. Indeed, nonvirtual subclasses of Matrix defined in package **Matrix** tend to have names of the form `xyyMatrix`, where the first character indicates the data type and the second and third characters together indicate the structure and format. This naming scheme is inspired by LAPACK: <https://netlib.org/lapack/lug/node24.html>

Methods

S4 methods are defined for the S4 generic functions listed by `.S4methods(class = "Matrix")`.

S3 methods are defined for the S3 generic functions listed by `.S3methods(class = "Matrix")`. They are defined so that Matrix objects are handled correctly even in environments where the S4 generic functions are not available, such as the evaluation frame of a function from a package that does not import the S4 generic function. Notably, the S3 method defined for `as.matrix` ensures that **base** functions like `apply`, `eigen`, and `svd` behave as documented even if their first argument is a Matrix (rather than matrix) object. Note that this mechanism is not needed for generic functions that dispatch “internally”; see [InternalGenerics](#).

Methods are documented by generic function rather than by class, hence see:

```
help.search(keyword = "methods", package = "Matrix")
help("<generic>-methods", package = "Matrix")
```

Methods defined in package **Matrix** for virtual class Matrix or one of its virtual subclasses can be dispatched for subclasses of Matrix defined in other packages. The behaviour is undefined if the “foreign” subclass does not extend any nonvirtual “local” subclass. Authors of packages that

define such foreign subclasses may need to override local methods with methods of their own. An example is class MultiCompanion in package **mcompanion** (version 0.6 at the time of writing).

See Also

Nonvirtual subclasses such as [dgCMatrix](#) and [dgeMatrix](#). Function [Matrix](#), a widely used (but not the only) constructor for Matrix objects.

Examples

```
cl <- getClass("Matrix")
showClass(cl) # output with all slots, superclasses, subclasses

subcl <- cl@subclasses
subcl <- subcl[vapply(subcl, packageSlot, "") == "Matrix"] # exclude foreign ones
(nms <- names(subcl))
stopifnot(all(endsWith(nms, "Matrix"))) # our convention

iv <- vapply(nms, isVirtualClass, FALSE)
(v <- nms[iv]) # virtual subclasses
(nv <- nms[!iv]) # nonvirtual subclasses
(nv.prefix <- sub("Matrix$", "", nv))
tabulate(nchar(nv.prefix), 6L) # 3-letter prefix, usually ...
```

MatrixClass	<i>The Matrix (Super-) Class of a Class</i>
-------------	---

Description

Return the (maybe super-)class of class `cl` from package **Matrix**, returning `character(0)` if there is none.

Usage

```
MatrixClass(cl, cld = getClassDef(cl), ...Matrix = TRUE,
            dropVirtual = TRUE, ...)
```

Arguments

<code>cl</code>	string, class name
<code>cld</code>	its class definition
<code>...Matrix</code>	logical indicating if the result must be of pattern "[dlniz]..Matrix" where the first letter "[dlniz]" denotes the content kind.
<code>dropVirtual</code>	logical indicating if virtual classes are included or not.
<code>...</code>	further arguments are passed to .selectSuperClasses() .

Value

a `character` string

Author(s)

Martin Maechler, 24 Mar 2009

See Also

`Matrix`, the mother of all **Matrix** classes.

Examples

```
mkA <- setClass("A", contains="dgCMatrix")
(A <- mkA())
stopifnot(identical(
  MatrixClass("A"),
  "dgCMatrix"))
```

MatrixFactorization-class

Virtual Class "MatrixFactorization" of Matrix Factorizations

Description

MatrixFactorization is the virtual class of factorizations of $m \times n$ matrices A , having the general form

$$P_1 A P_2 = A_1 \cdots A_p$$

or (equivalently)

$$A = P_1' A_1 \cdots A_p P_2'$$

where P_1 and P_2 are permutation matrices. Factorizations requiring symmetric A have the constraint $P_2 = P_1'$, and factorizations without row or column pivoting have the constraints $P_1 = I_m$ and $P_2 = I_n$, where I_m and I_n are the $m \times m$ and $n \times n$ identity matrices.

CholeskyFactorization, BunchKaufmanFactorization, SchurFactorization, LU, and QR are the virtual subclasses of MatrixFactorization containing all Cholesky, Bunch-Kaufman, Schur, LU, and QR factorizations, respectively.

Slots

Dim an integer vector of length 2 giving the dimensions of the factorized matrix.

Dimnames a list of length 2 preserving the dimnames of the factorized matrix. Each element must be NULL or a character vector of length equal to the corresponding element of **Dim**.

Methods

`determinant signature(x = "MatrixFactorization", logarithm = "missing")`: sets `logarithm = TRUE` and recalls the generic function.

`dim signature(x = "MatrixFactorization")`: returns `x@Dim`.

`dimnames signature(x = "MatrixFactorization")`: returns `x@Dimnames`.

`dimnames<- signature(x = "MatrixFactorization", value = "NULL")`: returns `x` with `x@Dimnames` set to `list(NULL, NULL)`.

`dimnames<- signature(x = "MatrixFactorization", value = "list")`: returns `x` with `x@Dimnames` set to `value`.

`length signature(x = "MatrixFactorization")`: returns `prod(x@Dim)`.

`show signature(object = "MatrixFactorization")`: prints the internal representation of the factorization using `str`.

`solve signature(a = "MatrixFactorization", b = .)`: see [solve-methods](#).

`unname signature(obj = "MatrixFactorization")`: returns `obj` with `obj@Dimnames` set to `list(NULL, NULL)`.

See Also

Classes extending `CholeskyFactorization`, namely `Cholesky`, `pCholesky`, and `CHMfactor`.

Classes extending `BunchKaufmanFactorization`, namely `BunchKaufman` and `pBunchKaufman`.

Classes extending `SchurFactorization`, namely `Schur`.

Classes extending `LU`, namely [denseLU](#) and [sparseLU](#).

Classes extending `QR`, namely [sparseQR](#).

Generic functions [Cholesky](#), [BunchKaufman](#), [Schur](#), [lu](#), and [qr](#) for *computing* factorizations.

Generic functions [expand1](#) and [expand2](#) for constructing matrix factors from `MatrixFactorization` objects.

Examples

```
showClass("MatrixFactorization")
```

Description

TODO.

nearPD

*Nearest Positive Definite Matrix***Description**

Compute the nearest positive definite matrix to an approximate one, typically a correlation or variance-covariance matrix.

Usage

```
nearPD(x, corr = FALSE, keepDiag = FALSE, base.matrix = FALSE,
       do2eigen = TRUE, doSym = FALSE,
       doDykstra = TRUE, only.values = FALSE,
       ensureSymmetry = !isSymmetric(x),
       eig.tol = 1e-06, conv.tol = 1e-07, posd.tol = 1e-08,
       maxit = 100, conv.norm.type = "I", trace = FALSE)
```

Arguments

x	numeric $n \times n$ approximately positive definite matrix, typically an approximation to a correlation or covariance matrix. If x is not symmetric (and ensureSymmetry is not false), <code>symmpart(x)</code> is used.
corr	logical indicating if the matrix should be a <i>correlation</i> matrix.
keepDiag	logical, generalizing corr: if TRUE, the resulting matrix should have the same diagonal (<code>diag(x)</code>) as the input matrix.
base.matrix	logical indicating if the resulting mat component should be a base matrix or (by default) a Matrix of class <code>dpoMatrix</code> .
do2eigen	logical indicating if a <code>posdefify()</code> eigen step should be applied to the result of the Higham algorithm.
doSym	logical indicating if $X \leftarrow (X + t(X))/2$ should be done, after $X \leftarrow \text{tcrossprod}(Qd, Q)$; some doubt if this is necessary.
doDykstra	logical indicating if Dykstra's correction should be used; true by default. If false, the algorithm is basically the direct fixpoint iteration $Y_k = P_U(P_S(Y_{k-1}))$.
only.values	logical; if TRUE, the result is just the vector of eigenvalues of the approximating matrix.
ensureSymmetry	logical; by default, <code>symmpart(x)</code> is used whenever <code>isSymmetric(x)</code> is not true. The user can explicitly set this to TRUE or FALSE, saving the symmetry test. <i>Beware</i> however that setting it FALSE for an asymmetric input x, is typically nonsense!
eig.tol	defines relative positiveness of eigenvalues compared to largest one, λ_1 . Eigenvalues λ_k are treated as if zero when $\lambda_k/\lambda_1 \leq \text{eig.tol}$.
conv.tol	convergence tolerance for Higham algorithm.
posd.tol	tolerance for enforcing positive definiteness (in the final <code>posdefify</code> step when <code>do2eigen</code> is TRUE).

<code>maxit</code>	maximum number of iterations allowed.
<code>conv.norm.type</code>	convergence norm type (<code>norm(*, type)</code>) used for Higham algorithm. The default is "I" (infinity), for reasons of speed (and back compatibility); using "F" is more in line with Higham's proposal.
<code>trace</code>	logical or integer specifying if convergence monitoring should be traced.

Details

This implements the algorithm of Higham (2002), and then (if `do2eigen` is true) forces positive definiteness using code from `posdefify`. The algorithm of Knol and ten Berge (1989) (not implemented here) is more general in that it allows constraints to (1) fix some rows (and columns) of the matrix and (2) force the smallest eigenvalue to have a certain value.

Note that setting `corr = TRUE` just sets `diag(.) <- 1` within the algorithm.

Higham (2002) uses Dykstra's correction, but the version by Jens Oehlschlägel did not use it (accidentally), and still gave reasonable results; this simplification, now only used if `doDykstra = FALSE`, was active in `nearPD()` up to Matrix version 0.999375-40.

Value

If only `.values = TRUE`, a numeric vector of eigenvalues of the approximating matrix; Otherwise, as by default, an S3 object of class "nearPD", basically a list with components

<code>mat</code>	a matrix of class <code>dpoMatrix</code> , the computed positive-definite matrix.
<code>eigenvalues</code>	numeric vector of eigenvalues of <code>mat</code> .
<code>corr</code>	logical, just the argument <code>corr</code> .
<code>normF</code>	the Frobenius norm (<code>norm(x-X, "F")</code>) of the difference between the original and the resulting matrix.
<code>iterations</code>	number of iterations needed.
<code>converged</code>	logical indicating if iterations converged.

Author(s)

Jens Oehlschlägel donated a first version. Subsequent changes by the Matrix package authors.

References

- Cheng, Sheung Hun and Higham, Nick (1998) A Modified Cholesky Algorithm Based on a Symmetric Indefinite Factorization; *SIAM J. Matrix Anal. Appl.*, **19**, 1097–1110.
- Knol DL, ten Berge JMF (1989) Least-squares approximation of an improper correlation matrix by a proper one. *Psychometrika* **54**, 53–61.
- Higham, Nick (2002) Computing the nearest correlation matrix - a problem from finance; *IMA Journal of Numerical Analysis* **22**, 329–343.

See Also

A first version of this (with non-optional `corr=TRUE`) has been available as `nearcor()`; and more simple versions with a similar purpose `posdefify()`, both from package `sfsmisc`.

Examples

```
## Higham(2002), p.334f - simple example
A <- matrix(1, 3,3); A[1,3] <- A[3,1] <- 0
n.A <- nearPD(A, corr=TRUE, do2eigen=FALSE)
n.A[c("mat", "normF")]
n.A.m <- nearPD(A, corr=TRUE, do2eigen=FALSE, base.matrix=TRUE)$mat
stopifnot(exprs = {                                     #-----
  all.equal(n.A$mat[1,2], 0.760689917)
  all.equal(n.A$normF, 0.52779033, tolerance=1e-9)
  all.equal(n.A.m, unname(as.matrix(n.A$mat)), tolerance = 1e-15)# seen rel.d.= 1.46e-16
})
set.seed(27)
m <- matrix(round(rnorm(25),2), 5, 5)
m <- m + t(m)
diag(m) <- pmax(0, diag(m)) + 1
(m <- round(cov2cor(m), 2))

str(near.m <- nearPD(m, trace = TRUE))
round(near.m$mat, 2)
norm(m - near.m$mat) # 1.102 / 1.08

if(requireNamespace("sfsmisc")) {
  m2 <- sfsmisc::posdefify(m) # a simpler approach
  norm(m - m2) # 1.185, i.e., slightly "less near"
}

round(nearPD(m, only.values=TRUE), 9)

## A longer example, extended from Jens' original,
## showing the effects of some of the options:

pr <- Matrix(c(1,      0.477, 0.644, 0.478, 0.651, 0.826,
               0.477, 1,      0.516, 0.233, 0.682, 0.75,
               0.644, 0.516, 1,      0.599, 0.581, 0.742,
               0.478, 0.233, 0.599, 1,      0.741, 0.8,
               0.651, 0.682, 0.581, 0.741, 1,      0.798,
               0.826, 0.75,  0.742, 0.8,  0.798, 1),
             nrow = 6, ncol = 6)

nc. <- nearPD(pr, conv.tol = 1e-7) # default
nc.$iterations # 2
nc.1 <- nearPD(pr, conv.tol = 1e-7, corr = TRUE)
nc.1$iterations # 11 / 12 (!)
ncr <- nearPD(pr, conv.tol = 1e-15)
str(ncr)# still 2 iterations
ncr.1 <- nearPD(pr, conv.tol = 1e-15, corr = TRUE)
ncr.1 $ iterations # 27 / 30 !

ncF <- nearPD(pr, conv.tol = 1e-15, conv.norm = "F")
stopifnot(all.equal(ncr, ncF))# norm type does not matter at all in this example

## But indeed, the 'corr = TRUE' constraint did ensure a better solution;
```

```
## cov2cor() does not just fix it up equivalently :
norm(pr - cov2cor(ncr$mat)) # = 0.09994
norm(pr -      ncr.1$mat)  # = 0.08746 / 0.08805

### 3) a real data example from a 'systemfit' model (3 eq.):
(load(system.file("external", "symW.rda", package="Matrix"))) # "symW"
dim(symW) # 24 x 24
class(symW) # "dsCMatrix": sparse symmetric
if(dev.interactive()) image(symW)
EV <- eigen(symW, only=TRUE)$values
summary(EV) ## looking more closely {EV sorted decreasingly}:
tail(EV) # all 6 are negative
EV2 <- eigen(sWpos <- nearPD(symW)$mat, only=TRUE)$values
stopifnot(EV2 > 0)
if(requireNamespace("sfsmisc")) {
  plot(pmax(1e-3, EV), EV2, type="o", log="xy", xaxt="n", yaxt="n")
  for(side in 1:2) sfsmisc::eaxis(side)
} else
  plot(pmax(1e-3, EV), EV2, type="o", log="xy")
abline(0, 1, col="red3", lty=2)
```

nMatrix-class

Virtual Class ‘nMatrix’ of Boolean Matrices

Description

nMatrix is a virtual subclass of [Matrix](#) representing boolean (or “nonzero pattern”) matrices, whose entries are FALSE or TRUE. ndenseMatrix and nsparseMatrix are virtual subclasses of nMatrix representing its intersections with [denseMatrix](#) and [sparseMatrix](#).

nMatrix objects tend to arise as the return value of op(A), where A is a Matrix object and op is a function that preserves matrix dimensions and returns FALSE or TRUE by element (e.g., op(A) := is.nan(A)).

Slots

Dim, Dimnames inherited from virtual superclass [Matrix](#).

Certain nonvirtual subclasses of nMatrix defined in package **Matrix** (see ‘Subclasses’) also have:

- x a [logical](#) vector storing matrix entries. Details such as *which* entries are stored vary by storage format. NA elements are valid and mapped to TRUE, hence methods often must operate on not x but rather x | is.na(x).

Subclasses

The nonvirtual subclasses of nMatrix are listed below with links to virtual superclasses defining their structure and storage format. Marked (*) classes have an x slot storing matrix entries. The details of the x slot vary by storage format. Unmarked classes do not have an x slot. These use sparse formats that store positions of nonzero entries instead of entries themselves.

Nonvirtual subclass	Structure	Storage format
* ngeMatrix	generalMatrix	unpackedMatrix
* nsyMatrix	symmetricMatrix	unpackedMatrix
* ntrMatrix	triangularMatrix	unpackedMatrix
* nspMatrix	symmetricMatrix	packedMatrix
* ntpMatrix	triangularMatrix	packedMatrix
ngCMatrix	generalMatrix	CsparseMatrix
nsCMatrix	symmetricMatrix	CsparseMatrix
ntCMatrix	triangularMatrix	CsparseMatrix
ngRMatrix	generalMatrix	RsparseMatrix
nsRMatrix	symmetricMatrix	RsparseMatrix
ntRMatrix	triangularMatrix	RsparseMatrix
ngTMatrix	generalMatrix	TsparseMatrix
nsTMatrix	symmetricMatrix	TsparseMatrix
ntTMatrix	triangularMatrix	TsparseMatrix
* ndiMatrix	diagonalMatrix	unnamed
indMatrix	indexMatrix	unnamed

Class `ndiMatrix` was not defined until **Matrix** version 1.6-2.

See Also

“Sister” classes `lMatrix`, `iMatrix`, `dMatrix`, and `zMatrix` representing logical, integer, double, and complex matrices.

Examples

```
getClass("nMatrix")

L3 <- Matrix(upper.tri(diag(3)))
L3 # an "ltCMatrix"
as(L3, "nMatrix") # -> ntC*

## similar, not using Matrix()
as(upper.tri(diag(3)), "nMatrix")# currently "ngTMatrix"

showClass("ndenseMatrix")
as(diag(3) > 0, "ndenseMatrix")# -> "nge"

(m <- Matrix(c(0,0,2:0), 3,5, dimnames=list(LETTERS[1:3],NULL)))
## ``extract the nonzero-pattern of (m) into an nMatrix'':
nm <- as(m, "nsparseMatrix") ## -> will be a "ngCMatrix"
str(nm) # no 'x' slot
nnm <- !nm # no longer sparse
## consistency check:
stopifnot(xor(as( nm, "matrix"),
               as(nnm, "matrix"))))

## low-level way of adding "non-structural zeros" :
nnm <- as(nnm, "lsparseMatrix") # "lgCMatrix"
nnm@x[2:4] <- c(FALSE, NA, NA)
```

```

nnm
as(nnm, "nMatrix") # NAs *and* non-structural 0 |---> 'TRUE'

data(KNex, package = "Matrix")
nnm <- as(KNex $ mm, "nMatrix")
str(xlx <- crossprod(nnm))# "nsCMatrix"
stopifnot(isSymmetric(xlx))
image(xlx, main=paste("crossprod(nnm) : Sparse", class(xlx)))

```

nnzero-methods

The Number of Non-Zero Values of a Matrix

Description

Returns the number of non-zero values of a numeric-like R object, and in particular an object `x` inheriting from class `Matrix`.

Usage

```

nnzero(x, ...)
## S4 method for signature 'denseMatrix'
nnzero(x, na.counted = NA, ...)
## S4 method for signature 'sparseMatrix'
nnzero(x, na.counted = NA, ...)

```

Arguments

<code>x</code>	an R object, typically inheriting from class <code>Matrix</code> or <code>numeric</code> .
<code>na.counted</code>	a <code>logical</code> describing how NAs should be counted. There are three possible settings for <code>na.counted</code> : TRUE NAs <i>are</i> counted as non-zero (since “they are not zero”). NA (default) the result will be NA if there are NA’s in <code>x</code> (since “NA’s are not known, i.e., <i>may be</i> zero”). FALSE NAs are <i>omitted</i> from <code>x</code> before the non-zero entries are counted. For sparse matrices, you may often want to use <code>na.counted = TRUE</code> .
<code>...</code>	optional arguments passed from or to other methods.

Value

the number of non zero entries in `x` (typically `integer`).

Note that for a *symmetric* sparse matrix `S` (i.e., inheriting from class `symmetricMatrix`), `nnzero(S)` is typically *twice* the `length(S@x)`.

Methods

`signature(x = "ANY")` the default method for non-**Matrix** class objects, simply counts the number 0s in `x`, counting NA's depending on the `na.counted` argument, see above.

`signature(x = "denseMatrix")` conceptually the same as for traditional **matrix** objects, care has to be taken for "**symmetricMatrix**" objects.

`signature(x = "diagonalMatrix")`, **and** `signature(x = "indMatrix")` fast simple methods for these special "sparseMatrix" classes.

`signature(x = "sparseMatrix")` typically, the most interesting method, also carefully taking "**symmetricMatrix**" objects into account.

See Also

The **Matrix** class also has a `length` method; typically, `length(M)` is much larger than `nnzero(M)` for a sparse matrix `M`, and the latter is a better indication of the *size* of `M`.

`drop0`, `zapsmall`.

Examples

```
m <- Matrix(0+1:28, nrow = 4)
m[-3,c(2,4:5,7)] <- m[ 3, 1:4] <- m[1:3, 6] <- 0
(mT <- as(m, "TsparseMatrix"))
nnzero(mT)
(S <- crossprod(mT))
nnzero(S)
str(S) # slots are smaller than nnzero()
stopifnot(nnzero(S) == sum(as.matrix(S) != 0))# failed earlier

data(KNex, package = "Matrix")
M <- KNex$mm
class(M)
dim(M)
length(M); stopifnot(length(M) == prod(dim(M)))
nnzero(M) # more relevant than length
## the above are also visible from
str(M)
```

norm-methods

Matrix Norms

Description

Computes a matrix norm of `x`, using Lapack for dense matrices. The norm can be the one ("0", or "1") norm, the infinity ("I") norm, the Frobenius ("F") norm, the maximum modulus ("M") among elements of a matrix, or the spectral norm or 2-norm ("2"), as determined by the value of `type`.

Usage

```
norm(x, type, ...)
```

Arguments

`x` a real or complex matrix.

`type` A character indicating the type of norm desired.
 "O", "o" **or** "1" specifies the one norm, (maximum absolute column sum);
 "I" **or** "i" specifies the infinity norm (maximum absolute row sum);
 "F", "f", "E" **or** "e" specifies the Frobenius norm (the Euclidean norm of `x` treated as if it were a vector);
 "M" **or** "m" specifies the maximum modulus of all the elements in `x`; and
 "2" specifies the "spectral norm" aka "2-norm", which is the largest singular value ([svd](#)) of `x`.
 The default is "O". Only the first character of `type[1]` is used.

... further arguments passed to or from other methods.

Details

For dense matrices, the methods eventually call the Lapack functions `dlange`, `dlansy`, `dlantr`, `zlange`, `zlansy`, and `zlantr`.

Value

A numeric value of class "norm", representing the quantity chosen according to `type`.

References

Anderson, E., et al. (1994). *LAPACK User's Guide*, 2nd edition, SIAM, Philadelphia.

See Also

[onenormest\(\)](#), an *approximate* randomized estimate of the 1-norm condition number, efficient for large sparse matrices.

The [norm\(\)](#) function from R's **base** package.

Examples

```
x <- Hilbert(9)
norm(x)# = "O" = "1"
stopifnot(identical(norm(x), norm(x, "1")))
norm(x, "I")# the same, because 'x' is symmetric

allnorms <- function(x) {
  ## norm(NA, "2") did not work until R 4.0.0
  do2 <- getRversion() >= "4.0.0" || !anyNA(x)
  vapply(c("1", "I", "F", "M", if(do2) "2"), norm, 0, x = x)
}
allnorms(x)
allnorms(Hilbert(10))

i <- c(1,3:8); j <- c(2,9,6:10); x <- 7 * (1:7)
```

```

A <- sparseMatrix(i, j, x = x) ## 8 x 10 "dgCMatrix"
(sA <- sparseMatrix(i, j, x = x, symmetric = TRUE)) ## 10 x 10 "dsCMatrix"
(tA <- sparseMatrix(i, j, x = x, triangular = TRUE)) ## 10 x 10 "dtCMatrix"
(allnorms(A) -> nA)
allnorms(sA)
allnorms(tA)
stopifnot(all.equal(nA, allnorms(as(A, "matrix"))),
  all.equal(nA, allnorms(tA))) # because tA == rbind(A, 0, 0)
A. <- A; A.[1,3] <- NA
stopifnot(is.na(allnorms(A.))) # gave error

```

not-methods

Logical Negation

Description

TODO.

Usage

```

## S4 method for signature 'ndenseMatrix'
!x
## S4 method for signature 'nsparseMatrix'
!x
## S4 method for signature 'ndiMatrix'
!x
## S4 method for signature 'nsparseVector'
!x

## S4 method for signature 'ldenseMatrix'
!x
## S4 method for signature 'lspareMatrix'
!x
## S4 method for signature 'ldiMatrix'
!x
## S4 method for signature 'lspareVector'
!x

## S4 method for signature 'Matrix'
!x
## S4 method for signature 'sparseVector'
!x

```

Arguments

x .

Description

Many **Matrix** package (binary operator) methods for **Matrix** objects (of super classes [Matrix](#) and [sparseVector](#)) combined with (base) [matrix](#), and atomic vectors [numeric](#), [logical](#), [complex](#), etc.

Methods are both defined as group methods, e.g., for the "Arith" group, and as regular methods for specific binary operators, e.g., +.

Usage

```
Ops(e1, e2)
Arith(e1, e2)
Compare(e1, e2)
Logic(e1, e2)

# e1 + e2
# e1 / e2
# e1 | e2   # etc
```

Arguments

e1, e2 typically “numeric-alike” vectors, “[sparseVector](#)”, or “[Matrix](#)” objects.

Value

typically a class “[Matrix](#)” object of a class related to the classes of e1 and e2.

See Also

Package **methods**: The overview [Methods](#); further [getGroup](#), [getGroupMembers](#)

Examples

```
## 'Ops': the union group of the 3 binary op groups:
getGroupMembers("Ops") # "Arith" "Compare" "Logic"
##
getGroupMembers("Arith") # "+" "-" "*" "^" "%%" "%/%" "/"
getGroupMembers("Compare") # "==" ">" "<" "!=" "<=" ">="
getGroupMembers("Logic") # "&" "|"                    ( *not* containing unary "!" )
```

Description

`pack()` coerces dense symmetric and dense triangular matrices from unpacked format (storing the full matrix) to packed format (storing only one of the upper and lower triangles). `unpack()` performs the reverse coercion. The two formats are formalized by the virtual classes "`packedMatrix`" and "`unpackedMatrix`".

Usage

```
pack(x, ...)
## S4 method for signature 'unpackedMatrix'
pack(x, symmetric = NA, upperTri = NA, ...)
## S4 method for signature 'matrix'
pack(x, symmetric = NA, upperTri = NA, ...)

unpack(x, ...)
```

Arguments

<code>x</code>	A dense symmetric or dense triangular matrix. For <code>pack()</code> : typically an " <code>unpackedMatrix</code> " or a standard " <code>matrix</code> ", though " <code>packedMatrix</code> " are allowed and returned unchanged. For <code>unpack()</code> : typically a " <code>packedMatrix</code> ", though " <code>unpackedMatrix</code> " are allowed and returned unchanged.
<code>symmetric</code>	logical (including NA) optionally indicating whether <code>x</code> is symmetric (or triangular).
<code>upperTri</code>	(for triangular <code>x</code> only) logical (including NA) indicating whether <code>x</code> is upper (or lower) triangular.
<code>...</code>	further arguments passed to or from other methods.

Details

`pack(x)` checks matrices `x` *not* inheriting from one of the virtual classes "`symmetricMatrix`" "`triangularMatrix`" for symmetry (via `isSymmetric()`) then for upper and lower triangularity (via `isTriangular()`) in order to identify a suitable coercion. Setting one or both of `symmetric` and `upperTri` to TRUE or FALSE rather than NA allows skipping of irrelevant tests for large matrices known to be symmetric or (upper or lower) triangular.

Users should *not* assume that `pack()` and `unpack()` are inverse operations. Specifically, `y <- unpack(pack(x))` may not reproduce an "`unpackedMatrix`" `x` in the sense of `identical()`. See the examples.

Value

For `pack()`: a "packedMatrix" giving the condensed representation of `x`.

For `unpack()`: an "unpackedMatrix" giving the full storage representation of `x`.

Examples

```
showMethods("pack")
(s <- crossprod(matrix(sample(15), 5,3))) # traditional symmetric matrix
(sp <- pack(s))
(mt <- as.matrix(tt <- tril(s))
(pt <- pack(mt))
stopifnot(identical(pt, pack(tt)),
  dim(s ) == dim(sp), all(s == sp),
  dim(mt) == dim(pt), all(mt == pt), all(mt == tt))

showMethods("unpack")
(cp4 <- chol(Hilbert(4))) # is triangular
(tp4 <- pack(cp4) # [t]riangular [p]acked
str(tp4)
(unpack(tp4))
stopifnot(identical(tp4, pack(unpack(tp4))))

z1 <- new("dsyMatrix", Dim = c(2L, 2L), x = as.double(1:4), uplo = "U")
z2 <- unpack(pack(z1))
stopifnot(!identical(z1, z2), # _not_ identical
  all(z1 == z2)) # but mathematically equal
cbind(z1@x, z2@x) # (unused!) lower triangle is "lost" in translation
```

packedMatrix-class

Virtual Class 'packedMatrix' of Packed Dense Format Matrices

Description

`packedMatrix` is a virtual subclass of `denseMatrix` representing packed dense format matrices, as defined by LAPACK: <https://netlib.org/lapack/lug/node121.html>. To represent an n -by- n symmetric, Hermitian, or triangular matrix A , this format stores a vector of length `choose(n+1, 2) == n*(n+1)/2` (the number of entries in the upper or lower triangle), of which $n*(n+1)/2$ or fewer elements are actually used (fewer if there are additional formal constraints on the structure of A).

Slots

`Dim`, `Dimnames` inherited from virtual superclass `Matrix`. `Dim[1]` and `Dim[2]` must be equal.

`uplo` a character string, either "U" or "L". "U" indicates that `x` stores only the upper triangular entries of A . "L" indicates that `x` stores only the lower triangular entries of A .

`x` a vector of length $n*(n+1)/2$, $n=Dim[1]=Dim[2]$, storing upper or lower triangular (depending on `uplo`) entries of A in column-major order. Authors of subclasses of `packedMatrix` may declare that only a subset k of the elements of `x` is used to determine A , defining implicitly a function F such that $A = F(x[k])$; see 'Subclasses' for examples in package **Matrix**.

Subclasses

The nonvirtual subclasses of `packedMatrix` defined in package **Matrix** are listed below with links to virtual superclasses defining their data type and structure.

Nonvirtual subclass	Data type	Structure
<code>nspMatrix</code>	nMatrix	symmetricMatrix
<code>lspMatrix</code>	lMatrix	symmetricMatrix
<code>ispMatrix</code>	iMatrix	symmetricMatrix
<code>dspMatrix</code>	dMatrix	symmetricMatrix
<code>zspMatrix</code>	zMatrix	symmetricMatrix
<code>dppMatrix</code>	dMatrix	posdefMatrix
<code>zppMatrix</code>	zMatrix	posdefMatrix
<code>ntpMatrix</code>	nMatrix	triangularMatrix
<code>ltpMatrix</code>	lMatrix	triangularMatrix
<code>itpMatrix</code>	iMatrix	triangularMatrix
<code>dtpMatrix</code>	dMatrix	triangularMatrix
<code>ztpMatrix</code>	zMatrix	triangularMatrix

The subclasses of `iMatrix` and `zMatrix` were not defined until **Matrix** version 1.8-0. The other subclasses predate `packedMatrix` itself, which was not defined until **Matrix** version 1.4-1; previously, those subclasses were direct extensions of `denseMatrix`.

The mapping from the `x` slot of length $n*(n+1)/2$ to the n -by- n represented matrix A is given by:

	$i < j$	$i = j$	$i > j$
	$A[i, j]$		
<code>.spMatrix, uplo="U"</code>	$x[i + j*(j-1)/2]$	$x[i + j*(j-1)/2]$	$x[j + i*(i-1)/2]$
<code>zspMatrix, uplo="U", trans="C"</code>	$x[i + j*(j-1)/2]$	$\text{Re}(x[i + j*(j-1)/2]) + 0i$	$\text{Conj}(x[j + i*(i-1)/2])$
<code>.spMatrix, uplo="L"</code>	$x[j + (2*n-i)*(i-1)/2]$	$x[i + (2*n-j)*(j-1)/2]$	$x[i + (2*n-j)*(j-1)/2]$
<code>zspMatrix, uplo="L", trans="C"</code>	$\text{Conj}(x[j + (2*n-i)*(i-1)/2])$	$\text{Re}(x[i + (2*n-j)*(j-1)/2]) + 0i$	$x[i + (2*n-j)*(j-1)/2]$
<code>.tpMatrix, uplo="U", diag="N"</code>	$x[i + j*(j-1)/2]$	$x[i + j*(j-1)/2]$	0
<code>.tpMatrix, uplo="U", diag="U"</code>	$x[i + j*(j-1)/2]$	1	0
<code>.tpMatrix, uplo="L", diag="N"</code>	0	$x[i + (2*n-j)*(j-1)/2]$	$x[i + (2*n-j)*(j-1)/2]$
<code>.tpMatrix, uplo="L", diag="U"</code>	0	1	$x[i + (2*n-j)*(j-1)/2]$

Classes `.spMatrix` represent Hermitian or symmetric A . Objects have `x` and `uplo` slots. If `uplo="U"`, then `x` stores the $n*(n+1)/2$ upper triangular entries of A . If `uplo="L"`, then `x` stores the $n*(n+1)/2$ lower triangular entries of A . Objects of class `zspMatrix` have an additional `trans` slot that distinguishes Hermitian and symmetric A , which use `trans="C"` and `"T"`, respectively. If `trans="C"`, then the imaginary parts of `x[du]` (`uplo="U"`) or `x[d1]` (`uplo="L"`) are not referenced, where `du=d`, `d1=length(x)+1-d`, and `d=cumsum(seq_len(n))`. (The diagonal entries of Hermitian matrices are real by definition.)

Classes `dppMatrix` and `zppMatrix` extend `dspMatrix` and `zspMatrix` and represent positive semidefinite A . Positive semidefiniteness of A induces a constraint on the `x` slot. However, this constraint is not strictly enforced by the validity methods, which test only that the diagonal entries of A are nonnegative (a necessary, not sufficient condition). For `zppMatrix`, only `trans="C"` is valid. Class `copMatrix` extends `dppMatrix` and represents correlation matrices; it has an additional `sd` slot, a numeric vector of length n listing standard deviations, enabling recovery of a covariance matrix.

Classes `.tpMatrix` represent triangular A. Objects have `x`, `uplo`, and `diag` slots. If `uplo="U"`, then A is upper triangular and `x[-du]` stores the $n*(n-1)/2$ strictly upper triangular entries. If `uplo="L"`, then A is lower triangular and `x[-dl]` stores the $n*(n-1)/2$ strictly lower triangular entries. If `diag="N"`, then `x[du]` (`uplo="U"`) or `x[dl]` (`uplo="L"`) stores the n diagonal entries of A. If `diag="U"`, then A is unit triangular and those elements of `x` are unreferenced.

See Also

“Complementary” class `unpackedMatrix` representing conventional (or “unpacked”) dense format matrices. Generic function `unpack`, an alternative to explicit coercion to `unpackedMatrix`.

Examples

```
showClass("packedMatrix")
showMethods(classes = "packedMatrix")

(s0 <- new("nsyMatrix"))

(M2 <- Matrix(c(TRUE, NA, FALSE, FALSE), 2, 2)) # logical dense (ltr)
(sM <- M2 & t(M2))                               # -> "lge"
class(sM <- as(sM, "nMatrix"))                   # -> "nge"
      (sM <- as(sM, "symmetricMatrix"))           # -> "nsy"
str(sM <- as(sM, "packedMatrix")) # -> "nsp", i.e., packed symmetric

(M2 <- Matrix(c(TRUE, NA, FALSE, FALSE), 2, 2)) # logical dense (ltr)
str(M2)
# can
(sM <- M2 | t(M2)) # "lge"
as(sM, "symmetricMatrix")
str(sM <- as(sM, "packedMatrix")) # packed symmetric

## Only upper triangular part matters (when uplo == "U" as per default)
(sy2 <- new("dsyMatrix", Dim = as.integer(c(2,2)), x = c(14, NA, 32, 77)))
str(t(sy2)) # uplo = "L", and the lower tri. (i.e. NA is replaced).

chol(sy2) #-> "Cholesky" matrix
(sp2 <- pack(sy2)) # a "dspMatrix"

## Coercing to dpoMatrix gives invalid object:
sy3 <- new("dsyMatrix", Dim = as.integer(c(2,2)), x = c(14, -1, 2, -7))
try(as(sy3, "dpoMatrix")) # -> error: not positive definite

## 4x4 example
m <- matrix(0,4,4); m[upper.tri(m)] <- 1:6
(sym <- m+t(m)+diag(11:14, 4))
(S1 <- pack(sym))
(S2 <- t(S1))
stopifnot(all(S1 == S2)) # equal "seen as matrix", but differ internally :
str(S1)
S2@x
```

```

showClass("ntrMatrix")

str(new("ntpMatrix"))
(nutr <- as(upper.tri(matrix(, 4, 4)), "ndenseMatrix"))
str(nutp <- pack(nutr)) # packed matrix: only 10 = 4*(4+1)/2 entries
!nutp # the logical negation (is *not* logical triangular !)
## but this one is:
stopifnot(all.equal(nutp, pack(!!nutp)))

showClass("ltrMatrix")

str(new("ltpMatrix"))
(lutr <- as(upper.tri(matrix(, 4, 4)), "ldenseMatrix"))
str(lutp <- pack(lutr)) # packed matrix: only 10 = 4*(4+1)/2 entries
!lutp # the logical negation (is *not* logical triangular !)
## but this one is:
stopifnot(all.equal(lutp, pack(!!lutp)))

(m <- rbind(2:3, 0:-1)+0)
(M <- as(m, "generalMatrix"))

(T <- as(M, "triangularMatrix")) # formally upper triangular
(T2 <- as(t(M), "triangularMatrix"))
stopifnot(T@uplo == "U", T2@uplo == "L", identical(T2, t(T)))

m <- matrix(0,4,4); m[upper.tri(m)] <- 1:6
(t1 <- Matrix(m+diag(,4)))
str(t1p <- pack(t1))
(t1pu <- diagN2U(t1p))
stopifnot(exprs = {
  inherits(t1, "dtrMatrix"); validObject(t1)
  inherits(t1p, "dtpMatrix"); validObject(t1p)
  inherits(t1pu, "dtCMatrix"); validObject(t1pu)
  t1pu@x == 1:6
  all(t1pu == t1p)
  identical((t1pu - t1)@x, numeric())# sparse all-0
})

showClass("dtpMatrix")

(p1 <- pack(T2))
str(p1)
(pp <- pack(T))
ip1 <- solve(p1)
stopifnot(length(p1@x) == 3, length(pp@x) == 3,
  p1 @ uplo == T2 @ uplo, pp @ uplo == T @ uplo,
  identical(t(pp), p1), identical(t(p1), pp),
  all((l.d <- p1 - T2) == 0), is(l.d, "dtpMatrix"),
  all((u.d <- pp - T) == 0), is(u.d, "dtpMatrix"),
  l.d@uplo == T2@uplo, u.d@uplo == T@uplo,
  identical(t(ip1), solve(pp)), is(ip1, "dtpMatrix"),
  all.equal(as(solve(p1,p1), "diagonalMatrix"), Diagonal(2)))

```

posdefMatrix-class *Virtual Class ‘posdefMatrix’ of Positive Semidefinite Matrices*

Description

posdefMatrix is a virtual subclass of [symmetricMatrix](#) representing positive semidefinite matrices. Positive semidefinite matrices tend to arise as the product of a matrix and its conjugate transpose ($\text{ct}(x) \%*\% x$ or $x \%*\% \text{ct}(x)$; see [ct-methods](#)). In particular, covariance and correlation matrices are positive semidefinite.

posdefMatrix was not defined until **Matrix** version 1.8-0.

Slots

Dim, Dimnames inherited from virtual superclass [Matrix](#).

factors inherited from virtual superclass [symmetricMatrix](#).

Subclasses

The direct, nonvirtual subclasses of posdefMatrix defined in package **Matrix** are listed below with links to virtual superclasses defining their data type and storage format.

Nonvirtual subclass	Data type	Storage format
dpoMatrix	dMatrix	unpackedMatrix
zpoMatrix	zMatrix	unpackedMatrix
dppMatrix	dMatrix	packedMatrix
zppMatrix	zMatrix	packedMatrix
dpCMatrix	dMatrix	CsparseMatrix
zpCMatrix	zMatrix	CsparseMatrix
dpRMatrix	dMatrix	RsparseMatrix
zpRMatrix	zMatrix	RsparseMatrix
dpTMatrix	dMatrix	TsparseMatrix
zpTMatrix	zMatrix	TsparseMatrix

The subclasses of [zMatrix](#), [CsparseMatrix](#), [RsparseMatrix](#), and [TsparseMatrix](#) were not defined until **Matrix** version 1.8-0.

Each of the above extends a nonvirtual class `ds.Matrix` or `zs.Matrix` representing Hermitian or symmetric matrices more generally; `dpoMatrix` and `dppMatrix` are extended by nonvirtual classes `corMatrix` and `copMatrix` representing correlation matrices. For implementation details, see the help topic for the storage format.

Note

Tests for positive semidefiniteness in finite precision arithmetic are generally expensive and inexact. Validity methods for nonvirtual subclasses of posdefMatrix defined in package **Matrix** test only that the diagonal entries of the represented matrix are nonnegative (a necessary, not sufficient condition). Meanwhile, methods for *coercion* to posdefMatrix defined in package **Matrix** test that

the represented matrix has a Cholesky factorization (a sufficient, not necessary condition, being equivalent to positive *definiteness*) by evaluating a suitable call to [Cholesky](#). A more rigorous test for positive semidefiniteness, supporting matrices which are not positive definite, would use the Bunch-Kaufman or pivoted Cholesky factorization; see [BunchKaufman](#) and [Cholesky](#).

See Also

Related classes [generalMatrix](#), [symmetricMatrix](#), [triangularMatrix](#), [diagonalMatrix](#), and [indexMatrix](#) representing general (unstructured, possibly nonsquare), Hermitian or symmetric, triangular, diagonal, and index matrices. Generic functions [crossprod](#) and [tcrossprod](#) for a common way to generate positive semidefinite matrices. Generic function [Cholesky](#) for (pivoted or unpivoted) Cholesky factorization of positive semidefinite matrices. Generic function [chol](#) for getting the upper triangular Cholesky factor.

Examples

```
h6 <- Hilbert(6)
rcond(h6)
str(h6)
h6 * 27720 # is ``integer``
solve(h6)
str(hp6 <- pack(h6))

### Note that as(*, "corMatrix") *scales* the matrix
(ch6 <- as(h6, "corMatrix"))
stopifnot(all.equal(as(h6 * 27720, "dsyMatrix"), round(27720 * h6),
                    tolerance = 1e-14),
          all.equal(ch6@sd^(-2), 2*(1:6)-1,
                    tolerance = 1e-12))
chch <- Cholesky(ch6, perm = FALSE)
stopifnot(identical(chch, ch6@factors[["denseCholesky+-"]]),
          all(abs(crossprod(as(chch, "dtrMatrix")) - ch6) < 1e-10))
```

printSpMatrix

Format and Print Sparse Matrices Flexibly

Description

Format and print sparse matrices flexibly. These are the “workhorses” used by the [format](#), [show](#) and [print](#) methods for sparse matrices. If *x* is large, `printSpMatrix2(x)` calls `printSpMatrix()` twice, namely, for the first and the last few rows, suppressing those in between, and also suppresses columns when *x* is too wide.

`printSpMatrix()` basically prints the result of `formatSpMatrix()`.

Usage

```
formatSpMatrix(x, digits = NULL, maxp = 1e9,
               cld = getClassDef(class(x)), zero.print = ".",
               col.names, note.dropping.colnames = TRUE, uniDiag = TRUE,
               align = c("fancy", "right"), ...)

printSpMatrix(x, digits = NULL, maxp = max(100L, getOption("max.print")),
              cld = getClassDef(class(x)),
              zero.print = ".", col.names, note.dropping.colnames = TRUE,
              uniDiag = TRUE, col.trailer = "",
              align = c("fancy", "right"), ...)

printSpMatrix2(x, digits = NULL, maxp = max(100L, getOption("max.print")),
               zero.print = ".", col.names, note.dropping.colnames = TRUE,
               uniDiag = TRUE, suppRows = NULL, suppCols = NULL,
               col.trailer = if(suppCols) "....." else "",
               align = c("fancy", "right"),
               width = getOption("width"), fitWidth = TRUE, ...)
```

Arguments

<code>x</code>	an R object inheriting from class <code>sparseMatrix</code> .
<code>digits</code>	significant digits to use for printing, see <code>print.default</code> , the default, <code>NULL</code> , corresponds to using <code>getOption("digits")</code> .
<code>maxp</code>	integer, default from <code>options(max.print)</code> , influences how many entries of large matrices are printed at all. Typically should not be smaller than around 1000; values smaller than 100 are silently “rounded up” to 100.
<code>cld</code>	the class definition of <code>x</code> ; must be equivalent to <code>getClassDef(class(x))</code> and exists mainly for possible speedup.
<code>zero.print</code>	character which should be printed for <i>structural</i> zeroes. The default <code>"."</code> may occasionally be replaced by <code>" "</code> (blank); using <code>"0"</code> would look almost like <code>print()</code> ing of non-sparse matrices.
<code>col.names</code>	logical or string specifying if and how column names of <code>x</code> should be printed, possibly abbreviated. The default is taken from <code>options("sparse.colnames")</code> if that is set, otherwise <code>FALSE</code> unless there are less than ten columns. When <code>TRUE</code> the full column names are printed. When <code>col.names</code> is a string beginning with <code>"abb"</code> or <code>"sub"</code> and ending with an integer <code>n</code> (i.e., of the form <code>"abb... <n>"</code>), the column names are <code>abbreviate()</code> d or <code>substring()</code> ed to (target) length <code>n</code> , see the examples.
<code>note.dropping.colnames</code>	logical specifying, when <code>col.names</code> is <code>FALSE</code> if the dropping of the column names should be noted, <code>TRUE</code> by default.
<code>uniDiag</code>	logical indicating if the diagonal entries of a sparse unit triangular or unit-diagonal matrix should be formatted as <code>"I"</code> instead of <code>"1"</code> (to emphasize that the 1's are “structural”).

col.trailer	a string to be appended to the right of each column; this is typically made use of by <code>show(<sparseMatrix>)</code> only, when suppressing columns.
suppRows, suppCols	logicals or NULL, for <code>printSpMatrix2()</code> specifying if rows or columns should be suppressed in printing. If NULL, sensible defaults are determined from <code>dim(x)</code> and <code>options(c("width", "max.print"))</code> . Setting both to FALSE may be a very bad idea.
align	a string specifying how the <code>zero.print</code> codes should be aligned, i.e., padded as strings. The default, "fancy", takes some effort to align the typical <code>zero.print = "."</code> with the position of 0, i.e., the first decimal (one left of decimal point) of the numbers printed, whereas <code>align = "right"</code> just makes use of <code>print(*, right = TRUE)</code> .
width	number, a positive integer, indicating the approximately desired (line) width of the output, see also <code>fitWidth</code> .
fitWidth	logical indicating if some effort should be made to match the desired width or temporarily enlarge that if deemed necessary.
...	unused optional arguments.

Details

formatSpMatrix: If `x` is large, only the first rows making up the approximately first `maxp` entries is used, otherwise all of `x`. `.formatSparseSimple()` is applied to (a dense version of) the matrix. Then, `formatSparseM` is used, unless in trivial cases or for sparse matrices without a slot.

Value

`formatSpMatrix()`
returns a character matrix with possibly empty column names, depending on `col.names` etc, see above.

`printSpMatrix*()`
return `x` *invisibly*, see `invisible`.

Author(s)

Martin Maechler

See Also

the virtual class `sparseMatrix` and the classes extending it; maybe `sparseMatrix` or `spMatrix` as simple constructors of such matrices.

The underlying utilities `formatSparseM` and `.formatSparseSimple()` (on the same page).

Examples

```
f1 <- gl(5, 3, labels = LETTERS[1:5])
X <- as(f1, "sparseMatrix")
X ## <==> show(X) <==> print(X)
```

```

t(X) ## shows column names, since only 5 columns

X2 <- as(g1(12, 3, labels = paste(LETTERS[1:12], "c", sep=".")),
        "sparseMatrix")
X2
## less nice, but possible:
print(X2, col.names = TRUE) # use [,1] [,2] .. => does not fit

## Possibilities with column names printing:
  t(X2) # suppressing column names
print(t(X2), col.names=TRUE)
print(t(X2), zero.print = "", col.names="abbr. 1")
print(t(X2), zero.print = "-", col.names="substring 2")

```

qr-methods

*Methods for QR Factorization***Description**

Computes the pivoted QR factorization of an $m \times n$ real matrix A , which has the general form

$$P_1 A P_2 = Q R$$

or (equivalently)

$$A = P_1' Q R P_2'$$

where P_1 and P_2 are permutation matrices, $Q = \prod_{j=1}^n H_j$ is an $m \times m$ orthogonal matrix equal to the product of n Householder matrices H_j , and R is an $m \times n$ upper trapezoidal matrix.

`denseMatrix` use the default method implemented in **base**, namely `qr.default`. It is built on LINPACK routine `dqrdc` and LAPACK routine `dgeqp3`, which do not pivot rows, so that P_1 is an identity matrix.

Methods for `sparseMatrix` are built on CXSparse routines `cs_sqr` and `cs_qr`, which require $m \geq n$.

Usage

```

qr(x, ...)
## S4 method for signature 'CsparseMatrix'
qr(x, order = 3L, ...)

```

Arguments

x a `finite` matrix or `Matrix` to be factorized, satisfying `nrow(x) >= ncol(x)` if sparse.

order an integer in 0:3 passed to CXSparse routine `cs_sqr`, indicating a strategy for choosing the column permutation P_2 . 0 means no column permutation. 1, 2, and 3 indicate a fill-reducing ordering of $A + A'$, $\tilde{A}'\tilde{A}$, and $A'A$, where $\tilde{A}$ is A with “dense” rows removed. Do not set to 0 unless you know that the column order of A is already sensible.

... further arguments passed to or from methods.

Details

If x is sparse and structurally rank deficient, having structural rank $r < n$, then x is augmented with $(n - r)$ rows of (partly non-structural) zeros, such that the augmented matrix has structural rank n . This augmented matrix is factorized as described above:

$$P_1 A P_2 = P_1 \begin{bmatrix} A_0 \\ 0 \end{bmatrix} P_2 = Q R$$

where A_0 denotes the original, user-supplied $(m - (n - r)) \times n$ matrix.

Value

An object representing the factorization, inheriting from virtual S4 class `QR` or S3 class `qr`. The specific class is `qr` unless x inherits from virtual class `sparseMatrix`, in which case it is `sparseQR`.

References

Davis, T. A. (2006). *Direct methods for sparse linear systems*. Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898718881

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

Class `sparseQR` and its methods.

Class `dgCMatrix`.

Generic function `qr` from **base**, whose default method `qr.default` “defines” the S3 class `qr` of dense QR factorizations.

Generic functions `expand1` and `expand2`, for constructing matrix factors from the result.

Generic functions `Cholesky`, `BunchKaufman`, `Schur`, and `lu`, for computing other factorizations.

Examples

```
showMethods("qr", inherited = FALSE)

## Rank deficient: columns 3 {b2} and 6 {c3} are "extra"
M <- as(cbind(a1 = 1,
              b1 = rep(c(1, 0), each = 3L),
              b2 = rep(c(0, 1), each = 3L),
              c1 = rep(c(1, 0, 0), 2L),
              c2 = rep(c(0, 1, 0), 2L),
```

```

        c3 = rep(c(0, 0, 1), 2L)),
        "CsparseMatrix")
rownames(M) <- paste0("r", seq_len(nrow(M)))
b <- 1:6
eps <- .Machine$double.eps

## .... [1] full rank .....
## ==> a least squares solution of  $Ax = b$  exists
##      and is unique _in exact arithmetic_

(A1 <- M[, -c(3L, 6L)])
(qr.A1 <- qr(A1))

stopifnot(exprs = {
  rankMatrix(A1) == ncol(A1)
  { d1 <- abs(diag(qr.A1@R)); sum(d1 < max(d1) * eps) == 0L }
  rcond(crossprod(A1)) >= eps
  all.equal(qr.coef(qr.A1, b), drop(solve(crossprod(A1), crossprod(A1, b))))
  all.equal(qr.fitted(qr.A1, b) + qr.resid(qr.A1, b), b)
})

## .... [2] numerically rank deficient with full structural rank .....
## ==> a least squares solution of  $Ax = b$  does not
##      exist or is not unique _in exact arithmetic_

(A2 <- M)
(qr.A2 <- qr(A2))

stopifnot(exprs = {
  rankMatrix(A2) == ncol(A2) - 2L
  { d2 <- abs(diag(qr.A2@R)); sum(d2 < max(d2) * eps) == 2L }
  rcond(crossprod(A2)) < eps

  ## 'qr.coef' computes unique least squares solution of "nearby" problem
  ##  $Zx = b$  for some full rank  $Z \sim A$ , currently without warning {FIXME} !
  tryCatch({ qr.coef(qr.A2, b); TRUE }, condition = function(x) FALSE)

  all.equal(qr.fitted(qr.A2, b) + qr.resid(qr.A2, b), b)
})

## .... [3] numerically and structurally rank deficient .....
## ==> factorization of _augmented_ matrix with
##      full structural rank proceeds as in [2]

## NB: implementation details are subject to change; see (*) below

A3 <- M
A3[, c(3L, 6L)] <- 0
A3
(qr.A3 <- qr(A3)) # with a warning ... "additional 2 row(s) of zeros"

stopifnot(exprs = {
  ## sparseQR object preserves the unaugmented dimensions (*)

```

```

dim(qr.A3 ) == dim(A3)
dim(qr.A3@V) == dim(A3) + c(2L, 0L)
dim(qr.A3@R) == dim(A3) + c(2L, 0L)

## The augmented matrix remains numerically rank deficient
rankMatrix(A3) == ncol(A3) - 2L
{ d3 <- abs(diag(qr.A3@R)); sum(d3 < max(d3) * eps) == 2L }
rcond(crossprod(A3)) < eps
})

## Auxiliary functions accept and return a vector or matrix
## with dimensions corresponding to the unaugmented matrix (*),
## in all cases with a warning
qr.coef (qr.A3, b)
qr.fitted(qr.A3, b)
qr.resid (qr.A3, b)

## .... [4] yet more examples .....

## By disabling column pivoting, one gets the "vanilla" factorization
## A = Q~ R, where Q~ := P1' Q is orthogonal because P1 and Q are

(qr.A1.pp <- qr(A1, order = 0L)) # partial pivoting

ae1 <- function(a, b, ...) all.equal(as(a, "matrix"), as(b, "matrix"), ...)
ae2 <- function(a, b, ...) ae1(unname(a), unname(b), ...)

stopifnot(exprs = {
  length(qr.A1 @q) == ncol(A1)
  length(qr.A1.pp@q) == 0L # indicating no column pivoting
  ae2(A1[, qr.A1@q + 1L], qr.Q(qr.A1 ) %*% qr.R(qr.A1 ))
  ae2(A1
      , qr.Q(qr.A1.pp) %*% qr.R(qr.A1.pp))
})

```

rankMatrix

*Rank of a Matrix***Description**

Compute ‘the’ matrix rank, a well-defined functional in theory(*), somewhat ambiguous in practice. We provide several methods, the default corresponding to Matlab’s definition.

(*) The rank of a $n \times m$ matrix A , $rk(A)$, is the maximal number of linearly independent columns (or rows); hence $rk(A) \leq \min(n, m)$.

Usage

```

rankMatrix(x, tol = NULL,
           method = c("tolNorm2", "qr.R", "qrLINPACK", "qr",
                     "useGrad", "maybeGrad"),

```

```
sval = svd(x, 0, 0)$d, warn.t = TRUE, warn.qr = TRUE)
```

```
qr2rankMatrix(qr, tol = NULL, isBqr = is.qr(qr), do.warn = TRUE)
```

Arguments

<code>x</code>	numeric matrix, of dimension $n \times m$, say.
<code>tol</code>	nonnegative number specifying a (relative, “scalefree”) tolerance for testing of “practically zero” with specific meaning depending on method; by default, $\max(\dim(x)) * .Machine$double.eps$ is according to Matlab’s default (for its only method which is our <code>method="tolNorm2"</code>).
<code>method</code>	a character string specifying the computational method for the rank, can be abbreviated: <code>"tolNorm2"</code>: the number of singular values $\geq \text{tol} * \max(\text{sval})$; <code>"qrLINPACK"</code>: for a dense matrix, this is the rank of <code>qr(x, tol, LAPACK=FALSE)</code> (which is <code>qr(...)\$rank</code>); This (<code>"qr"</code>, dense) version used to be <i>the</i> recommended way to compute a matrix rank for a while in the past. For sparse <code>x</code>, this is equivalent to <code>"qr.R"</code>. <code>"qr.R"</code>: this is the rank of triangular matrix R, where <code>qr()</code> uses LAPACK or a <code>"sparseQR"</code> method (see qr-methods) to compute the decomposition QR. The rank of R is then defined as the number of “non-zero” diagonal entries d_i of R, and “non-zero”s fulfill $d_i \geq \text{tol} \cdot \max(d_i)$. <code>"qr"</code>: is for back compatibility; for dense <code>x</code>, it corresponds to <code>"qrLINPACK"</code>, whereas for sparse <code>x</code>, it uses <code>"qr.R"</code>. For all the <code>"qr"</code> methods, singular values <code>sval</code> are not used, which may be crucially important for a large sparse matrix <code>x</code>, as in that case, when <code>sval</code> is not specified, the default, computing <code>svd()</code> currently coerces <code>x</code> to a dense matrix. <code>"useGrad"</code>: considering the “gradient” of the (decreasing) singular values, the index of the <i>smallest</i> gap. <code>"maybeGrad"</code>: choosing method <code>"useGrad"</code> only when that seems <i>reasonable</i>; otherwise using <code>"tolNorm2"</code>.
<code>sval</code>	numeric vector of non-increasing singular values of <code>x</code> ; typically unspecified and computed from <code>x</code> when needed, i.e., unless <code>method = "qr"</code> .
<code>warn.t</code>	logical indicating if <code>rankMatrix()</code> should warn when it needs <code>t(x)</code> instead of <code>x</code> . Currently, for <code>method = "qr"</code> only, gives a warning by default because the caller often could have passed <code>t(x)</code> directly, more efficiently.
<code>warn.qr</code>	in the QR cases (i.e., if method starts with <code>"qr"</code>), <code>rankMatrix()</code> calls <code>qr2rankMarix(..., do.warn = warn.qr)</code> , see below.
<code>qr</code>	an R object resulting from <code>qr(x, ...)</code> , i.e., typically inheriting from <code>class "qr"</code> or <code>"sparseQR"</code> .
<code>isBqr</code>	<code>logical</code> indicating if <code>qr</code> is resulting from <code>base qr()</code> . (Otherwise, it is typically from Matrix package <code>sparse qr</code> .)
<code>do.warn</code>	logical; if true, warn about non-finite diagonal entries in the R matrix of the QR decomposition. Do not change lightly!

Details

`qr2rankMatrix()` is typically called from `rankMatrix()` for the "qr"* methods, but can be used directly - much more efficiently in case the qr-decomposition is available anyway.

Value

If `x` is a matrix of all 0 (or of zero dimension), the rank is zero; otherwise, typically a positive integer in `1:min(dim(x))` with attributes detailing the method used.

There are rare cases where the sparse *QR* decomposition “fails” in so far as the diagonal entries of *R*, the d_i (see above), end with non-finite, typically `NaN` entries. Then, a warning is signalled (unless `warn.qr / do.warn` is not true) and `NA` (specifically, `NA_integer_`) is returned.

Note

For large sparse matrices `x`, unless you can specify `sval` yourself, currently `method = "qr"` may be the only feasible one, as the others need `sval` and call `svd()` which currently coerces `x` to a `denseMatrix` which may be very slow or impossible, depending on the matrix dimensions.

Note that in the case of sparse `x`, `method = "qr"`, all non-strictly zero diagonal entries d_i where counted, up to including **Matrix** version 1.1-0, i.e., that method implicitly used `tol = 0`, see also the `set.seed(42)` example below.

Author(s)

Martin Maechler; for the "*Grad" methods building on suggestions by Ravi Varadhan.

See Also

`qr`, `svd`.

Examples

```
rankMatrix(cbind(1, 0, 1:3)) # 2

(meths <- eval(formals(rankMatrix)$method))

## a "border" case:
H12 <- Hilbert(12)
rankMatrix(H12, tol = 1e-20) # 12; but 11 with default method & tol.
sapply(meths, function(.m.) rankMatrix(H12, method = .m.))
## tolNorm2 qr.R qrLINPACK qr useGrad maybeGrad
##      11      11      12      12      11      11
## The meaning of 'tol' for method="qrLINPACK" and *dense* x is not entirely "scale free"
rMQL <- function(ex, M) rankMatrix(M, method="qrLINPACK", tol = 10^-ex)
rMQR <- function(ex, M) rankMatrix(M, method="qr.R",      tol = 10^-ex)
sapply(5:15, rMQL, M = H12) # result is platform dependent
##  7  7  8 10 10 11 11 11 12 12 12 {x86_64}
sapply(5:15, rMQL, M = 1000 * H12) # not identical unfortunately
##  7  7  8 10 11 11 12 12 12 12 12
sapply(5:15, rMQR, M = H12)
##  5  6  7  8  8  9  9 10 10 11 11
```

```

sapply(5:15, rMQR, M = 1000 * H12) # the *same*

## "sparse" case:
M15 <- kronecker(diag(x=c(100,1,10)), Hilbert(5))
sapply(meths, function(.m.) rankMatrix(M15, method = .m.))
#--> all 15, but 'useGrad' has 14.
sapply(meths, function(.m.) rankMatrix(M15, method = .m., tol = 1e-7)) # all 14

## "large" sparse
n <- 250000; p <- 33; nnz <- 10000
L <- sparseMatrix(i = sample.int(n, nnz, replace=TRUE),
                  j = sample.int(p, nnz, replace=TRUE),
                  x = rnorm(nnz))
(st1 <- system.time(r1 <- rankMatrix(L))) # warning+ ~1.5 sec (2013)
(st2 <- system.time(r2 <- rankMatrix(L, method = "qr"))) # considerably faster!
r1[[1]] == print(r2[[1]]) ## --> ( 33 TRUE )

## another sparse-"qr" one, which ``failed'' till 2013-11-23:
set.seed(42)
f1 <- factor(sample(50, 1000, replace=TRUE))
f2 <- factor(sample(50, 1000, replace=TRUE))
f3 <- factor(sample(50, 1000, replace=TRUE))
D <- t(do.call(rbind, lapply(list(f1,f2,f3), as, 'sparseMatrix'))))
dim(D); nnzero(D) ## 1000 x 150 // 3000 non-zeros (= 2%)
stopifnot(rankMatrix(D, method='qr') == 148,
           rankMatrix(crossprod(D),method='qr') == 148)

## zero matrix has rank 0 :
stopifnot(sapply(meths, function(.m.)
                 rankMatrix(matrix(0, 2, 2), method = .m.)) == 0)

```

Description

Estimate the reciprocal of the condition number of a matrix.

This is a generic function with several methods, as seen by [showMethods\(rcond\)](#).

Usage

```

kappa(z, ...)
rcond(x, norm, ...)

## S4 method for signature 'sparseMatrix,character'
rcond(x, norm, exact = FALSE, inverse = solve(x),
      warn = TRUE, ...)

```

Arguments

<code>x, z</code>	an R object that inherits from the <code>Matrix</code> class.
<code>norm</code>	character string indicating the type of norm to be used in the estimate. The default is "0" for the 1-norm ("0" is equivalent to "1"). For sparse matrices, when <code>useInv=TRUE</code> , <code>norm</code> can be any of the kinds allowed for norm ; otherwise, the other possible value is "I" for the infinity norm, see also norm .
<code>exact</code>	.
<code>inverse</code>	.
<code>warn</code>	.
<code>...</code>	further arguments passed to or from other methods.

Value

An estimate of the reciprocal condition number of `x`.

BACKGROUND

The condition number of a regular (square) matrix is the product of the [norm](#) of the matrix and the norm of its inverse (or pseudo-inverse).

More generally, the condition number is defined (also for non-square matrices A) as

$$\kappa(A) = \frac{\max_{\|v\|=1} \|Av\|}{\min_{\|v\|=1} \|Av\|}.$$

Whenever `x` is *not* a square matrix, in our method definitions, this is typically computed via `rcond(qr.R(qr(X)), ...)` where X is `x` or `t(x)`.

The condition number takes on values between 1 and infinity, inclusive, and can be viewed as a factor by which errors in solving linear systems with this matrix as coefficient matrix could be magnified.

`rcond()` computes the *reciprocal* condition number $1/\kappa$ with values in $[0, 1]$ and can be viewed as a scaled measure of how close a matrix is to being rank deficient (aka "singular").

Condition numbers are usually estimated, since exact computation is costly in terms of floating-point operations. An (over) estimate of reciprocal condition number is given, since by doing so overflow is avoided. Matrices are well-conditioned if the reciprocal condition number is near 1 and ill-conditioned if it is near zero.

References

Golub, G., and Van Loan, C. F. (1989). *Matrix Computations*, 2nd edition, Johns Hopkins, Baltimore.

See Also

[norm](#), [kappa\(\)](#) from package **base** computes an *approximate* condition number of a "traditional" matrix, even non-square ones, with respect to the $p = 2$ (Euclidean) [norm](#). [solve](#).

[condest](#), a newer *approximate* estimate of the (1-norm) condition number, particularly efficient for large sparse matrices.

Examples

```
x <- Matrix(rnorm(9), 3, 3)
rcond(x)
## typically "the same" (with more computational effort):
1 / (norm(x) * norm(solve(x)))
rcond(Hilbert(9)) # should be about 9.1e-13

## For non-square matrices:
rcond(x1 <- cbind(1,1:10))# 0.05278
rcond(x2 <- cbind(x1, 2:11))# practically 0, since x2 does not have full rank

## sparse
(S1 <- Matrix(rbind(0:1,0, diag(3:-2))))
rcond(S1)
m1 <- as(S1, "denseMatrix")
all.equal(rcond(S1), rcond(m1))

## wide and sparse
rcond(Matrix(cbind(0, diag(2:-1))))

## Large sparse example -----
m <- Matrix(c(3,0:2), 2,2)
M <- bdiag(kronecker(Diagonal(2), m), kronecker(m,m))
36*(iM <- solve(M)) # still sparse
MM <- kronecker(Diagonal(10), kronecker(Diagonal(5),kronecker(m,M)))
dim(M3 <- kronecker(bdiag(M,M),MM)) # 12'800 ^ 2
if(interactive()) ## takes about 2 seconds if you have >= 8 GB RAM
  system.time(r <- rcond(M3))
## whereas this is *fast* even though it computes solve(M3)
system.time(r. <- rcond(M3, exact=TRUE))
if(interactive()) ## the values are not the same
  c(r, r.) # 0.05555 0.013888
## for all norms available for sparseMatrix, except "2" :
cbind(rr <- sapply(c("0","I","F","M"),
  function(N) rcond(M3, norm=N, exact=TRUE)))
```

Description

S4 methods defined in package **Matrix** for S4 generic functions [rep](#), [rep.int](#), and [rep.len](#).

Usage

```
## S4 method for signature 'denseMatrix'
rep(x, ...)
## S4 method for signature 'sparseMatrix'
```

```

rep(x, ...)
## S4 method for signature 'sparseVector'
rep(x, times, length.out, each, ...)

## S4 method for signature 'denseMatrix'
rep.int(x, times)
## S4 method for signature 'sparseMatrix'
rep.int(x, times)
## S4 method for signature 'sparseVector'
rep.int(x, times)

## S4 method for signature 'denseMatrix'
rep_len(x, length.out)
## S4 method for signature 'sparseMatrix'
rep_len(x, length.out)
## S4 method for signature 'sparseVector'
rep_len(x, length.out)

```

Arguments

x a [Matrix](#) object or a [sparseVector](#) object.
times, length.out, each non-negative integers. times may be a vector of length length(x) * each storing non-negative integers.
... optional arguments passed from or to other methods.

Value

The methods with signature x="sparseVector" returns a new [sparseVector](#) object that “repeats” x according to the conventions of the internal default methods, hence see [rep](#).

The methods with signature x="denseMatrix" or x="sparseMatrix" coerce x to vector or [sparseVector](#), respectively, then dispatch.

Examples

```

x <- new("sparseVector", length = 4L, i = 2L:4L, x = 1L:3L)

rep(x) # no-op
rep(x, times = 2L)
rep(x, times = 0L:3L)
rep(x, length.out = 10L)

rep(x, each = 2L)
rep(x, times = 2L, each = 2L)
rep(x, times = c(0L:3L, 0L:3L), each = 2L)
rep(x, length.out = 10L, each = 2L)

stopifnot(identical(rep.int(x, 2L), rep(x, times = 2L)),
           identical(rep_len(x, 10L), rep(x, length.out = 10L)))

```

rep2abI	<i>Replicate Vectors into 'abIndex' Result</i>
---------	--

Description

rep2abI(x, times) conceptually computes `rep.int(x, times)` but with an `abIndex` class result.

Usage

```
rep2abI(x, times)
```

Arguments

x	numeric vector
times	integer (valued) scalar: the number of repetitions

Value

a vector of `class abIndex`

See Also

`rep.int()`, the base function; `abIseq`, `abIndex`.

Examples

```
(ab <- rep2abI(2:7, 4))
stopifnot(identical(as(ab, "numeric"),
  rep(2:7, 4)))
```

rleDiff-class	<i>Class "rleDiff" of rle(diff(.)) Stored Vectors</i>
---------------	---

Description

Class "rleDiff" is for compactly storing long vectors which mainly consist of *linear* stretches. For such a vector x, `diff(x)` consists of *constant* stretches and is hence well compressable via `rle()`.

Objects from the Class

Objects can be created by calls of the form `new("rleDiff", ...)`.

Currently experimental, see below.

Slots

first: A single number (of class "numLike", a class union of "numeric" and "logical").

rle: Object of class "rle", basically a [list](#) with components "lengths" and "values", see [rle\(\)](#). As this is used to encode potentially huge index vectors, lengths may be of type [double](#) here.

Methods

There is a simple [show](#) method only.

Note

This is currently an *experimental* auxiliary class for the class [abIndex](#), see there.

See Also

[rle](#), [abIndex](#).

Examples

```
showClass("rleDiff")

ab <- c(abIseq(2, 100), abIseq(20, -2))
ab@rleD # is "rleDiff"
```

rsparsematrix

Random Sparse Matrix

Description

Generate a random sparse matrix efficiently. The default has rounded gaussian non-zero entries, and `rand.x = NULL` generates random pattern matrices, i.e. inheriting from [nsparseMatrix](#).

Usage

```
rsparsematrix(nrow, ncol, density, nnz = round(density * maxE),
              symmetric = FALSE,
              rand.x = function(n) signif(rnorm(n), 2), ...)
```

Arguments

nrow, ncol	number of rows and columns, i.e., the matrix dimension (dim).
density	optional number in $[0, 1]$, the density is the proportion of non-zero entries among all matrix entries. If specified it determines the default for nnz, otherwise nnz needs to be specified.
nnz	number of non-zero entries, for a sparse matrix typically considerably smaller than <code>nrow*ncol</code> . Must be specified if density is not.

symmetric	logical indicating if result should be a matrix of class <code>symmetricMatrix</code> . Note that in the symmetric case, nnz denotes the number of non zero entries of the upper (or lower) part of the matrix, including the diagonal.
rand.x	<code>NULL</code> or the random number generator for the x slot, a <code>function</code> such that <code>rand.x(n)</code> generates a numeric vector of length n. Typical examples are <code>rand.x = rnorm</code> , or <code>rand.x = runif</code> ; the default is nice for didactical purposes.
...	optionally further arguments passed to <code>sparseMatrix()</code> , notably <code>repr</code> .

Details

The algorithm first samples “encoded” (i, j) s without replacement, via one dimensional indices, if not symmetric `sample.int(nrow*ncol, nnz)`, then—if `rand.x` is not `NULL`—gets `x <- rand.x(nnz)` and calls `sparseMatrix(i=i, j=j, x=x, ...)`. When `rand.x=NULL`, `sparseMatrix(i=i, j=j, ...)` will return a pattern matrix (i.e., inheriting from `nsparseMatrix`).

Value

a `sparseMatrix`, say `M` of dimension $(nrow, ncol)$, i.e., with `dim(M) == c(nrow, ncol)`, if `symmetric` is not true, with `nzM <- nnzero(M)` fulfilling `nzM <= nnz` and typically, `nzM == nnz`.

Author(s)

Martin Maechler

Examples

```
set.seed(17) # to be reproducible
M <- rsparsematrix(8, 12, nnz = 30) # small example, not very sparse
M
M1 <- rsparsematrix(1000, 20, nnz = 123, rand.x = runif)
summary(M1)

## a random *symmetric* Matrix
(S9 <- rsparsematrix(9, 9, nnz = 10, symmetric=TRUE)) # dsCMatrix
nnzero(S9) # ~ 20: as 'nnz' only counts one "triangle"

## a random pattern aka boolean Matrix (no 'x' slot):
(n7 <- rsparsematrix(5, 12, nnz = 10, rand.x = NULL))

## a [T]riplet representation sparseMatrix:
T2 <- rsparsematrix(40, 12, nnz = 99, repr = "T")
head(T2)
```

RsparseMatrix-class	<i>Virtual Class ‘RsparseMatrix’ of Compressed Sparse Row Format Matrices</i>
---------------------	---

Description

RsparseMatrix is a virtual subclass of [sparseMatrix](#) representing compressed sparse row (CSR) format matrices. To represent an m -by- n matrix A , this format stores the positions of $m \times n$ or fewer (often $O(\max\{m, n\})$) entries of A , ordering these by row then by column. Authors of subclasses of RsparseMatrix are free to declare how this information is used to specify A . See ‘Subclasses’ for subclasses defined in package **Matrix**.

Slots

Dim, Dimnames inherited from virtual superclass [Matrix](#).

p an integer vector of length $\text{Dim}[1]+1$ corresponding to a subset V of the set of entries of A . $p[i+1]$ is the number of elements of V belonging to one of the first i rows of A , implying

$$0 = p[1] \leq p[i] \leq p[i+1] \leq p[\text{length}(p)] = |V|$$

where $|V|$ denotes the size of V . The number of elements of V belonging to row i of A is given by $p[i+1]-p[i]$ or, equivalently, $\text{diff}(p)[i]$.

j an integer vector of length $p[\text{length}(p)]$ storing the 0-based column indices of the elements of V ordered by row then by column. Hence, to be valid, j must satisfy $0 \leq j < \text{Dim}[2]$ and be increasing within the contiguous segments that partition it by row of A . The segment corresponding to row i can be accessed as $j[(p[i]+1):p[i+1]]$ if $p[i] < p[i+1]$ and by $j[\text{seq.int}(from=p[i], length.out=p[i+1]-p[i])]$ more generally. Experimentally, $\text{length}(j)$ is allowed to exceed $p[\text{length}(p)]$; if it does, then the excess elements of j are unreferenced.

Subclasses

The nonvirtual subclasses of RsparseMatrix defined in package **Matrix** are listed below with links to virtual superclasses defining their data type and structure.

Nonvirtual subclass	Data type	Structure
ngRMatrix	nMatrix	generalMatrix
lgRMatrix	lMatrix	generalMatrix
igRMatrix	iMatrix	generalMatrix
dgRMatrix	dMatrix	generalMatrix
zgRMatrix	zMatrix	generalMatrix
nsRMatrix	nMatrix	symmetricMatrix
lsRMatrix	lMatrix	symmetricMatrix
isRMatrix	iMatrix	symmetricMatrix
dsRMatrix	dMatrix	symmetricMatrix
zsRMatrix	zMatrix	symmetricMatrix
dpRMatrix	dMatrix	posdefMatrix

zpRMatrix	zMatrix	posdefMatrix
ntRMatrix	nMatrix	triangularMatrix
ltRMatrix	lMatrix	triangularMatrix
itRMatrix	iMatrix	triangularMatrix
dtRMatrix	dMatrix	triangularMatrix
ztRMatrix	zMatrix	triangularMatrix

The subclasses of `iMatrix`, `zMatrix`, and `posdefMatrix` were not defined until **Matrix** version 1.8-0.

Classes `.gRMatrix` represent general (unstructured, possibly nonsquare) A . Objects have `p` and `j` slots specifying a subset V of the set of entries of A , as described in ‘Slots’. For `ngRMatrix`, whose data type is boolean, the entries in V have value 1, and the entries not in V have value 0. For other `.gRMatrix`, the entries in V have values taken from an additional `x` slot (a logical, integer, double, or complex vector), and the entries not in V have value 0. The `j` and `x` slots are entirely parallel, both using the ordering of V by row then by column.

Classes `.sRMatrix` represent Hermitian or symmetric A . Compared to `.gRMatrix`, objects have an additional `uplo` slot indicating a restriction on V . If `uplo="U"`, then V contains only upper triangular entries of A . If `uplo="L"`, then V contains only lower triangular entries of A . The entries in V have value 1 (`nsRMatrix`) or values taken from an `x` slot (other `.sRMatrix`). The entries opposite the main diagonal, namely $A[j, i]$ for all $A[i, j]$ in V , have value `op(A[i, j])`, where `op=Conj` for Hermitian A and `op=identity` for symmetric A . All other entries of A have value 0. Objects of class `zsRMatrix` have an additional `trans` slot that distinguishes Hermitian and symmetric A , which use `trans="C"` and `"T"`, respectively. If `trans="C"` and V contains diagonal entries of A , then the imaginary parts of the corresponding elements of x are not referenced. (The diagonal entries of Hermitian matrices are real by definition.)

Classes `dpRMatrix` and `zpRMatrix` extend `dsRMatrix` and `zsRMatrix` and represent positive semidefinite A . Positive semidefiniteness of A induces a constraint on the `x` slot. However, this constraint is not strictly enforced by the validity methods, which test only that the diagonal entries of A are nonnegative (a necessary, not sufficient condition). For `zpRMatrix`, only `trans="C"` is valid.

Classes `.tRMatrix` represent triangular A . Compared to `.gRMatrix`, objects have additional `uplo` and `diag` slots indicating restrictions on V . If `uplo="U"`, then A is upper triangular and V contains only upper triangular entries. If `uplo="L"`, then A is lower triangular and V contains only lower triangular entries. If `diag="U"` (instead of `"N"`), then A is unit triangular and V contains only *strictly* upper or lower triangular entries. With the exception of diagonal entries when `diag="U"`, which have value 1, entries not in V have value 0.

See Also

“Sister” classes `CsparseMatrix` and `TsparseMatrix` for compressed sparse column and triplet format matrices. Generic functions `rbind2`, `rowSums`, `[` (when called as `x[i, ]`), etc., whose methods for `RsparseMatrix` are particularly efficient.

Examples

```
showClass("RsparseMatrix")

(m0 <- new("dsRMatrix"))
m2 <- new("dsRMatrix", Dim = c(2L, 2L),
```

```

      x = c(3,1), j = c(1L,1L), p = 0:2)
m2
stopifnot(colSums(as(m2, "TsparseMatrix")) == 3:4)
str(m2)
(ds2 <- forceSymmetric(diag(2))) # dsy*
dR <- as(ds2, "RsparseMatrix")
dR # dsRMatrix

(m0 <- new("dtRMatrix"))
(m2 <- new("dtRMatrix", Dim = c(2L,2L),
      x = c(5, 1:2), p = c(0L,2:3), j= c(0:1,1L)))
str(m2)
(m3 <- as(Diagonal(2), "RsparseMatrix"))# --> dtRMatrix

```

Schur-methods

*Methods for Schur Factorization***Description**

Computes the Schur factorization of an $n \times n$ real matrix A , which has the general form

$$A = QTQ'$$

where Q is an orthogonal matrix and T is a block upper triangular matrix with 1×1 and 2×2 diagonal blocks specifying the real and complex conjugate eigenvalues of A . The column vectors of Q are the Schur vectors of A , and T is the Schur form of A .

Methods are built on LAPACK routine dgees.

Usage

```
Schur(x, vectors = TRUE, ...)
```

Arguments

<code>x</code>	a finite square matrix or Matrix to be factorized.
<code>vectors</code>	a logical. If TRUE (the default), then Schur vectors are computed in addition to the Schur form.
<code>...</code>	further arguments passed to or from methods.

Value

An object representing the factorization, inheriting from virtual class `SchurFactorization`. Currently, the specific class is always `Schur`.

References

The LAPACK source code, including documentation; see <https://netlib.org/lapack/double/dgees.f>.

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

Class `Schur` and its methods.

Class `dgeMatrix`.

Generic functions `expand1` and `expand2`, for constructing matrix factors from the result.

Generic functions `Cholesky`, `BunchKaufman`, `lu`, and `qr`, for computing other factorizations.

Examples

```
showMethods("Schur", inherited = FALSE)
set.seed(0)

Schur(Hilbert(9L)) # real eigenvalues

n <- 5L
(A <- Matrix(round(rnorm(n * n, sd = 100)), n, n))
(sch.A <- Schur(A)) # complex eigenvalues

## A ~ Q T Q' in floating point
str(e.sch.A <- expand2(sch.A), max.level = 2L)
stopifnot(all.equal(A, Reduce(`%*%`, e.sch.A)))

(e1 <- eigen(matrix(sch.A@x, n, n), only.values = TRUE)$values)
(e2 <- eigen(A, only.values = TRUE)$values)
(e3 <- sch.A@values)

stopifnot(exprs = {
  all.equal(e1, e2, tolerance = 1e-13)
  all.equal(e1, e3[order(Mod(e3), decreasing = TRUE)], tolerance = 1e-13)
  identical(Schur(A, vectors = FALSE),
    `slot<-`(sch.A, "vectors", TRUE, double(0L)))
})
```

show-methods

Show a Matrix

Description

TODO.

Usage

```
show(object)
```

Arguments

object .

Description

Computes the skew-symmetric or skew-Hermitian part of a square matrix x , defined as $(x - \text{op}(x))/2$, where op denotes matrix transpose (for skew-symmetric) or matrix conjugate transpose (for skew-Hermitian).

Usage

```
skewpart(x, ...)

## S4 method for signature 'denseMatrix'
skewpart(x, trans = "C", ...)
## S4 method for signature 'CsparseMatrix'
skewpart(x, trans = "C", ...)
## S4 method for signature 'RsparseMatrix'
skewpart(x, trans = "C", ...)
## S4 method for signature 'TsparseMatrix'
skewpart(x, trans = "C", ...)
## S4 method for signature 'diagonalMatrix'
skewpart(x, trans = "C", ...)
## S4 method for signature 'indMatrix'
skewpart(x, ...)
## S4 method for signature 'matrix'
skewpart(x, trans = "C", ...)
```

Arguments

<code>x</code>	an R object representing a square matrix. Package Matrix provides methods for Matrix objects and traditional matrices.
<code>trans</code>	a character string, "C" or "T", indicating either conjugate transpose or transpose, i.e., whether the result is skew-Hermitian or skew-symmetric. It matters only for x with complex data. The default value is "C" for consistency with methods for isSymmetric .
<code>...</code>	optional arguments passed from or to other methods.

Value

An R object representing a skew-symmetric or skew-Hermitian matrix, inheriting from virtual class [Matrix](#) if x does. The dimension names are the result of symmetrizing `dimnames(x)`, giving precedence to the column names unless those are NULL and the row names are non-NULL.

See Also

[symmpart](#), [isSymmetric](#), [forceSymmetric](#).

Examples

```
## see help("sympart") for examples
```

solve-methods

*Methods in Package **Matrix** for Function solve*

Description

Methods for generic function [solve](#) for solving linear systems of equations, i.e., for X in $AX = B$, where A is a square matrix and X and B are matrices with dimensions consistent with A .

Usage

```
solve(a, b, ...)

## S4 method for signature 'denseMatrix,missing'
solve(a, b, tol = .Machine$double.eps, ...)
## S4 method for signature 'denseMatrix,denseMatrix'
solve(a, b, tol = .Machine$double.eps, ...)

## S4 method for signature 'CsparseMatrix,missing'
solve(a, b, sparse = TRUE, ...)
## S4 method for signature 'CsparseMatrix,denseMatrix'
solve(a, b, sparse = FALSE, ...)
## S4 method for signature 'CsparseMatrix,CsparseMatrix'
solve(a, b, sparse = TRUE, ...)

## S4 method for signature 'denseLU,denseMatrix'
solve(a, b, ...)
## S4 method for signature 'denseBunchKaufman,denseMatrix'
solve(a, b, ...)
## S4 method for signature 'denseCholesky,denseMatrix'
solve(a, b, ...)
## S4 method for signature 'sparseQR,CsparseMatrix'
solve(a, b, sparse = TRUE, ...)
## S4 method for signature 'sparseLU,CsparseMatrix'
solve(a, b, sparse = TRUE, tol = .Machine$double.eps, ...)
## S4 method for signature 'sparseCholesky,CsparseMatrix'
solve(a, b, sparse = TRUE,
      system = c("A", "LDLt", "LD", "DLt", "L", "Lt", "D", "P", "Pt"), ...)
```

Arguments

a a [finite](#) square matrix or [Matrix](#) containing the coefficients of the linear system, or otherwise a [MatrixFactorization](#), in which case methods behave (by default) as if the factorized matrix were specified.

<code>b</code>	a vector, <code>sparseVector</code> , matrix, or <code>Matrix</code> satisfying <code>NROW(b) == nrow(a)</code> , giving the right-hand side(s) of the linear system. Vectors <code>b</code> are treated as <code>length(b)-1</code> matrices. If <code>b</code> is missing, then methods take <code>b</code> to be an identity matrix.
<code>tol</code>	a non-negative number. For <code>a</code> inheriting from <code>denseMatrix</code> , an error is signaled if the reciprocal one-norm condition number (see <code>rcond</code>) of <code>a</code> is less than <code>tol</code> , indicating that <code>a</code> is near-singular. For <code>a</code> of class <code>sparseLU</code> , an error is signaled if the ratio <code>min(d)/max(d)</code> is less than <code>tol</code> , where <code>d = abs(diag(a@U))</code> . (Interpret with care, as this ratio is a cheap heuristic and <i>not</i> in general equal to or even proportional to the reciprocal one-norm condition number.) Setting <code>tol = 0</code> disables the test.
<code>sparse</code>	a logical indicating if the result should be formally sparse, i.e., if the result should inherit from virtual class <code>sparseMatrix</code> . Only methods for sparse <code>a</code> and missing or matrix <code>b</code> have this argument. Methods for missing or sparse <code>b</code> use <code>sparse = TRUE</code> by default. Methods for dense <code>b</code> use <code>sparse = FALSE</code> by default.
<code>system</code>	a string specifying a linear system to be solved. Only methods for <code>a</code> inheriting from <code>CHMfactor</code> have this argument. See ‘Details’.
<code>...</code>	further arguments passed to or from methods.

Details

Methods for general and symmetric matrices `a` compute a triangular factorization (LU, Bunch-Kaufman, or Cholesky) and call the method for the corresponding factorization class. The factorization is sparse if `a` is. Methods for sparse, symmetric matrices `a` attempt a Cholesky factorization and perform an LU factorization only if that fails (typically because `a` is not positive definite).

Triangular, diagonal, and permutation matrices do not require factorization (they are already “factors”), hence methods for those are implemented directly. For triangular `a`, solutions are obtained by forward or backward substitution; for diagonal `a`, they are obtained by scaling the rows of `b`; and for permutations `a`, they are obtained by permuting the rows of `b`.

Methods for dense `a` are built on 14 LAPACK routines: class `d..Matrix`, where `..=(ge|tr|tp|sy|sp|po|pp)`, uses routines `d..tri` and `d..trs` for missing and non-missing `b`, respectively. A corollary is that these methods always give a dense result.

Methods for sparse `a` are built on CXSparse routines `cs_lsolve`, `cs_usolve`, and `cs_spsolve` and CHOLMOD routines `cholmod_solve` and `cholmod_spsolve`. By default, these methods give a vector result if `b` is a vector, a sparse matrix result if `b` is missing or a sparse matrix, and a dense matrix result if `b` is a dense matrix. One can override this behaviour by setting the `sparse` argument, where available, but that should be done with care. Note that a sparse result may be sparse only in the formal sense and not at all in the mathematical sense, depending on the nonzero patterns of `a` and `b`. Furthermore, whereas dense results are fully preallocated, sparse results must be “grown” in a loop over the columns of `b`.

Methods for `a` of class `sparseQR` are simple wrappers around `qr.coef`, giving the *least squares* solution in overdetermined cases.

Methods for `a` inheriting from `CHMfactor` can solve systems other than the default one $AX = B$. The correspondence between its `system` argument the system actually solved is outlined in the table below. See `CHMfactor-class` for a definition of notation.

system	<code>isLDL(a)=TRUE</code>	<code>isLDL(a)=FALSE</code>
"A"	$AX = B$	$AX = B$
"LDLt"	$L_1 D L_1' X = B$	$LL' X = B$
"LD"	$L_1 D X = B$	$LX = B$
"DLt"	$D L_1' X = B$	$L' X = B$
"L"	$L_1 X = B$	$LX = B$
"Lt"	$L_1' X = B$	$L' X = B$
"D"	$DX = B$	$X = B$
"P"	$X = P_1 B$	$X = P_1 B$
"Pt"	$X = P_1' B$	$X = P_1' B$

See Also

Virtual class [MatrixFactorization](#) and its subclasses.

Generic functions [Cholesky](#), [BunchKaufman](#), [Schur](#), [lu](#), and [qr](#) for *computing* factorizations.

Generic function [solve](#) from **base**.

Function [qr.coef](#) from **base** for computing least squares solutions of overdetermined linear systems.

Examples

```
## A close to symmetric example with "quite sparse" inverse:
n1 <- 7; n2 <- 3
dd <- data.frame(a = gl(n1,n2), b = gl(n2,1,n1*n2))# balanced 2-way
X <- sparse.model.matrix(~ -1+ a + b, dd)# no intercept --> even sparser
XXt <- tcrossprod(X)
diag(XXt) <- rep(c(0,0,1,0), length.out = nrow(XXt))

n <- nrow(ZZ <- kronecker(XXt, Diagonal(x=c(4,1))))
image(a <- 2*Diagonal(n) + ZZ %*% Diagonal(x=c(10, rep(1, n-1))))
isSymmetric(a) # FALSE
image(drop0(skewpart(a)))
image(ia0 <- solve(a, tol = 0)) # checker board, dense [but really, a is singular!]
try(solve(a, sparse=TRUE))##-> error [ TODO: assertError ]
ia. <- solve(a, sparse=TRUE, tol = 1e-19)##-> *no* error
if(R.version$arch == "x86_64")
  ## Fails on 32-bit [Fedora 19, R 3.0.2] from Matrix 1.1-0 on [FIXME ??] only
  stopifnot(all.equal(as.matrix(ia.), as.matrix(ia0)))
a <- a + Diagonal(n)
iad <- solve(a)
ias <- solve(a, sparse=FALSE)
stopifnot(all.equal(as(iad,"denseMatrix"), ias, tolerance=1e-14))
I. <- iad %*% a ; image(I.)
I0 <- drop0(zapsmall(I.)); image(I0)
.I <- a %*% iad
.I0 <- drop0(zapsmall(.I))
stopifnot( all.equal(as(I0, "diagonalMatrix"), Diagonal(n)),
           all.equal(as(.I0,"diagonalMatrix"), Diagonal(n)) )
```

Description

S4 methods defined in package **Matrix** for S4 generic function `sort`, generalizing `base::sort`.

Usage

```
## S4 method for signature 'denseMatrix'
sort(x, decreasing = FALSE, ...)
## S4 method for signature 'sparseMatrix'
sort(x, decreasing = FALSE, ...)
## S4 method for signature 'sparseVector'
sort(x, decreasing = FALSE, na.last = NA, ...)
```

Arguments

<code>x</code>	a Matrix object or a sparseVector object.
<code>decreasing</code>	a logical indicating if the result should be decreasing rather than increasing.
<code>na.last</code>	a logical indicating a treatment of NA in the data. If <code>TRUE</code> , then they are put last; if <code>FALSE</code> , then they are put first; if <code>NA</code> , then they are removed.
<code>...</code>	optional arguments passed from or to other methods and eventually to sort.int .

Value

The method with signature `x="sparseVector"` returns a fully sorted [sparseVector](#) object. The sort is stable in the sense that the order of structural and nonstructural zeros in `x` is preserved.

The methods with signatures `x="denseMatrix"` and `x="sparseMatrix"` coerce `x` to vector and [sparseVector](#), respectively, then dispatch.

See Also

[xtfrm](#), [order](#), [rank](#).

Examples

```
z <- new("zsparseVector", length = 8L, i = 2L:7L,
        x = c(0, 1, 1i, -1, -1i, 0))
z
(s01 <- sort(z))
(s10 <- sort(z, decreasing = TRUE))

stopifnot(identical(z[z == 0i], s01[s01 == 0i]),
          identical(z[z == 0i], s10[s10 == 0i]))
```

sparse.model.matrix *Construct Sparse Design / Model Matrices*

Description

Construct a sparse model or “design” matrix, from a formula and data frame (`sparse.model.matrix`) or a single factor (`fac2sparse`).

The `fac2[Ss]parse()` functions are utilities, also used internally in the principal user level function `sparse.model.matrix()`.

Usage

```
sparse.model.matrix(object, data = environment(object),
  contrasts.arg = NULL, xlev = NULL, transpose = FALSE,
  drop.unused.levels = FALSE, row.names = TRUE,
  sep = "", verbose = FALSE, ...)

fac2sparse(from, to = c("d", "l", "n"),
  drop.unused.levels = TRUE, repr = c("C", "R", "T"), giveCsparse)
fac2Sparse(from, to = c("d", "l", "n"),
  drop.unused.levels = TRUE, repr = c("C", "R", "T"), giveCsparse,
  factorPatt12, contrasts.arg = NULL)
```

Arguments

object	an object of an appropriate class. For the default method, a model formula or terms object.
data	a data frame created with <code>model.frame</code> . If another sort of object, <code>model.frame</code> is called first.
contrasts.arg	for <code>sparse.model.matrix()</code> : A list, whose entries are contrasts suitable for input to the <code>contrasts</code> replacement function and whose names are the names of columns of data containing <code>factors</code> . for <code>fac2Sparse()</code> : character string or NULL or (coercable to) " <code>sparseMatrix</code> ", specifying the contrasts to be applied to the factor levels.
xlev	to be used as argument of <code>model.frame</code> if data has no "terms" attribute.
transpose	logical indicating if the <i>transpose</i> should be returned; if the transposed is used anyway, setting <code>transpose = TRUE</code> is more efficient.
drop.unused.levels	should factors have unused levels dropped? The default for <code>sparse.model.matrix</code> has been changed to FALSE, 2010-07, for compatibility with R's standard (dense) <code>model.matrix()</code> .
row.names	logical indicating if row names should be used.
sep	<code>character</code> string passed to <code>paste()</code> when constructing column names from the variable name and its levels.

verbose	logical or integer indicating if (and how much) progress output should be printed.
...	further arguments passed to or from other methods.
from	(for fac2sparse()) a factor .
to	a character indicating the “kind” of sparse matrix to be returned. The default, “d” is for double .
giveCsparse	deprecated , replaced with repr; logical indicating if the result must be a CsparseMatrix .
repr	character string, one of “C”, “T”, or “R”, specifying the sparse <i>representation</i> to be used for the result, i.e., one from the super classes CsparseMatrix , TsparseMatrix , or RsparseMatrix .
factorPatt12	logical vector, say fp, of length two; when fp[1] is true, return “contrasted” t(X); when fp[2] is true, the original (“dummy”) t(X), i.e, the result of fac2sparse() .

Value

a sparse matrix, extending [CsparseMatrix](#) (for fac2sparse() if repr = “C” as per default; a [TsparseMatrix](#) or [RsparseMatrix](#), otherwise).

For fac2Sparse(), a [list](#) of length two, both components with the corresponding transposed model matrix, where the corresponding factorPatt12 is true.

fac2sparse(), the basic workhorse of sparse.model.matrix(), returns the *transpose* ([t](#)) of the model matrix.

Note

[model.Matrix](#)(sparse = TRUE) from package **MatrixModels** may be preferable nowadays to sparse.model.matrix, as model.Matrix returns an object of class [modelMatrix](#) with additional slots assign and contrasts relating to the model variables.

Author(s)

Doug Bates and Martin Maechler, with initial suggestions from Tim Hesterberg.

See Also

[model.matrix](#) in package **stats**, part of base R.

[model.Matrix](#) in package **MatrixModels**; see ‘Note’.

as(f, “sparseMatrix”) (see coerce(from = “factor”, . .) in the class doc [sparseMatrix](#)) produces the *transposed* sparse model matrix for a single factor f (and *no* contrasts).

Examples

```
dd <- data.frame(a = gl(3,4), b = gl(4,1,12))# balanced 2-way
options("contrasts") # the default: "contr.treatment"
sparse.model.matrix(~ a + b, dd)
sparse.model.matrix(~ -1+ a + b, dd)# no intercept --> even sparser
sparse.model.matrix(~ a + b, dd, contrasts.arg = list(a = "contr.sum"))
sparse.model.matrix(~ a + b, dd, contrasts.arg = list(b = "contr.SAS"))
```

```
## Sparse method is equivalent to the traditional one :
stopifnot(all(sparse.model.matrix(~ a + b, dd) ==
  Matrix(model.matrix(~ a + b, dd), sparse=TRUE)),
  all(sparse.model.matrix(~0 + a + b, dd) ==
  Matrix(model.matrix(~0 + a + b, dd), sparse=TRUE)))

(ff <- gl(3,4,, c("X","Y", "Z"))))
fac2sparse(ff) # 3 x 12 sparse Matrix of class "dgCMatrix"
##
## X 1 1 1 1 . . . . .
## Y . . . . 1 1 1 1 . . .
## Z . . . . . 1 1 1 1

## can also be computed via sparse.model.matrix():
f30 <- gl(3,0 )
f12 <- gl(3,0, 12)
stopifnot(
  all.equal(t( fac2sparse(ff) ),
    sparse.model.matrix(~ 0+ff),
    tolerance = 0, check.attributes=FALSE),
  is(M <- fac2sparse(f30, drop= TRUE),"CsparseMatrix"), dim(M) == c(0, 0),
  is(M <- fac2sparse(f30, drop=FALSE),"CsparseMatrix"), dim(M) == c(3, 0),
  is(M <- fac2sparse(f12, drop= TRUE),"CsparseMatrix"), dim(M) == c(0,12),
  is(M <- fac2sparse(f12, drop=FALSE),"CsparseMatrix"), dim(M) == c(3,12)
)
```

sparseCholesky-class *Sparse Cholesky Factorizations*

Description

CHMfactor is the virtual class of sparse Cholesky factorizations of $n \times n$ real, symmetric matrices A , having the general form

$$P_1 A P_1' = L_1 D L_1' \stackrel{D_{jj} \geq 0}{=} L L'$$

or (equivalently)

$$A = P_1' L_1 D L_1' P_1 \stackrel{D_{jj} \geq 0}{=} P_1' L L' P_1$$

where P_1 is a permutation matrix, L_1 is a unit lower triangular matrix, D is a diagonal matrix, and $L = L_1 \sqrt{D}$. The second equalities hold only for positive semidefinite A , for which the diagonal entries of D are non-negative and $\sqrt{D}$ is well-defined.

The implementation of class CHMfactor is based on CHOLMOD's C-level cholmod_factor_struct. Virtual subclasses CHMsimpl and CHMsuper separate the simplicial and supernodal variants. These have nonvirtual subclasses [dn]CHMsimpl and [dn]CHMsuper, where prefix 'd' and prefix 'n' are reserved for numeric and symbolic factorizations, respectively.

Usage

```
isLDL(x)
```

Arguments

x an object inheriting from virtual class CHMfactor, almost always the result of a call to generic function [Cholesky](#).

Value

`isLDL(x)` returns TRUE or FALSE: TRUE if `x` stores the lower triangular entries of $L_1 - I + D$, FALSE if `x` stores the lower triangular entries of L .

Slots

Of CHMfactor:

`Dim`, `Dimnames` inherited from virtual class [MatrixFactorization](#).

`colcount` an integer vector of length `Dim[1]` giving an *estimate* of the number of nonzero entries in each column of the lower triangular Cholesky factor. If symbolic analysis was performed prior to factorization, then the estimate is exact.

`perm` a 0-based integer vector of length `Dim[1]` specifying the permutation applied to the rows and columns of the factorized matrix. `perm` of length 0 is valid and equivalent to the identity permutation, implying no pivoting.

`type` an integer vector of length 6 specifying details of the factorization. The elements correspond to members `ordering`, `is_ll`, `is_super`, `is_monotonic`, `maxcsize`, and `maxesize` of the original `cholmod_factor_struct`. Simplicial and supernodal factorizations are distinguished by `is_super`. Simplicial factorizations do not use `maxcsize` or `maxesize`. Supernodal factorizations do not use `is_ll` or `is_monotonic`.

Of CHMsimpl (all unused by `nCHMsimpl`):

`nz` an integer vector of length `Dim[1]` giving the number of nonzero entries in each column of the lower triangular Cholesky factor. There is at least one nonzero entry in each column, because the diagonal elements of the factor are stored explicitly.

`p` an integer vector of length `Dim[1]+1`. Row indices of nonzero entries in column `j` of the lower triangular Cholesky factor are obtained as `i[p[j]+seq_len(nz[j]))+1`.

`i` an integer vector of length greater than or equal to `sum(nz)` containing the row indices of nonzero entries in the lower triangular Cholesky factor. These are grouped by column and sorted within columns, but the columns themselves need not be ordered monotonically. Columns may be overallocated, i.e., the number of elements of `i` reserved for column `j` may exceed `nz[j]`.

`prv`, `nxt` integer vectors of length `Dim[1]+2` indicating the order in which the columns of the lower triangular Cholesky factor are stored in `i` and `x`. Starting from `j <- Dim[1]+2`, the recursion `j <- nxt[j+1]+1` traverses the columns in forward order and terminates when `nxt[j+1] = -1`. Starting from `j <- Dim[1]+1`, the recursion `j <- prv[j+1]+1` traverses the columns in backward order and terminates when `prv[j+1] = -1`.

Of dCHMsimpl:

`x` a numeric vector parallel to `i` containing the corresponding nonzero entries of the lower triangular Cholesky factor L or (if and only if `type[2]` is 0) of the lower triangular matrix $L_1 - I + D$.

Of CHMSuper:

`super`, `pi`, `px` integer vectors of length `nsuper+1`, where `nsuper` is the number of supernodes. `super[j]+1` is the index of the leftmost column of supernode `j`. The row indices of supernode `j` are obtained as `s[pi[j]+seq_len(pi[j+1]-pi[j])]+1`. The numeric entries of supernode `j` are obtained as `x[px[j]+seq_len(px[j+1]-px[j])]+1` (if slot `x` is available).

`s` an integer vector of length greater than or equal to `Dim[1]` containing the row indices of the supernodes. `s` may contain duplicates, but not within a supernode, where the row indices must be increasing.

Of `dCHMSuper`:

`x` a numeric vector of length less than or equal to `prod(Dim)` containing the numeric entries of the supernodes.

Extends

Class [MatrixFactorization](#), directly.

Instantiation

Objects can be generated directly by calls of the form `new("dCHMSimpl", ...)`, etc., but `dCHMSimpl` and `dCHMSuper` are more typically obtained as the value of `Cholesky(x, ...)` for `x` inheriting from [sparseMatrix](#) (often [dsCMatrix](#)).

There is currently no API outside of calls to `new` for generating `nCHMSimpl` and `nCHMSuper`. These classes are vestigial and may be formally deprecated in a future version of **Matrix**.

Methods

`coerce` signature(`from` = "CHMSimpl", `to` = "dtCMatrix"): returns a [dtCMatrix](#) representing the lower triangular Cholesky factor L or the lower triangular matrix $L_1 - I + D$, the latter if and only if `from@type[2]` is 0.

`coerce` signature(`from` = "CHMSuper", `to` = "dgCMatrix"): returns a [dgCMatrix](#) representing the lower triangular Cholesky factor L . Note that, for supernodes spanning two or more columns, the supernodal algorithm by design stores non-structural zeros above the main diagonal, hence [dgCMatrix](#) is indeed more appropriate than [dtCMatrix](#) as a coercion target.

`determinant` signature(`x` = "CHMfactor", `logarithm` = "logical"): behaves according to an optional argument `sqrt`. If `sqrt` = FALSE, then this method computes the determinant of the factorized matrix A or its logarithm. If `sqrt` = TRUE, then this method computes the determinant of the factor $L = L_1 \text{sqrt}(D)$ or its logarithm, giving NaN for the modulus when D has negative diagonal elements. For backwards compatibility, the default value of `sqrt` is TRUE, but that can be expected change in a future version of **Matrix**, hence defensive code will always set `sqrt` (to TRUE, if the code must remain backwards compatible with **Matrix** < 1.6-0). Calls to this method not setting `sqrt` may warn about the pending change. The warnings can be disabled with `options(Matrix.warnSqrtDefault = 0)`.

`diag` signature(`x` = "CHMfactor"): returns a numeric vector of length n containing the diagonal elements of D , which (if they are all non-negative) are the squared diagonal elements of L .

`expand` signature(`x` = "CHMfactor"): see [expand-methods](#).

`expand1` signature(`x` = "CHMSimpl"): see [expand1-methods](#).

`expand1` signature(`x = "CHMSuper"`): see [expand1-methods](#).
`expand2` signature(`x = "CHMsimpl"`): see [expand2-methods](#).
`expand2` signature(`x = "CHMSuper"`): see [expand2-methods](#).
`image` signature(`x = "CHMfactor"`): see [image-methods](#).
`nnzero` signature(`x = "CHMfactor"`): see [nnzero-methods](#).
`solve` signature(`a = "CHMfactor"`, `b = .`): see [solve-methods](#).
`update` signature(`object = "CHMfactor"`): returns a copy of object with the same nonzero pattern but with numeric entries updated according to additional arguments `parent` and `mult`, where `parent` is (coercible to) a [dsCMatrix](#) or a [dgCMatrix](#) and `mult` is a numeric vector of positive length.
 The numeric entries are updated with those of the Cholesky factor of $F(\text{parent}) + \text{mult}[1] * I$, i.e., $F(\text{parent})$ plus $\text{mult}[1]$ times the identity matrix, where $F = \text{identity}$ for symmetric `parent` and $F = \text{tcrossprod}$ for other `parent`. The nonzero pattern of $F(\text{parent})$ must match that of S if `object = Cholesky(S, ...)`.
`updown` signature(`update = .`, `C = .`, `object = "CHMfactor"`): see [updown-methods](#).

References

The CHOLMOD source code; see <https://github.com/DrTimothyAldenDavis/SuiteSparse>, notably the header file 'CHOLMOD/Include/cholmod.h' defining `cholmod_factor_struct`.

Chen, Y., Davis, T. A., Hager, W. W., & Rajamanickam, S. (2008). Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software*, 35(3), Article 22, 1-14. doi:10.1145/1391989.1391995

Amestoy, P. R., Davis, T. A., & Duff, I. S. (2004). Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 17(4), 886-905. doi:10.1145/1024074.1024081

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

Class [dsCMatrix](#).

Generic functions [Cholesky](#), [updown](#), [expand1](#) and [expand2](#).

Examples

```

showClass("sparseCholesky")
set.seed(2)

m <- 1000L
n <- 200L
M <- rsparsematrix(m, n, 0.01)
A <- crossprod(M)

## With dimnames, to see that they are propagated :
dimnames(A) <- dn <- rep.int(list(paste0("x", seq_len(n))), 2L)

```

```

(ch.A <- Cholesky(A)) # pivoted, by default
str(e.ch.A <- expand2(ch.A, LDL = TRUE), max.level = 2L)
str(E.ch.A <- expand2(ch.A, LDL = FALSE), max.level = 2L)

ae1 <- function(a, b, ...) all.equal(as(a, "matrix"), as(b, "matrix"), ...)
ae2 <- function(a, b, ...) ae1(unname(a), unname(b), ...)

## A ~ P1' L1 D L1' P1 ~ P1' L L' P1 in floating point
stopifnot(exprs = {
  identical(names(e.ch.A), c("P1.", "L1", "D", "L1.", "P1"))
  identical(names(E.ch.A), c("P1.", "L", "L.", "P1"))
  identical(e.ch.A[["P1"]],
    new("pMatrix", Dim = c(n, n), Dimnames = c(list(NULL), dn[2L]),
      margin = 2L, perm = invertPerm(ch.A@perm, 0L, 1L)))
  identical(e.ch.A[["P1."]], t(e.ch.A[["P1"]]))
  identical(e.ch.A[["L1."]], t(e.ch.A[["L1"]]))
  identical(E.ch.A[["L." ]], t(E.ch.A[["L" ]]))
  identical(e.ch.A[["D"]], Diagonal(x = diag(ch.A)))
  all.equal(E.ch.A[["L"]], with(e.ch.A, L1 %*% sqrt(D)))
  ae1(A, with(e.ch.A, P1. %*% L1 %*% D %*% L1. %*% P1))
  ae1(A, with(E.ch.A, P1. %*% L %*% L. %*% P1))
  ae2(A[ch.A@perm + 1L, ch.A@perm + 1L], with(e.ch.A, L1 %*% D %*% L1.))
  ae2(A[ch.A@perm + 1L, ch.A@perm + 1L], with(E.ch.A, L %*% L. ))
})

## Factorization handled as factorized matrix
## (in some cases only optionally, depending on arguments)
b <- rnorm(n)
stopifnot(all.equal(det(A), det(ch.A, sqrt = FALSE)),
  all.equal(solve(A, b), solve(ch.A, b, system = "A")))

u1 <- update(ch.A, A, mult = sqrt(2))
u2 <- update(ch.A, t(M), mult = sqrt(2)) # updating with crossprod(M), not M
stopifnot(all.equal(u1, u2, tolerance = 1e-14))

```

sparseLU-class

Sparse LU Factorizations

Description

sparseLU is the class of sparse, row- and column-pivoted LU factorizations of $n \times n$ real matrices A , having the general form

$$P_1 A P_2 = LU$$

or (equivalently)

$$A = P_1' L U P_2'$$

where P_1 and P_2 are permutation matrices, L is a unit lower triangular matrix, and U is an upper triangular matrix.

Slots

Dim, Dimnames inherited from virtual class [MatrixFactorization](#).

L an object of class [dtCMatrix](#), the unit lower triangular L factor.

U an object of class [dtCMatrix](#), the upper triangular U factor.

p, q 0-based integer vectors of length Dim[1], specifying the permutations applied to the rows and columns of the factorized matrix. q of length 0 is valid and equivalent to the identity permutation, implying no column pivoting. Using R syntax, the matrix $P_1 A P_2$ is precisely $A[p+1, q+1]$ ($A[p+1,]$ when q has length 0).

Extends

Class [LU](#), directly. Class [MatrixFactorization](#), by class [LU](#), distance 2.

Instantiation

Objects can be generated directly by calls of the form `new("sparseLU", ...)`, but they are more typically obtained as the value of `lu(x)` for x inheriting from [sparseMatrix](#) (often [dgCMatrix](#)).

Methods

determinant signature(x = "sparseLU", logarithm = "logical"): computes the determinant of the factorized matrix A or its logarithm.

expand signature(x = "sparseLU"): see [expand-methods](#).

expand1 signature(x = "sparseLU"): see [expand1-methods](#).

expand2 signature(x = "sparseLU"): see [expand2-methods](#).

solve signature(a = "sparseLU", b = .): see [solve-methods](#).

References

Davis, T. A. (2006). *Direct methods for sparse linear systems*. Society for Industrial and Applied Mathematics. doi:[10.1137/1.9780898718881](#)

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:[10.56021/9781421407944](#)

See Also

Class [denseLU](#) for dense LU factorizations.

Class [dgCMatrix](#).

Generic functions [lu](#), [expand1](#) and [expand2](#).

Examples

```
showClass("sparseLU")
set.seed(2)

A <- as(readMM(system.file("external", "pores_1.mtx", package = "Matrix")),
        "CsparseMatrix")
(n <- A@Dim[1L])

## With dimnames, to see that they are propagated :
dimnames(A) <- dn <- list(paste0("r", seq_len(n)),
                          paste0("c", seq_len(n)))

(lu.A <- lu(A))
str(e.lu.A <- expand2(lu.A), max.level = 2L)

ae1 <- function(a, b, ...) all.equal(as(a, "matrix"), as(b, "matrix"), ...)
ae2 <- function(a, b, ...) ae1(unname(a), unname(b), ...)

## A ~ P1' L U P2' in floating point
stopifnot(exprs = {
  identical(names(e.lu.A), c("P1.", "L", "U", "P2."))
  identical(e.lu.A[["P1."]],
            new("pMatrix", Dim = c(n, n), Dimnames = c(dn[1L], list(NULL)),
              margin = 1L, perm = invertPerm(lu.A@p, 0L, 1L)))
  identical(e.lu.A[["P2."]],
            new("pMatrix", Dim = c(n, n), Dimnames = c(list(NULL), dn[2L]),
              margin = 2L, perm = invertPerm(lu.A@q, 0L, 1L)))
  identical(e.lu.A[["L"]], lu.A@L)
  identical(e.lu.A[["U"]], lu.A@U)
  ae1(A, with(e.lu.A, P1. %*% L %*% U %*% P2.))
  ae2(A[lu.A@p + 1L, lu.A@q + 1L], with(e.lu.A, L %*% U))
})

## Factorization handled as factorized matrix
b <- rnorm(n)
stopifnot(identical(det(A), det(lu.A)),
          identical(solve(A, b), solve(lu.A, b)))
```

sparseMatrix

General Sparse Matrix Construction from Nonzero Entries

Description

User-friendly construction of sparse matrices (inheriting from virtual [class CsparseMatrix](#), [RsparseMatrix](#), or [TsparseMatrix](#)) from the positions and values of their nonzero entries.

This interface is recommended over direct construction via calls such as `new("..[CRT]Matrix", ...)`.

Usage

```
sparseMatrix(i, j, p, x, dims, dimnames,
             symmetric = FALSE, triangular = FALSE, index1 = TRUE,
             repr = c("C", "R", "T"), giveCsparse,
             check = TRUE, use.last.ij = FALSE,
             keepInteger = Matrix.keepInteger())
```

Arguments

<code>i, j</code>	integer vectors of equal length specifying the positions (row and column indices) of the nonzero (or non-TRUE) entries of the matrix. Note that, when <code>x</code> is non-missing, the x_k corresponding to <i>repeated</i> pairs (i_k, j_k) are <i>added</i> , for consistency with the definition of class TsparseMatrix , unless <code>use.last.ij</code> is TRUE, in which case only the <i>last</i> such x_k is used.
<code>p</code>	integer vector of pointers, one for each column (or row), to the initial (zero-based) index of elements in the column (or row). Exactly one of <code>i</code> , <code>j</code> , and <code>p</code> must be missing.
<code>x</code>	optional, typically nonzero values for the matrix entries. If specified, then the length must equal that of <code>i</code> (or <code>j</code>) or equal 1, in which case <code>x</code> is recycled as necessary. If missing, then the result is a nonzero pattern matrix, i.e., inheriting from class nsparseMatrix .
<code>dims</code>	optional length-2 integer vector of matrix dimensions. If missing, then <code>!index1+c(max(i),max(j))</code> is used.
<code>dimnames</code>	optional list of dimnames ; if missing, then <code>NULL</code> ones are used.
<code>symmetric</code>	logical indicating if the resulting matrix should be symmetric. In that case, (i, j, p) should specify only one triangle (upper or lower).
<code>triangular</code>	logical indicating if the resulting matrix should be triangular. In that case, (i, j, p) should specify only one triangle (upper or lower).
<code>index1</code>	logical. If TRUE (the default), then <code>i</code> and <code>j</code> are interpreted as 1-based indices, following the R convention. That is, counting of rows and columns starts at 1. If FALSE, then they are interpreted as 0-based indices.
<code>repr</code>	character string, one of "C", "R", and "T", specifying the representation of the sparse matrix result, i.e., specifying one of the virtual classes CsparseMatrix , RsparseMatrix , and TsparseMatrix .
<code>giveCsparse</code>	(deprecated, replaced by repr) logical indicating if the result should inherit from CsparseMatrix or TsparseMatrix . Note that operations involving CsparseMatrix are very often (but not always) more efficient.
<code>check</code>	logical indicating whether to check that the result is formally valid before returning. Do not set to FALSE unless you know what you are doing!
<code>use.last.ij</code>	logical indicating if, in the case of repeated (duplicated) pairs (i_k, j_k) , only the last pair should be used. FALSE (the default) is consistent with the definition of class TsparseMatrix .
<code>keepInteger</code>	an optional logical argument introduced in Matrix version 1.8-0. If FALSE, then integer <code>x</code> are treated as double . This behaviour is compatible with earlier versions of Matrix , but it is not so desirable now that matrices with integer data

are representable as objects inheriting from virtual class `iMatrix`. The default value will change to `TRUE` in the near future; currently, it depends on the value of the environment variable `R_MATRIX_KEEP_INTEGER` at load time (false if unset), but that can be overridden by `options(Matrix.keepInteger=)`.

Details

Exactly one of the arguments `i`, `j` and `p` must be missing.

In typical usage, `p` is missing, `i` and `j` are vectors of positive integers and `x` is a numeric vector. These three vectors, which must have the same length, form the triplet representation of the sparse matrix.

If `i` or `j` is missing then `p` must be a non-decreasing integer vector whose first element is zero. It provides the compressed, or “pointer” representation of the row or column indices, whichever is missing. The expanded form of `p`, `rep(seq_along(dp), dp)` where `dp <- diff(p)`, is used as the (1-based) row or column indices.

You cannot set both `singular` and `triangular` to `true`; rather use `Diagonal()` (or its alternatives, see there).

The values of `i`, `j`, `p` and `index1` are used to create 1-based index vectors `i` and `j` from which a `TsparseMatrix` is constructed, with numerical values given by `x`, if non-missing. Note that in that case, when some pairs (i_k, j_k) are repeated (aka “duplicated”), the corresponding x_k are *added*, in consistency with the definition of the `TsparseMatrix` class, unless `use.last.ij` is set to `true`.

By default, when `repr = "C"`, the `CsparseMatrix` derived from this triplet form is returned, where `repr = "R"` now allows to directly get an `RsparseMatrix` and `repr = "T"` leaves the result as `TsparseMatrix`.

The reason for returning a `CsparseMatrix` object instead of the triplet format by default is that the compressed column form is easier to work with when performing matrix operations. In particular, if there are no zeros in `x` then a `CsparseMatrix` is a unique representation of the sparse matrix.

Value

A sparse matrix, by default in compressed sparse column format and (formally) without symmetric or triangular structure, i.e., by default inheriting from both `CsparseMatrix` and `generalMatrix`.

Note

You *do* need to use `index1 = FALSE` (or add `+ 1` to `i` and `j`) if you want use the 0-based `i` (and `j`) slots from existing sparse matrices.

See Also

`Matrix(*, sparse=TRUE)` for the constructor of such matrices from a *dense* matrix. That is easier in small sample, but much less efficient (or impossible) for large matrices, where something like `sparseMatrix()` is needed. Further `bdiag` and `Diagonal` for (block-)diagonal and `bandSparse` for banded sparse matrix constructors.

Random sparse matrices via `rsparsematrix()`.

The standard `R xtabs(*, sparse=TRUE)`, for sparse tables and `sparse.model.matrix()` for building sparse model matrices.

Consider `CsparseMatrix` and similar class definition help files.

Examples

```
## simple example
i <- c(1,3:8); j <- c(2,9,6:10); x <- 7 * (1:7)
(A <- sparseMatrix(i, j, x = x)) ## 8 x 10 "dgCMatrix"
summary(A)
str(A) # note that *internally* 0-based row indices are used

(sA <- sparseMatrix(i, j, x = x, symmetric = TRUE)) ## 10 x 10 "dsCMatrix"
(tA <- sparseMatrix(i, j, x = x, triangular = TRUE)) ## 10 x 10 "dtCMatrix"
stopifnot( all(sA == tA + t(tA)) ,
           identical(sA, as(tA + t(tA), "symmetricMatrix")))

## dims can be larger than the maximum row or column indices
(AA <- sparseMatrix(c(1,3:8), c(2,9,6:10), x = 7 * (1:7), dims = c(10,20)))
summary(AA)

## i, j and x can be in an arbitrary order, as long as they are consistent
set.seed(1); (perm <- sample(1:7))
(A1 <- sparseMatrix(i[perm], j[perm], x = x[perm]))
stopifnot(identical(A, A1))

## The slots are 0-index based, so
try( sparseMatrix(i=A@i, p=A@p, x= seq_along(A@x)) )
## fails and you should say so: 1-indexing is FALSE:
sparseMatrix(i=A@i, p=A@p, x= seq_along(A@x), index1 = FALSE)

## the (i,j) pairs can be repeated, in which case the x's are summed
(args <- data.frame(i = c(i, 1), j = c(j, 2), x = c(x, 2)))
(Aa <- do.call(sparseMatrix, args))
## explicitly ask for elimination of such duplicates, so
## that the last one is used:
(A. <- do.call(sparseMatrix, c(args, list(use.last.ij = TRUE))))
stopifnot(Aa[1,2] == 9, # 2+7 == 9
          A.[1,2] == 2) # 2 was *after* 7

## for a pattern matrix, of course there is no "summing":
(nA <- do.call(sparseMatrix, args[c("i","j")]))

dn <- list(LETTERS[1:3], letters[1:5])
## pointer vectors can be used, and the (i,x) slots are sorted if necessary:
m <- sparseMatrix(i = c(3,1, 3:2, 2:1), p = c(0:2, 4,4,6), x = 1:6, dimnames = dn)
m
str(m)
stopifnot(identical(dimnames(m), dn))

sparseMatrix(x = 2.72, i=1:3, j=2:4) # recycling x
sparseMatrix(x = TRUE, i=1:3, j=2:4) # recycling x, |--> "lgCMatrix"

## no 'x' --> patter*n* matrix:
(n <- sparseMatrix(i=1:6, j=rev(2:7)))# -> ngCMatrix
```

```
## an empty sparse matrix:
(e <- sparseMatrix(dims = c(4,6), i={}, j={}))

## a symmetric one:
(sy <- sparseMatrix(i= c(2,4,3:5), j= c(4,7:5,5), x = 1:5,
                    dims = c(7,7), symmetric=TRUE))
stopifnot(isSymmetric(sy),
          identical(sy, ## switch i <-> j {and transpose }
                    t( sparseMatrix(j= c(2,4,3:5), i= c(4,7:5,5), x = 1:5,
                                   dims = c(7,7), symmetric=TRUE))))

## rsparsematrix() calls sparseMatrix() :
M1 <- rsparsematrix(1000, 20, nnz = 200)
summary(M1)

## pointers example in converting from other sparse matrix representations.
if(requireNamespace("SparseM") &&
    packageVersion("SparseM") >= "0.87" &&
    nzchar(dfil <- system.file("extdata", "rua_32_ax.rua", package = "SparseM"))) {
  X <- SparseM::model.matrix(SparseM::read.matrix.hb(dfil))
  XX <- sparseMatrix(j = X@ja, p = X@ia - 1L, x = X@ra, dims = X@dimension)
  validObject(XX)

  ## Alternatively, and even more user friendly :
  X. <- as(X, "Matrix") # or also
  X2 <- as(X, "sparseMatrix")
  stopifnot(identical(XX, X.), identical(X., X2))
}
```

sparseMatrix-class	<i>Virtual Class ‘sparseMatrix’ of Sparse Format Matrices</i>
--------------------	---

Description

sparseMatrix is a virtual subclass of [Matrix](#) representing sparse format matrices. Formally, a sparse format is a storage format in which the space required to represent an $m \times n$ matrix with p nonzero entries is $O(\max\{m, n, p\})$.

Slots

Dim, Dimnames inherited from virtual superclass [Matrix](#).

Subclasses

Package **Matrix** defines two virtual subclasses of denseMatrix: [unpackedMatrix](#) and [packedMatrix](#). These correspond to the conventional rectangular and packed storage formats used by LAPACK: <https://netlib.org/lapack/lug/node121.html>. See those help topics for implementation details and (recursively) further subclasses.

See Also

“Complementary” class `denseMatrix` representing dense (meaning *not* sparse) format matrices. Function `sparseMatrix`, a widely used (but not the only) constructor for `sparseMatrix` objects. Other options for construction, not including explicit coercion: `Matrix`(sparse = TRUE), `xtabs`(sparse = TRUE), `sparse.model.matrix`(), ...

Examples

```
showClass("sparseMatrix") ## and look at the help() of its subclasses
M <- Matrix(0, 10000, 100)
M[1,1] <- M[2,3] <- 3.14
M ## show(.) method suppresses printing of the majority of rows

data(CAex, package = "Matrix")
dim(CAex) # 72 x 72 matrix
determinant(CAex) # works via sparse lu(.)

## factor -> t( <sparse design matrix> ) :
(fact <- gl(5, 3, 30, labels = LETTERS[1:5]))
(Xt <- as(fact, "sparseMatrix")) # indicator rows

## missing values --> all-0 columns:
f.mis <- fact
i.mis <- c(3:5, 17)
is.na(f.mis) <- i.mis
Xt != (X. <- as(f.mis, "sparseMatrix")) # differ only in columns 3:5,17
stopifnot(all(X.[,i.mis] == 0), all(Xt[, -i.mis] == X.[, -i.mis]))
```

sparseQR-class

Sparse QR Factorizations

Description

sparseQR is the class of sparse, row- and column-pivoted QR factorizations of $m \times n$ ($m \geq n$) real matrices, having the general form

$$P_1 A P_2 = Q R = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1$$

or (equivalently)

$$A = P_1' Q R P_2' = P_1' \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} P_2' = P_1' Q_1 R_1 P_2'$$

where P_1 and P_2 are permutation matrices, $Q = \prod_{j=1}^n H_j$ is an $m \times m$ orthogonal matrix (Q_1 contains the first n column vectors) equal to the product of n Householder matrices H_j , and R is an $m \times n$ upper trapezoidal matrix (R_1 contains the first n row vectors and is upper *triangular*).

Details

The method for `qr.Q` does not return Q but rather the (also orthogonal) product $P_1'Q$. This behaviour is algebraically consistent with the **base** implementation (see [qr](#)), which can be seen by noting that `qr.default` in **base** does not pivot rows, constraining P_1 to be an identity matrix. It follows that `qr.Q(qr.default(x))` also returns $P_1'Q$.

Similarly, the methods for `qr.qy` and `qr.qty` multiply on the left by $P_1'Q$ and $Q'P_1$ rather than Q and Q' .

It is wrong to expect the values of `qr.Q` (or `qr.R`, `qr.qy`, `qr.qty`) computed from “equivalent” sparse and dense factorizations (say, `qr(x)` and `qr(as(x, "matrix"))`) for `x` of class [dgCMatrix](#) to compare equal. The underlying factorization algorithms are quite different, notably as they employ different pivoting strategies, and in general the factorization is not unique even for fixed P_1 and P_2 .

On the other hand, the values of `qr.X`, `qr.coef`, `qr.fitted`, and `qr.resid` are well-defined, and in those cases the sparse and dense computations *should* compare equal (within some tolerance).

The method for `qr.R` is a simple wrapper around `qrR`, but not back-permuting by default and never giving row names. It did not support `backPermute = TRUE` until **Matrix** 1.6-0, hence code needing the back-permuted result should call `qrR` if **Matrix** $\geq$ 1.6-0 is not known.

Slots

`Dim`, `Dimnames` inherited from virtual class [MatrixFactorization](#).

`beta` a numeric vector of length `Dim[2]`, used to construct Householder matrices; see `V` below.

`V` an object of class [dgCMatrix](#) with `Dim[2]` columns. The number of rows `nrow(V)` is at least `Dim[1]` and at most `Dim[1]+Dim[2]`. `V` is lower trapezoidal, and its column vectors generate the Householder matrices H_j that compose the orthogonal Q factor. Specifically, H_j is constructed as `diag(Dim[1]) - beta[j] * tcrossprod(V[, j])`.

`R` an object of class [dgCMatrix](#) with `nrow(V)` rows and `Dim[2]` columns. `R` is the upper trapezoidal R factor.

`p`, `q` 0-based integer vectors of length `nrow(V)` and `Dim[2]`, respectively, specifying the permutations applied to the rows and columns of the factorized matrix. `q` of length 0 is valid and equivalent to the identity permutation, implying no column pivoting. Using R syntax, the matrix P_1AP_2 is precisely `A[p+1, q+1]` (`A[p+1, ]` when `q` has length 0).

Extends

Class [QR](#), directly. Class [MatrixFactorization](#), by class [QR](#), distance 2.

Instantiation

Objects can be generated directly by calls of the form `new("sparseQR", ...)`, but they are more typically obtained as the value of `qr(x)` for `x` inheriting from [sparseMatrix](#) (often [dgCMatrix](#)).

Methods

`determinant` `signature(x = "sparseQR", logarithm = "logical")`: computes the determinant of the factorized matrix A or its logarithm.

`expand1` `signature(x = "sparseQR")`: see [expand1-methods](#).

`expand2` signature(`x = "sparseQR"`): see [expand2-methods](#).

`qr.Q` signature(`qr = "sparseQR"`): returns as a [dgeMatrix](#) either $P_1'Q$ or $P_1'Q_1$, depending on optional argument `complete`. The default is `FALSE`, indicating $P_1'Q_1$.

`qr.R` signature(`qr = "sparseQR"`): `qrR` returns R , R_1 , RP_2' , or R_1P_2' , depending on optional arguments `complete` and `backPermute`. The default in both cases is `FALSE`, indicating R_1 , for compatibility with **base**. The class of the result in that case is [dtcMatrix](#). In the other three cases, it is [dgCMatrix](#).

`qr.X` signature(`qr = "sparseQR"`): returns A as a [dgeMatrix](#), by default. If $m > n$ and optional argument `ncol` is greater than n , then the result is augmented with $P_1'QJ$, where J is composed of columns $(n + 1)$ through `ncol` of the $m \times m$ identity matrix.

`qr.coef` signature(`qr = "sparseQR"`, `y = .`): returns as a [dgeMatrix](#) or vector the result of multiplying `y` on the left by $P_2R_1^{-1}Q_1'P_1$.

`qr.fitted` signature(`qr = "sparseQR"`, `y = .`): returns as a [dgeMatrix](#) or vector the result of multiplying `y` on the left by $P_1'Q_1Q_1'P_1$.

`qr.resid` signature(`qr = "sparseQR"`, `y = .`): returns as a [dgeMatrix](#) or vector the result of multiplying `y` on the left by $P_1'Q_2Q_2'P_1$.

`qr.qty` signature(`qr = "sparseQR"`, `y = .`): returns as a [dgeMatrix](#) or vector the result of multiplying `y` on the left by $Q'P_1$.

`qr.qy` signature(`qr = "sparseQR"`, `y = .`): returns as a [dgeMatrix](#) or vector the result of multiplying `y` on the left by $P_1'Q$.

`solve` signature(`a = "sparseQR"`, `b = .`): see [solve-methods](#).

References

Davis, T. A. (2006). *Direct methods for sparse linear systems*. Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898718881

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press. doi:10.56021/9781421407944

See Also

Class [dgCMatrix](#).

Generic function `qr` from **base**, whose default method `qr.default` “defines” the S3 class `qr` of dense QR factorizations.

[qr-methods](#) for methods defined in **Matrix**.

Generic functions [expand1](#) and [expand2](#).

The many auxiliary functions for QR factorizations: [qr.Q](#), [qr.R](#), [qr.X](#), [qr.coef](#), [qr.fitted](#), [qr.resid](#), [qr.qty](#), [qr.qy](#), and [qr.solve](#).

Examples

```
showClass("sparseQR")
set.seed(2)

m <- 300L
```

```

n <- 60L
A <- rsparsematrix(m, n, 0.05)

## With dimnames, to see that they are propagated :
dimnames(A) <- dn <- list(paste0("r", seq_len(m)),
                          paste0("c", seq_len(n)))

(qr.A <- qr(A))
str(e.qr.A <- expand2(qr.A, complete = FALSE), max.level = 2L)
str(E.qr.A <- expand2(qr.A, complete = TRUE), max.level = 2L)

t(sapply(e.qr.A, dim))
t(sapply(E.qr.A, dim))

## Horribly inefficient, but instructive :
slowQ <- function(V, beta) {
  d <- dim(V)
  Q <- diag(d[1L])
  if(d[2L] > 0L) {
    for(j in d[2L]:1L) {
      cat(j, "\n", sep = "")
      Q <- Q - (beta[j] * tcrossprod(V[, j])) %*% Q
    }
  }
  Q
}

ae1 <- function(a, b, ...) all.equal(as(a, "matrix"), as(b, "matrix"), ...)
ae2 <- function(a, b, ...) ae1(unname(a), unname(b), ...)

##  $A \sim P1' Q R P2' \sim P1' Q1 R1 P2'$  in floating point
stopifnot(exprs = {
  identical(names(e.qr.A), c("P1.", "Q1", "R1", "P2."))
  identical(names(E.qr.A), c("P1.", "Q", "R", "P2."))
  identical(e.qr.A[["P1."]],
            new("pMatrix", Dim = c(m, m), Dimnames = c(dn[1L], list(NULL)),
              margin = 1L, perm = invertPerm(qr.A@p, 0L, 1L)))
  identical(e.qr.A[["P2."]],
            new("pMatrix", Dim = c(n, n), Dimnames = c(list(NULL), dn[2L]),
              margin = 2L, perm = invertPerm(qr.A@q, 0L, 1L)))
  identical(e.qr.A[["R1"]], triu(E.qr.A[["R"]][seq_len(n), ]))
  identical(e.qr.A[["Q1"]], E.qr.A[["Q"]][, seq_len(n)])
  identical(E.qr.A[["R"]], qr.A@R)
  ## ae1(E.qr.A[["Q"]], slowQ(qr.A@V, qr.A@beta))
  ae1(crossprod(E.qr.A[["Q"]], diag(m))
    ae1(A, with(e.qr.A, P1. %*% Q1 %*% R1 %*% P2.))
    ae1(A, with(E.qr.A, P1. %*% Q %*% R %*% P2.))
    ae2(A.perm <- A[qr.A@p + 1L, qr.A@q + 1L], with(e.qr.A, Q1 %*% R1))
    ae2(A.perm, with(E.qr.A, Q %*% R ))
})

## More identities
b <- rnorm(m)

```

```

stopifnot(exprs = {
  ae1(qrX <- qr.X      (qr.A  ), A)
  ae2(qrQ <- qr.Q      (qr.A  ), with(e.qr.A, P1. %*% Q1))
  ae2(      qr.R      (qr.A  ), with(e.qr.A, R1))
  ae2(qrc <- qr.coef   (qr.A, b), with(e.qr.A, solve(R1 %*% P2., t(qrQ)) %*% b))
  ae2(qrf <- qr.fitted(qr.A, b), with(e.qr.A, tcrossprod(qrQ) %*% b))
  ae2(qrr <- qr.resid  (qr.A, b), b - qrf)
  ae2(qrq <- qr.qy     (qr.A, b), with(E.qr.A, P1. %*% Q %*% b))
  ae2(qr.qty(qr.A, qr.q), b)
})

## Sparse and dense computations should agree here
qr.Am <- qr(as(A, "matrix")) # <=> qr.default(A)
stopifnot(exprs = {
  ae2(qrX, qr.X      (qr.Am  ))
  ae2(qrc, qr.coef   (qr.Am, b))
  ae2(qrf, qr.fitted(qr.Am, b))
  ae2(qrr, qr.resid  (qr.Am, b))
})

```

sparseVector

Sparse Vector Construction from Nonzero Entries

Description

User friendly construction of sparse vectors, i.e., objects inheriting from [class sparseVector](#), from indices and values of its non-zero entries.

Usage

```
sparseVector(x, i, length)
```

Arguments

x	vector of the non zero entries; may be missing in which case a "nsparseVector" will be returned.
i	integer vector (of the same length as x) specifying the indices of the non-zero (or non-TRUE) entries of the sparse vector.
length	length of the sparse vector.

Details

zero entries in x are dropped automatically, analogously as [drop0\(\)](#) acts on sparse matrices.

Value

a sparse vector, i.e., inheriting from [class sparseVector](#).

Author(s)

Martin Maechler

See Also[sparseMatrix\(\)](#) constructor for sparse matrices; the class [sparseVector](#).**Examples**

```
str(sv <- sparseVector(x = 1:10, i = sample(999, 10), length=1000))

sx <- c(0,0,3, 3.2, 0,0,0,-3:1,0,0,2,0,0,5,0,0)
ss <- as(sx, "sparseVector")
stopifnot(identical(ss,
  sparseVector(x = c(2, -1, -2, 3, 1, -3, 5, 3.2),
    i = c(15L, 10:9, 3L,12L,8L,18L, 4L), length = 20L)))

(ns <- sparseVector(i= c(7, 3, 2), length = 10))
stopifnot(identical(ns,
  new("nsparseVector", length = 10, i = c(2, 3, 7))))
```

sparseVector-class	<i>Sparse Vector Classes</i>
--------------------	------------------------------

Description

Sparse Vector Classes: The virtual mother class "sparseVector" has the five actual daughter classes "dsparseVector", "isparseVector", "lsparseVector", "nsparseVector", and "zsparseVector", where we've mainly implemented methods for the d*, l* and n* ones.

Slots

length: class "numeric" - the [length](#) of the sparse vector. Note that "numeric" can be considerably larger than the maximal "integer", [.Machine\\$integer.max](#), on purpose.

i: class "numeric" - the (1-based) indices of the non-zero entries. Must *not* be NA and strictly sorted increasingly.

Note that "integer" is "part of" "numeric", and can (and often will) be used for non-huge sparseVectors.

x: (for all but "nsparseVector"): the non-zero entries. This is of class "numeric" for class "dsparseVector", "logical" for class "lsparseVector", etc.

Methods

length signature(x = "sparseVector"): simply extracts the length slot.

show signature(object = "sparseVector"): The [show](#) method for sparse vectors prints “*structural*” zeroes as “.” using the non-exported `prSpVector` function which allows further customization such as replacing “.” by “ ” (blank).

Note that [options](#)(max.print) will influence how many entries of large sparse vectors are printed at all.

as.vector signature(x = "sparseVector", mode = "character") coerces sparse vectors to “regular”, i.e., atomic vectors. This is the same as `as(x, "vector")`.

as ...: see coerce below

coerce signature(from = "sparseVector", to = "sparseMatrix"), and

coerce signature(from = "sparseMatrix", to = "sparseVector"), etc: coercions to and from sparse matrices ([sparseMatrix](#)) are provided and work analogously as in standard R, i.e., a vector is coerced to a 1-column matrix.

dim<- signature(x = "sparseVector", value = "integer") coerces a sparse vector to a sparse Matrix, i.e., an object inheriting from [sparseMatrix](#), of the appropriate dimension.

head signature(x = "sparseVector"): as with R’s (package [util](#)) [head](#), `head(x, n)` (for $n \geq 1$) is equivalent to `x[1:n]`, but here can be much more efficient, see the example.

tail signature(x = "sparseVector"): analogous to [head](#), see above.

toeplitz signature(x = "sparseVector"): as [toeplitz](#)(x), produce the $n \times n$ Toeplitz matrix from x, where $n = \text{length}(x)$.

rep signature(x = "sparseVector") repeat x, with the same argument list (x, times, length.out, each, ...) as the default method for `rep()`.

which signature(x = "nsparseVector") and

which signature(x = "lsparseVector") return the indices of the non-zero entries (which is trivial for sparse vectors).

Ops signature(e1 = "sparseVector", e2 = "*"): define arithmetic, compare and logic operations, (see [Ops](#)).

Summary signature(x = "sparseVector"): define all the [Summary](#) methods.

is.na, is.finite, is.infinite (x = "sparseVector"), and

is.na, is.finite, is.infinite (x = "nsparseVector"): return [logical](#) or “nsparseVector” of the same length as x, indicating if/where x is [NA](#) (or NaN), finite or infinite, entirely analogously to the corresponding base R functions.

zapsmall signature(x = "sparseVectors"): typically used for numeric sparse vector: [round](#)() entries such that (relatively) very small entries become zero exactly.

`c.sparseVector()` is an S3 method for all “sparseVector”s, but automatic dispatch only happens for the first argument, so it is useful also as regular R function, see the examples.

See Also

[sparseVector](#)() for friendly construction of sparse vectors (apart from `as(*, "sparseVector")`).

Examples

```

getClass("sparseVector")
getClass("dsparseVector")

sx <- c(0,0,3, 3.2, 0,0,0,-3:1,0,0,2,0,0,5,0,0)
(ss <- as(sx, "sparseVector"))

ix <- as.integer(round(sx))
(is <- as(ix, "sparseVector")) ## an "isparseVector" (!)
(ns <- sparseVector(i= c(7, 3, 2), length = 10)) # "nsparseVector"
## rep() works too:
(ri <- rep(is, length.out= 25))

## Using `dim<-` as in base R :
r <- ss
dim(r) <- c(4,5) # becomes a sparse Matrix:
r
## or coercion (as as.matrix() in base R):
as(ss, "Matrix")
stopifnot(all(ss == print(as(ss, "CsparseMatrix"))))

## currently has "non-structural" FALSE -- printing as ":"
(lis <- is & FALSE)
(nn <- is[is == 0]) # all "structural" FALSE

## NA-case
sN <- sx; sN[4] <- NA
(svN <- as(sN, "sparseVector"))

v <- as(c(0,0,3, 3.2, rep(0,9),-3,0,-1, rep(0,20),5,0),
        "sparseVector")
v <- rep(rep(v, 50), 5000)
set.seed(1); v[sample(v@i, 1e6)] <- 0
str(v)
system.time(for(i in 1:4) hv <- head(v, 1e6))
## user system elapsed
## 0.033 0.000 0.032
system.time(for(i in 1:4) h2 <- v[1:1e6])
## user system elapsed
## 1.317 0.000 1.319

stopifnot(identical(hv, h2),
          identical(is | FALSE, is != 0),
          validObject(svN), validObject(lis), as.logical(is.na(svN[4])),
          identical(is^2 > 0, is & TRUE),
          all(!lis), !any(lis), length(nn@i) == 0, !any(nn), all(!nn),
          sum(lis) == 0, !prod(lis), range(lis) == c(0,0))

## create and use the t(.) method:
t(x20 <- sparseVector(c(9,3:1), i=c(1:2,4,7), length=20))
(T20 <- toeplitz(x20))
stopifnot(is(T20, "symmetricMatrix"), is(T20, "sparseMatrix"),

```

```

        identical(unname(as.matrix(T20)),
                  toeplitz(as.vector(x20))))

## c() method for "sparseVector" - also available as regular function
(c1 <- c(x20, 0,0,0, -10*x20))
(c2 <- c(ns, is, FALSE))
(c3 <- c(ns, !ns, TRUE, NA, FALSE))
(c4 <- c(ns, rev(ns)))
## here, c() would produce a list {not dispatching to c.sparseVector()}
(c5 <- c.sparseVector(0,0, x20))

## checking (consistency)
.v <- as.vector
.s <- function(v) as(v, "sparseVector")
stopifnot(exprs = {
  all.equal(c1, .s(c(.v(x20), 0,0,0, -10*.v(x20))),      tol = 0)
  all.equal(c2, .s(c(.v(ns), .v(is), FALSE)),           tol = 0)
  all.equal(c3, .s(c(.v(ns), !.v(ns), TRUE, NA, FALSE)), tol = 0)
  all.equal(c4, .s(c(.v(ns), rev(.v(ns)))),             tol = 0,
              check.class = FALSE)
  all.equal(c5, .s(c(0,0, .v(x20))),                   tol = 0)
})

```

spMatrix

Sparse Matrix Constructor From Triplet

Description

User friendly construction of a sparse matrix (inheriting from class [TsparseMatrix](#)) from the triplet representation.

This is much less flexible than [sparseMatrix\(\)](#) and hence somewhat *deprecated*.

Usage

```
spMatrix(nrow, ncol, i = integer(0L), j = integer(0L), x = double(0L))
```

Arguments

nrow, ncol	integers specifying the desired number of rows and columns.
i, j	integer vectors of the same length specifying the locations of the non-zero (or non-TRUE) entries of the matrix.
x	atomic vector of the same length as i and j, specifying the values of the non-zero entries.

Value

A sparse matrix in triplet form, as an R object inheriting from both [TsparseMatrix](#) and [generalMatrix](#).

The matrix M will have $M[i[k], j[k]] == x[k]$, for $k = 1, 2, \dots, n$, where $n = \text{length}(i)$ and $M[i', j'] == 0$ for all other pairs (i', j') .

See Also

`Matrix(*, sparse=TRUE)` for the more usual constructor of such matrices. Then, `sparseMatrix` is more general and flexible than `spMatrix()` and by default returns a `CsparseMatrix` which is often slightly more desirable. Further, `bdiag` and `Diagonal` for (block-)diagonal matrix constructors.

Consider `TsparseMatrix` and similar class definition help files.

Examples

```
## simple example
A <- spMatrix(10,20, i = c(1,3:8),
               j = c(2,9,6:10),
               x = 7 * (1:7))

A # a "dgTMatrix"
summary(A)
str(A) # note that *internally* 0-based indices (i,j) are used

L <- spMatrix(9, 30, i = rep(1:9, 3), 1:27,
              (1:27) %% 4 != 1)
L # an "lgTMatrix"

## A simplified predecessor of Matrix' rsparsematrix() function :

rSpMatrix <- function(nrow, ncol, nnz,
                      rand.x = function(n) round(rnorm(nnz), 2))
{
  ## Purpose: random sparse matrix
  ## -----
  ## Arguments: (nrow,ncol): dimension
  ##           nnz : number of non-zero entries
  ##           rand.x: random number generator for 'x' slot
  ## -----
  ## Author: Martin Maechler, Date: 14.-16. May 2007
  stopifnot((nnz <- as.integer(nnz)) >= 0,
            nrow >= 0, ncol >= 0, nnz <= nrow * ncol)
  spMatrix(nrow, ncol,
           i = sample(nrow, nnz, replace = TRUE),
           j = sample(ncol, nnz, replace = TRUE),
           x = rand.x(nnz))
}

M1 <- rSpMatrix(100000, 20, nnz = 200)
summary(M1)
```

Description

Methods for "`[<-`", i.e., extraction or subsetting mostly of matrices, in package **Matrix**.

Note: Contrary to standard `matrix` assignment in base R, in `x[. .] <- val` it is typically an **error** (see `stop`) when the `type` or `class` of `val` would require the class of `x` to be changed, e.g., when `x` is logical, say `"lsparseMatrix"`, and `val` is numeric. In other cases, e.g., when `x` is a `"nsparseMatrix"` and `val` is not `TRUE` or `FALSE`, a warning is signalled, and `val` is "interpreted" as `logical`, and (logical) `NA` is interpreted as `TRUE`.

Methods

There are *many many* more than these:

`x = "Matrix", i = "missing", j = "missing", value = "ANY"` is currently a simple fallback method implementation which ensures "readable" error messages.

`x = "Matrix", i = "ANY", j = "ANY", value = "ANY"` currently gives an error

`x = "denseMatrix", i = "index", j = "missing", value = "numeric" ...`

`x = "denseMatrix", i = "index", j = "index", value = "numeric" ...`

`x = "denseMatrix", i = "missing", j = "index", value = "numeric" ...`

See Also

[\[-methods](#) for subsetting "Matrix" objects; the [index](#) class; [Extract](#) about the standard subset assignment (and extraction).

Examples

```
set.seed(101)
(a <- m <- Matrix(round(rnorm(7*4),2), nrow = 7))

a[] <- 2.2 # <- replaces every entry
a
## as do these:
a[,] <- 3 ; a[TRUE,] <- 4

m[2, 3] <- 3.14 # simple number
m[3, 3:4] <- 3:4 # simple numeric of length 2

## sub matrix assignment:
m[-(4:7), 3:4] <- cbind(1,2:4) #-> upper right corner of 'm'
m[3:5, 2:3] <- 0
m[6:7, 1:2] <- Diagonal(2)
m

## rows or columns only:
m[1,] <- 10
m[,2] <- 1:7
m[-(1:6), ] <- 3:0 # not the first 6 rows, i.e. only the 7th
as(m, "sparseMatrix")
```

Description

S4 methods defined in package **Matrix** for S4 generic function [\[](#).

Usage

```
## S4 method for signature 'Matrix,ANY,ANY,logical'
x[i, j, ..., drop = TRUE]
## S4 method for signature 'Matrix,ANY,ANY,missing'
x[i, j, ..., drop = TRUE]
## S4 method for signature 'Matrix,ANY,missing,logical'
x[i, j, ..., drop = TRUE]
## S4 method for signature 'Matrix,ANY,missing,missing'
x[i, j, ..., drop = TRUE]
## S4 method for signature 'Matrix,missing,ANY,logical'
x[i, j, ..., drop = TRUE]
## S4 method for signature 'Matrix,missing,ANY,missing'
x[i, j, ..., drop = TRUE]
## S4 method for signature 'Matrix,missing,missing,logical'
x[i, j, ..., drop = TRUE]
## S4 method for signature 'Matrix,missing,missing,missing'
x[i, j, ..., drop = TRUE]
## S4 method for signature 'sparseVector,ANY,missing,missing'
x[i, j, ..., drop = TRUE]
## S4 method for signature 'sparseVector,missing,missing,missing'
x[i, j, ..., drop = TRUE]
```

Arguments

<code>x</code>	a Matrix object or a sparseVector object.
<code>i, j, ...</code>	indices following the conventions of the internal default method, hence see [. Here, an index can also be a Matrix object (to be treated as the equivalent traditional matrix) or a sparseVector object (to be treated as the equivalent traditional vector). NA row and column subscripts (elements of <code>i</code> and <code>j</code> in calls <code>x[i, j]</code>) are not supported for sparse format <code>x</code> .
<code>drop</code>	a logical indicating if dimensions are dropped from subsets <code>x[i, j]</code> when <code>x</code> is a Matrix object and the subset would otherwise have exactly one row or exactly one column. The return value in this case is a traditional vector storing the row or column, exactly as if drop were called.

Details

Methods for `x` inheriting from virtual class [Matrix](#) return either a new [Matrix](#) object or a traditional vector, the latter for calls `x[i]` or when dimensions are dropped in calls `x[i, j]`.

Subsets `x[i, j, drop=FALSE]` preserve the data type and storage format of `x`, but not always its structure. For example, subsets of `x` of class `dsCMatrix` preserve inheritance from virtual classes `dMatrix` and `CsparseMatrix`, but not necessarily `symmetricMatrix`. Whether subsets preserve structure depends on the indices provided, namely `i` and `j` after conversion to integer vectors with elements in `seq_len(nrow(x))` and `seq_len(ncol(x))`, respectively:

- `x[i, j, drop=FALSE]` will inherit from `symmetricMatrix` if `x` does and `i` and `j` are identical.
- `x[i, j, drop=FALSE]` will inherit from `triangularMatrix` if `x` does; `i` and `j` have equal length `K`; and `i[ki] op j[kj]` (`op` is `>` if `x` is formally upper triangular, `<` if `x` is formally lower triangular) for all `ki, kj` in `seq_len(K)` such that `ki < kj` or for all `ki, kj` in `seq_len(K)` such that `ki > kj`. In this case, the subset is formally *unit* triangular if `x` is and `i` and `j` are identical.
- `x[i, j, drop=FALSE]` will inherit from `diagonalMatrix` if `x` does; `i` and `j` have equal length `K`; and `i[ki] != j[kj]` for all `ki, kj` in `seq_len(K)` such that `ki != kj`. In this case, the subset is formally *unit* diagonal if `x` is and `i` and `j` are identical.
- `x[i, j, drop=FALSE]` will inherit from `indexMatrix` if `x` does; `i` is a permutation of `seq_len(nrow(x))` if `x` is formally a column index matrix; and `j` is a permutation of `seq_len(ncol(x))` if `x` is formally a row index matrix. As a special case, `x[i, j, drop=FALSE]` will inherit from `pMatrix` if `x` does and `i` and `j` are both permutations of `seq_len(nrow(x))`.
- `x[i, j, drop=FALSE]` will inherit from `generalMatrix` otherwise.

Exceptions to the rule, that subsets `x[i, j, drop=FALSE]` preserve the storage format of `x`, occur when a subset is not representable in the storage format of `x`, typically because that format cannot represent general matrices. In such cases, the subset is represented in a suitable different format. A dense format is chosen for dense format `x`, and a sparse format is chosen for sparse format `x`. Hence, without exception, subsets `x[i, j, drop=FALSE]` preserve inheritance from `denseMatrix` and inheritance from `sparseMatrix`.

Methods for `x` inheriting from virtual class `sparseVector` return a new `sparseVector` object, preserving data type.

Value

A `Matrix` object, a `sparseVector` object, or a traditional vector; see ‘Details’.

See Also

`[<-` for subassignment to objects, `drop` for (conditionally) dropping dimensions.

Examples

```
selectSuperClasses("itCMatrix", directOnly = FALSE)
m <- new("itCMatrix",
        Dim = c(4L, 4L),
        Dimnames = list(A = paste0("a", 1L:4L), B = paste0("b", 1L:4L)),
        p = c(0L, 1L, 2L, 2L, 2L), i = c(2L, 3L), x = c(2L, 4L),
        uplo = "L", diag = "U")
m

selectSuperClasses("nsparseVector", directOnly = FALSE)
v <- new("nsparseVector", length = 8L, i = c(1L, 2L, 4L, 8L))
```

```

v

## 1-ary subsets
stopifnot(identical(m, m[]),
           identical(v, v[]))

m[ 0L:8L ]
m[ 0L:8L + 0.75] # same
m[-(0L:8L)]
m[-(0L:8L) - 0.75] # same
m[c(TRUE, FALSE)]
m[m != 0L]
m[cbind(      1L:4L,      1L:4L)]
m[cbind(rownames(m), colnames(m))] # same
v[ v@i]
v[ v ] # same
v[-v@i]
v[!v ] # same

## 2-ary subsets: error for "sparseVector"
stopifnot(identical(m, m[, ]),
           inherits(tryCatch(v[, ], error = identity), "error"))
m[1L,      ] # first row, names are colnames(x)
m[ , "b3"] # third col, names are rownames(x)
m[1L, "b3"] # no names ..... see help("drop")
m[4L, 2L:4L]
m[4L, 2L:4L, drop = FALSE]
m[c(2L, 4L), c(2L, 4L)] # lower unit triangular
m[c(4L, 2L), c(4L, 2L)] # upper unit triangular

## 3-ary subsets: error for "sparseVector" *and* for "Matrix"
stopifnot(inherits(tryCatch(m[, , ], error = identity), "error"),
           inherits(tryCatch(v[, , ], error = identity), "error"))

```

Summary-methods

Methods for Group Generic Function Summary

Description

S4 methods defined in package **Matrix** for S4 group generic function [Summary](#).

Usage

```

## S4 method for signature 'denseMatrix'
Summary(x, ..., na.rm = FALSE)
## S4 method for signature 'CsparseMatrix'
Summary(x, ..., na.rm = FALSE)
## S4 method for signature 'RsparseMatrix'
Summary(x, ..., na.rm = FALSE)
## S4 method for signature 'TsparseMatrix'
Summary(x, ..., na.rm = FALSE)

```

```
## S4 method for signature 'diagonalMatrix'
Summary(x, ..., na.rm = FALSE)
## S4 method for signature 'indMatrix'
Summary(x, ..., na.rm = FALSE)
## S4 method for signature 'sparseVector'
Summary(x, ..., na.rm = FALSE)
```

Arguments

<code>x</code>	a Matrix object or a sparseVector object.
<code>...</code>	optional arguments to be considered in addition to <code>x</code> . Supported are Matrix objects, sparseVector objects, traditional arrays and vectors, and NULL .
<code>na.rm</code>	a logical indicating if NA and NaN are removed from the data.

Value

The return value is identical to that of the internal default method if all [Matrix](#) and [sparseVector](#) arguments were

<code>min, max</code>	integer or double of length 1.
<code>range</code>	integer or double of length 2.
<code>range</code>	integer or double of length 2.

See Also

[Ops](#), [Math](#).

Examples

```
getGroupMembers(Summary)

## Dispatch is on first argument
v <- new("nsparseVector")
selectMethod("any", c(x = "nsparseVector"))
any(v, v, FALSE) # FALSE
any(v, v, TRUE) # TRUE
selectMethod("any", c(x = "logical"))
try(any(FALSE, v, v, TRUE)) # signals an error
any(TRUE, v, v, FALSE) # TRUE: loop over arguments short-circuits
```

symmetricMatrix-class *Virtual Class ‘symmetricMatrix’ of Hermitian or Symmetric Matrices*

Description

`symmetricMatrix` is a virtual subclass of [Matrix](#) representing Hermitian or symmetric matrices. Objects inheriting from `symmetricMatrix` have symmetric dimension names.

Slots

`Dim` inherited from virtual superclass [Matrix](#). `Dim[1]` and `Dim[2]` must be equal.

`Dimnames` inherited from virtual superclass [Matrix](#). The actual dimension names of objects `x` are obtained by `dimnames(x)`, not by `x@Dimnames`. The method for `dimnames` symmetrizes the `Dimnames` slot, which can be asymmetric, and returns the result; see [dimnames-methods](#). The validity method for class `symmetricMatrix` may be changed in a future version of **Matrix** to require a symmetric `Dimnames` slot, so that the actual dimension names of objects `x` can be obtained directly as `x@Dimnames`.

`factors` a list of [MatrixFactorization](#) objects caching factorizations of the represented matrix. It is updated “automagically” by methods for generic functions that compute matrix factorizations, such as `lu`. The caching mechanism is not idiomatic as it circumvents **R**’s usual copy semantics; it may be deprecated in a future version of **Matrix**, hence code intended to be forwards compatible will not access the `factors` slot.

`uplo` .

All direct, nonvirtual subclasses of `symmetricMatrix` defined in package **Matrix** have an additional slot:

`uplo` a character string, either “U” or “L”. “U” indicates that only the upper triangular entries of the represented matrix are stored. “L” indicates that only the lower triangular entries of the represented matrix are stored.

Subclasses

The direct, nonvirtual subclasses of `symmetricMatrix` defined in package **Matrix** are listed below with links to virtual superclasses defining their data type and storage format.

Nonvirtual subclass	Data type	Storage format
<code>nsyMatrix</code>	nMatrix	unpackedMatrix
<code>lsyMatrix</code>	lMatrix	unpackedMatrix
<code>isyMatrix</code>	iMatrix	unpackedMatrix
<code>dsyMatrix</code>	dMatrix	unpackedMatrix
<code>zsyMatrix</code>	zMatrix	unpackedMatrix
<code>nspMatrix</code>	nMatrix	packedMatrix
<code>lspMatrix</code>	lMatrix	packedMatrix
<code>ispMatrix</code>	iMatrix	packedMatrix
<code>dspMatrix</code>	dMatrix	packedMatrix
<code>zspMatrix</code>	zMatrix	packedMatrix
<code>ngCMatrix</code>	nMatrix	CsparseMatrix
<code>lsCMatrix</code>	lMatrix	CsparseMatrix
<code>isCMatrix</code>	iMatrix	CsparseMatrix
<code>dsCMatrix</code>	dMatrix	CsparseMatrix
<code>zsCMatrix</code>	zMatrix	CsparseMatrix
<code>nsRMatrix</code>	nMatrix	RsparseMatrix
<code>lsRMatrix</code>	lMatrix	RsparseMatrix
<code>isRMatrix</code>	iMatrix	RsparseMatrix
<code>dsRMatrix</code>	dMatrix	RsparseMatrix
<code>zsRMatrix</code>	zMatrix	RsparseMatrix

nsTMatrix	nMatrix	TsparseMatrix
lsTMatrix	lMatrix	TsparseMatrix
isTMatrix	iMatrix	TsparseMatrix
dsTMatrix	dMatrix	TsparseMatrix
zsTMatrix	zMatrix	TsparseMatrix

The subclasses of iMatrix and zMatrix were not defined until **Matrix** version 1.8-0.

See Also

Related classes [generalMatrix](#), [posdefMatrix](#), [triangularMatrix](#), [diagonalMatrix](#), and [indexMatrix](#) representing general (unstructured, possibly nonsquare), positive semidefinite, triangular, diagonal, and index matrices. Generic function [isSymmetric](#) for testing for testing if a matrix is (possibly *ap-proximately*) Hermitian or symmetric. Generic function [forceSymmetric](#) for inducing Hermitian or symmetric structure. Generic function [symmpart](#) for getting the Hermitian or symmetric part of a square matrix. Generic function [skewpart](#) for getting the skew-Hermitian or skew-symmetric part of a square matrix. Generic function [BunchKaufman](#) for Bunch-Kaufman factorization of Hermitian or symmetric matrices.

Examples

```
## An example about the symmetric Dimnames:
sy <- sparseMatrix(i= c(2,4,3:5), j= c(4,7:5,5), x = 1:5, dims = c(7,7),
                  symmetric=TRUE, dimnames = list(NULL, letters[1:7]))
sy # shows symmetrical dimnames
sy@Dimnames # internally only one part is stored
dimnames(sy) # both parts - as sy *is* symmetrical

showClass("symmetricMatrix")

## The names of direct subclasses:
scl <- getClass("symmetricMatrix")@subclasses
directly <- sapply(lapply(scl, slot, "by"), length) == 0
names(scl)[directly]

## Methods -- applicable to all subclasses above:
showMethods(classes = "symmetricMatrix")
```

symmpart-methods

Symmetric and Hermitian Parts of a Square Matrix

Description

Computes the symmetric or Hermitian part of a square matrix x , defined as $(x + \text{op}(x))/2$, where op denotes matrix transpose (for symmetric) or matrix conjugate transpose (for Hermitian).

Usage

```

symmpart(x, ...)

## S4 method for signature 'denseMatrix'
symmpart(x, uplo = "U", trans = "C", ...)
## S4 method for signature 'CsparseMatrix'
symmpart(x, uplo = "U", trans = "C", ...)
## S4 method for signature 'RsparseMatrix'
symmpart(x, uplo = "U", trans = "C", ...)
## S4 method for signature 'TsparseMatrix'
symmpart(x, uplo = "U", trans = "C", ...)
## S4 method for signature 'diagonalMatrix'
symmpart(x,                trans = "C", ...)
## S4 method for signature 'indMatrix'
symmpart(x,                ...)
## S4 method for signature 'matrix'
symmpart(x,                trans = "C", ...)

```

Arguments

<code>x</code>	an R object representing a square matrix. Package Matrix provides methods for Matrix objects and traditional matrices.
<code>uplo</code>	a character string, "U" or "L", indicating which of the upper and lower triangles of the symmetric (Hermitian) part is stored in the result.
<code>trans</code>	a character string, "C" or "T", indicating either conjugate transpose or transpose, i.e., whether the result is Hermitian or symmetric. It matters only for <code>x</code> with complex data. The default value is "C" for consistency with methods for isSymmetric .
<code>...</code>	optional arguments passed from or to other methods.

Value

An R object representing a symmetric or Hermitian matrix, inheriting from virtual class [Matrix](#) if `x` does. The dimension names are the result of symmetrizing `dimnames(x)`, giving precedence to the column names unless those are NULL and the row names are non-NULL.

See Also

[skewpart](#), [isSymmetric](#), [forceSymmetric](#).

Examples

```

X <- new("zgeMatrix",
        Dim = c(3L, 3L),
        Dimnames = list(A = paste0("a", 1:3), B = paste0("b", 1:3)),
        x = 0:8 * rep(c(1, -1i), length.out = 9L))
X
(Xc0 <- symmpart(X))           #      Hermitian

```

```

(Xc1 <- skewpart(X))          # skew-Hermitian
(Xt0 <- symmpart(X, trans = "T")) # symmetric
(Xt1 <- skewpart(X, trans = "T")) # skew-symmetric
stopifnot(isSymmetric(Xc0),
  isSymmetric(Xc1 * 1i),
  isSymmetric(Xt0, trans = "T"),
  isSymmetric(Xt1 * ifelse(upper.tri(Xt1), 1, -1), trans = "T"),
  all(Xc0 + Xc1 == X),
  all(Xt0 + Xt1 == X),
  all(Im(diag(Xc0)) == 0),
  all(Re(diag(Xc1)) == 0),
  all(diag(Xt1) == 0i),
  max(abs(Im(eigen(Xc0)$values))) < 16 * .Machine$double.eps,
  max(abs(Re(eigen(Xc1)$values))) < 16 * .Machine$double.eps,
  identical(lapply(list(Xc0, Xc1, Xt0, Xt1), dimnames),
    rep(list(dimnames(X)[c(2L, 2L)]), 4L)))

```

t-methods

Matrix Transpose

Description

S4 methods defined in package **Matrix** for S4 generic function `t`, generalizing base::`t`.

Usage

```

## S4 method for signature 'denseMatrix'
t(x)
## S4 method for signature 'CsparseMatrix'
t(x)
## S4 method for signature 'RsparseMatrix'
t(x)
## S4 method for signature 'TsparseMatrix'
t(x)
## S4 method for signature 'diagonalMatrix'
t(x)
## S4 method for signature 'indMatrix'
t(x)
## S4 method for signature 'pMatrix'
t(x)
## S4 method for signature 'sparseVector'
t(x)

.tCRT(x, lazy = TRUE)

```

Arguments

`x` a **Matrix** object or a **sparseVector** object. Vectors must have length in the range of type **integer**; they are handled as one-column matrices.

For `.tCRT`, `x` must inherit from one of `[CRT]sparseMatrix`.

`lazy` a logical indicating if transposition can change the storage format.

Details

For consistency, methods for `t` preserve the storage format of matrix arguments, even if that is not most efficient. In particular, if `x` is a `CsparseMatrix` object, then `t(x)` is a `CsparseMatrix` object, even though the `RsparseMatrix` representation of the transpose is obtained more cheaply. For “lazy” transposing of compressed sparse column or row format matrices, interchanging the column and row formats, use `.tCRT`.

If `x` is formally symmetric (see `symmetricMatrix`), then `x` and `t(x)` represent the same matrix with opposite uplo slots, such that one stores only the upper triangle and the other stores only the lower triangle.

Value

A `Matrix` object representing the transpose of `x`. The class is a superclass of the class of `x` if `x` is a `Matrix` object.

See Also

`ct` and `.ctCRT` for matrix conjugate transpose. `crossprod` for matrix products involving transposes.

Examples

```
(X <- new("igRMatrix",
  Dim = c(3L, 5L),
  Dimnames = list(A = paste0("a", 1:3), B = paste0("b", 1:5)),
  p = c(0L, 1L, 2L, 5L), j = c(1L, 0L, 0L, 1L, 4L), x = -2:2))
(tX <- t(X)) # compressed sparse row
(.tX <- .tCRT(X)) # compressed sparse column

stopifnot(identical(dim(tX), dim(X)[2:1]),
  identical(dimnames(tX), dimnames(X)[2:1]),
  identical(X, t(tX)),
  identical(X, .tCRT(.tX)),
  identical(tX, as(.tX, "RsparseMatrix")),
  identical(.tX, as(.tX, "CsparseMatrix")),
  identical(tX, .tCRT(X, lazy = FALSE)))

(S <- crossprod(X)) # formally symmetric
(tS <- t(S)) # *output* is identical but stored triangle is different
stopifnot(is(S, "dpRMatrix"),
  all.equal(S, tS, tolerance = 0),
  S@uplo == "U", tS@uplo == "L",
  identical(S, t(tS)))
```

Description

S4 methods defined in package **Matrix** for S4 generic function `toeplitz`, generalizing `stats::toeplitz`.

The S4 generic function definition is obtained from package **methods** through the `implicitGeneric` mechanism.

Usage

```
## S4 method for signature 'sparseVector'
toeplitz(x, r = NULL, symmetric = is.null(r),
         repr = c("C", "R", "T"), giveCsparse, ...)
```

Arguments

<code>x</code>	a <code>sparseVector</code> object.
<code>r</code>	a <code>sparseVector</code> object or <code>NULL</code> , not used if <code>symmetric</code> is <code>TRUE</code> .
<code>symmetric</code>	a logical indicating if the return value is formally symmetric.
<code>repr</code>	a character string indicating the format of the return value.
<code>giveCsparse</code>	deprecated.
<code>...</code>	optional arguments not yet used.

Value

A `sparseMatrix` object representing the Toeplitz matrix with first row `x` or `r` (`x` if `symmetric` is `TRUE` or `r` is `NULL`) and first column `x`. `x[1]` determines the main diagonal; `r[1]` is not used.

Examples

```
x <- new("isparseVector", length = 6L,
        i = c(2L, 4L, 6L), x = -1L:1L)
y <- new("dsparseVector", length = 8L,
        i = c(2L, 8L), x = c(1.1, 2.2))
(t1s <- toeplitz(x))
(t1g <- toeplitz(x, symmetric = FALSE))
(t2g <- toeplitz(x, y))
stopifnot(is(t1s, "isCMatrix"),
          is(t1g, "igCMatrix"),
          is(t2g, "dgCMatrix"),
          identical(t1g, as(t1s, "generalMatrix")),
          identical(tril(t2g[, 1L:6L]), as(tril(t1g), "dMatrix")))
```

triangularMatrix-class

Virtual Class 'triangularMatrix' of Triangular Matrices

Description

triangularMatrix is a virtual subclass of [Matrix](#) representing triangular matrices.

Slots

Dim, Dimnames inherited from virtual superclass [Matrix](#). Dim[1] and Dim[2] must be equal.

uplo, diag .

All direct, nonvirtual subclasses of triangularMatrix defined in package **Matrix** have two additional slots:

uplo a character string, either "U" or "L". "U" indicates that the represented matrix is upper triangular and that only the upper triangular entries are stored. "L" indicates that the represented matrix is lower triangular and that only the lower triangular entries are stored.

diag a character string, either "N" or "U". "N" indicates that the diagonal entries of the represented matrix are stored. "U" indicates that the represented matrix is unit triangular and that the diagonal entries, which are all equal to 1, are not stored.

Subclasses

The direct, nonvirtual subclasses of triangularMatrix defined in package **Matrix** are listed below with links to virtual superclasses defining their data type and storage format.

Nonvirtual subclass	Data type	Storage format
ntrMatrix	nMatrix	unpackedMatrix
ltrMatrix	lMatrix	unpackedMatrix
itrMatrix	iMatrix	unpackedMatrix
dtrMatrix	dMatrix	unpackedMatrix
ztrMatrix	zMatrix	unpackedMatrix
ntpMatrix	nMatrix	packedMatrix
ltpMatrix	lMatrix	packedMatrix
itpMatrix	iMatrix	packedMatrix
dtpMatrix	dMatrix	packedMatrix
ztpMatrix	zMatrix	packedMatrix
ntCMatrix	nMatrix	CsparseMatrix
ltCMatrix	lMatrix	CsparseMatrix
itCMatrix	iMatrix	CsparseMatrix
dtCMatrix	dMatrix	CsparseMatrix
ztCMatrix	zMatrix	CsparseMatrix
ntRMatrix	nMatrix	RsparseMatrix
ltrMatrix	lMatrix	RsparseMatrix
itrMatrix	iMatrix	RsparseMatrix

dtRMatrix	dMatrix	RsparseMatrix
ztRMatrix	zMatrix	RsparseMatrix
ntTMatrix	nMatrix	TsparseMatrix
ltTMatrix	lMatrix	TsparseMatrix
itTMatrix	iMatrix	TsparseMatrix
dtTMatrix	dMatrix	TsparseMatrix
ztTMatrix	zMatrix	TsparseMatrix

The subclasses of `iMatrix` and `zMatrix` were not defined until **Matrix** version 1.8-0.

See Also

Related classes `generalMatrix`, `symmetricMatrix`, `posdefMatrix`, `diagonalMatrix`, and `indexMatrix` representing general (unstructured, possibly nonsquare), Hermitian or symmetric, positive semidefinite, diagonal, and index matrices. Generic function `isTriangular` for testing if a matrix is upper or lower triangular. Generic function `forceTriangular` for inducing upper or lower triangular structure. Generic functions `band`, `triu`, and `tril` for inducing banded, upper trapezoidal, and lower trapezoidal structure.

Examples

```
showClass("triangularMatrix")

## The names of direct subclasses:
scl <- getClass("triangularMatrix")@subclasses
directly <- sapply(lapply(scl, slot, "by"), length) == 0
names(scl)[directly]

(m <- matrix(c(5, 1, 0, 3), 2))
as(m, "triangularMatrix")
```

TsparseMatrix-class	<i>Virtual Class ‘TsparseMatrix’ of Triplet Format Matrices</i>
---------------------	---

Description

`TsparseMatrix` is a virtual subclass of `sparseMatrix` representing triplet format matrices. To represent an m -by- n matrix A , this format stores the positions of $m \times n$ or fewer (often $O(\max\{m, n\})$) entries of A . Authors of subclasses of `TsparseMatrix` are free to declare how this information is used to specify A . See ‘Subclasses’ for subclasses defined in package **Matrix**.

The triplet format is convenient for reading and writing sparse matrices but often not amenable to direct computation. Accordingly, methods for `TsparseMatrix` defined in package **Matrix** tend to work by coercion to and from a more tractable *compressed* sparse format. Users can avoid repeated coercions by methods by coercing sparse matrix operands to either `CsparseMatrix` or `RsparseMatrix` *before* computing with them.

Slots

`Dim`, `Dimnames` inherited from virtual superclass [Matrix](#).

`i`, `j` integer vectors of equal length corresponding to a subset V of the set of entries of A . Respectively, `i` and `j` store the 0-based row and column indices of the elements of V . Hence, to be valid, `i` and `j` must satisfy $0 \leq i < \text{Dim}[1]$ and $0 \leq j < \text{Dim}[2]$. The common length of `i` and `j` can exceed the size of V , allowing for duplication of the form `i[k1] == i[k2]`, `j[k1] == j[k2]`, $k1 \neq k2$.

Subclasses

The nonvirtual subclasses of `TsparseMatrix` defined in package **Matrix** are listed below with links to virtual superclasses defining their data type and structure.

Nonvirtual subclass	Data type	Structure
<code>ngTMatrix</code>	nMatrix	generalMatrix
<code>lgTMatrix</code>	lMatrix	generalMatrix
<code>igTMatrix</code>	iMatrix	generalMatrix
<code>dgTMatrix</code>	dMatrix	generalMatrix
<code>zgTMatrix</code>	zMatrix	generalMatrix
<code>nsTMatrix</code>	nMatrix	symmetricMatrix
<code>lsTMatrix</code>	lMatrix	symmetricMatrix
<code>isTMatrix</code>	iMatrix	symmetricMatrix
<code>dsTMatrix</code>	dMatrix	symmetricMatrix
<code>zsTMatrix</code>	zMatrix	symmetricMatrix
<code>dpTMatrix</code>	dMatrix	posdefMatrix
<code>zpTMatrix</code>	zMatrix	posdefMatrix
<code>ntTMatrix</code>	nMatrix	triangularMatrix
<code>ltTMatrix</code>	lMatrix	triangularMatrix
<code>itTMatrix</code>	iMatrix	triangularMatrix
<code>dtTMatrix</code>	dMatrix	triangularMatrix
<code>ztTMatrix</code>	zMatrix	triangularMatrix

The subclasses of `iMatrix`, `zMatrix`, and `posdefMatrix` were not defined until **Matrix** version 1.8-0.

Classes `.gTMatrix` represent general (unstructured, possibly nonsquare) A . Objects have `i` and `j` slots specifying a subset V of the set of entries of A , as described in ‘Slots’. For `ngTMatrix`, whose data type is boolean, the entries in V have value 1, and the entries not in V have value 0. For other `.gTMatrix`, the entries in V have values taken from an additional `x` slot (a logical, integer, double, or complex vector), and the entries not in V have value 0. The `i`, `j`, and `x` slots have a common length, and the value of `A[i[k], j[k]]` is the result of aggregating `x[i == i[k] & j == j[k]]` by summation (reduction by `|` if `x` is logical, by `+` otherwise).

Classes `.sCMatrix` represent Hermitian or symmetric A . Compared to `.gTMatrix`, objects have an additional `uplo` slot indicating a restriction on V . If `uplo="U"`, then V contains only upper triangular entries of A . If `uplo="L"`, then V contains only lower triangular entries of A . The entries in V have value 1 (`nsCMatrix`) or values taken from an `x` slot (other `.sCMatrix`). The entries opposite the main diagonal, namely `A[j, i]` for all `A[i, j]` in V , have value `op(A[i, j])`, where `op=Conj` for Hermitian A and `op=identity` for symmetric A . All other entries of A have value 0. Objects of class

zsTMatrix have an additional trans slot that distinguishes Hermitian and symmetric A, which use trans="C" and "T", respectively. If trans="C" and V contains diagonal entries of A, then the imaginary parts of the corresponding elements of x are not referenced. (The diagonal entries of Hermitian matrices are real by definition.)

Classes dpTMatrix and zpTMatrix extend dsTMatrix and zsTMatrix and represent positive semidefinite A. Positive semidefiniteness of A induces a constraint on the x slot. However, this constraint is not strictly enforced by the validity methods, which test only that the diagonal entries of A are nonnegative (a necessary, not sufficient condition). For zpTMatrix, only trans="C" is valid.

Classes .tTMatrix represent triangular A. Compared to .gTMatrix, objects have additional uplo and diag slots indicating restrictions on V. If uplo="U", then A is upper triangular and V contains only upper triangular entries. If uplo="L", then A is lower triangular and V contains only lower triangular entries. If diag="U" (instead of "N"), then A is unit triangular and V contains only *strictly* upper or lower triangular entries. The entries in V have value 1 (ntCMatrix) or values taken from an x slot (other .tCMatrix). With the exception of diagonal entries when diag="U", which have value 1, entries not in V have value 0.

See Also

“Sister” classes [CsparseMatrix](#) and [RsparseMatrix](#) for compressed sparse column and compressed sparse row format matrices. Functions [aggregateT](#) and [asUniqueT](#) for “canonicalizing” objects inheriting from TsparseMatrix.

Examples

```
showClass("TsparseMatrix")
## or just the subclasses' names
names(getClass("TsparseMatrix")@subclasses)

T3 <- spMatrix(3,4, i=c(1,3:1), j=c(2,4:2), x=1:4)
T3 # only 3 non-zero entries, 5 = 1+4 !

m <- Matrix(0+1:28, nrow = 4)
m[-3,c(2,4:5,7)] <- m[ 3, 1:4] <- m[1:3, 6] <- 0
(mT <- as(m, "TsparseMatrix"))
str(mT)
mT[1,]
mT[4, drop = FALSE]
stopifnot(identical(mT[lower.tri(mT)],
                    m [lower.tri(m) ]))
mT[lower.tri(mT,diag=TRUE)] <- 0
mT

## Triplet representation with repeated (i,j) entries
## *adds* the corresponding x's:
T2 <- new("dgTMatrix",
          i = as.integer(c(1,1,0,3,3)),
          j = as.integer(c(2,2,4,0,0)), x=10*1:5, Dim=4:5)
str(T2) # contains (i,j,x) slots exactly as above, but
T2 ## has only three non-zero entries, as for repeated (i,j)'s,
## the corresponding x's are "implicitly" added
```

```

stopifnot(nnzero(T2) == 3)

mm <- Matrix(toeplitz(c(10, 0, 1, 0, 3)), sparse = TRUE)
mm # automatically dsCMatrix
str(mm)
mT <- as(as(mm, "generalMatrix"), "TsparseMatrix")

## Either
(symM <- as(mT, "symmetricMatrix")) # dsT
(symC <- as(symM, "CsparseMatrix")) # dsC
## or
sT <- Matrix(mT, sparse=TRUE, forceCheck=TRUE) # dsT

sym2 <- as(symC, "TsparseMatrix")
## --> the same as 'symM', a "dsTMatrix"

showClass("dtCMatrix")
showClass("dtTMatrix")
t1 <- new("dtTMatrix", x= c(3,7), i= 0:1, j=3:2, Dim= as.integer(c(4,4)))
t1
## from 0-diagonal to unit-diagonal {low-level step}:
tu <- t1 ; tu@diag <- "U"
tu
(cu <- as(tu, "CsparseMatrix"))
str(cu)# only two entries in @i and @x
stopifnot(cu@i == 1:0,
           all(2 * symmpart(cu) == Diagonal(4) + forceSymmetric(cu)))

t1[1,2:3] <- -1:-2
diag(t1) <- 10*c(1:2,3:2)
t1 # still triangular
(it1 <- solve(t1))
t1. <- solve(it1)
all(abs(t1 - t1.) < 10 * .Machine$double.eps)

## 2nd example
U5 <- new("dtCMatrix", i= c(1L, 0:3), p=c(0L,0L,0:2, 5L), Dim = c(5L, 5L),
         x = rep(1, 5), diag = "U")
U5
(iu <- solve(U5)) # contains one '0'
validObject(iu2 <- solve(U5, Diagonal(5)))# failed in earlier versions

I5 <- iu %*% U5 # should equal the identity matrix
i5 <- iu2 %*% U5
m53 <- matrix(1:15, 5,3, dimnames=list(NULL,letters[1:3]))
asDiag <- function(M) as(drop0(M), "diagonalMatrix")
stopifnot(
  all.equal(Diagonal(5), asDiag(I5), tolerance=1e-14) ,
  all.equal(Diagonal(5), asDiag(i5), tolerance=1e-14) ,
  identical(list(NULL, dimnames(m53)[[2]]), dimnames(solve(U5, m53)))
)

```

unpackedMatrix-class	<i>Virtual Class ‘unpackedMatrix’ of Conventional Dense Format Matrices</i>
----------------------	---

Description

unpackedMatrix is a virtual subclass of [denseMatrix](#) representing conventional dense format matrices, as defined by LAPACK: <https://netlib.org/lapack/lug/node121.html>. To represent an m-by-n matrix A, this format stores a vector of length m*n, of which m*n or fewer elements are actually used (fewer if there are formal constraints on the structure of A). Within package **Matrix**, this format is called “unpacked” to emphasize that it is complementary to the packed format also defined by LAPACK.

Slots

- Dim, Dimnames inherited from virtual superclass [Matrix](#).
- x a vector of length prod(Dim) storing entries of A in column-major order. Authors of subclasses of unpackedMatrix may declare that only a subset k of the elements of x is used to determine A, defining implicitly a function F such that A = F(x[k]); see ‘Subclasses’ for examples in package **Matrix**.

Subclasses

The nonvirtual subclasses of unpackedMatrix defined in package **Matrix** are listed below with links to virtual superclasses defining their data type and structure.

Nonvirtual subclass	Data type	Structure
ngeMatrix	nMatrix	generalMatrix
lgeMatrix	lMatrix	generalMatrix
igeMatrix	iMatrix	generalMatrix
dgeMatrix	dMatrix	generalMatrix
zgeMatrix	zMatrix	generalMatrix
nsyMatrix	nMatrix	symmetricMatrix
lsyMatrix	lMatrix	symmetricMatrix
isyMatrix	iMatrix	symmetricMatrix
dsyMatrix	dMatrix	symmetricMatrix
zsyMatrix	zMatrix	symmetricMatrix
dpoMatrix	dMatrix	posdefMatrix
zpoMatrix	zMatrix	posdefMatrix
ntrMatrix	nMatrix	triangularMatrix
ltrMatrix	lMatrix	triangularMatrix
itrMatrix	iMatrix	triangularMatrix
dtrMatrix	dMatrix	triangularMatrix
ztrMatrix	zMatrix	triangularMatrix

The subclasses of [iMatrix](#) and [zMatrix](#) were not defined until **Matrix** version 1.8-0. The other subclasses predate unpackedMatrix itself, which was not defined until **Matrix** version 1.5-0; previously, those subclasses were direct extensions of [denseMatrix](#).

The mapping from the x slot of length $m \times n$ to the m -by- n represented matrix A is given by:

	A[i, j]		
	i < j	i = j	i > j
.geMatrix	$x[(j-1)*m+i]$	$x[(j-1)*m+i]$	$x[(j-1)*m+i]$
.syMatrix, uplo="U"	$x[(j-1)*m+i]$	$x[(j-1)*m+i]$	$x[(i-1)*m+j]$
zsyMatrix, uplo="U", trans="C"	$x[(j-1)*m+i]$	$\text{Re}(x[(j-1)*m+i]) + 0i$	$\text{Conj}(x[(i-1)*m+j])$
.syMatrix, uplo="L"	$x[(i-1)*m+j]$	$x[(j-1)*m+i]$	$x[(j-1)*m+i]$
zsyMatrix, uplo="L", trans="C"	$\text{Conj}(x[(i-1)*m+j])$	$\text{Re}(x[(j-1)*m+i]) + 0i$	$x[(j-1)*m+i]$
.trMatrix, uplo="U", diag="N"	$x[(j-1)*m+i]$	$x[(j-1)*m+i]$	0
.trMatrix, uplo="U", diag="U"	$x[(j-1)*m+i]$	1	0
.trMatrix, uplo="L", diag="N"	0	$x[(j-1)*m+i]$	$x[(j-1)*m+j]$
.trMatrix, uplo="L", diag="U"	0	1	$x[(j-1)*m+j]$

Classes `.geMatrix` represent general (that is, unstructured) A . Objects have an x slot that stores all $m \times n$ entries of A . These classes closely approximate traditional matrices. For example, a traditional matrix z of type integer and its analogue Z of formal class `igeMatrix` are related by:

```
z <- array(Z@x, dim = Z@Dim, dimnames = Z@Dimnames)
Z <- new("igeMatrix", Dim = dim(z), Dimnames = dimnames(z), x = as.vector(z))
```

Classes `.syMatrix` represent Hermitian or symmetric A . Objects have x and `uplo` slots. If `uplo="U"`, then $x[\text{row}(A) \leq \text{col}(A)]$ stores the $n*(n+1)/2$ upper triangular entries of A and $x[\text{row}(A) > \text{col}(A)]$ is unreferenced. If `uplo="L"`, then $x[\text{row}(A) \geq \text{col}(A)]$ stores the $n*(n+1)/2$ lower triangular entries of A and $x[\text{row}(A) < \text{col}(A)]$ is unreferenced. Objects of class `zsyMatrix` have an additional `trans` slot that distinguishes Hermitian and symmetric A , which use `trans="C"` and `"T"`, respectively. If `trans="C"`, then the imaginary parts of $x[\text{row}(A) == \text{col}(A)]$ are not referenced. (The diagonal entries of Hermitian matrices are real by definition.)

Classes `dpoMatrix` and `zpoMatrix` extend `dsyMatrix` and `zsyMatrix` and represent positive semidefinite A . Positive semidefiniteness of A induces a constraint on the x slot. However, this constraint is not strictly enforced by the validity methods, which test only that the diagonal entries of A are nonnegative (a necessary, not sufficient condition). For `zpoMatrix`, only `trans="C"` is valid. Class `corMatrix` extends `dpoMatrix` and represents correlation matrices; it has an additional `sd` slot, a numeric vector of length n listing standard deviations, enabling recovery of a covariance matrix.

Classes `.trMatrix` represent triangular A . Objects have x , `uplo`, and `diag` slots. If `uplo="U"`, then A is upper triangular, $x[\text{row}(A) < \text{col}(A)]$ stores the $n*(n-1)/2$ strictly upper triangular entries, and $x[\text{row}(A) > \text{col}(A)]$ is unreferenced. If `uplo="L"`, then A is lower triangular, $x[\text{row}(A) > \text{col}(A)]$ stores the $n*(n-1)/2$ strictly lower triangular entries, and $x[\text{row}(A) < \text{col}(A)]$ is unreferenced. If `diag="N"`, then $x[\text{row}(A) == \text{col}(A)]$ stores the n diagonal entries of A . If `diag="U"`, then A is unit triangular and those elements of x are unreferenced.

See Also

“Complementary” class `packedMatrix` representing packed dense format matrices. Generic function `pack`, an alternative to explicit coercion to `packedMatrix`.

Examples

```

showClass("unpackedMatrix")
showMethods(classes = "unpackedMatrix")

showClass("ngeMatrix")
## "lgeMatrix" is really more relevant

showClass("lgeMatrix")
str(new("lgeMatrix"))
set.seed(1)
(lM <- Matrix(matrix(rnorm(28), 4,7) > 0))# a simple random lgeMatrix
set.seed(11)
(lC <- Matrix(matrix(rnorm(28), 4,7) > 0))# a simple random lgCMatrix
as(lM, "CsparseMatrix")

(s0 <- new("nsyMatrix"))

(M2 <- Matrix(c(TRUE, NA, FALSE, FALSE), 2, 2)) # logical dense (ltr)
(sM <- M2 & t(M2)) # -> "lge"
class(sM <- as(sM, "nMatrix")) # -> "nge"
(sM <- as(sM, "symmetricMatrix")) # -> "nsy"
str(sM <- as(sM, "packedMatrix")) # -> "nsp", i.e., packed symmetric

(M2 <- Matrix(c(TRUE, NA, FALSE, FALSE), 2, 2)) # logical dense (ltr)
str(M2)
# can
(sM <- M2 | t(M2)) # "lge"
as(sM, "symmetricMatrix")
str(sM <- as(sM, "packedMatrix")) # packed symmetric

## Only upper triangular part matters (when uplo == "U" as per default)
(sy2 <- new("dsyMatrix", Dim = as.integer(c(2,2)), x = c(14, NA,32,77)))
str(t(sy2)) # uplo = "L", and the lower tri. (i.e. NA is replaced).

chol(sy2) #-> "Cholesky" matrix
(sp2 <- pack(sy2)) # a "dspMatrix"

## Coercing to dpoMatrix gives invalid object:
sy3 <- new("dsyMatrix", Dim = as.integer(c(2,2)), x = c(14, -1, 2, -7))
try(as(sy3, "dpoMatrix")) # -> error: not positive definite

## 4x4 example
m <- matrix(0,4,4); m[upper.tri(m)] <- 1:6
(sym <- m+t(m)+diag(11:14, 4))
(S1 <- pack(sym))
(S2 <- t(S1))
stopifnot(all(S1 == S2)) # equal "seen as matrix", but differ internally :
str(S1)
S2@x

showClass("ntrMatrix")

```

```

str(new("ntpMatrix"))
(nutr <- as(upper.tri(matrix(, 4, 4)), "ndenseMatrix"))
str(nutp <- pack(nutr)) # packed matrix: only 10 = 4*(4+1)/2 entries
!nutp # the logical negation (is *not* logical triangular !)
## but this one is:
stopifnot(all.equal(nutp, pack(!nutp)))

showClass("ltrMatrix")

str(new("ltpMatrix"))
(lutr <- as(upper.tri(matrix(, 4, 4)), "ldenseMatrix"))
str(lutp <- pack(lutr)) # packed matrix: only 10 = 4*(4+1)/2 entries
!lutp # the logical negation (is *not* logical triangular !)
## but this one is:
stopifnot(all.equal(lutp, pack(!lutp)))

(m <- rbind(2:3, 0:-1))
(M <- as(m, "generalMatrix"))

(T <- as(M, "triangularMatrix")) # formally upper triangular
(T2 <- as(t(M), "triangularMatrix"))
stopifnot(T@uplo == "U", T2@uplo == "L", identical(T2, t(T)))

m <- matrix(0,4,4); m[upper.tri(m)] <- 1:6
(t1 <- Matrix(m+diag(,4)))
str(t1p <- pack(t1))
(t1pu <- diagN2U(t1p))
stopifnot(exprs = {
  inherits(t1, "dtrMatrix"); validObject(t1)
  inherits(t1p, "dtpMatrix"); validObject(t1p)
  inherits(t1pu, "dtCMatrix"); validObject(t1pu)
  t1pu@x == 1:6
  all(t1pu == t1p)
  identical((t1pu - t1)@x, numeric())# sparse all-0
})

```

Description

S4 methods defined in package **Matrix** for S4 generic function `update`, generalizing `stats::update`.

The method for virtual class `sparseCholesky` adjusts the numeric entries of a sparse Cholesky factorization to a new sparse matrix. It changes factorizations of A

$$P_1 A P_1' = L_1 D L_1' \stackrel{D_{jj} \geq 0}{=} L L'$$

to factorizations of $\tilde{A}$

$$P_1 \tilde{A} P_1' = \tilde{L}_1 \tilde{D} \tilde{L}_1' \stackrel{\tilde{D}_{jj} \geq 0}{=} \tilde{L} \tilde{L}'$$

where the new matrix $\tilde{A}$ is constrained to have nonzero pattern identical to that of the old matrix A .

Usage

```
## S4 method for signature 'sparseCholesky'
update(object, parent, mult = 0, ...)
```

Arguments

<code>object</code>	a sparseCholesky object.
<code>parent</code>	a sparseMatrix object, coerced to CsparseMatrix as necessary.
<code>mult</code>	a numeric or complex vector of positive length.
<code>...</code>	optional arguments not yet used.

Value

A new [sparseCholesky](#) object representing the Cholesky factorization of `op(parent)` plus `mult[1]` times the identity matrix, where `op` is [identity](#) for formally Hermitian parent and [tcrossprod](#)(`trans = "C"`) for all other parent.

See Also

Virtual class [sparseCholesky](#).

Generic functions [Cholesky](#) and [updown](#).

Examples

```
set.seed(44L)
X <- rsparsematrix(1000L, 200L, 0.01)
A <- crossprod(X)

u0 <- u0. <- Cholesky(A, LDL = FALSE)
u1 <- update(u0, parent = 4 * A)
u2 <- update(u0, parent = A, mult = 2)
u3 <- update(u0, parent = t(X), mult = 2)

u0.@x <- 2 * u0.@x

stopifnot(all.equal(u1, u0., tolerance = 1e-14),
          all.equal(u3, u2, tolerance = 1e-14))
```

Description

Computes a rank- k update or downdate of a Cholesky factorization

$$P_1 A P_1' = L_1 D L_1' = L L'$$

which for some k -column matrix C is the factorization

$$P_1 (A + s C C') P_1' = \tilde{L}_1 \tilde{D} \tilde{L}_1' = \tilde{L} \tilde{L}'$$

Here, $s = 1$ for an update and $s = -1$ for a downdate.

Usage

```
updown(update, C, L)

## S4 method for signature 'character,ANY,ANY'
updown(update, C, L)
## S4 method for signature 'logical,Matrix,sparseCholesky'
updown(update, C, L)
## S4 method for signature 'logical,matrix,sparseCholesky'
updown(update, C, L)
```

Arguments

update	a logical (TRUE or FALSE) or a character string ("+" or "-") indicating if L should be updated (or otherwise downdated).
C	an R object representing the k -column matrix C , such that <code>tcrossprod(C)</code> has the dimensions of L. Package Matrix provides methods for Matrix objects and traditional matrices; these coerce to CsparseMatrix as necessary.
L	an R object representing the original Cholesky factorization. Package Matrix provides methods for virtual class sparseCholesky .

Value

An R object representing the updated or downdated Cholesky factorization.

Methods provided by package **Matrix** return a new **sparseCholesky** object. The new factorization is square root-free, even if the old one is not, hence it is required that $A + s C C'$ has nonzero leading principal minors, not that it is positive definite.

Author(s)

Nicholas Nagle (University of Tennessee, Knoxville) provided the original implementation.

References

Davis, T. A., Hager, W. W. (2001). Multiple-rank modifications of a sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 22(4), 997-1013. doi:10.1137/S0895479899357346

See Also

Virtual class [sparseCholesky](#).

Generic function [Cholesky](#).

The method for generic function [update](#) with signature `c(object="sparseCholesky")`, which is *not* equivalent to `updown(update = TRUE, C = ., L = object)`.

Examples

```
X <- new("dgCMatrix", Dim = c(3L, 5L), p = c(0L, 1L, 2L, 4L, 4L, 6L),
        i = c(2L, 0L, 1L, 2L, 0L, 1L), x = as.double(1:6))
I5 <- new("ddiMatrix", Dim = c(5L, 5L), diag = "U")
A <- crossprod(X) + I5
u0 <- Cholesky(A)
u1 <- updown("+", I5, u0)
u2 <- updown("-", I5, u1)
stopifnot(all.equal(u0, u2))
```

USCounties

Contiguity Matrix of U.S. Counties

Description

This matrix gives the contiguities of 3111 U.S. counties, using the queen criterion of at least one shared vertex or edge.

Usage

```
data(USCounties)
```

Format

A 3111×3111 sparse, symmetric matrix of class [dsCMatrix](#), with 9101 nonzero entries.

Source

GAL lattice file ‘usc_q.GAL’ (retrieved in 2008 from ‘<http://sal.uiuc.edu/weights/zips/usc.zip>’ with permission from Luc Anselin for use and distribution) was read into R using function `read.gal` from package [spdep](#).

Neighbour lists were augmented with row-standardized (and then symmetrized) spatial weights, using functions `nb2listw` and `similar.listw` from packages [spdep](#) and [spatialreg](#). The resulting listw object was coerced to class [dsTMatrix](#) using `as_dsTMatrix_listw` from [spatialreg](#), and subsequently to class [dsCMatrix](#).

References

Ord, J. K. (1975). Estimation methods for models of spatial interaction. *Journal of the American Statistical Association*, 70(349), 120-126. doi:[10.2307/2285387](https://doi.org/10.2307/2285387)

Examples

```
data(USCounties, package = "Matrix")
(n <- ncol(USCounties))
I <- .symDiagonal(n)

set.seed(1)
r <- 50L
rho <- 1 / runif(r, 0, 0.5)

system.time(MJ0 <- sapply(rho, function(mult)
  determinant(USCounties + mult * I, logarithm = TRUE)$modulus))

## Can be done faster by updating the Cholesky factor:

C1 <- Cholesky(USCounties, Imult = 2)
system.time(MJ1 <- sapply(rho, function(mult)
  determinant(update(C1, USCounties, mult), sqrt = FALSE)$modulus))
stopifnot(all.equal(MJ0, MJ1))

C2 <- Cholesky(USCounties, super = TRUE, Imult = 2)
system.time(MJ2 <- sapply(rho, function(mult)
  determinant(update(C2, USCounties, mult), sqrt = FALSE)$modulus))
stopifnot(all.equal(MJ0, MJ2))
```

which-methods

Which Indices are TRUE?

Description

S4 methods defined in package **Matrix** for S4 generic function `which`, generalizing base::[which](#).

Usage

```
## S4 method for signature 'ndenseMatrix'
which(x, arr.ind = FALSE, useNames = TRUE)
## S4 method for signature 'nsparseMatrix'
which(x, arr.ind = FALSE, useNames = TRUE)
## S4 method for signature 'ndiMatrix'
which(x, arr.ind = FALSE, useNames = TRUE)
## S4 method for signature 'nsparseVector'
which(x, arr.ind = FALSE, useNames = TRUE)

## S4 method for signature 'ldenseMatrix'
which(x, arr.ind = FALSE, useNames = TRUE)
## S4 method for signature 'lsparseMatrix'
which(x, arr.ind = FALSE, useNames = TRUE)
## S4 method for signature 'ldiMatrix'
which(x, arr.ind = FALSE, useNames = TRUE)
## S4 method for signature 'lsparseVector'
which(x, arr.ind = FALSE, useNames = TRUE)
```

Arguments

<code>x</code>	a Matrix object or a sparseVector object with pattern or logical data.
<code>arr.ind</code>	a logical indicating if array indices are computed and returned when <code>x</code> is a Matrix object.
<code>useNames</code>	a logical, passed to arrayInd when <code>x</code> is a Matrix and <code>arr.ind</code> is TRUE.

Value

An integer or double vector or a two-column integer matrix, following base: [which](#).

Examples

```
## Column index matrix: matrix whose columns are standard unit vectors
n <- 6L
r <- 4L
J <- new("indMatrix", Dim = c(n, r), Dimnames = list(u = NULL, v = NULL),
        margin = 2L, perm = sample(n, r, replace = TRUE))
J
(w0 <- which(J))
(w1 <- which(J, arr.ind = TRUE))
stopifnot(identical(w0, 0L:(r - 1L) * n + J@perm),
          identical(w1, cbind(u = J@perm, v = 1L:r)))
```

wrld_1deg

Contiguity Matrix of World One-Degree Grid Cells

Description

This matrix gives the contiguities of 15260 one-degree grid cells of world land areas, using a criterion based on the great-circle distance between centers.

Usage

```
data(wrld_1deg)
```

Format

A 15260×15260 sparse, symmetric matrix of class [dsCMatrix](#), with 55973 nonzero entries.

Source

Shoreline data were read into R from the GSHHS database using function `Rgshhs` from package **maptools**. Antarctica was excluded. An approximately one-degree grid was generated using function `Sobj_SpatialGrid`, also from **maptools**. Grid cells with centers on land were identified using the `over` method for classes `SpatialPolygons` and `SpatialGrid`, defined in package **sp**. Neighbours of these were identified by passing the resulting `SpatialPixels` object to function `dnearneigh` from package **spdep**, using as a cut-off a great-circle distance of $\sqrt{2}$ kilometers between centers.

Neighbour lists were augmented with row-standardized (and then symmetrized) spatial weights, using functions `nb2listw` and `similar.listw` from packages **spdep** and **spatialreg**. The resulting `listw` object was coerced to class `dsTMatrix` using `as_dsTMatrix_listw` from **spatialreg**, and subsequently to class `dsCMatrix`.

References

Ord, J. K. (1975). Estimation methods for models of spatial interaction. *Journal of the American Statistical Association*, 70(349), 120-126. doi:[10.2307/2285387](https://doi.org/10.2307/2285387)

Examples

```
data(wrld_1deg, package = "Matrix")
(n <- ncol(wrld_1deg))
I <- .symDiagonal(n)

doExtras <- interactive() || nzchar(Sys.getenv("R_MATRIX_CHECK_EXTRA"))
set.seed(1)
r <- if(doExtras) 20L else 3L
rho <- 1 / runif(r, 0, 0.5)

system.time(MJ0 <- sapply(rho, function(mult)
  determinant(wrld_1deg + mult * I, logarithm = TRUE)$modulus))

## Can be done faster by updating the Cholesky factor:

C1 <- Cholesky(wrld_1deg, Imult = 2)
system.time(MJ1 <- sapply(rho, function(mult)
  determinant(update(C1, wrld_1deg, mult), sqrt = FALSE)$modulus))
stopifnot(all.equal(MJ0, MJ1))

C2 <- Cholesky(wrld_1deg, super = TRUE, Imult = 2)
system.time(MJ2 <- sapply(rho, function(mult)
  determinant(update(C2, wrld_1deg, mult), sqrt = FALSE)$modulus))
stopifnot(all.equal(MJ0, MJ2))
```

xtfrm-methods

Auxiliary Function for Sorting and Ranking

Description

S4 methods defined in package **Matrix** for S4 generic function `xtfrm`.

Usage

```
## S4 method for signature 'denseMatrix'
xtfrm(x)
## S4 method for signature 'sparseMatrix'
xtfrm(x)
## S4 method for signature 'sparseVector'
xtfrm(x)
```

Arguments

`x` a [Matrix](#) object or a [sparseVector](#) object.

Value

The method with signature `x="sparseVector"` returns a numeric vector listing ranks. Ties are assigned one rank equal to the minimum of the indices. Structural and nonstructural zeros are assigned equal rank. [NA](#) and [NaN](#) are assigned rank NA.

The methods with signatures `x="denseMatrix"` and `x="sparseMatrix"` coerce `x` to vector and [sparseVector](#), respectively, then dispatch.

See Also

[sort](#), [order](#), [rank](#).

Examples

```
z <- new("zsparseVector", length = 8L, i = 2L:7L,
        x = c(0, 1, 1i, -1, -1i, 0))
z
xtfrm(z)
(i01 <- order(z)) # uses 'xtfrm'
(i10 <- order(z, decreasing = TRUE))

stopifnot(identical(i01, order(xtfrm(z))),
          identical(i10, order(Re(z), Im(z), decreasing = TRUE)))
```

zapsmall-methods

Rounding of Numbers: Zapping Small Ones to Zero

Description

S4 methods defined in package **Matrix** for S4 generic function `zapsmall`, generalizing `base::zapsmall`.

The S4 generic function definition is obtained from package **methods** through the [implicitGeneric](#) mechanism.

Usage

```
## S4 method for signature 'Matrix'
zapsmall(x, digits = getOption("digits"),
        mFUN = function(x, ina) max(abs(x[!ina])),
        min.d = 0L, ...)
## S4 method for signature 'sparseVector'
zapsmall(x, digits = getOption("digits"),
        mFUN = function(x, ina) max(abs(x[!ina])),
        min.d = 0L, ...)
```

Arguments

`x` a [Matrix](#) object or a [sparseVector](#) object. Pattern, logical, and integer data are converted to double.

`digits, mFUN, min.d` additional arguments passed to `base::zapsmall`.

`...` optional arguments not yet used.

Details

Methods preserve the nonzero pattern of sparse format `x`: zeros introduced by rounding are non-structural.

Value

An object representing the result of rounding `x`. The class is a superclass of the class of `x` unless `x` has pattern, logical, or integer data.

See Also

Function [drop0](#) for deleting small (on an absolute scale) entries of sparse format objects.

Methods for group generic function [Math2](#) for rounding to a decimal place or a number of significant digits.

Examples

```
x <- new("dgCMatrix", Dim = c(3L, 2L),
        p = c(0L, 3L, 4L), i = c(0L, 1L, 2L, 1L),
        x = c(0, 1, -51, 1e+6)) # maximum absolute value is 10^6
x
zapsmall(x, digits = 4L, min.d = -Inf) # rounds to nearest 10^(6-4)
zapsmall(x, digits = 3L, min.d = -Inf) # rounds to nearest 10^(6-3)
```

zMatrix-class

Virtual Class ‘zMatrix’ of Complex Matrices

Description

`zMatrix` is a virtual subclass of [Matrix](#) representing complex matrices. `zdenseMatrix` and `zsparseMatrix` are virtual subclasses of `zMatrix` representing its intersections with [denseMatrix](#) and [sparseMatrix](#).

Slots

`Dim, Dimnames` inherited from virtual superclass [Matrix](#).

`x` a [complex](#) vector storing matrix entries. Details such as *which* entries are stored vary by storage format.

Subclasses

The nonvirtual subclasses of zMatrix are listed below with links to virtual superclasses defining their structure and storage format.

Nonvirtual subclass	Structure	Storage format
zgeMatrix	generalMatrix	unpackedMatrix
zsyMatrix	symmetricMatrix	unpackedMatrix
zpoMatrix	posdefMatrix	unpackedMatrix
ztrMatrix	triangularMatrix	unpackedMatrix
zspMatrix	symmetricMatrix	packedMatrix
zppMatrix	posdefMatrix	packedMatrix
ztpMatrix	triangularMatrix	packedMatrix
zgCMatrix	generalMatrix	CsparseMatrix
zsCMatrix	symmetricMatrix	CsparseMatrix
zpCMatrix	posdefMatrix	CsparseMatrix
ztCMatrix	triangularMatrix	CsparseMatrix
zgRMatrix	generalMatrix	RsparseMatrix
zsRMatrix	symmetricMatrix	RsparseMatrix
zpRMatrix	posdefMatrix	RsparseMatrix
ztRMatrix	triangularMatrix	RsparseMatrix
zgTMatrix	generalMatrix	TsparseMatrix
zsTMatrix	symmetricMatrix	TsparseMatrix
zpTMatrix	posdefMatrix	TsparseMatrix
ztTMatrix	triangularMatrix	TsparseMatrix
zdiMatrix	diagonalMatrix	<i>unnamed</i>

Subclasses of zMatrix were not defined until **Matrix** version 1.8-0.

See Also

“Sister” classes [nMatrix](#), [lMatrix](#), [iMatrix](#), and [dMatrix](#) representing boolean, logical, integer, and double precision matrices.

Examples

```
## TODO
```

Index

- ! (not-methods), [132](#)
- !,Matrix-method (not-methods), [132](#)
- !,indMatrix-method (not-methods), [132](#)
- !,ldenseMatrix-method (not-methods), [132](#)
- !,ldiMatrix-method (not-methods), [132](#)
- !,lspareMatrix-method (not-methods), [132](#)
- !,lspareVector-method (not-methods), [132](#)
- !,ndenseMatrix-method (not-methods), [132](#)
- !,ndiMatrix-method (not-methods), [132](#)
- !,nspareMatrix-method (not-methods), [132](#)
- !,nspareVector-method (not-methods), [132](#)
- !,sparseVector-method (not-methods), [132](#)
- !-methods (not-methods), [132](#)
- * **Choleski**
 - Cholesky-methods, [25](#)
- * **NA**
 - is.na-methods, [96](#)
- * **algebra**
 - boolmatmult-methods, [17](#)
 - BunchKaufman-methods, [18](#)
 - chol-methods, [21](#)
 - chol2inv-methods, [24](#)
 - Cholesky-methods, [25](#)
 - colSums-methods, [33](#)
 - condest, [35](#)
 - cov2cor-methods, [37](#)
 - ct-methods, [40](#)
 - denseBunchKaufman-class, [43](#)
 - denseCholesky-class, [45](#)
 - denseLU-class, [48](#)
 - denseSchur-class, [51](#)
 - determinant-methods, [52](#)
 - diag-methods, [53](#)
 - dimScale, [63](#)
 - dmperm, [65](#)
 - expand-methods, [69](#)
 - KhatriRao, [103](#)
 - kronecker-methods, [106](#)
 - lu-methods, [110](#)
 - matmult-methods, [115](#)
 - MatrixFactorization-class, [122](#)
 - nearPD, [124](#)
 - norm-methods, [130](#)
 - qr-methods, [143](#)
 - rankMatrix, [146](#)
 - rcond-methods, [149](#)
 - Schur-methods, [158](#)
 - skewpart-methods, [160](#)
 - solve-methods, [161](#)
 - sparseCholesky-class, [167](#)
 - sparseLU-class, [171](#)
 - sparseQR-class, [178](#)
 - sympart-methods, [194](#)
 - t-methods, [196](#)
 - update-methods, [207](#)
 - updown-methods, [208](#)
- * **arith**
 - all.equal-methods, [8](#)
 - colSums-methods, [33](#)
 - diff-methods, [59](#)
 - dimScale, [63](#)
 - facmul-methods, [75](#)
 - KhatriRao, [103](#)
 - kronecker-methods, [106](#)
 - matmult-methods, [115](#)
 - Ops-methods, [133](#)
 - skewpart-methods, [160](#)
 - sort-methods, [164](#)
 - Summary-methods, [191](#)
 - sympart-methods, [194](#)
- * **array**
 - asUniqueT, [8](#)
 - band-methods, [10](#)
 - bandSparse, [11](#)

- bdiag, 13
- bind-methods, 15
- boolmatmult-methods, 17
- BunchKaufman-methods, 18
- chol-methods, 21
- chol2inv-methods, 24
- Cholesky-methods, 25
- colSums-methods, 33
- cov2cor-methods, 37
- CsparseMatrix-class, 37
- ct-methods, 40
- denseBunchKaufman-class, 43
- denseCholesky-class, 45
- denseLU-class, 48
- denseMatrix-class, 50
- denseSchur-class, 51
- Diagonal, 54
- diagonalMatrix-class, 56
- diagU2N, 57
- dim-methods, 60
- dimnames-methods, 61
- dimScale, 63
- dMatrix-class, 64
- dmperm, 65
- drop-methods, 67
- drop0, 68
- expand-methods, 69
- expm-methods, 72
- facmul-methods, 75
- forceCanonical-methods, 80
- forceDiagonal-methods, 81
- forceSymmetric-methods, 81
- forceTriangular-methods, 83
- generalMatrix-class, 85
- head-methods, 86
- Hilbert, 87
- iMatrix-class, 90
- indexMatrix-class, 91
- is.null.DN, 97
- isCanonical-methods, 98
- isDiagonal-methods, 100
- isSymmetric-methods, 101
- isTriangular-methods, 102
- KhatriRao, 103
- kronecker-methods, 106
- length-methods, 107
- lMatrix-class, 109
- lu-methods, 110
- mat2triplet, 112
- matmult-methods, 115
- Matrix, 117
- Matrix-class, 119
- MatrixFactorization-class, 122
- nearPD, 124
- nMatrix-class, 127
- nnzero-methods, 129
- pack-methods, 134
- packedMatrix-class, 135
- posdefMatrix-class, 139
- qr-methods, 143
- rsparsematrix, 154
- RsparseMatrix-class, 156
- Schur-methods, 158
- skewpart-methods, 160
- solve-methods, 161
- sparse.model.matrix, 165
- sparseCholesky-class, 167
- sparseLU-class, 171
- sparseMatrix, 173
- sparseMatrix-class, 177
- sparseQR-class, 178
- spMatrix, 186
- subassign-methods, 187
- subset-methods, 189
- symmetricMatrix-class, 192
- sympart-methods, 194
- t-methods, 196
- toeplitz-methods, 198
- triangularMatrix-class, 199
- TsparseMatrix-class, 200
- unpackedMatrix-class, 204
- update-methods, 207
- updown-methods, 208
- zapsmall-methods, 214
- zMatrix-class, 215
- * attribute**
 - diagU2N, 57
 - dim-methods, 60
 - dimnames-methods, 61
 - is.null.DN, 97
- * character**
 - formatSparseM, 83
 - printSpMatrix, 140
 - show-methods, 159
- * classes**
 - abIndex-class, 5

- CsparseMatrix-class, 37
- denseBunchKaufman-class, 43
- denseCholesky-class, 45
- denseLU-class, 48
- denseMatrix-class, 50
- denseSchur-class, 51
- diagonalMatrix-class, 56
- dMatrix-class, 64
- generalMatrix-class, 85
- iMatrix-class, 90
- index-class, 91
- indexMatrix-class, 91
- lMatrix-class, 109
- Matrix-class, 119
- MatrixFactorization-class, 122
- nMatrix-class, 127
- packedMatrix-class, 135
- posdefMatrix-class, 139
- rleDiff-class, 153
- RsparseMatrix-class, 156
- sparseCholesky-class, 167
- sparseLU-class, 171
- sparseMatrix-class, 177
- sparseQR-class, 178
- sparseVector-class, 183
- symmetricMatrix-class, 192
- triangularMatrix-class, 199
- TsparseMatrix-class, 200
- unpackedMatrix-class, 204
- zMatrix-class, 215
- * **complex**
 - Complex-methods, 35
- * **connection**
 - externalFormats, 73
- * **datasets**
 - CAex, 20
 - KNex, 105
 - USCounties, 210
 - wrld_1deg, 212
- * **distribution**
 - rsparsematrix, 154
- * **file**
 - externalFormats, 73
- * **hplot**
 - image-methods, 87
- * **logic**
 - all.equal-methods, 8
 - asUniqueT, 8
 - boolmatmult-methods, 17
 - nnzero-methods, 129
 - not-methods, 132
 - Ops-methods, 133
 - Summary-methods, 191
 - which-methods, 211
- * **manip**
 - abIseq, 7
 - asUniqueT, 8
 - bind-methods, 15
 - dim-methods, 60
 - dimnames-methods, 61
 - drop0, 68
 - length-methods, 107
 - rep-methods, 151
 - rep2abI, 153
 - sort-methods, 164
 - sparseVector-class, 183
 - xtfrm-methods, 213
 - zapsmall-methods, 214
- * **math**
 - condest, 35
 - determinant-methods, 52
 - diag-methods, 53
 - expm-methods, 72
 - is.na-methods, 96
 - Math-methods, 114
 - Math2-methods, 114
 - norm-methods, 130
 - rcond-methods, 149
- * **methods**
 - all.equal-methods, 8
 - band-methods, 10
 - bind-methods, 15
 - boolmatmult-methods, 17
 - BunchKaufman-methods, 18
 - chol-methods, 21
 - chol2inv-methods, 24
 - Cholesky-methods, 25
 - coerce-methods, 31
 - coerce-methods-graph, 31
 - coerce-methods-SparseM, 32
 - colSums-methods, 33
 - Complex-methods, 35
 - cov2cor-methods, 37
 - ct-methods, 40
 - determinant-methods, 52
 - diag-methods, 53

- diff-methods, 59
- dim-methods, 60
- dimnames-methods, 61
- drop-methods, 67
- expand-methods, 69
- expm-methods, 72
- externalFormats, 73
- facmul-methods, 75
- forceCanonical-methods, 80
- forceDiagonal-methods, 81
- forceSymmetric-methods, 81
- forceTriangular-methods, 83
- head-methods, 86
- image-methods, 87
- initialize-methods, 94
- is.na-methods, 96
- isCanonical-methods, 98
- isDiagonal-methods, 100
- isSymmetric-methods, 101
- isTriangular-methods, 102
- kronecker-methods, 106
- length-methods, 107
- lu-methods, 110
- Math-methods, 114
- Math2-methods, 114
- mean-methods, 123
- nnzero-methods, 129
- norm-methods, 130
- not-methods, 132
- Ops-methods, 133
- pack-methods, 134
- qr-methods, 143
- rcond-methods, 149
- rep-methods, 151
- Schur-methods, 158
- show-methods, 159
- skewpart-methods, 160
- solve-methods, 161
- sort-methods, 164
- subassign-methods, 187
- subset-methods, 189
- Summary-methods, 191
- sympart-methods, 194
- t-methods, 196
- toeplitz-methods, 198
- update-methods, 207
- updown-methods, 208
- which-methods, 211
- xtfrm-methods, 213
- zapsmall-methods, 214
- * models**
 - sparse.model.matrix, 165
- * package**
 - Matrix-package, 5
- * print**
 - formatSparseM, 83
 - printSpMatrix, 140
 - show-methods, 159
- * programming**
 - all.equal-methods, 8
 - forceCanonical-methods, 80
 - is.na-methods, 96
 - is.null.DN, 97
 - isCanonical-methods, 98
 - isDiagonal-methods, 100
 - isSymmetric-methods, 101
 - isTriangular-methods, 102
 - sparseCholesky-class, 167
- * ts**
 - toeplitz-methods, 198
- * univar**
 - mean-methods, 123
 - sort-methods, 164
 - xtfrm-methods, 213
- * utilities**
 - abIseq, 7
 - asUniqueT, 8
 - bandSparse, 11
 - bdiag, 13
 - coerce-methods-graph, 31
 - condest, 35
 - Diagonal, 54
 - diagU2N, 57
 - dimScale, 63
 - dmperm, 65
 - drop0, 68
 - externalFormats, 73
 - fastMisc, 76
 - formatSparseM, 83
 - Hilbert, 87
 - invertPerm, 94
 - is.null.DN, 97
 - KhatriRao, 103
 - mat2triplet, 112
 - Matrix, 117
 - MatrixClass, 121

- nearPD, [124](#)
- printSpMatrix, [140](#)
- rankMatrix, [146](#)
- rep2abI, [153](#)
- rsparsematrix, [154](#)
- sparse.model.matrix, [165](#)
- sparseCholesky-class, [167](#)
- sparseMatrix, [173](#)
- sparseQR-class, [178](#)
- sparseVector, [182](#)
- spMatrix, [186](#)
- *Matrix, ddiMatrix-method (Ops-methods), [133](#)
- *Matrix, idiMatrix-method (Ops-methods), [133](#)
- *Matrix, ldiMatrix-method (Ops-methods), [133](#)
- *Matrix, ndiMatrix-method (Ops-methods), [133](#)
- *Matrix, zdiMatrix-method (Ops-methods), [133](#)
- *ddenseMatrix, ddiMatrix-method (Ops-methods), [133](#)
- *ddenseMatrix, idiMatrix-method (Ops-methods), [133](#)
- *ddenseMatrix, ldiMatrix-method (Ops-methods), [133](#)
- *ddenseMatrix, ndiMatrix-method (Ops-methods), [133](#)
- *ddenseMatrix, zdiMatrix-method (Ops-methods), [133](#)
- *ddiMatrix, Matrix-method (Ops-methods), [133](#)
- *ddiMatrix, ddenseMatrix-method (Ops-methods), [133](#)
- *ddiMatrix, idenseMatrix-method (Ops-methods), [133](#)
- *ddiMatrix, ldenseMatrix-method (Ops-methods), [133](#)
- *ddiMatrix, ndenseMatrix-method (Ops-methods), [133](#)
- *ddiMatrix, zdenseMatrix-method (Ops-methods), [133](#)
- *idenseMatrix, ddiMatrix-method (Ops-methods), [133](#)
- *idenseMatrix, idiMatrix-method (Ops-methods), [133](#)
- *idenseMatrix, ldiMatrix-method (Ops-methods), [133](#)

- (Ops-methods), [133](#)
- `*, idenseMatrix, ndiMatrix-method`
(Ops-methods), [133](#)
- `*, idenseMatrix, zdiMatrix-method`
(Ops-methods), [133](#)
- `*, idiMatrix, Matrix-method`
(Ops-methods), [133](#)
- `*, idiMatrix, ddenseMatrix-method`
(Ops-methods), [133](#)
- `*, idiMatrix, idenseMatrix-method`
(Ops-methods), [133](#)
- `*, idiMatrix, ldenseMatrix-method`
(Ops-methods), [133](#)
- `*, idiMatrix, ndenseMatrix-method`
(Ops-methods), [133](#)
- `*, idiMatrix, zdenseMatrix-method`
(Ops-methods), [133](#)
- `*, ldenseMatrix, ddiMatrix-method`
(Ops-methods), [133](#)
- `*, ldenseMatrix, idiMatrix-method`
(Ops-methods), [133](#)
- `*, ldenseMatrix, ldiMatrix-method`
(Ops-methods), [133](#)
- `*, ldenseMatrix, ndiMatrix-method`
(Ops-methods), [133](#)
- `*, ldenseMatrix, zdiMatrix-method`
(Ops-methods), [133](#)
- `*, ldiMatrix, Matrix-method`
(Ops-methods), [133](#)
- `*, ldiMatrix, ddenseMatrix-method`
(Ops-methods), [133](#)
- `*, ldiMatrix, idenseMatrix-method`
(Ops-methods), [133](#)
- `*, ldiMatrix, ldenseMatrix-method`
(Ops-methods), [133](#)
- `*, ldiMatrix, ndenseMatrix-method`
(Ops-methods), [133](#)
- `*, ldiMatrix, zdenseMatrix-method`
(Ops-methods), [133](#)
- `*, ndenseMatrix, ddiMatrix-method`
(Ops-methods), [133](#)
- `*, ndenseMatrix, idiMatrix-method`
(Ops-methods), [133](#)
- `*, ndenseMatrix, ldiMatrix-method`
(Ops-methods), [133](#)
- `*, ndenseMatrix, ndiMatrix-method`
(Ops-methods), [133](#)
- `*, ndenseMatrix, zdiMatrix-method`

- (Ops-methods), [133](#)
- `*`, `ndiMatrix`, `Matrix`-method (Ops-methods), [133](#)
- `*`, `ndiMatrix`, `ddenseMatrix`-method (Ops-methods), [133](#)
- `*`, `ndiMatrix`, `idenseMatrix`-method (Ops-methods), [133](#)
- `*`, `ndiMatrix`, `ldenseMatrix`-method (Ops-methods), [133](#)
- `*`, `ndiMatrix`, `ndenseMatrix`-method (Ops-methods), [133](#)
- `*`, `ndiMatrix`, `zdenseMatrix`-method (Ops-methods), [133](#)
- `*`, `zdenseMatrix`, `ddiMatrix`-method (Ops-methods), [133](#)
- `*`, `zdenseMatrix`, `idiMatrix`-method (Ops-methods), [133](#)
- `*`, `zdenseMatrix`, `ldiMatrix`-method (Ops-methods), [133](#)
- `*`, `zdenseMatrix`, `ndiMatrix`-method (Ops-methods), [133](#)
- `*`, `zdenseMatrix`, `zdiMatrix`-method (Ops-methods), [133](#)
- `*`, `zdiMatrix`, `Matrix`-method (Ops-methods), [133](#)
- `*`, `zdiMatrix`, `ddenseMatrix`-method (Ops-methods), [133](#)
- `*`, `zdiMatrix`, `idenseMatrix`-method (Ops-methods), [133](#)
- `*`, `zdiMatrix`, `ldenseMatrix`-method (Ops-methods), [133](#)
- `*`, `zdiMatrix`, `ndenseMatrix`-method (Ops-methods), [133](#)
- `*`, `zdiMatrix`, `zdenseMatrix`-method (Ops-methods), [133](#)
- `+`, `Matrix`, `missing`-method (Ops-methods), [133](#)
- `+`, `dgTMatrix`, `dgTMatrix`-method (Ops-methods), [133](#)
- `+`, `sparseVector`, `missing`-method (Ops-methods), [133](#)
- `-`, `Matrix`, `missing`-method (Ops-methods), [133](#)
- `-`, `sparseVector`, `missing`-method (Ops-methods), [133](#)
- `.CR2RC` (`fastMisc`), [76](#)
- `.CR2T` (`fastMisc`), [76](#)
- `.M2C` (`fastMisc`), [76](#)
- `.M2R` (`fastMisc`), [76](#)
- `.M2T` (`fastMisc`), [76](#)
- `.M2V` (`fastMisc`), [76](#)
- `.M2diag` (`fastMisc`), [76](#)
- `.M2gen` (`fastMisc`), [76](#)
- `.M2kind` (`fastMisc`), [76](#)
- `.M2m` (`fastMisc`), [76](#)
- `.M2packed` (`fastMisc`), [76](#)
- `.M2sym` (`fastMisc`), [76](#)
- `.M2tri` (`fastMisc`), [76](#)
- `.M2unpacked` (`fastMisc`), [76](#)
- `.M2v` (`fastMisc`), [76](#)
- `.Machine`, [147](#), [183](#)
- `.S3methods`, [120](#)
- `.S4methods`, [120](#)
- `.T2CR` (`fastMisc`), [76](#)
- `.bdiag` (`bdiag`), [13](#)
- `.ctCRT`, [197](#)
- `.ctCRT` (`ct`-methods), [40](#)
- `.dense2g` (`fastMisc`), [76](#)
- `.dense2kind` (`fastMisc`), [76](#)
- `.dense2m` (`fastMisc`), [76](#)
- `.dense2sparse` (`fastMisc`), [76](#)
- `.dense2v` (`fastMisc`), [76](#)
- `.diag.dsC` (`fastMisc`), [76](#)
- `.diag2dense` (`fastMisc`), [76](#)
- `.diag2sparse` (`fastMisc`), [76](#)
- `.diagN2U` (`diagU2N`), [57](#)
- `.diagU2N` (`diagU2N`), [57](#)
- `.formatSparseSimple`, [142](#)
- `.formatSparseSimple` (`formatSparseM`), [83](#)
- `.ind2dense` (`fastMisc`), [76](#)
- `.ind2sparse` (`fastMisc`), [76](#)
- `.m2V` (`fastMisc`), [76](#)
- `.m2dense` (`fastMisc`), [76](#)
- `.m2sparse` (`fastMisc`), [76](#)
- `.selectSuperClasses`, [121](#)
- `.solve.dgC.chol` (`fastMisc`), [76](#)
- `.solve.dgC.lu` (`fastMisc`), [76](#)
- `.solve.dgC.qr` (`fastMisc`), [76](#)
- `.sparse2dense` (`fastMisc`), [76](#)
- `.sparse2g` (`fastMisc`), [76](#)
- `.sparse2kind` (`fastMisc`), [76](#)
- `.sparse2m` (`fastMisc`), [76](#)
- `.sparse2v` (`fastMisc`), [76](#)
- `.sparseDiagonal` (`Diagonal`), [54](#)
- `.symDiagonal` (`Diagonal`), [54](#)
- `.tCR2RC` (`fastMisc`), [76](#)

- .tCRT, [42](#)
- .tCRT (t-methods), [196](#)
- .trDiagonal (Diagonal), [54](#)
- .updateCHMfactor (fastMisc), [76](#)
- .validateCsparse (CsparseMatrix-class), [37](#)
- /, ddiMatrix, Matrix-method (Ops-methods), [133](#)
- /, ddiMatrix, ddenseMatrix-method (Ops-methods), [133](#)
- /, ddiMatrix, idenseMatrix-method (Ops-methods), [133](#)
- /, ddiMatrix, ldenseMatrix-method (Ops-methods), [133](#)
- /, ddiMatrix, ndenseMatrix-method (Ops-methods), [133](#)
- /, ddiMatrix, zdenseMatrix-method (Ops-methods), [133](#)
- /, idiMatrix, Matrix-method (Ops-methods), [133](#)
- /, idiMatrix, ddenseMatrix-method (Ops-methods), [133](#)
- /, idiMatrix, idenseMatrix-method (Ops-methods), [133](#)
- /, idiMatrix, ldenseMatrix-method (Ops-methods), [133](#)
- /, idiMatrix, ndenseMatrix-method (Ops-methods), [133](#)
- /, idiMatrix, zdenseMatrix-method (Ops-methods), [133](#)
- /, ldiMatrix, Matrix-method (Ops-methods), [133](#)
- /, ldiMatrix, ddenseMatrix-method (Ops-methods), [133](#)
- /, ldiMatrix, idenseMatrix-method (Ops-methods), [133](#)
- /, ldiMatrix, ldenseMatrix-method (Ops-methods), [133](#)
- /, ldiMatrix, ndenseMatrix-method (Ops-methods), [133](#)
- /, ldiMatrix, zdenseMatrix-method (Ops-methods), [133](#)
- /, ndiMatrix, Matrix-method (Ops-methods), [133](#)
- /, ndiMatrix, ddenseMatrix-method (Ops-methods), [133](#)
- /, ndiMatrix, idenseMatrix-method (Ops-methods), [133](#)
- /, ndiMatrix, ldenseMatrix-method (Ops-methods), [133](#)
- /, ndiMatrix, ndenseMatrix-method (Ops-methods), [133](#)
- /, ndiMatrix, zdenseMatrix-method (Ops-methods), [133](#)
- [, [39](#), [67](#), [91](#), [157](#), [189](#)
- [, Matrix, ANY, ANY, logical-method (subset-methods), [189](#)
- [, Matrix, ANY, ANY, missing-method (subset-methods), [189](#)
- [, Matrix, ANY, missing, logical-method (subset-methods), [189](#)
- [, Matrix, ANY, missing, missing-method (subset-methods), [189](#)
- [, Matrix, missing, ANY, logical-method (subset-methods), [189](#)
- [, Matrix, missing, ANY, missing-method (subset-methods), [189](#)
- [, Matrix, missing, missing, logical-method (subset-methods), [189](#)
- [, Matrix, missing, missing, missing-method (subset-methods), [189](#)
- [, abIndex, ANY, missing, missing-method (abIndex-class), [5](#)
- [, sparseVector, ANY, missing, missing-method (subset-methods), [189](#)
- [, sparseVector, missing, missing, missing-method (subset-methods), [189](#)
- [-methods (subset-methods), [189](#)
- [<- (subassign-methods), [187](#)
- [<-, CsparseMatrix, Matrix, missing, replValue-method (subassign-methods), [187](#)
- [<-, CsparseMatrix, index, index, replValue-method (subassign-methods), [187](#)
- [<-, CsparseMatrix, index, index, sparseVector-method

- (subassign-methods), 187
- [<-, indMatrix, index, index, ANY-method (subassign-methods), 187
- [<-, indMatrix, index, missing, ANY-method (subassign-methods), 187
- [<-, indMatrix, missing, index, ANY-method (subassign-methods), 187
- [<-, indMatrix, missing, missing, ANY-method (subassign-methods), 187
- [<-, sparseMatrix, ANY, ANY, sparseMatrix-method (subassign-methods), 187
- [<-, sparseMatrix, ANY, missing, sparseMatrix-method (subassign-methods), 187
- [<-, sparseMatrix, missing, ANY, sparseMatrix-method (subassign-methods), 187
- [<-, sparseMatrix, missing, missing, ANY-method (subassign-methods), 187
- [<-, sparseVector, ANY, missing, ANY-method (subassign-methods), 187
- [<-, sparseVector, missing, missing, ANY-method (subassign-methods), 187
- [<--methods (subassign-methods), 187
- %%%, (matmult-methods), 115
- %%%, ANY, Matrix-method (matmult-methods), 115
- %%%, ANY, sparseVector-method (matmult-methods), 115
- %%%, CsparseMatrix, CsparseMatrix-method (matmult-methods), 115
- %%%, CsparseMatrix, RsparseMatrix-method (matmult-methods), 115
- %%%, CsparseMatrix, TsparseMatrix-method (matmult-methods), 115
- %%%, CsparseMatrix, denseMatrix-method (matmult-methods), 115
- %%%, CsparseMatrix, diagonalMatrix-method (matmult-methods), 115
- %%%, CsparseMatrix, matrix-method (matmult-methods), 115
- %%%, CsparseMatrix, vector-method (matmult-methods), 115
- %%%, Matrix, ANY-method (matmult-methods), 115
- %%%, Matrix, indMatrix-method (matmult-methods), 115
- %%%, Matrix, pMatrix-method (matmult-methods), 115
- %%%, Matrix, sparseVector-method (matmult-methods), 115
- %%%, RsparseMatrix, CsparseMatrix-method (matmult-methods), 115
- %%%, RsparseMatrix, RsparseMatrix-method (matmult-methods), 115
- %%%, RsparseMatrix, TsparseMatrix-method (matmult-methods), 115
- %%%, RsparseMatrix, denseMatrix-method (matmult-methods), 115
- %%%, RsparseMatrix, diagonalMatrix-method (matmult-methods), 115
- %%%, RsparseMatrix, matrix-method (matmult-methods), 115
- %%%, RsparseMatrix, vector-method (matmult-methods), 115
- %%%, TsparseMatrix, CsparseMatrix-method (matmult-methods), 115
- %%%, TsparseMatrix, RsparseMatrix-method (matmult-methods), 115
- %%%, TsparseMatrix, TsparseMatrix-method (matmult-methods), 115
- %%%, TsparseMatrix, denseMatrix-method (matmult-methods), 115
- %%%, TsparseMatrix, diagonalMatrix-method (matmult-methods), 115
- %%%, TsparseMatrix, matrix-method (matmult-methods), 115
- %%%, TsparseMatrix, vector-method (matmult-methods), 115
- %%%, denseMatrix, CsparseMatrix-method (matmult-methods), 115
- %%%, denseMatrix, RsparseMatrix-method (matmult-methods), 115
- %%%, denseMatrix, TsparseMatrix-method (matmult-methods), 115
- %%%, denseMatrix, denseMatrix-method (matmult-methods), 115
- %%%, denseMatrix, diagonalMatrix-method (matmult-methods), 115
- %%%, denseMatrix, matrix-method (matmult-methods), 115
- %%%, denseMatrix, vector-method (matmult-methods), 115
- %%%, diagonalMatrix, CsparseMatrix-method (matmult-methods), 115
- %%%, diagonalMatrix, RsparseMatrix-method (matmult-methods), 115
- %%%, diagonalMatrix, TsparseMatrix-method (matmult-methods), 115

- (matmult-methods), 115
- %%,diagonalMatrix,denseMatrix-method
(matmult-methods), 115
- %%,diagonalMatrix,diagonalMatrix-method
(matmult-methods), 115
- %%,diagonalMatrix,matrix-method
(matmult-methods), 115
- %%,diagonalMatrix,vector-method
(matmult-methods), 115
- %%,indMatrix,Matrix-method
(matmult-methods), 115
- %%,indMatrix,indMatrix-method
(matmult-methods), 115
- %%,indMatrix,matrix-method
(matmult-methods), 115
- %%,indMatrix,pMatrix-method
(matmult-methods), 115
- %%,indMatrix,vector-method
(matmult-methods), 115
- %%,matrix,CsparseMatrix-method
(matmult-methods), 115
- %%,matrix,RsparseMatrix-method
(matmult-methods), 115
- %%,matrix,TsparseMatrix-method
(matmult-methods), 115
- %%,matrix,denseMatrix-method
(matmult-methods), 115
- %%,matrix,diagonalMatrix-method
(matmult-methods), 115
- %%,matrix,indMatrix-method
(matmult-methods), 115
- %%,matrix,pMatrix-method
(matmult-methods), 115
- %%,matrix,sparseVector-method
(matmult-methods), 115
- %%,pMatrix,Matrix-method
(matmult-methods), 115
- %%,pMatrix,indMatrix-method
(matmult-methods), 115
- %%,pMatrix,matrix-method
(matmult-methods), 115
- %%,pMatrix,pMatrix-method
(matmult-methods), 115
- %%,pMatrix,vector-method
(matmult-methods), 115
- %%,sparseVector,ANY-method
(matmult-methods), 115
- %%,sparseVector,Matrix-method
(matmult-methods), 115
- %%,sparseVector,matrix-method
(matmult-methods), 115
- %%,sparseVector,sparseVector-method
(matmult-methods), 115
- %%,sparseVector,vector-method
(matmult-methods), 115
- %%,vector,CsparseMatrix-method
(matmult-methods), 115
- %%,vector,RsparseMatrix-method
(matmult-methods), 115
- %%,vector,TsparseMatrix-method
(matmult-methods), 115
- %%,vector,denseMatrix-method
(matmult-methods), 115
- %%,vector,diagonalMatrix-method
(matmult-methods), 115
- %%,vector,indMatrix-method
(matmult-methods), 115
- %%,vector,pMatrix-method
(matmult-methods), 115
- %%,vector,sparseVector-method
(matmult-methods), 115
- %%-methods (matmult-methods), 115
- %%/,ddiMatrix,Matrix-method
(Ops-methods), 133
- %%/,ddiMatrix,ddenseMatrix-method
(Ops-methods), 133
- %%/,ddiMatrix,idenseMatrix-method
(Ops-methods), 133
- %%/,ddiMatrix,ldenseMatrix-method
(Ops-methods), 133
- %%/,ddiMatrix,ndenseMatrix-method
(Ops-methods), 133
- %%/,ddiMatrix,zdenseMatrix-method
(Ops-methods), 133
- %%/,idiMatrix,Matrix-method
(Ops-methods), 133
- %%/,idiMatrix,ddenseMatrix-method
(Ops-methods), 133
- %%/,idiMatrix,idenseMatrix-method
(Ops-methods), 133
- %%/,idiMatrix,ldenseMatrix-method
(Ops-methods), 133
- %%/,idiMatrix,ndenseMatrix-method
(Ops-methods), 133
- %%/,idiMatrix,zdenseMatrix-method
(Ops-methods), 133

- %% (boolmatmult-methods), 17
- %%, ANY, ANY-method
(boolmatmult-methods), 17
- %%, ANY, Matrix-method
(boolmatmult-methods), 17
- %%, ANY, matrix-method
(boolmatmult-methods), 17
- %%, ANY, sparseVector-method
(boolmatmult-methods), 17
- %%, ANY, vector-method
(boolmatmult-methods), 17
- %%, CsparseMatrix, CsparseMatrix-method
(boolmatmult-methods), 17
- %%, CsparseMatrix, RsparseMatrix-method
(boolmatmult-methods), 17
- %%, CsparseMatrix, TsparseMatrix-method
(boolmatmult-methods), 17
- %%, CsparseMatrix, denseMatrix-method
(boolmatmult-methods), 17
- %%, CsparseMatrix, diagonalMatrix-method
(boolmatmult-methods), 17
- %%, CsparseMatrix, matrix-method
(boolmatmult-methods), 17
- %%, CsparseMatrix, vector-method
(boolmatmult-methods), 17
- %%, Matrix, ANY-method
(boolmatmult-methods), 17
- %%, Matrix, indMatrix-method
(boolmatmult-methods), 17
- %%, Matrix, pMatrix-method
(boolmatmult-methods), 17
- %%, Matrix, sparseVector-method
(boolmatmult-methods), 17
- %%, RsparseMatrix, CsparseMatrix-method
(boolmatmult-methods), 17
- %%, RsparseMatrix, RsparseMatrix-method
(boolmatmult-methods), 17
- %%, RsparseMatrix, TsparseMatrix-method
(boolmatmult-methods), 17
- %%, RsparseMatrix, denseMatrix-method
(boolmatmult-methods), 17
- %%, RsparseMatrix, diagonalMatrix-method
(boolmatmult-methods), 17
- %%, RsparseMatrix, matrix-method
(boolmatmult-methods), 17
- %%, RsparseMatrix, vector-method
(boolmatmult-methods), 17
- %%, TsparseMatrix, CsparseMatrix-method
(boolmatmult-methods), 17
- %%, TsparseMatrix, RsparseMatrix-method
(boolmatmult-methods), 17
- %%, TsparseMatrix, TsparseMatrix-method
(boolmatmult-methods), 17
- %%, TsparseMatrix, denseMatrix-method
(boolmatmult-methods), 17
- %%, TsparseMatrix, diagonalMatrix-method
(boolmatmult-methods), 17
- %%, TsparseMatrix, matrix-method
(boolmatmult-methods), 17
- %%, TsparseMatrix, vector-method
(boolmatmult-methods), 17
- %%, denseMatrix, CsparseMatrix-method
(boolmatmult-methods), 17
- %%, denseMatrix, RsparseMatrix-method
(boolmatmult-methods), 17
- %%, denseMatrix, TsparseMatrix-method
(boolmatmult-methods), 17
- %%, denseMatrix, denseMatrix-method
(boolmatmult-methods), 17
- %%, denseMatrix, diagonalMatrix-method
(boolmatmult-methods), 17
- %%, denseMatrix, matrix-method
(boolmatmult-methods), 17
- %%, denseMatrix, vector-method
(boolmatmult-methods), 17
- %%, diagonalMatrix, CsparseMatrix-method
(boolmatmult-methods), 17
- %%, diagonalMatrix, RsparseMatrix-method
(boolmatmult-methods), 17
- %%, diagonalMatrix, TsparseMatrix-method
(boolmatmult-methods), 17
- %%, diagonalMatrix, denseMatrix-method
(boolmatmult-methods), 17
- %%, diagonalMatrix, diagonalMatrix-method
(boolmatmult-methods), 17
- %%, diagonalMatrix, matrix-method
(boolmatmult-methods), 17
- %%, diagonalMatrix, vector-method
(boolmatmult-methods), 17
- %%, indMatrix, Matrix-method
(boolmatmult-methods), 17
- %%, indMatrix, indMatrix-method
(boolmatmult-methods), 17
- %%, indMatrix, matrix-method
(boolmatmult-methods), 17
- %%, indMatrix, pMatrix-method
(boolmatmult-methods), 17

- (boolmatmult-methods), [17](#)
- %%,indMatrix,vector-method
 - (boolmatmult-methods), [17](#)
- %%,matrix,ANY-method
 - (boolmatmult-methods), [17](#)
- %%,matrix,CsparseMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,matrix,RsparseMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,matrix,TsparseMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,matrix,denseMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,matrix,diagonalMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,matrix,indMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,matrix,matrix-method
 - (boolmatmult-methods), [17](#)
- %%,matrix,pMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,matrix,sparseVector-method
 - (boolmatmult-methods), [17](#)
- %%,matrix,vector-method
 - (boolmatmult-methods), [17](#)
- %%,pMatrix,Matrix-method
 - (boolmatmult-methods), [17](#)
- %%,pMatrix,indMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,pMatrix,matrix-method
 - (boolmatmult-methods), [17](#)
- %%,pMatrix,pMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,pMatrix,vector-method
 - (boolmatmult-methods), [17](#)
- %%,sparseVector,ANY-method
 - (boolmatmult-methods), [17](#)
- %%,sparseVector,Matrix-method
 - (boolmatmult-methods), [17](#)
- %%,sparseVector,matrix-method
 - (boolmatmult-methods), [17](#)
- %%,sparseVector,sparseVector-method
 - (boolmatmult-methods), [17](#)
- %%,sparseVector,vector-method
 - (boolmatmult-methods), [17](#)
- %%,vector,ANY-method
 - (boolmatmult-methods), [17](#)
- %%,vector,CsparseMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,vector,RsparseMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,vector,TsparseMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,vector,denseMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,vector,diagonalMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,vector,indMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,vector,matrix-method
 - (boolmatmult-methods), [17](#)
- %%,vector,pMatrix-method
 - (boolmatmult-methods), [17](#)
- %%,vector,sparseVector-method
 - (boolmatmult-methods), [17](#)
- %%,vector,vector-method
 - (boolmatmult-methods), [17](#)
- %%-methods (boolmatmult-methods), [17](#)
- &,Matrix,ddiMatrix-method
 - (Ops-methods), [133](#)
- &,Matrix,idiMatrix-method
 - (Ops-methods), [133](#)
- &,Matrix,ldiMatrix-method
 - (Ops-methods), [133](#)
- &,Matrix,ndiMatrix-method
 - (Ops-methods), [133](#)
- &,Matrix,zdiMatrix-method
 - (Ops-methods), [133](#)
- &,ddenseMatrix,ddiMatrix-method
 - (Ops-methods), [133](#)
- &,ddenseMatrix,idiMatrix-method
 - (Ops-methods), [133](#)
- &,ddenseMatrix,ldiMatrix-method
 - (Ops-methods), [133](#)
- &,ddenseMatrix,ndiMatrix-method
 - (Ops-methods), [133](#)
- &,ddenseMatrix,zdiMatrix-method
 - (Ops-methods), [133](#)
- &,ddiMatrix,Matrix-method
 - (Ops-methods), [133](#)
- &,ddiMatrix,ddenseMatrix-method
 - (Ops-methods), [133](#)
- &,ddiMatrix,idenseMatrix-method
 - (Ops-methods), [133](#)
- &,ddiMatrix,ldenseMatrix-method
 - (Ops-methods), [133](#)

- `^,Matrix,idiMatrix-method`
(Ops-methods), [133](#)
- `^,Matrix,ldiMatrix-method`
(Ops-methods), [133](#)
- `^,Matrix,ndiMatrix-method`
(Ops-methods), [133](#)
- `^,Matrix,zdiMatrix-method`
(Ops-methods), [133](#)
- `^,ddenseMatrix,ddiMatrix-method`
(Ops-methods), [133](#)
- `^,ddenseMatrix,idiMatrix-method`
(Ops-methods), [133](#)
- `^,ddenseMatrix,ldiMatrix-method`
(Ops-methods), [133](#)
- `^,ddenseMatrix,ndiMatrix-method`
(Ops-methods), [133](#)
- `^,ddenseMatrix,zdiMatrix-method`
(Ops-methods), [133](#)
- `^,idenseMatrix,ddiMatrix-method`
(Ops-methods), [133](#)
- `^,idenseMatrix,idiMatrix-method`
(Ops-methods), [133](#)
- `^,idenseMatrix,ldiMatrix-method`
(Ops-methods), [133](#)
- `^,idenseMatrix,ndiMatrix-method`
(Ops-methods), [133](#)
- `^,idenseMatrix,zdiMatrix-method`
(Ops-methods), [133](#)
- `^,ldenseMatrix,ddiMatrix-method`
(Ops-methods), [133](#)
- `^,ldenseMatrix,idiMatrix-method`
(Ops-methods), [133](#)
- `^,ldenseMatrix,ldiMatrix-method`
(Ops-methods), [133](#)
- `^,ldenseMatrix,ndiMatrix-method`
(Ops-methods), [133](#)
- `^,ldenseMatrix,zdiMatrix-method`
(Ops-methods), [133](#)
- `^,ndenseMatrix,ddiMatrix-method`
(Ops-methods), [133](#)
- `^,ndenseMatrix,idiMatrix-method`
(Ops-methods), [133](#)
- `^,ndenseMatrix,ldiMatrix-method`
(Ops-methods), [133](#)
- `^,ndenseMatrix,ndiMatrix-method`
(Ops-methods), [133](#)
- `^,ndenseMatrix,zdiMatrix-method`
(Ops-methods), [133](#)
- `^,zdenseMatrix,ddiMatrix-method`
(Ops-methods), [133](#)
- `^,zdenseMatrix,idiMatrix-method`
(Ops-methods), [133](#)
- `^,zdenseMatrix,ldiMatrix-method`
(Ops-methods), [133](#)
- `^,zdenseMatrix,ndiMatrix-method`
(Ops-methods), [133](#)
- `^,zdenseMatrix,zdiMatrix-method`
(Ops-methods), [133](#)
- `abbreviate`, [141](#)
- `abIndex`, [7](#), [153](#), [154](#)
- `abIndex-class`, [5](#)
- `abIseq`, [5](#), [7](#), [153](#)
- `abIseq1 (abIseq)`, [7](#)
- `abs`, [88](#)
- `aggregateT`, [202](#)
- `aggregateT (asUniqueT)`, [8](#)
- `all.equal`, [8](#)
- `all.equal (all.equal-methods)`, [8](#)
- `all.equal, abIndex, abIndex-method`
(abIndex-class), [5](#)
- `all.equal, abIndex, numLike-method`
(abIndex-class), [5](#)
- `all.equal, Matrix, Matrix-method`
(all.equal-methods), [8](#)
- `all.equal, Matrix, sparseVector-method`
(all.equal-methods), [8](#)
- `all.equal, Matrix, vector-method`
(all.equal-methods), [8](#)
- `all.equal, numLike, abIndex-method`
(abIndex-class), [5](#)
- `all.equal, sparseVector, Matrix-method`
(all.equal-methods), [8](#)
- `all.equal, sparseVector, sparseVector-method`
(all.equal-methods), [8](#)
- `all.equal, sparseVector, vector-method`
(all.equal-methods), [8](#)
- `all.equal, vector, Matrix-method`
(all.equal-methods), [8](#)
- `all.equal, vector, sparseVector-method`
(all.equal-methods), [8](#)
- `all.equal-methods`, [8](#)
- `all.equal.numeric`, [8](#)
- `anyDuplicated`, [9](#)
- `anyDuplicatedT (asUniqueT)`, [8](#)
- `anyNA`, [96](#)
- `anyNA (is.na-methods)`, [96](#)

- anyNA, CsparseMatrix-method
(is.na-methods), [96](#)
- anyNA, denseMatrix-method
(is.na-methods), [96](#)
- anyNA, diagonalMatrix-method
(is.na-methods), [96](#)
- anyNA, indMatrix-method (is.na-methods),
[96](#)
- anyNA, RsparseMatrix-method
(is.na-methods), [96](#)
- anyNA, sparseVector-method
(is.na-methods), [96](#)
- anyNA, TsparseMatrix-method
(is.na-methods), [96](#)
- anyNA-methods (is.na-methods), [96](#)
- apply, [120](#)
- Arith (Ops-methods), [133](#)
- Arith, abIndex, abIndex-method
(abIndex-class), [5](#)
- Arith, abIndex, numLike-method
(abIndex-class), [5](#)
- Arith, complex, CsparseMatrix-method
(Ops-methods), [133](#)
- Arith, complex, dgCMatix-method
(Ops-methods), [133](#)
- Arith, complex, dgeMatrix-method
(Ops-methods), [133](#)
- Arith, complex, diagonalMatrix-method
(Ops-methods), [133](#)
- Arith, complex, dsparseMatrix-method
(Ops-methods), [133](#)
- Arith, complex, igCMatix-method
(Ops-methods), [133](#)
- Arith, complex, igeMatrix-method
(Ops-methods), [133](#)
- Arith, complex, isparseMatrix-method
(Ops-methods), [133](#)
- Arith, complex, lgCMatix-method
(Ops-methods), [133](#)
- Arith, complex, lgeMatrix-method
(Ops-methods), [133](#)
- Arith, complex, lsparseMatrix-method
(Ops-methods), [133](#)
- Arith, complex, ngCMatix-method
(Ops-methods), [133](#)
- Arith, complex, ngeMatrix-method
(Ops-methods), [133](#)
- Arith, complex, nsparseMatrix-method
(Ops-methods), [133](#)
- Arith, complex, zgCMatix-method
(Ops-methods), [133](#)
- Arith, complex, zgeMatrix-method
(Ops-methods), [133](#)
- Arith, complex, zsparseMatrix-method
(Ops-methods), [133](#)
- Arith, CsparseMatrix, complex-method
(Ops-methods), [133](#)
- Arith, CsparseMatrix, CsparseMatrix-method
(Ops-methods), [133](#)
- Arith, CsparseMatrix, logical-method
(Ops-methods), [133](#)
- Arith, CsparseMatrix, numeric-method
(Ops-methods), [133](#)
- Arith, denseMatrix, logical-method
(Ops-methods), [133](#)
- Arith, denseMatrix, numeric-method
(Ops-methods), [133](#)
- Arith, denseMatrix, sparseVector-method
(Ops-methods), [133](#)
- Arith, dgCMatix, complex-method
(Ops-methods), [133](#)
- Arith, dgCMatix, dgCMatix-method
(Ops-methods), [133](#)
- Arith, dgCMatix, logical-method
(Ops-methods), [133](#)
- Arith, dgCMatix, numeric-method
(Ops-methods), [133](#)
- Arith, dgeMatrix, complex-method
(Ops-methods), [133](#)
- Arith, dgeMatrix, dgeMatrix-method
(Ops-methods), [133](#)
- Arith, dgeMatrix, logical-method
(Ops-methods), [133](#)
- Arith, dgeMatrix, numeric-method
(Ops-methods), [133](#)
- Arith, dgeMatrix, sparseVector-method
(Ops-methods), [133](#)
- Arith, diagonalMatrix, complex-method
(Ops-methods), [133](#)
- Arith, diagonalMatrix, logical-method
(Ops-methods), [133](#)
- Arith, diagonalMatrix, numeric-method
(Ops-methods), [133](#)
- Arith, dpoMatrix, logical-method
(Ops-methods), [133](#)
- Arith, dpoMatrix, numeric-method

- (Ops-methods), [133](#)
- Arith,dppMatrix,logical-method
(Ops-methods), [133](#)
- Arith,dppMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,dsCMatrix,dsCMatrix-method
(Ops-methods), [133](#)
- Arith,dsparseMatrix,complex-method
(Ops-methods), [133](#)
- Arith,dsparseMatrix,logical-method
(Ops-methods), [133](#)
- Arith,dsparseMatrix,nsparseMatrix-method
(Ops-methods), [133](#)
- Arith,dsparseMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,dsparseVector,dsparseVector-method
(Ops-methods), [133](#)
- Arith,dtCMatrix,dtCMatrix-method
(Ops-methods), [133](#)
- Arith,igCMatrix,complex-method
(Ops-methods), [133](#)
- Arith,igCMatrix,igCMatrix-method
(Ops-methods), [133](#)
- Arith,igCMatrix,logical-method
(Ops-methods), [133](#)
- Arith,igCMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,igeMatrix,complex-method
(Ops-methods), [133](#)
- Arith,igeMatrix,igeMatrix-method
(Ops-methods), [133](#)
- Arith,igeMatrix,logical-method
(Ops-methods), [133](#)
- Arith,igeMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,igeMatrix,sparseVector-method
(Ops-methods), [133](#)
- Arith,iMatrix,lMatrix-method
(Ops-methods), [133](#)
- Arith,iMatrix,nMatrix-method
(Ops-methods), [133](#)
- Arith,isCMatrix,isCMatrix-method
(Ops-methods), [133](#)
- Arith,isparsedMatrix,complex-method
(Ops-methods), [133](#)
- Arith,isparsedMatrix,logical-method
(Ops-methods), [133](#)
- Arith,isparsedMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,itCMatrix,itCMatrix-method
(Ops-methods), [133](#)
- Arith,lgCMatrix,complex-method
(Ops-methods), [133](#)
- Arith,lgCMatrix,lgCMatrix-method
(Ops-methods), [133](#)
- Arith,lgCMatrix,logical-method
(Ops-methods), [133](#)
- Arith,lgCMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,lgeMatrix,complex-method
(Ops-methods), [133](#)
- Arith,lgeMatrix,lgeMatrix-method
(Ops-methods), [133](#)
- Arith,lgeMatrix,logical-method
(Ops-methods), [133](#)
- Arith,lgeMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,lgeMatrix,sparseVector-method
(Ops-methods), [133](#)
- Arith,lgTMatrix,lgTMatrix-method
(Ops-methods), [133](#)
- Arith,lMatrix,iMatrix-method
(Ops-methods), [133](#)
- Arith,lMatrix,logical-method
(Ops-methods), [133](#)
- Arith,lMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,logical,CsparseMatrix-method
(Ops-methods), [133](#)
- Arith,logical,denseMatrix-method
(Ops-methods), [133](#)
- Arith,logical,dgCMatrix-method
(Ops-methods), [133](#)
- Arith,logical,dgeMatrix-method
(Ops-methods), [133](#)
- Arith,logical,diagonalMatrix-method
(Ops-methods), [133](#)
- Arith,logical,dpoMatrix-method
(Ops-methods), [133](#)
- Arith,logical,dppMatrix-method
(Ops-methods), [133](#)
- Arith,logical,dsparseMatrix-method
(Ops-methods), [133](#)
- Arith,logical,igCMatrix-method
(Ops-methods), [133](#)
- Arith,logical,igeMatrix-method

- (Ops-methods), [133](#)
- Arith,logical,isparsedMatrix-method
(Ops-methods), [133](#)
- Arith,logical,lgeMatrix-method
(Ops-methods), [133](#)
- Arith,logical,lgeMatrix-method
(Ops-methods), [133](#)
- Arith,logical,lMatrix-method
(Ops-methods), [133](#)
- Arith,logical,lsparseMatrix-method
(Ops-methods), [133](#)
- Arith,logical,ngCMatrix-method
(Ops-methods), [133](#)
- Arith,logical,ngeMatrix-method
(Ops-methods), [133](#)
- Arith,logical,nMatrix-method
(Ops-methods), [133](#)
- Arith,logical,nsparseMatrix-method
(Ops-methods), [133](#)
- Arith,logical,zgCMatrix-method
(Ops-methods), [133](#)
- Arith,logical,zgeMatrix-method
(Ops-methods), [133](#)
- Arith,logical,zsparseMatrix-method
(Ops-methods), [133](#)
- Arith,lsCMatrix,lsCMatrix-method
(Ops-methods), [133](#)
- Arith,lsparseMatrix,complex-method
(Ops-methods), [133](#)
- Arith,lsparseMatrix,logical-method
(Ops-methods), [133](#)
- Arith,lsparseMatrix,Matrix-method
(Ops-methods), [133](#)
- Arith,lsparseMatrix,nsparseMatrix-method
(Ops-methods), [133](#)
- Arith,lsparseMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,ltCMatrix,ltCMatrix-method
(Ops-methods), [133](#)
- Arith,Matrix,lsparseMatrix-method
(Ops-methods), [133](#)
- Arith,Matrix,Matrix-method
(Ops-methods), [133](#)
- Arith,Matrix,nsparseMatrix-method
(Ops-methods), [133](#)
- Arith,Matrix,zMatrix-method
(Ops-methods), [133](#)
- Arith,ngCMatrix,complex-method
(Ops-methods), [133](#)
- Arith,ngCMatrix,logical-method
(Ops-methods), [133](#)
- Arith,ngCMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,ngeMatrix,complex-method
(Ops-methods), [133](#)
- Arith,ngeMatrix,logical-method
(Ops-methods), [133](#)
- Arith,ngeMatrix,ngeMatrix-method
(Ops-methods), [133](#)
- Arith,ngeMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,ngeMatrix,sparseVector-method
(Ops-methods), [133](#)
- Arith,nMatrix,iMatrix-method
(Ops-methods), [133](#)
- Arith,nMatrix,logical-method
(Ops-methods), [133](#)
- Arith,nMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,nsCMatrix,nsCMatrix-method
(Ops-methods), [133](#)
- Arith,nsparseMatrix,complex-method
(Ops-methods), [133](#)
- Arith,nsparseMatrix,dsparseMatrix-method
(Ops-methods), [133](#)
- Arith,nsparseMatrix,logical-method
(Ops-methods), [133](#)
- Arith,nsparseMatrix,lsparseMatrix-method
(Ops-methods), [133](#)
- Arith,nsparseMatrix,Matrix-method
(Ops-methods), [133](#)
- Arith,nsparseMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,ntCMatrix,ntCMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,CsparseMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,denseMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,dgCMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,dgeMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,diagonalMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,dpoMatrix-method

- (Ops-methods), [133](#)
- Arith,numeric,dppMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,dsparseMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,igCMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,igeMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,isparsedMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,lgCMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,lgeMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,lMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,lsparsedMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,ngCMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,ngeMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,nMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,nsparsedMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,zgCMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,zgeMatrix-method
(Ops-methods), [133](#)
- Arith,numeric,zsparseMatrix-method
(Ops-methods), [133](#)
- Arith,numLike,abIndex-method
(abIndex-class), [5](#)
- Arith,sparseVector,denseMatrix-method
(Ops-methods), [133](#)
- Arith,sparseVector,dgeMatrix-method
(Ops-methods), [133](#)
- Arith,sparseVector,igeMatrix-method
(Ops-methods), [133](#)
- Arith,sparseVector,lgeMatrix-method
(Ops-methods), [133](#)
- Arith,sparseVector,ngeMatrix-method
(Ops-methods), [133](#)
- Arith,sparseVector,sparseVector-method
(Ops-methods), [133](#)
- Arith,sparseVector,zgeMatrix-method
(Ops-methods), [133](#)
- Arith,triangularMatrix,diagonalMatrix-method
(Ops-methods), [133](#)
- Arith,zgCMatrix,complex-method
(Ops-methods), [133](#)
- Arith,zgCMatrix,logical-method
(Ops-methods), [133](#)
- Arith,zgCMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,zgCMatrix,zgCMatrix-method
(Ops-methods), [133](#)
- Arith,zgeMatrix,complex-method
(Ops-methods), [133](#)
- Arith,zgeMatrix,logical-method
(Ops-methods), [133](#)
- Arith,zgeMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,zgeMatrix,sparseVector-method
(Ops-methods), [133](#)
- Arith,zgeMatrix,zgeMatrix-method
(Ops-methods), [133](#)
- Arith,zMatrix,Matrix-method
(Ops-methods), [133](#)
- Arith,zsCMatrix,zsCMatrix-method
(Ops-methods), [133](#)
- Arith,zsparseMatrix,complex-method
(Ops-methods), [133](#)
- Arith,zsparseMatrix,logical-method
(Ops-methods), [133](#)
- Arith,zsparseMatrix,numeric-method
(Ops-methods), [133](#)
- Arith,ztCMatrix,ztCMatrix-method
(Ops-methods), [133](#)
- Arith-methods (Ops-methods), [133](#)
- arrayInd, [212](#)
- as, [31](#), [32](#)
- as.array,Matrix-method
(coerce-methods), [31](#)
- as.array,sparseVector-method
(coerce-methods), [31](#)
- as.array-methods (coerce-methods), [31](#)
- as.complex,Matrix-method
(coerce-methods), [31](#)
- as.complex,sparseVector-method
(coerce-methods), [31](#)
- as.complex-methods (coerce-methods), [31](#)
- as.integer, [60](#)
- as.integer,abIndex-method

- (abIndex-class), 5
- as.integer, Matrix-method
 - (coerce-methods), 31
- as.integer, sparseVector-method
 - (coerce-methods), 31
- as.integer-methods (coerce-methods), 31
- as.logical, Matrix-method
 - (coerce-methods), 31
- as.logical, sparseVector-method
 - (coerce-methods), 31
- as.logical-methods (coerce-methods), 31
- as.matrix, 120
- as.matrix, Matrix-method
 - (coerce-methods), 31
- as.matrix, sparseVector-method
 - (coerce-methods), 31
- as.matrix-methods (coerce-methods), 31
- as.numeric, abIndex-method
 - (abIndex-class), 5
- as.numeric, Matrix-method
 - (coerce-methods), 31
- as.numeric, sparseVector-method
 - (coerce-methods), 31
- as.numeric-methods (coerce-methods), 31
- as.vector, 8
- as.vector, abIndex-method
 - (abIndex-class), 5
- as.vector, Matrix-method
 - (coerce-methods), 31
- as.vector, sparseVector-method
 - (coerce-methods), 31
- as.vector-methods (coerce-methods), 31
- asPerm, 48
- asPerm (invertPerm), 94
- asUniqueT, 8, 113, 202
- attribute, 103

- band, 12, 55, 200
- band (band-methods), 10
- band, CsparseMatrix-method
 - (band-methods), 10
- band, denseMatrix-method (band-methods), 10
- band, diagonalMatrix-method
 - (band-methods), 10
- band, indMatrix-method (band-methods), 10
- band, matrix-method (band-methods), 10
- band, RsparseMatrix-method
 - (band-methods), 10

- band, TsparseMatrix-method
 - (band-methods), 10
- band-methods, 10
- bandSparse, 11, 11, 14, 55, 118, 175
- bdiag, 12, 13, 118, 175, 187
- bind-methods, 15
- boolmatmult-methods, 17
- BunchKaufman, 27, 43, 44, 71, 112, 123, 140, 144, 159, 163, 194
- BunchKaufman (BunchKaufman-methods), 18
- BunchKaufman, denseMatrix-method
 - (BunchKaufman-methods), 18
- BunchKaufman, matrix-method
 - (BunchKaufman-methods), 18
- BunchKaufman, sparseMatrix-method
 - (BunchKaufman-methods), 18
- BunchKaufman-class
 - (MatrixFactorization-class), 122
- BunchKaufman-methods, 18

- c, 5
- c.abIndex (abIseq), 7
- c.Matrix (bind-methods), 15
- c.sparseVector (bind-methods), 15
- CAex, 20
- cbind, 15, 16
- cbind2, 15, 16, 39
- cbind2 (bind-methods), 15
- cbind2, Matrix, Matrix-method
 - (bind-methods), 15
- cbind2, Matrix, matrix-method
 - (bind-methods), 15
- cbind2, matrix, Matrix-method
 - (bind-methods), 15
- cbind2, Matrix, missing-method
 - (bind-methods), 15
- cbind2, Matrix, NULL-method
 - (bind-methods), 15
- cbind2, Matrix, vector-method
 - (bind-methods), 15
- cbind2, NULL, Matrix-method
 - (bind-methods), 15
- cbind2, vector, Matrix-method
 - (bind-methods), 15
- cbind2-methods (bind-methods), 15
- character, 5, 12, 54, 91, 121, 122, 165, 166, 174
- chol, 22, 24–27, 140

chol (chol-methods), 21
 chol, CsparseMatrix-method
 (chol-methods), 21
 chol, denseMatrix-method (chol-methods),
 21
 chol, diagonalMatrix-method
 (chol-methods), 21
 chol, indMatrix-method (chol-methods), 21
 chol, RsparseMatrix-method
 (chol-methods), 21
 chol, TsparseMatrix-method
 (chol-methods), 21
 chol-methods, 21
 chol2inv, 24, 25
 chol2inv (chol2inv-methods), 24
 chol2inv, denseMatrix-method
 (chol2inv-methods), 24
 chol2inv, sparseMatrix-method
 (chol2inv-methods), 24
 chol2inv-methods, 24
 Cholesky, 19, 21, 22, 46, 47, 71, 77, 78, 112,
 123, 140, 144, 159, 163, 168–170,
 208, 210
 Cholesky (Cholesky-methods), 25
 Cholesky, CsparseMatrix-method
 (Cholesky-methods), 25
 Cholesky, denseMatrix-method
 (Cholesky-methods), 25
 Cholesky, diagonalMatrix-method
 (Cholesky-methods), 25
 Cholesky, indMatrix-method
 (Cholesky-methods), 25
 Cholesky, matrix-method
 (Cholesky-methods), 25
 Cholesky, RsparseMatrix-method
 (Cholesky-methods), 25
 Cholesky, TsparseMatrix-method
 (Cholesky-methods), 25
 Cholesky-class
 (MatrixFactorization-class),
 122
 Cholesky-methods, 25
 class, 5, 12, 16, 57, 58, 97, 100, 102, 121,
 125, 147, 153, 173, 182, 188
 coerce (coerce-methods), 31
 coerce, abIndex, integer-method
 (abIndex-class), 5
 coerce, abIndex, numeric-method

 (abIndex-class), 5
 coerce, abIndex, seqMat-method
 (abIndex-class), 5
 coerce, abIndex, vector-method
 (abIndex-class), 5
 coerce, ANY, denseMatrix-method
 (coerce-methods), 31
 coerce, ANY, Matrix-method
 (coerce-methods), 31
 coerce, ANY, sparseMatrix-method
 (coerce-methods), 31
 coerce, ANY, sparseVector-method
 (coerce-methods), 31
 coerce, ddenseBunchKaufman, dtpMatrix-method
 (denseBunchKaufman-class), 43
 coerce, ddenseBunchKaufman, dtrMatrix-method
 (denseBunchKaufman-class), 43
 coerce, ddenseCholesky, dtpMatrix-method
 (denseCholesky-class), 45
 coerce, ddenseCholesky, dtrMatrix-method
 (denseCholesky-class), 45
 coerce, ddenseLU, dgeMatrix-method
 (denseLU-class), 48
 coerce, ddenseSchur, dgeMatrix-method
 (denseSchur-class), 51
 coerce, dgCMatrix, matrix.csc-method
 (coerce-methods-SparseM), 32
 coerce, dgRMatrix, matrix.csr-method
 (coerce-methods-SparseM), 32
 coerce, dgTMatrix, matrix.coo-method
 (coerce-methods-SparseM), 32
 coerce, diagonalMatrix, symmetricMatrix-method
 (coerce-methods), 31
 coerce, diagonalMatrix, triangularMatrix-method
 (coerce-methods), 31
 coerce, dpoMatrix, corMatrix-method
 (coerce-methods), 31
 coerce, dppMatrix, copMatrix-method
 (coerce-methods), 31
 coerce, dsimplicialCholesky, dtCMatrix-method
 (sparseCholesky-class), 167
 coerce, dspMatrix, copMatrix-method
 (coerce-methods), 31
 coerce, dsupernodalCholesky, dgCMatrix-method
 (sparseCholesky-class), 167
 coerce, dsyMatrix, corMatrix-method
 (coerce-methods), 31
 coerce, factor, sparseMatrix-method

- (coerce-methods), 31
- coerce, generalMatrix, packedMatrix-method
(coerce-methods), 31
- coerce, graph, CsparseMatrix-method
(coerce-methods-graph), 31
- coerce, graph, Matrix-method
(coerce-methods-graph), 31
- coerce, graph, RsparseMatrix-method
(coerce-methods-graph), 31
- coerce, graph, sparseMatrix-method
(coerce-methods-graph), 31
- coerce, graph, TsparseMatrix-method
(coerce-methods-graph), 31
- coerce, graphAM, TsparseMatrix-method
(coerce-methods-graph), 31
- coerce, graphNEL, TsparseMatrix-method
(coerce-methods-graph), 31
- coerce, indMatrix, packedMatrix-method
(coerce-methods), 31
- coerce, indMatrix, pMatrix-method
(coerce-methods), 31
- coerce, list, indMatrix-method
(coerce-methods), 31
- coerce, logical, abIndex-method
(abIndex-class), 5
- coerce, Matrix, copMatrix-method
(coerce-methods), 31
- coerce, matrix, copMatrix-method
(coerce-methods), 31
- coerce, Matrix, corMatrix-method
(coerce-methods), 31
- coerce, matrix, corMatrix-method
(coerce-methods), 31
- coerce, Matrix, CsparseMatrix-method
(coerce-methods), 31
- coerce, matrix, CsparseMatrix-method
(coerce-methods), 31
- coerce, Matrix, ddenseMatrix-method
(coerce-methods), 31
- coerce, matrix, ddenseMatrix-method
(coerce-methods), 31
- coerce, Matrix, denseMatrix-method
(coerce-methods), 31
- coerce, matrix, denseMatrix-method
(coerce-methods), 31
- coerce, matrix, dgCMatrix-method
(coerce-methods), 31
- coerce, Matrix, diagonalMatrix-method
(coerce-methods), 31
- coerce, matrix, diagonalMatrix-method
(coerce-methods), 31
- coerce, Matrix, dMatrix-method
(coerce-methods), 31
- coerce, matrix, dMatrix-method
(coerce-methods), 31
- coerce, Matrix, dsparseMatrix-method
(coerce-methods), 31
- coerce, matrix, dsparseMatrix-method
(coerce-methods), 31
- coerce, Matrix, generalMatrix-method
(coerce-methods), 31
- coerce, matrix, generalMatrix-method
(coerce-methods), 31
- coerce, Matrix, graph-method
(coerce-methods-graph), 31
- coerce, Matrix, graphNEL-method
(coerce-methods-graph), 31
- coerce, Matrix, idenseMatrix-method
(coerce-methods), 31
- coerce, matrix, idenseMatrix-method
(coerce-methods), 31
- coerce, Matrix, iMatrix-method
(coerce-methods), 31
- coerce, matrix, iMatrix-method
(coerce-methods), 31
- coerce, Matrix, indMatrix-method
(coerce-methods), 31
- coerce, matrix, indMatrix-method
(coerce-methods), 31
- coerce, Matrix, isparseMatrix-method
(coerce-methods), 31
- coerce, matrix, isparseMatrix-method
(coerce-methods), 31
- coerce, Matrix, ldenseMatrix-method
(coerce-methods), 31
- coerce, matrix, ldenseMatrix-method
(coerce-methods), 31
- coerce, Matrix, lMatrix-method
(coerce-methods), 31
- coerce, matrix, lMatrix-method
(coerce-methods), 31
- coerce, Matrix, lsparseMatrix-method
(coerce-methods), 31
- coerce, matrix, lsparseMatrix-method
(coerce-methods), 31
- coerce, Matrix, matrix-method

- (coerce-methods), 31
- coerce,matrix,Matrix-method (coerce-methods), 31
- coerce,matrix,matrix.coo-method (coerce-methods-SparseM), 32
- coerce,Matrix,matrix.csc-method (coerce-methods-SparseM), 32
- coerce,Matrix,matrix.csr-method (coerce-methods-SparseM), 32
- coerce,Matrix,ndenseMatrix-method (coerce-methods), 31
- coerce,matrix,ndenseMatrix-method (coerce-methods), 31
- coerce,Matrix,nMatrix-method (coerce-methods), 31
- coerce,matrix,nMatrix-method (coerce-methods), 31
- coerce,Matrix,nsparseMatrix-method (coerce-methods), 31
- coerce,matrix,nsparseMatrix-method (coerce-methods), 31
- coerce,Matrix,packedMatrix-method (coerce-methods), 31
- coerce,matrix,packedMatrix-method (coerce-methods), 31
- coerce,Matrix,pMatrix-method (coerce-methods), 31
- coerce,matrix,pMatrix-method (coerce-methods), 31
- coerce,Matrix,posdefMatrix-method (coerce-methods), 31
- coerce,matrix,posdefMatrix-method (coerce-methods), 31
- coerce,Matrix,RsparseMatrix-method (coerce-methods), 31
- coerce,matrix,RsparseMatrix-method (coerce-methods), 31
- coerce,Matrix,sparseMatrix-method (coerce-methods), 31
- coerce,matrix,sparseMatrix-method (coerce-methods), 31
- coerce,Matrix,sparseVector-method (coerce-methods), 31
- coerce,matrix,sparseVector-method (coerce-methods), 31
- coerce,Matrix,symmetricMatrix-method (coerce-methods), 31
- coerce,matrix,symmetricMatrix-method (coerce-methods), 31
- coerce,Matrix,triangularMatrix-method (coerce-methods), 31
- coerce,matrix,triangularMatrix-method (coerce-methods), 31
- coerce,Matrix,TsparseMatrix-method (coerce-methods), 31
- coerce,matrix,TsparseMatrix-method (coerce-methods), 31
- coerce,Matrix,unpackedMatrix-method (coerce-methods), 31
- coerce,matrix,unpackedMatrix-method (coerce-methods), 31
- coerce,Matrix,zdenseMatrix-method (coerce-methods), 31
- coerce,matrix,zdenseMatrix-method (coerce-methods), 31
- coerce,Matrix,zMatrix-method (coerce-methods), 31
- coerce,matrix,zMatrix-method (coerce-methods), 31
- coerce,Matrix,zsparseMatrix-method (coerce-methods), 31
- coerce,matrix,zsparseMatrix-method (coerce-methods), 31
- coerce,matrix.coo,CsparseMatrix-method (coerce-methods-SparseM), 32
- coerce,matrix.coo,dgCMatrix-method (coerce-methods-SparseM), 32
- coerce,matrix.coo,dgTMatrix-method (coerce-methods-SparseM), 32
- coerce,matrix.coo,Matrix-method (coerce-methods-SparseM), 32
- coerce,matrix.coo,RsparseMatrix-method (coerce-methods-SparseM), 32
- coerce,matrix.coo,sparseMatrix-method (coerce-methods-SparseM), 32
- coerce,matrix.coo,TsparseMatrix-method (coerce-methods-SparseM), 32
- coerce,matrix.csc,CsparseMatrix-method (coerce-methods-SparseM), 32
- coerce,matrix.csc,dgCMatrix-method (coerce-methods-SparseM), 32
- coerce,matrix.csc,Matrix-method (coerce-methods-SparseM), 32
- coerce,matrix.csc,RsparseMatrix-method (coerce-methods-SparseM), 32
- coerce,matrix.csc,sparseMatrix-method

- (coerce-methods-SparseM), 32
- coerce,matrix.csc,TsparseMatrix-method
(coerce-methods-SparseM), 32
- coerce,matrix.csr,CsparseMatrix-method
(coerce-methods-SparseM), 32
- coerce,matrix.csr,dgCMatrix-method
(coerce-methods-SparseM), 32
- coerce,matrix.csr,dgRMatrix-method
(coerce-methods-SparseM), 32
- coerce,matrix.csr,Matrix-method
(coerce-methods-SparseM), 32
- coerce,matrix.csr,RsparseMatrix-method
(coerce-methods-SparseM), 32
- coerce,matrix.csr,sparseMatrix-method
(coerce-methods-SparseM), 32
- coerce,matrix.csr,TsparseMatrix-method
(coerce-methods-SparseM), 32
- coerce,nsparseMatrix,indMatrix-method
(coerce-methods), 31
- coerce,nsparseMatrix,pMatrix-method
(coerce-methods), 31
- coerce,numeric,abIndex-method
(abIndex-class), 5
- coerce,numeric,indMatrix-method
(coerce-methods), 31
- coerce,numeric,pMatrix-method
(coerce-methods), 31
- coerce,numeric,seqMat-method
(abIndex-class), 5
- coerce,seqMat,abIndex-method
(abIndex-class), 5
- coerce,seqMat,numeric-method
(abIndex-class), 5
- coerce,sparseVector,CsparseMatrix-method
(coerce-methods), 31
- coerce,sparseVector,denseMatrix-method
(coerce-methods), 31
- coerce,sparseVector,dsparseVector-method
(coerce-methods), 31
- coerce,sparseVector,generalMatrix-method
(coerce-methods), 31
- coerce,sparseVector,isparsedVector-method
(coerce-methods), 31
- coerce,sparseVector,lsparseVector-method
(coerce-methods), 31
- coerce,sparseVector,Matrix-method
(coerce-methods), 31
- coerce,sparseVector,nsparseVector-method
(coerce-methods), 31
- coerce,sparseVector,RsparseMatrix-method
(coerce-methods), 31
- coerce,sparseVector,sparseMatrix-method
(coerce-methods), 31
- coerce,sparseVector,TsparseMatrix-method
(coerce-methods), 31
- coerce,sparseVector,unpackedMatrix-method
(coerce-methods), 31
- coerce,sparseVector,zsparseVector-method
(coerce-methods), 31
- coerce,TsparseMatrix,graphNEL-method
(coerce-methods-graph), 31
- coerce,vector,CsparseMatrix-method
(coerce-methods), 31
- coerce,vector,ddenseMatrix-method
(coerce-methods), 31
- coerce,vector,denseMatrix-method
(coerce-methods), 31
- coerce,vector,dMatrix-method
(coerce-methods), 31
- coerce,vector,dsparseMatrix-method
(coerce-methods), 31
- coerce,vector,dsparseVector-method
(coerce-methods), 31
- coerce,vector,generalMatrix-method
(coerce-methods), 31
- coerce,vector,idenseMatrix-method
(coerce-methods), 31
- coerce,vector,iMatrix-method
(coerce-methods), 31
- coerce,vector,isparsedMatrix-method
(coerce-methods), 31
- coerce,vector,isparsedVector-method
(coerce-methods), 31
- coerce,vector,ldenseMatrix-method
(coerce-methods), 31
- coerce,vector,lMatrix-method
(coerce-methods), 31
- coerce,vector,lsparseMatrix-method
(coerce-methods), 31
- coerce,vector,lsparseVector-method
(coerce-methods), 31
- coerce,vector,Matrix-method
(coerce-methods), 31
- coerce,vector,ndenseMatrix-method
(coerce-methods), 31
- coerce,vector,nMatrix-method

- (coerce-methods), [31](#)
- coerce,vector,nsparseMatrix-method
(coerce-methods), [31](#)
- coerce,vector,nsparseVector-method
(coerce-methods), [31](#)
- coerce,vector,RsparseMatrix-method
(coerce-methods), [31](#)
- coerce,vector,sparseMatrix-method
(coerce-methods), [31](#)
- coerce,vector,sparseVector-method
(coerce-methods), [31](#)
- coerce,vector,TsparseMatrix-method
(coerce-methods), [31](#)
- coerce,vector,unpackedMatrix-method
(coerce-methods), [31](#)
- coerce,vector,zdenseMatrix-method
(coerce-methods), [31](#)
- coerce,vector,zMatrix-method
(coerce-methods), [31](#)
- coerce,vector,zsparseMatrix-method
(coerce-methods), [31](#)
- coerce,vector,zsparseVector-method
(coerce-methods), [31](#)
- coerce-methods, [31](#)
- coerce-methods-graph, [31](#)
- coerce-methods-SparseM, [32](#)
- colMeans(colSums-methods), [33](#)
- colMeans,CsparseMatrix-method
(colSums-methods), [33](#)
- colMeans,denseMatrix-method
(colSums-methods), [33](#)
- colMeans,diagonalMatrix-method
(colSums-methods), [33](#)
- colMeans,indMatrix-method
(colSums-methods), [33](#)
- colMeans,RsparseMatrix-method
(colSums-methods), [33](#)
- colMeans,TsparseMatrix-method
(colSums-methods), [33](#)
- colMeans-methods(colSums-methods), [33](#)
- colScale(dimScale), [63](#)
- colSums, [33](#), [34](#), [39](#)
- colSums(colSums-methods), [33](#)
- colSums,CsparseMatrix-method
(colSums-methods), [33](#)
- colSums,denseMatrix-method
(colSums-methods), [33](#)
- colSums,diagonalMatrix-method
(colSums-methods), [33](#)
- colSums,indMatrix-method
(colSums-methods), [33](#)
- colSums,RsparseMatrix-method
(colSums-methods), [33](#)
- colSums,TsparseMatrix-method
(colSums-methods), [33](#)
- colSums-methods, [33](#)
- Compare(Ops-methods), [133](#)
- Compare,complex,dMatrix-method
(Ops-methods), [133](#)
- Compare,complex,iMatrix-method
(Ops-methods), [133](#)
- Compare,complex,lMatrix-method
(Ops-methods), [133](#)
- Compare,complex,nMatrix-method
(Ops-methods), [133](#)
- Compare,complex,zMatrix-method
(Ops-methods), [133](#)
- Compare,CsparseMatrix,CsparseMatrix-method
(Ops-methods), [133](#)
- Compare,dMatrix,complex-method
(Ops-methods), [133](#)
- Compare,dMatrix,logical-method
(Ops-methods), [133](#)
- Compare,dMatrix,numeric-method
(Ops-methods), [133](#)
- Compare,iMatrix,complex-method
(Ops-methods), [133](#)
- Compare,iMatrix,logical-method
(Ops-methods), [133](#)
- Compare,iMatrix,numeric-method
(Ops-methods), [133](#)
- Compare,lgeMatrix,lgeMatrix-method
(Ops-methods), [133](#)
- Compare,lMatrix,complex-method
(Ops-methods), [133](#)
- Compare,lMatrix,logical-method
(Ops-methods), [133](#)
- Compare,lMatrix,numeric-method
(Ops-methods), [133](#)
- Compare,logical,dMatrix-method
(Ops-methods), [133](#)
- Compare,logical,iMatrix-method
(Ops-methods), [133](#)
- Compare,logical,lMatrix-method
(Ops-methods), [133](#)
- Compare,logical,nMatrix-method

- (Ops-methods), [133](#)
- Compare, logical, zMatrix-method (Ops-methods), [133](#)
- Compare, ngeMatrix, ngeMatrix-method (Ops-methods), [133](#)
- Compare, nMatrix, complex-method (Ops-methods), [133](#)
- Compare, nMatrix, logical-method (Ops-methods), [133](#)
- Compare, nMatrix, nMatrix-method (Ops-methods), [133](#)
- Compare, nMatrix, numeric-method (Ops-methods), [133](#)
- Compare, numeric, dMatrix-method (Ops-methods), [133](#)
- Compare, numeric, iMatrix-method (Ops-methods), [133](#)
- Compare, numeric, lMatrix-method (Ops-methods), [133](#)
- Compare, numeric, nMatrix-method (Ops-methods), [133](#)
- Compare, numeric, zMatrix-method (Ops-methods), [133](#)
- Compare, triangularMatrix, diagonalMatrix-method (Ops-methods), [133](#)
- Compare, zMatrix, complex-method (Ops-methods), [133](#)
- Compare, zMatrix, logical-method (Ops-methods), [133](#)
- Compare, zMatrix, numeric-method (Ops-methods), [133](#)
- Compare-methods (Ops-methods), [133](#)
- Complex (Complex-methods), [35](#)
- complex, [101](#), [133](#), [215](#)
- Complex, denseMatrix-method (Complex-methods), [35](#)
- Complex, diagonalMatrix-method (Complex-methods), [35](#)
- Complex, indMatrix-method (Complex-methods), [35](#)
- Complex, sparseMatrix-method (Complex-methods), [35](#)
- Complex, sparseVector-method (Complex-methods), [35](#)
- Complex-methods, [35](#)
- condest, [35](#), [36](#), [150](#)
- Conj, [41](#)
- contrasts, [165](#)
- copMatrix-class (packedMatrix-class), [135](#)
- corMatrix-class (unpackedMatrix-class), [204](#)
- cov2cor, [63](#)
- cov2cor (cov2cor-methods), [37](#)
- cov2cor, Matrix-method (cov2cor-methods), [37](#)
- cov2cor-methods, [37](#)
- crossprod, [42](#), [115](#), [116](#), [140](#), [197](#)
- crossprod (matmult-methods), [115](#)
- crossprod, ANY, Matrix-method (matmult-methods), [115](#)
- crossprod, ANY, sparseVector-method (matmult-methods), [115](#)
- crossprod, CsparseMatrix, CsparseMatrix-method (matmult-methods), [115](#)
- crossprod, CsparseMatrix, denseMatrix-method (matmult-methods), [115](#)
- crossprod, CsparseMatrix, diagonalMatrix-method (matmult-methods), [115](#)
- crossprod, CsparseMatrix, matrix-method (matmult-methods), [115](#)
- crossprod, CsparseMatrix, missing-method (matmult-methods), [115](#)
- crossprod, CsparseMatrix, RsparseMatrix-method (matmult-methods), [115](#)
- crossprod, CsparseMatrix, TsparseMatrix-method (matmult-methods), [115](#)
- crossprod, CsparseMatrix, vector-method (matmult-methods), [115](#)
- crossprod, denseMatrix, CsparseMatrix-method (matmult-methods), [115](#)
- crossprod, denseMatrix, denseMatrix-method (matmult-methods), [115](#)
- crossprod, denseMatrix, diagonalMatrix-method (matmult-methods), [115](#)
- crossprod, denseMatrix, matrix-method (matmult-methods), [115](#)
- crossprod, denseMatrix, missing-method (matmult-methods), [115](#)
- crossprod, denseMatrix, RsparseMatrix-method (matmult-methods), [115](#)
- crossprod, denseMatrix, TsparseMatrix-method (matmult-methods), [115](#)
- crossprod, denseMatrix, vector-method (matmult-methods), [115](#)
- crossprod, diagonalMatrix, CsparseMatrix-method

- (matmult-methods), 115
- crossprod,diagonalMatrix,denseMatrix-method (matmult-methods), 115
- crossprod,diagonalMatrix,diagonalMatrix-method (matmult-methods), 115
- crossprod,diagonalMatrix,matrix-method (matmult-methods), 115
- crossprod,diagonalMatrix,missing-method (matmult-methods), 115
- crossprod,diagonalMatrix,RsparseMatrix-method (matmult-methods), 115
- crossprod,diagonalMatrix,TsparseMatrix-method (matmult-methods), 115
- crossprod,diagonalMatrix,vector-method (matmult-methods), 115
- crossprod,indMatrix,Matrix-method (matmult-methods), 115
- crossprod,indMatrix,matrix-method (matmult-methods), 115
- crossprod,indMatrix,missing-method (matmult-methods), 115
- crossprod,indMatrix,vector-method (matmult-methods), 115
- crossprod,Matrix,ANY-method (matmult-methods), 115
- crossprod,matrix,CsparseMatrix-method (matmult-methods), 115
- crossprod,matrix,denseMatrix-method (matmult-methods), 115
- crossprod,matrix,diagonalMatrix-method (matmult-methods), 115
- crossprod,Matrix,indMatrix-method (matmult-methods), 115
- crossprod,matrix,indMatrix-method (matmult-methods), 115
- crossprod,Matrix,pMatrix-method (matmult-methods), 115
- crossprod,matrix,pMatrix-method (matmult-methods), 115
- crossprod,matrix,RsparseMatrix-method (matmult-methods), 115
- crossprod,Matrix,sparseVector-method (matmult-methods), 115
- crossprod,matrix,sparseVector-method (matmult-methods), 115
- crossprod,matrix,TsparseMatrix-method (matmult-methods), 115
- crossprod,pMatrix,missing-method (matmult-methods), 115
- crossprod,pMatrix,pMatrix-method (matmult-methods), 115
- crossprod,RsparseMatrix,CsparseMatrix-method (matmult-methods), 115
- crossprod,RsparseMatrix,denseMatrix-method (matmult-methods), 115
- crossprod,RsparseMatrix,diagonalMatrix-method (matmult-methods), 115
- crossprod,RsparseMatrix,matrix-method (matmult-methods), 115
- crossprod,RsparseMatrix,missing-method (matmult-methods), 115
- crossprod,RsparseMatrix,RsparseMatrix-method (matmult-methods), 115
- crossprod,RsparseMatrix,TsparseMatrix-method (matmult-methods), 115
- crossprod,RsparseMatrix,vector-method (matmult-methods), 115
- crossprod,sparseVector,ANY-method (matmult-methods), 115
- crossprod,sparseVector,Matrix-method (matmult-methods), 115
- crossprod,sparseVector,matrix-method (matmult-methods), 115
- crossprod,sparseVector,missing-method (matmult-methods), 115
- crossprod,sparseVector,sparseVector-method (matmult-methods), 115
- crossprod,sparseVector,vector-method (matmult-methods), 115
- crossprod,TsparseMatrix,CsparseMatrix-method (matmult-methods), 115
- crossprod,TsparseMatrix,denseMatrix-method (matmult-methods), 115
- crossprod,TsparseMatrix,diagonalMatrix-method (matmult-methods), 115
- crossprod,TsparseMatrix,matrix-method (matmult-methods), 115
- crossprod,TsparseMatrix,missing-method (matmult-methods), 115
- crossprod,TsparseMatrix,RsparseMatrix-method (matmult-methods), 115
- crossprod,TsparseMatrix,TsparseMatrix-method (matmult-methods), 115
- crossprod,TsparseMatrix,vector-method (matmult-methods), 115
- crossprod,vector,CsparseMatrix-method (matmult-methods), 115

- (matmult-methods), 115
- crossprod,vector,denseMatrix-method
(matmult-methods), 115
- crossprod,vector,diagonalMatrix-method
(matmult-methods), 115
- crossprod,vector,indMatrix-method
(matmult-methods), 115
- crossprod,vector,pMatrix-method
(matmult-methods), 115
- crossprod,vector,RsparseMatrix-method
(matmult-methods), 115
- crossprod,vector,sparseVector-method
(matmult-methods), 115
- crossprod,vector,TsparseMatrix-method
(matmult-methods), 115
- crossprod-methods (matmult-methods), 115
- CsparseMatrix, 12, 14, 41, 54, 55, 60, 64, 68,
77, 78, 85, 86, 90, 104, 109, 116,
120, 128, 139, 157, 166, 173–176,
187, 190, 193, 197, 199, 200, 202,
208, 209, 216
- CsparseMatrix-class, 37
- ct, 197
- ct (ct-methods), 40
- ct,CsparseMatrix-method (ct-methods), 40
- ct,denseMatrix-method (ct-methods), 40
- ct,diagonalMatrix-method (ct-methods),
40
- ct,indMatrix-method (ct-methods), 40
- ct,pMatrix-method (ct-methods), 40
- ct,RsparseMatrix-method (ct-methods), 40
- ct,sparseVector-method (ct-methods), 40
- ct,TsparseMatrix-method (ct-methods), 40
- ct,vector-method (ct-methods), 40
- ct-methods, 40
- data.frame, 74
- ddenseBunchKaufman-class
(denseBunchKaufman-class), 43
- ddenseCholesky-class
(denseCholesky-class), 45
- ddenseLU-class (denseLU-class), 48
- ddenseMatrix-class (dMatrix-class), 64
- ddenseSchur-class (denseSchur-class), 51
- ddiMatrix, 70
- ddiMatrix-class (diagonalMatrix-class),
56
- denseBunchKaufman-class, 43
- denseCholesky-class, 45
- denseLU, 70, 111, 112, 123, 172
- denseLU-class, 48
- denseMatrix, 16, 21, 25, 48, 58, 64, 68, 77,
78, 90, 109, 110, 120, 127, 135, 143,
148, 162, 178, 190, 204, 215
- denseMatrix-class, 50
- denseSchur-class, 51
- det (determinant-methods), 52
- determinant (determinant-methods), 52
- determinant,denseBunchKaufman,logical-method
(determinant-methods), 52
- determinant,denseCholesky,logical-method
(determinant-methods), 52
- determinant,denseLU,logical-method
(determinant-methods), 52
- determinant,denseSchur,logical-method
(determinant-methods), 52
- determinant,diagonalMatrix,logical-method
(determinant-methods), 52
- determinant,generalMatrix,logical-method
(determinant-methods), 52
- determinant,indMatrix,logical-method
(determinant-methods), 52
- determinant,Matrix,missing-method
(determinant-methods), 52
- determinant,MatrixFactorization,missing-method
(determinant-methods), 52
- determinant,pMatrix,logical-method
(determinant-methods), 52
- determinant,sparseCholesky,logical-method
(determinant-methods), 52
- determinant,sparseLU,logical-method
(determinant-methods), 52
- determinant,sparseQR,logical-method
(determinant-methods), 52
- determinant,symmetricMatrix,logical-method
(determinant-methods), 52
- determinant,triangularMatrix,logical-method
(determinant-methods), 52
- determinant-methods, 52
- dgCMatix, 20, 65, 70, 71, 78, 111, 112, 116,
120, 121, 144, 169, 170, 172, 179,
180
- dgCMatix-class (CsparseMatrix-class),
37
- dgeMatrix, 48, 49, 51, 52, 71, 111, 112, 121,
159, 180
- dgeMatrix-class (unpackedMatrix-class),

- 204
- dgRMatrix-class (RsparseMatrix-class), 156
- dgTMatrix, 31, 89
- dgTMatrix-class (TsparseMatrix-class), 200
- diag, 12, 55, 81, 124
- diag (diag-methods), 53
- diag, CsparseMatrix-method (diag-methods), 53
- diag, denseCholesky-method (diag-methods), 53
- diag, denseMatrix-method (diag-methods), 53
- diag, diagonalMatrix-method (diag-methods), 53
- diag, indMatrix-method (diag-methods), 53
- diag, RsparseMatrix-method (diag-methods), 53
- diag, sparseCholesky-method (diag-methods), 53
- diag, TsparseMatrix-method (diag-methods), 53
- diag-methods, 53
- diag<- (diag-methods), 53
- diag<-, CsparseMatrix-method (diag-methods), 53
- diag<-, denseMatrix-method (diag-methods), 53
- diag<-, diagonalMatrix-method (diag-methods), 53
- diag<-, indMatrix-method (diag-methods), 53
- diag<-, RsparseMatrix-method (diag-methods), 53
- diag<-, TsparseMatrix-method (diag-methods), 53
- diag<--methods (diag-methods), 53
- diagN2U (diagU2N), 57
- Diagonal, 14, 54, 56, 118, 175, 187
- diagonalMatrix, 14, 16, 21, 22, 24, 54, 55, 57, 58, 65, 78, 81, 86, 91, 92, 100, 109, 113, 117, 118, 120, 128, 140, 190, 194, 200, 216
- diagonalMatrix-class, 56
- diagU2N, 57
- diff, 153
- diff (diff-methods), 59
- diff, denseMatrix-method (diff-methods), 59
- diff, sparseMatrix-method (diff-methods), 59
- diff, sparseVector-method (diff-methods), 59
- diff-methods, 59
- dim, 60, 62, 108, 142, 154
- dim, Matrix-method (dim-methods), 60
- dim, MatrixFactorization-method (dim-methods), 60
- dim-methods, 60
- dim<-, denseMatrix, NULL-method (dim-methods), 60
- dim<-, denseMatrix, numeric-method (dim-methods), 60
- dim<-, sparseMatrix, NULL-method (dim-methods), 60
- dim<-, sparseMatrix, numeric-method (dim-methods), 60
- dim<-, sparseVector, numeric-method (dim-methods), 60
- dim<--methods (dim-methods), 60
- dimnames, 34, 60, 61, 74, 84, 97, 98, 104, 108, 117, 118, 174
- dimnames, Matrix-method (dimnames-methods), 61
- dimnames, MatrixFactorization-method (dimnames-methods), 61
- dimnames, symmetricMatrix-method (dimnames-methods), 61
- dimnames-methods, 61
- dimnames<-, generalMatrix, list-method (dimnames-methods), 61
- dimnames<-, generalMatrix, NULL-method (dimnames-methods), 61
- dimnames<-, Matrix, list-method (dimnames-methods), 61
- dimnames<-, Matrix, NULL-method (dimnames-methods), 61
- dimnames<-, MatrixFactorization, list-method (dimnames-methods), 61
- dimnames<-, MatrixFactorization, NULL-method (dimnames-methods), 61
- dimnames<-, symmetricMatrix, list-method (dimnames-methods), 61
- dimnames<-, symmetricMatrix, NULL-method (dimnames-methods), 61

- dimnames<--methods (dimnames-methods), 61
- dimScale, 63
- dinMatrix-class (indexMatrix-class), 91
- dMatrix, 17, 38, 56, 63, 72, 85, 86, 91, 109, 120, 128, 136, 139, 156, 157, 190, 193, 194, 199–201, 204, 216
- dMatrix-class, 64
- dmperm, 65
- double, 12, 13, 55, 64, 118, 154, 166, 174
- dpCMatrix-class (CsparseMatrix-class), 37
- dpoMatrix, 27, 46, 47, 87, 124, 125
- dpoMatrix-class (unpackedMatrix-class), 204
- dppMatrix, 27, 46, 47
- dppMatrix-class (packedMatrix-class), 135
- dpRMatrix-class (RsparseMatrix-class), 156
- dpTMatrix-class (TsparseMatrix-class), 200
- drop, 67, 189, 190
- drop, abIndex-method (abIndex-class), 5
- drop, Matrix-method (drop-methods), 67
- drop, sparseVector-method (drop-methods), 67
- drop-methods, 67
- drop0, 17, 67, 68, 130, 182, 215
- dsCMatrix, 27, 70, 78, 169, 170, 190, 210, 212, 213
- dsCMatrix-class (CsparseMatrix-class), 37
- dsimplicialCholesky-class (sparseCholesky-class), 167
- dsparseCholesky-class (sparseCholesky-class), 167
- dsparseLU-class (sparseLU-class), 171
- dsparseMatrix, 54
- dsparseMatrix-class (dMatrix-class), 64
- dsparseQR-class (sparseQR-class), 178
- dsparseVector-class (sparseVector-class), 183
- dspMatrix, 19, 43, 46
- dspMatrix-class (packedMatrix-class), 135
- dsRMatrix-class (RsparseMatrix-class), 156
- dsTMatrix, 210, 213
- dsTMatrix-class (TsparseMatrix-class), 200
- dsupernodalCholesky-class (sparseCholesky-class), 167
- dsyMatrix, 19, 43, 44, 46
- dsyMatrix-class (unpackedMatrix-class), 204
- dtCMatrix, 58, 65, 70, 71, 169, 172, 180
- dtCMatrix-class (CsparseMatrix-class), 37
- dtpMatrix, 44, 46, 70
- dtpMatrix-class (packedMatrix-class), 135
- dtrMatrix, 44, 46, 70, 71
- dtrMatrix-class (RsparseMatrix-class), 156
- dtrMatrix-class (unpackedMatrix-class), 204
- dtTMatrix-class (TsparseMatrix-class), 200
- eigen, 20, 120
- error, 111
- expand (expand-methods), 69
- expand, denseLU-method (expand-methods), 69
- expand, sparseCholesky-method (expand-methods), 69
- expand, sparseLU-method (expand-methods), 69
- expand-methods, 69
- expand1, 19, 27, 44, 47, 49, 52, 112, 123, 144, 159, 170, 172, 180
- expand1 (expand-methods), 69
- expand1, denseBunchKaufman-method (expand-methods), 69
- expand1, denseCholesky-method (expand-methods), 69
- expand1, denseLU-method (expand-methods), 69
- expand1, denseSchur-method (expand-methods), 69
- expand1, simplicialCholesky-method (expand-methods), 69
- expand1, sparseLU-method (expand-methods), 69
- expand1, sparseQR-method (expand-methods), 69

- expand1, supernodalCholesky-method
(expand-methods), 69
- expand1-methods (expand-methods), 69
- expand2, 19, 27, 44, 47, 49, 52, 75, 112, 123,
144, 159, 170, 172, 180
- expand2 (expand-methods), 69
- expand2, denseBunchKaufman-method
(expand-methods), 69
- expand2, denseCholesky-method
(expand-methods), 69
- expand2, denseLU-method
(expand-methods), 69
- expand2, denseSchur-method
(expand-methods), 69
- expand2, simplicialCholesky-method
(expand-methods), 69
- expand2, sparseLU-method
(expand-methods), 69
- expand2, sparseQR-method
(expand-methods), 69
- expand2, supernodalCholesky-method
(expand-methods), 69
- expand2-methods (expand-methods), 69
- expm (expm-methods), 72
- expm, denseMatrix-method (expm-methods),
72
- expm, diagonalMatrix-method
(expm-methods), 72
- expm, matrix-method (expm-methods), 72
- expm, sparseMatrix-method
(expm-methods), 72
- expm-methods, 72
- extends, 117
- externalFormats, 73
- Extract, 188
- fac2Sparse (sparse.model.matrix), 165
- fac2sparse, 166
- fac2sparse (sparse.model.matrix), 165
- facmul (facmul-methods), 75
- facmul-methods, 75
- factor, 165, 166
- FALSE, 97
- fastMisc, 76
- finite, 19, 21, 26, 111, 143, 158, 161
- forceCanonical, 99
- forceCanonical
(forceCanonical-methods), 80
- forceCanonical, CsparseMatrix-method
(forceCanonical-methods), 80
- forceCanonical, denseMatrix-method
(forceCanonical-methods), 80
- forceCanonical, diagonalMatrix-method
(forceCanonical-methods), 80
- forceCanonical, indMatrix-method
(forceCanonical-methods), 80
- forceCanonical, RsparseMatrix-method
(forceCanonical-methods), 80
- forceCanonical, sparseVector-method
(forceCanonical-methods), 80
- forceCanonical, TsparseMatrix-method
(forceCanonical-methods), 80
- forceCanonical-methods, 80
- forceDiagonal, 56
- forceDiagonal (forceDiagonal-methods),
81
- forceDiagonal, CsparseMatrix-method
(forceDiagonal-methods), 81
- forceDiagonal, denseMatrix-method
(forceDiagonal-methods), 81
- forceDiagonal, diagonalMatrix-method
(forceDiagonal-methods), 81
- forceDiagonal, indMatrix-method
(forceDiagonal-methods), 81
- forceDiagonal, matrix-method
(forceDiagonal-methods), 81
- forceDiagonal, RsparseMatrix-method
(forceDiagonal-methods), 81
- forceDiagonal, TsparseMatrix-method
(forceDiagonal-methods), 81
- forceDiagonal-methods, 81
- forceSymmetric, 102, 160, 194, 195
- forceSymmetric
(forceSymmetric-methods), 81
- forceSymmetric, CsparseMatrix-method
(forceSymmetric-methods), 81
- forceSymmetric, denseMatrix-method
(forceSymmetric-methods), 81
- forceSymmetric, diagonalMatrix-method
(forceSymmetric-methods), 81
- forceSymmetric, indMatrix-method
(forceSymmetric-methods), 81
- forceSymmetric, matrix-method
(forceSymmetric-methods), 81
- forceSymmetric, RsparseMatrix-method
(forceSymmetric-methods), 81

- forceSymmetric, TsparseMatrix-method
(forceSymmetric-methods), 81
- forceSymmetric-methods, 81
- forceTriangular, 200
- forceTriangular
(forceTriangular-methods), 83
- forceTriangular, CsparseMatrix-method
(forceTriangular-methods), 83
- forceTriangular, denseMatrix-method
(forceTriangular-methods), 83
- forceTriangular, diagonalMatrix-method
(forceTriangular-methods), 83
- forceTriangular, indMatrix-method
(forceTriangular-methods), 83
- forceTriangular, matrix-method
(forceTriangular-methods), 83
- forceTriangular, RsparseMatrix-method
(forceTriangular-methods), 83
- forceTriangular, TsparseMatrix-method
(forceTriangular-methods), 83
- forceTriangular-methods, 83
- format, 83, 84, 140
- format, sparseMatrix-method
(show-methods), 159
- formatSparseM, 83, 142
- formatSpMatrix, 83, 84
- formatSpMatrix (printSpMatrix), 140
- function, 104, 155
- generalMatrix, 38, 54, 56, 64, 77, 78, 90, 92,
109, 120, 128, 140, 156, 175, 186,
190, 194, 200, 201, 204, 216
- generalMatrix-class, 85
- get.gpar, 88
- getClassDef, 141
- getGroup, 133
- getGroupMembers, 133
- getOption, 141
- graph, 31
- graph2T (coerce-methods-graph), 31
- grey, 88
- grid.raster, 88
- grid.rect, 88
- head, 184
- head (head-methods), 86
- head, Matrix-method (head-methods), 86
- head, sparseVector-method
(head-methods), 86
- head-methods, 86
- Hilbert, 87
- idenseMatrix-class (iMatrix-class), 90
- identical, 134
- identity, 170, 208
- idiMatrix-class (diagonalMatrix-class),
56
- igCMatrix-class (CsparseMatrix-class),
37
- igeMatrix-class (unpackedMatrix-class),
204
- igRMatrix-class (RsparseMatrix-class),
156
- igTMatrix-class (TsparseMatrix-class),
200
- iinMatrix-class (indexMatrix-class), 91
- image, 87
- image (image-methods), 87
- image, ANY-method (image-methods), 87
- image, dgTMatrix-method (image-methods),
87
- image, Matrix-method (image-methods), 87
- image, sparseCholesky-method
(image-methods), 87
- image-methods, 87
- iMatrix, 12, 13, 38, 55, 56, 65, 85, 86, 109,
118, 120, 128, 136, 156, 157, 175,
193, 194, 199–201, 204, 216
- iMatrix-class, 90
- implicitGeneric, 198, 214
- index, 188
- index-class, 91
- indexMatrix, 56, 86, 120, 128, 140, 190, 194,
200
- indexMatrix-class, 91
- indMatrix, 16, 68, 78
- indMatrix-class (indexMatrix-class), 91
- Inf, 96, 97
- initialize (initialize-methods), 94
- initialize, Matrix-method
(initialize-methods), 94
- initialize, MatrixFactorization-method
(initialize-methods), 94
- initialize, sparseVector-method
(initialize-methods), 94
- initialize-methods, 94
- integer, 12, 13, 41, 55, 90, 118, 129, 174, 196
- InternalGenerics, 120

- invertPerm, [92](#), [94](#)
- invisible, [142](#)
- invPerm (invertPerm), [94](#)
- is, [88](#)
- is.finite, [96](#)
- is.finite (is.na-methods), [96](#)
- is.finite, abIndex-method
(abIndex-class), [5](#)
- is.finite, CsparseMatrix-method
(is.na-methods), [96](#)
- is.finite, denseMatrix-method
(is.na-methods), [96](#)
- is.finite, diagonalMatrix-method
(is.na-methods), [96](#)
- is.finite, indMatrix-method
(is.na-methods), [96](#)
- is.finite, RsparseMatrix-method
(is.na-methods), [96](#)
- is.finite, sparseVector-method
(is.na-methods), [96](#)
- is.finite, TsparseMatrix-method
(is.na-methods), [96](#)
- is.finite-methods (is.na-methods), [96](#)
- is.infinite, [96](#)
- is.infinite (is.na-methods), [96](#)
- is.infinite, abIndex-method
(abIndex-class), [5](#)
- is.infinite, CsparseMatrix-method
(is.na-methods), [96](#)
- is.infinite, denseMatrix-method
(is.na-methods), [96](#)
- is.infinite, diagonalMatrix-method
(is.na-methods), [96](#)
- is.infinite, indMatrix-method
(is.na-methods), [96](#)
- is.infinite, RsparseMatrix-method
(is.na-methods), [96](#)
- is.infinite, sparseVector-method
(is.na-methods), [96](#)
- is.infinite, TsparseMatrix-method
(is.na-methods), [96](#)
- is.infinite-methods (is.na-methods), [96](#)
- is.na, [96](#)
- is.na (is.na-methods), [96](#)
- is.na, abIndex-method (abIndex-class), [5](#)
- is.na, CsparseMatrix-method
(is.na-methods), [96](#)
- is.na, denseMatrix-method
(is.na-methods), [96](#)
- is.na, diagonalMatrix-method
(is.na-methods), [96](#)
- is.na, indMatrix-method (is.na-methods),
[96](#)
- is.na, RsparseMatrix-method
(is.na-methods), [96](#)
- is.na, sparseVector-method
(is.na-methods), [96](#)
- is.na, TsparseMatrix-method
(is.na-methods), [96](#)
- is.na-methods, [96](#)
- is.nan, [96](#)
- is.nan (is.na-methods), [96](#)
- is.nan, CsparseMatrix-method
(is.na-methods), [96](#)
- is.nan, denseMatrix-method
(is.na-methods), [96](#)
- is.nan, diagonalMatrix-method
(is.na-methods), [96](#)
- is.nan, indMatrix-method
(is.na-methods), [96](#)
- is.nan, RsparseMatrix-method
(is.na-methods), [96](#)
- is.nan, sparseVector-method
(is.na-methods), [96](#)
- is.nan, TsparseMatrix-method
(is.na-methods), [96](#)
- is.nan-methods (is.na-methods), [96](#)
- is.null, [97](#), [98](#)
- is.null.DN, [97](#)
- isCanonical, [80](#)
- isCanonical (isCanonical-methods), [98](#)
- isCanonical, CsparseMatrix-method
(isCanonical-methods), [98](#)
- isCanonical, denseMatrix-method
(isCanonical-methods), [98](#)
- isCanonical, diagonalMatrix-method
(isCanonical-methods), [98](#)
- isCanonical, indMatrix-method
(isCanonical-methods), [98](#)
- isCanonical, RsparseMatrix-method
(isCanonical-methods), [98](#)
- isCanonical, sparseVector-method
(isCanonical-methods), [98](#)
- isCanonical, TsparseMatrix-method
(isCanonical-methods), [98](#)
- isCanonical-methods, [98](#)

- isCMatrix-class (CsparseMatrix-class), 37
- isDiagonal, 56, 81
- isDiagonal (isDiagonal-methods), 100
- isDiagonal, CsparseMatrix-method (isDiagonal-methods), 100
- isDiagonal, denseMatrix-method (isDiagonal-methods), 100
- isDiagonal, diagonalMatrix-method (isDiagonal-methods), 100
- isDiagonal, indMatrix-method (isDiagonal-methods), 100
- isDiagonal, matrix-method (isDiagonal-methods), 100
- isDiagonal, RsparseMatrix-method (isDiagonal-methods), 100
- isDiagonal, table-method (isDiagonal-methods), 100
- isDiagonal, TsparseMatrix-method (isDiagonal-methods), 100
- isDiagonal-methods, 100
- isLDL, 163
- isLDL (sparseCholesky-class), 167
- isparsedMatrix-class (iMatrix-class), 90
- isparsedVector-class (sparseVector-class), 183
- isPerm, 92
- isPerm (invertPerm), 94
- ispMatrix-class (packedMatrix-class), 135
- isRMatrix-class (RsparseMatrix-class), 156
- isSymmetric, 77, 82, 100, 102, 103, 124, 134, 160, 194, 195
- isSymmetric (isSymmetric-methods), 101
- isSymmetric, CsparseMatrix-method (isSymmetric-methods), 101
- isSymmetric, denseMatrix-method (isSymmetric-methods), 101
- isSymmetric, diagonalMatrix-method (isSymmetric-methods), 101
- isSymmetric, indMatrix-method (isSymmetric-methods), 101
- isSymmetric, RsparseMatrix-method (isSymmetric-methods), 101
- isSymmetric, TsparseMatrix-method (isSymmetric-methods), 101
- isSymmetric-methods, 101
- isSymmetric.matrix, 101, 102
- isTMatrix-class (TsparseMatrix-class), 200
- isTriangular, 134, 200
- isTriangular (isTriangular-methods), 102
- isTriangular, CsparseMatrix-method (isTriangular-methods), 102
- isTriangular, denseMatrix-method (isTriangular-methods), 102
- isTriangular, diagonalMatrix-method (isTriangular-methods), 102
- isTriangular, indMatrix-method (isTriangular-methods), 102
- isTriangular, matrix-method (isTriangular-methods), 102
- isTriangular, RsparseMatrix-method (isTriangular-methods), 102
- isTriangular, TsparseMatrix-method (isTriangular-methods), 102
- isTriangular-methods, 102
- isUniqueT (asUniqueT), 8
- isyMatrix-class (unpackedMatrix-class), 204
- itCMatrix-class (CsparseMatrix-class), 37
- itpMatrix-class (packedMatrix-class), 135
- itRMatrix-class (RsparseMatrix-class), 156
- itrMatrix-class (unpackedMatrix-class), 204
- itTMatrix-class (TsparseMatrix-class), 200
- kappa, 150
- kappa (rcond-methods), 149
- kappa, Matrix-method (rcond-methods), 149
- kappa-methods (rcond-methods), 149
- KhatriRao, 103
- KNex, 105
- kronecker, 14, 104, 106
- kronecker (kronecker-methods), 106
- kronecker, CsparseMatrix, CsparseMatrix-method (kronecker-methods), 106
- kronecker, CsparseMatrix, diagonalMatrix-method (kronecker-methods), 106
- kronecker, CsparseMatrix, Matrix-method (kronecker-methods), 106

- kronecker, denseMatrix, denseMatrix-method
(kronecker-methods), 106
- kronecker, denseMatrix, Matrix-method
(kronecker-methods), 106
- kronecker, diagonalMatrix, CsparseMatrix-method
(kronecker-methods), 106
- kronecker, diagonalMatrix, diagonalMatrix-method
(kronecker-methods), 106
- kronecker, diagonalMatrix, indMatrix-method
(kronecker-methods), 106
- kronecker, diagonalMatrix, Matrix-method
(kronecker-methods), 106
- kronecker, diagonalMatrix, RsparseMatrix-method
(kronecker-methods), 106
- kronecker, diagonalMatrix, TsparseMatrix-method
(kronecker-methods), 106
- kronecker, indMatrix, diagonalMatrix-method
(kronecker-methods), 106
- kronecker, indMatrix, indMatrix-method
(kronecker-methods), 106
- kronecker, indMatrix, Matrix-method
(kronecker-methods), 106
- kronecker, Matrix, matrix-method
(kronecker-methods), 106
- kronecker, matrix, Matrix-method
(kronecker-methods), 106
- kronecker, Matrix, vector-method
(kronecker-methods), 106
- kronecker, RsparseMatrix, diagonalMatrix-method
(kronecker-methods), 106
- kronecker, RsparseMatrix, Matrix-method
(kronecker-methods), 106
- kronecker, RsparseMatrix, RsparseMatrix-method
(kronecker-methods), 106
- kronecker, TsparseMatrix, diagonalMatrix-method
(kronecker-methods), 106
- kronecker, TsparseMatrix, Matrix-method
(kronecker-methods), 106
- kronecker, TsparseMatrix, TsparseMatrix-method
(kronecker-methods), 106
- kronecker, vector, Matrix-method
(kronecker-methods), 106
- kronecker-methods, 106
- ldenseMatrix-class (lMatrix-class), 109
- ldiMatrix, 17
- ldiMatrix-class (diagonalMatrix-class), 56
- length, 54, 60, 62, 65, 107, 130, 183
- length, abIndex-method (abIndex-class), 5
- length, Matrix-method (length-methods), 107
- length, MatrixFactorization-method
(length-methods), 107
- length, sparseVector-method
(length-methods), 107
- length-methods, 107
- length<-, denseMatrix, numeric-method
(length-methods), 107
- length<-, sparseMatrix, numeric-method
(length-methods), 107
- length<-, sparseVector, numeric-method
(length-methods), 107
- length<--methods (length-methods), 107
- levelplot, 87–89
- lgCMatrix-class (CsparseMatrix-class), 37
- lgeMatrix-class (unpackedMatrix-class), 204
- lgRMatrix-class (RsparseMatrix-class), 156
- lgTMatrix-class (TsparseMatrix-class), 200
- linMatrix-class (indexMatrix-class), 91
- list, 13, 36, 65, 66, 88, 113, 154, 166
- lMatrix, 17, 38, 56, 65, 85, 86, 91, 115, 118, 120, 128, 136, 156, 157, 193, 194, 199–201, 204, 216
- lMatrix-class, 109
- log (Math-methods), 114
- log, denseMatrix-method (Math-methods), 114
- log, diagonalMatrix-method
(Math-methods), 114
- log, indMatrix-method (Math-methods), 114
- log, sparseMatrix-method (Math-methods), 114
- log, sparseVector-method (Math-methods), 114
- log-methods (Math-methods), 114
- Logic (Ops-methods), 133
- Logic, ANY, Matrix-method (Ops-methods), 133
- Logic, complex, dMatrix-method
(Ops-methods), 133
- Logic, complex, iMatrix-method
(Ops-methods), 133

- Logic,complex,lMatrix-method
(Ops-methods), [133](#)
- Logic,complex,nMatrix-method
(Ops-methods), [133](#)
- Logic,complex,zMatrix-method
(Ops-methods), [133](#)
- Logic,CsparseMatrix,CsparseMatrix-method
(Ops-methods), [133](#)
- Logic,dMatrix,complex-method
(Ops-methods), [133](#)
- Logic,dMatrix,logical-method
(Ops-methods), [133](#)
- Logic,dMatrix,numeric-method
(Ops-methods), [133](#)
- Logic,dMatrix,sparseVector-method
(Ops-methods), [133](#)
- Logic,iMatrix,complex-method
(Ops-methods), [133](#)
- Logic,iMatrix,logical-method
(Ops-methods), [133](#)
- Logic,iMatrix,numeric-method
(Ops-methods), [133](#)
- Logic,iMatrix,sparseVector-method
(Ops-methods), [133](#)
- Logic,ldenseMatrix,lsparseMatrix-method
(Ops-methods), [133](#)
- Logic,lgCMatrix,lgCMatrix-method
(Ops-methods), [133](#)
- Logic,lgeMatrix,lgeMatrix-method
(Ops-methods), [133](#)
- Logic,lgTMatrix,lgTMatrix-method
(Ops-methods), [133](#)
- Logic,lMatrix,complex-method
(Ops-methods), [133](#)
- Logic,lMatrix,logical-method
(Ops-methods), [133](#)
- Logic,lMatrix,numeric-method
(Ops-methods), [133](#)
- Logic,lMatrix,sparseVector-method
(Ops-methods), [133](#)
- Logic,logical,dMatrix-method
(Ops-methods), [133](#)
- Logic,logical,iMatrix-method
(Ops-methods), [133](#)
- Logic,logical,lMatrix-method
(Ops-methods), [133](#)
- Logic,logical,nMatrix-method
(Ops-methods), [133](#)
- Logic,logical,zMatrix-method
(Ops-methods), [133](#)
- Logic,lsCMatrix,lsCMatrix-method
(Ops-methods), [133](#)
- Logic,lsparseMatrix,ldenseMatrix-method
(Ops-methods), [133](#)
- Logic,lsparseMatrix,lsparseMatrix-method
(Ops-methods), [133](#)
- Logic,lsparseVector,lsparseVector-method
(Ops-methods), [133](#)
- Logic,ltCMatrix,ltCMatrix-method
(Ops-methods), [133](#)
- Logic,Matrix,ANY-method (Ops-methods),
[133](#)
- Logic,Matrix,nMatrix-method
(Ops-methods), [133](#)
- Logic,ngeMatrix,ngeMatrix-method
(Ops-methods), [133](#)
- Logic,nMatrix,complex-method
(Ops-methods), [133](#)
- Logic,nMatrix,logical-method
(Ops-methods), [133](#)
- Logic,nMatrix,Matrix-method
(Ops-methods), [133](#)
- Logic,nMatrix,nMatrix-method
(Ops-methods), [133](#)
- Logic,nMatrix,numeric-method
(Ops-methods), [133](#)
- Logic,nMatrix,sparseVector-method
(Ops-methods), [133](#)
- Logic,numeric,dMatrix-method
(Ops-methods), [133](#)
- Logic,numeric,iMatrix-method
(Ops-methods), [133](#)
- Logic,numeric,lMatrix-method
(Ops-methods), [133](#)
- Logic,numeric,nMatrix-method
(Ops-methods), [133](#)
- Logic,numeric,zMatrix-method
(Ops-methods), [133](#)
- Logic,sparseVector,dMatrix-method
(Ops-methods), [133](#)
- Logic,sparseVector,iMatrix-method
(Ops-methods), [133](#)
- Logic,sparseVector,lMatrix-method
(Ops-methods), [133](#)
- Logic,sparseVector,nMatrix-method
(Ops-methods), [133](#)

- Logic,sparseVector,sparseVector-method
(Ops-methods), [133](#)
- Logic,sparseVector,zMatrix-method
(Ops-methods), [133](#)
- Logic,triangularMatrix,diagonalMatrix-method
(Ops-methods), [133](#)
- Logic,zMatrix,complex-method
(Ops-methods), [133](#)
- Logic,zMatrix,logical-method
(Ops-methods), [133](#)
- Logic,zMatrix,numeric-method
(Ops-methods), [133](#)
- Logic,zMatrix,sparseVector-method
(Ops-methods), [133](#)
- Logic-methods (Ops-methods), [133](#)
- logical, [54](#), [91](#), [97](#), [100](#), [103](#), [109](#), [113](#), [115](#),
[118](#), [121](#), [127](#), [129](#), [133](#), [147](#), [184](#),
[188](#)
- lsCMatrix-class (CsparseMatrix-class),
[37](#)
- lsparseMatrix, [54](#)
- lsparseMatrix-class (lMatrix-class), [109](#)
- lsparseVector-class
(sparseVector-class), [183](#)
- lspMatrix-class (packedMatrix-class),
[135](#)
- lsRMatrix-class (RsparseMatrix-class),
[156](#)
- lsTMatrix-class (TsparseMatrix-class),
[200](#)
- lsyMatrix-class (unpackedMatrix-class),
[204](#)
- ltCMatrix-class (CsparseMatrix-class),
[37](#)
- ltpMatrix-class (packedMatrix-class),
[135](#)
- ltRMatrix-class (RsparseMatrix-class),
[156](#)
- ltrMatrix-class (unpackedMatrix-class),
[204](#)
- ltTMatrix-class (TsparseMatrix-class),
[200](#)
- LU, [48](#), [111](#), [172](#)
- lu, [19](#), [27](#), [36](#), [48](#), [49](#), [71](#), [85](#), [86](#), [123](#), [144](#),
[159](#), [163](#), [172](#), [193](#)
- lu (lu-methods), [110](#)
- lu,CsparseMatrix-method (lu-methods),
[110](#)
- lu,denseMatrix-method (lu-methods), [110](#)
- lu,diagonalMatrix-method (lu-methods),
[110](#)
- lu,indMatrix-method (lu-methods), [110](#)
- lu,matrix-method (lu-methods), [110](#)
- lu,RsparseMatrix-method (lu-methods),
[110](#)
- lu,TsparseMatrix-method (lu-methods),
[110](#)
- LU-class (MatrixFactorization-class),
[122](#)
- lu-methods, [110](#)
- mat2triplet, [112](#)
- Math, [192](#)
- Math (Math-methods), [114](#)
- Math,denseMatrix-method (Math-methods),
[114](#)
- Math,diagonalMatrix-method
(Math-methods), [114](#)
- Math,indMatrix-method (Math-methods),
[114](#)
- Math,sparseMatrix-method
(Math-methods), [114](#)
- Math,sparseVector-method
(Math-methods), [114](#)
- Math-methods, [114](#)
- Math2, [215](#)
- Math2 (Math2-methods), [114](#)
- Math2,denseMatrix-method
(Math2-methods), [114](#)
- Math2,diagonalMatrix-method
(Math2-methods), [114](#)
- Math2,indMatrix-method (Math2-methods),
[114](#)
- Math2,sparseMatrix-method
(Math2-methods), [114](#)
- Math2,sparseVector-method
(Math2-methods), [114](#)
- Math2-methods, [114](#)
- matmult-methods, [115](#)
- Matrix, [8](#), [9](#), [12](#), [16](#), [19](#), [21](#), [24](#), [26](#), [27](#), [34](#), [38](#),
[41](#), [42](#), [50](#), [51](#), [54–56](#), [60–64](#), [67](#), [68](#),
[70](#), [73](#), [77](#), [78](#), [81–83](#), [85](#), [90–92](#), [96](#),
[97](#), [100–102](#), [106](#), [108](#), [109](#), [111](#),
[115](#), [116](#), [117](#), [117](#), [118](#), [121](#), [122](#),
[124](#), [127](#), [129](#), [130](#), [133](#), [135](#), [139](#),
[143](#), [152](#), [156](#), [158](#), [160–162](#), [164](#),
[175](#), [177](#), [178](#), [187](#), [189](#), [190](#), [192](#),

- [193, 195–197, 199, 201, 204, 209, 212, 214, 215](#)
- [matrix, 11, 17, 84, 97, 98, 100, 102, 113, 117–119, 124, 130, 133, 188](#)
- [Matrix-class, 119](#)
- [Matrix-package, 5](#)
- [matrix.csr, 33](#)
- [Matrix.Version \(Matrix-package\), 5](#)
- [MatrixClass, 121](#)
- [MatrixFactorization, 43, 45, 46, 48, 51, 60–62, 69, 71, 75, 85, 108, 161, 163, 168, 169, 172, 179, 193](#)
- [MatrixFactorization-class, 122](#)
- [mean \(mean-methods\), 123](#)
- [mean, denseMatrix-method \(mean-methods\), 123](#)
- [mean, sparseMatrix-method \(mean-methods\), 123](#)
- [mean, sparseVector-method \(mean-methods\), 123](#)
- [mean-methods, 123](#)
- [method, 101](#)
- [Methods, 133](#)
- [model.frame, 165](#)
- [model.Matrix, 166](#)
- [model.matrix, 165, 166](#)
- [modelMatrix, 166](#)
- [NA, 96, 97, 100, 103, 106, 108, 129, 164, 184, 188, 189, 192, 214](#)
- [NA_integer_, 148](#)
- [names, 34, 54](#)
- [NaN, 96, 97, 148, 192, 214](#)
- [ndenseMatrix-class \(nMatrix-class\), 127](#)
- [ndiMatrix-class \(diagonalMatrix-class\), 56](#)
- [nearcor, 125](#)
- [nearPD, 124](#)
- [new, 169, 173](#)
- [ngCMatrix-class \(CsparseMatrix-class\), 37](#)
- [ngeMatrix-class \(unpackedMatrix-class\), 204](#)
- [ngRMatrix-class \(RsparseMatrix-class\), 156](#)
- [ngTMatrix, 31](#)
- [ngTMatrix-class \(TsparseMatrix-class\), 200](#)
- [ninMatrix-class \(indexMatrix-class\), 91](#)
- [nMatrix, 12, 17, 38, 56, 65, 77, 85, 86, 91, 92, 96, 109, 113, 115, 120, 136, 156, 157, 193, 194, 199–201, 204, 216](#)
- [nMatrix-class, 127](#)
- [nnzero, 68, 155](#)
- [nnzero \(nnzero-methods\), 129](#)
- [nnzero, ANY-method \(nnzero-methods\), 129](#)
- [nnzero, denseMatrix-method \(nnzero-methods\), 129](#)
- [nnzero, diagonalMatrix-method \(nnzero-methods\), 129](#)
- [nnzero, indMatrix-method \(nnzero-methods\), 129](#)
- [nnzero, sparseCholesky-method \(nnzero-methods\), 129](#)
- [nnzero, sparseMatrix-method \(nnzero-methods\), 129](#)
- [nnzero, sparseVector-method \(nnzero-methods\), 129](#)
- [nnzero-methods, 129](#)
- [norm, 35, 36, 125, 131, 150](#)
- [norm \(norm-methods\), 130](#)
- [norm, ANY, missing-method \(norm-methods\), 130](#)
- [norm, denseMatrix, character-method \(norm-methods\), 130](#)
- [norm, diagonalMatrix, character-method \(norm-methods\), 130](#)
- [norm, indMatrix, character-method \(norm-methods\), 130](#)
- [norm, pMatrix, character-method \(norm-methods\), 130](#)
- [norm, sparseMatrix, character-method \(norm-methods\), 130](#)
- [norm-methods, 130](#)
- [not-methods, 132](#)
- [nsCMatrix-class \(CsparseMatrix-class\), 37](#)
- [nsimplicialCholesky-class \(sparseCholesky-class\), 167](#)
- [nsparseCholesky-class \(sparseCholesky-class\), 167](#)
- [nsparseMatrix, 17, 54, 113, 115, 154, 155, 174](#)
- [nsparseMatrix-class \(nMatrix-class\), 127](#)
- [nsparseVector, 77, 96](#)
- [nsparseVector-class \(sparseVector-class\), 183](#)

- nspMatrix-class (packedMatrix-class), [135](#)
- nsRMatrix-class (RsparseMatrix-class), [156](#)
- nsTMatrix, [31](#)
- nsTMatrix-class (TsparseMatrix-class), [200](#)
- nsupernodalCholesky-class
(sparseCholesky-class), [167](#)
- nsyMatrix-class (unpackedMatrix-class), [204](#)
- ntCMatrix-class (CsparseMatrix-class), [37](#)
- ntpMatrix-class (packedMatrix-class), [135](#)
- ntRMatrix-class (RsparseMatrix-class), [156](#)
- ntrMatrix-class (unpackedMatrix-class), [204](#)
- ntTMatrix-class (TsparseMatrix-class), [200](#)
- NULL, [60](#), [61](#), [73](#), [81](#), [84](#), [97](#), [117](#), [141](#), [155](#), [174](#), [192](#)
- numeric, [5](#), [6](#), [33](#), [34](#), [91](#), [106](#), [129](#), [133](#)

- onenormest, [131](#)
- onenormest (condest), [35](#)
- Ops, [6](#), [184](#), [192](#)
- Ops (Ops-methods), [133](#)
- Ops, abIndex, abIndex-method
(abIndex-class), [5](#)
- Ops, abIndex, ANY-method (abIndex-class), [5](#)
- Ops, ANY, abIndex-method (abIndex-class), [5](#)
- Ops, ANY, ddiMatrix-method (Ops-methods), [133](#)
- Ops, ANY, diagonalMatrix-method
(Ops-methods), [133](#)
- Ops, ANY, iMatrix-method (Ops-methods), [133](#)
- Ops, ANY, ldiMatrix-method (Ops-methods), [133](#)
- Ops, ANY, Matrix-method (Ops-methods), [133](#)
- Ops, ANY, sparseVector-method
(Ops-methods), [133](#)
- Ops, ANY, zMatrix-method (Ops-methods), [133](#)

- Ops, ddiMatrix, ANY-method (Ops-methods), [133](#)
- Ops, ddiMatrix, ddiMatrix-method
(Ops-methods), [133](#)
- Ops, ddiMatrix, dMatrix-method
(Ops-methods), [133](#)
- Ops, ddiMatrix, idiMatrix-method
(Ops-methods), [133](#)
- Ops, ddiMatrix, ldiMatrix-method
(Ops-methods), [133](#)
- Ops, ddiMatrix, logical-method
(Ops-methods), [133](#)
- Ops, ddiMatrix, Matrix-method
(Ops-methods), [133](#)
- Ops, ddiMatrix, ndiMatrix-method
(Ops-methods), [133](#)
- Ops, ddiMatrix, numeric-method
(Ops-methods), [133](#)
- Ops, ddiMatrix, sparseMatrix-method
(Ops-methods), [133](#)
- Ops, ddiMatrix, zdiMatrix-method
(Ops-methods), [133](#)
- Ops, diagonalMatrix, ANY-method
(Ops-methods), [133](#)
- Ops, diagonalMatrix, dMatrix-method
(Ops-methods), [133](#)
- Ops, diagonalMatrix, Matrix-method
(Ops-methods), [133](#)
- Ops, diagonalMatrix, sparseMatrix-method
(Ops-methods), [133](#)
- Ops, diagonalMatrix, triangularMatrix-method
(Ops-methods), [133](#)
- Ops, dMatrix, ddiMatrix-method
(Ops-methods), [133](#)
- Ops, dMatrix, diagonalMatrix-method
(Ops-methods), [133](#)
- Ops, dMatrix, dMatrix-method
(Ops-methods), [133](#)
- Ops, dMatrix, ldiMatrix-method
(Ops-methods), [133](#)
- Ops, dMatrix, lMatrix-method
(Ops-methods), [133](#)
- Ops, dMatrix, nMatrix-method
(Ops-methods), [133](#)
- Ops, dpoMatrix, logical-method
(Ops-methods), [133](#)
- Ops, dpoMatrix, numeric-method
(Ops-methods), [133](#)

- Ops, dppMatrix, logical-method
(Ops-methods), [133](#)
- Ops, dppMatrix, numeric-method
(Ops-methods), [133](#)
- Ops, dsparseMatrix, nsparseMatrix-method
(Ops-methods), [133](#)
- Ops, idiMatrix, ddiMatrix-method
(Ops-methods), [133](#)
- Ops, idiMatrix, idiMatrix-method
(Ops-methods), [133](#)
- Ops, idiMatrix, ldiMatrix-method
(Ops-methods), [133](#)
- Ops, idiMatrix, ndiMatrix-method
(Ops-methods), [133](#)
- Ops, idiMatrix, zdiMatrix-method
(Ops-methods), [133](#)
- Ops, iMatrix, ANY-method (Ops-methods),
[133](#)
- Ops, iMatrix, iMatrix-method
(Ops-methods), [133](#)
- Ops, iMatrix, Matrix-method
(Ops-methods), [133](#)
- Ops, ldenseMatrix, ldenseMatrix-method
(Ops-methods), [133](#)
- Ops, ldiMatrix, ANY-method (Ops-methods),
[133](#)
- Ops, ldiMatrix, ddiMatrix-method
(Ops-methods), [133](#)
- Ops, ldiMatrix, dMatrix-method
(Ops-methods), [133](#)
- Ops, ldiMatrix, idiMatrix-method
(Ops-methods), [133](#)
- Ops, ldiMatrix, ldiMatrix-method
(Ops-methods), [133](#)
- Ops, ldiMatrix, logical-method
(Ops-methods), [133](#)
- Ops, ldiMatrix, Matrix-method
(Ops-methods), [133](#)
- Ops, ldiMatrix, ndiMatrix-method
(Ops-methods), [133](#)
- Ops, ldiMatrix, numeric-method
(Ops-methods), [133](#)
- Ops, ldiMatrix, sparseMatrix-method
(Ops-methods), [133](#)
- Ops, ldiMatrix, zdiMatrix-method
(Ops-methods), [133](#)
- Ops, lMatrix, dMatrix-method
(Ops-methods), [133](#)
- Ops, lMatrix, lMatrix-method
(Ops-methods), [133](#)
- Ops, lMatrix, nMatrix-method
(Ops-methods), [133](#)
- Ops, lMatrix, numeric-method
(Ops-methods), [133](#)
- Ops, logical, dpoMatrix-method
(Ops-methods), [133](#)
- Ops, logical, dppMatrix-method
(Ops-methods), [133](#)
- Ops, lsparseMatrix, lsparseMatrix-method
(Ops-methods), [133](#)
- Ops, lsparseMatrix, nsparseMatrix-method
(Ops-methods), [133](#)
- Ops, Matrix, ANY-method (Ops-methods), [133](#)
- Ops, Matrix, ddiMatrix-method
(Ops-methods), [133](#)
- Ops, Matrix, diagonalMatrix-method
(Ops-methods), [133](#)
- Ops, Matrix, iMatrix-method
(Ops-methods), [133](#)
- Ops, Matrix, ldiMatrix-method
(Ops-methods), [133](#)
- Ops, Matrix, matrix-method (Ops-methods),
[133](#)
- Ops, matrix, Matrix-method (Ops-methods),
[133](#)
- Ops, Matrix, NULL-method (Ops-methods),
[133](#)
- Ops, Matrix, sparseVector-method
(Ops-methods), [133](#)
- Ops, matrix, sparseVector-method
(Ops-methods), [133](#)
- Ops, Matrix, zMatrix-method
(Ops-methods), [133](#)
- Ops, ndenseMatrix, ndenseMatrix-method
(Ops-methods), [133](#)
- Ops, ndiMatrix, ddiMatrix-method
(Ops-methods), [133](#)
- Ops, ndiMatrix, idiMatrix-method
(Ops-methods), [133](#)
- Ops, ndiMatrix, ldiMatrix-method
(Ops-methods), [133](#)
- Ops, ndiMatrix, ndiMatrix-method
(Ops-methods), [133](#)
- Ops, ndiMatrix, zdiMatrix-method
(Ops-methods), [133](#)
- Ops, nMatrix, dMatrix-method

- (Ops-methods), [133](#)
- Ops, nMatrix, lMatrix-method
(Ops-methods), [133](#)
- Ops, nMatrix, nMatrix-method
(Ops-methods), [133](#)
- Ops, nMatrix, numeric-method
(Ops-methods), [133](#)
- Ops, nsparseMatrix, dsparseMatrix-method
(Ops-methods), [133](#)
- Ops, nsparseMatrix, lsparseMatrix-method
(Ops-methods), [133](#)
- Ops, nsparseMatrix, sparseMatrix-method
(Ops-methods), [133](#)
- Ops, NULL, Matrix-method (Ops-methods),
[133](#)
- Ops, NULL, sparseVector-method
(Ops-methods), [133](#)
- Ops, numeric, dpoMatrix-method
(Ops-methods), [133](#)
- Ops, numeric, dppMatrix-method
(Ops-methods), [133](#)
- Ops, numeric, lMatrix-method
(Ops-methods), [133](#)
- Ops, numeric, nMatrix-method
(Ops-methods), [133](#)
- Ops, numeric, sparseMatrix-method
(Ops-methods), [133](#)
- Ops, sparseMatrix, ddiMatrix-method
(Ops-methods), [133](#)
- Ops, sparseMatrix, diagonalMatrix-method
(Ops-methods), [133](#)
- Ops, sparseMatrix, ldiMatrix-method
(Ops-methods), [133](#)
- Ops, sparseMatrix, nsparseMatrix-method
(Ops-methods), [133](#)
- Ops, sparseMatrix, numeric-method
(Ops-methods), [133](#)
- Ops, sparseMatrix, sparseMatrix-method
(Ops-methods), [133](#)
- Ops, sparseVector, ANY-method
(Ops-methods), [133](#)
- Ops, sparseVector, Matrix-method
(Ops-methods), [133](#)
- Ops, sparseVector, matrix-method
(Ops-methods), [133](#)
- Ops, sparseVector, NULL-method
(Ops-methods), [133](#)
- Ops, sparseVector, sparseVector-method
(Ops-methods), [133](#)
- Ops, sparseVector, vector-method
(Ops-methods), [133](#)
- Ops, vector, sparseVector-method
(Ops-methods), [133](#)
- Ops, zdiMatrix, ddiMatrix-method
(Ops-methods), [133](#)
- Ops, zdiMatrix, idiMatrix-method
(Ops-methods), [133](#)
- Ops, zdiMatrix, ldiMatrix-method
(Ops-methods), [133](#)
- Ops, zdiMatrix, ndiMatrix-method
(Ops-methods), [133](#)
- Ops, zdiMatrix, zdiMatrix-method
(Ops-methods), [133](#)
- Ops, zMatrix, ANY-method (Ops-methods),
[133](#)
- Ops, zMatrix, Matrix-method
(Ops-methods), [133](#)
- Ops, zMatrix, zMatrix-method
(Ops-methods), [133](#)
- Ops-methods, [133](#)
- options, [12](#), [13](#), [55](#), [118](#), [141](#), [142](#), [169](#), [175](#),
[184](#)
- order, [95](#), [113](#), [164](#), [214](#)
- pack, [205](#)
- pack (pack-methods), [134](#)
- pack, matrix-method (pack-methods), [134](#)
- pack, packedMatrix-method
(pack-methods), [134](#)
- pack, sparseMatrix-method
(pack-methods), [134](#)
- pack, unpackedMatrix-method
(pack-methods), [134](#)
- pack-methods, [134](#)
- packedMatrix, [19](#), [27](#), [50](#), [64](#), [77](#), [78](#), [90](#), [109](#),
[120](#), [128](#), [134](#), [139](#), [177](#), [193](#), [199](#),
[205](#), [216](#)
- packedMatrix-class, [135](#)
- panel.levelplot.raster, [88](#)
- paste, [165](#)
- plot.default, [88](#)
- pMatrix, [66](#), [70](#), [71](#), [95](#), [190](#)
- pMatrix-class (indexMatrix-class), [91](#)
- posdefify, [124](#), [125](#)
- posdefMatrix, [38](#), [56](#), [64](#), [86](#), [92](#), [120](#), [136](#),
[156](#), [157](#), [194](#), [200](#), [201](#), [204](#), [216](#)
- posdefMatrix-class, [139](#)

- print, [84](#), [87](#), [140](#), [142](#)
- print, diagonalMatrix-method (show-methods), [159](#)
- print, sparseMatrix-method (show-methods), [159](#)
- print.default, [84](#), [141](#)
- print.diagSummary (show-methods), [159](#)
- print.sparseSummary (show-methods), [159](#)
- print.trellis, [89](#)
- printSpMatrix, [84](#), [140](#)
- printSpMatrix2 (printSpMatrix), [140](#)
- QR, [144](#), [179](#)
- qr, [19](#), [27](#), [71](#), [86](#), [112](#), [123](#), [144](#), [147](#), [148](#), [159](#), [163](#), [179](#), [180](#)
- qr (qr-methods), [143](#)
- qr, CsparseMatrix-method (qr-methods), [143](#)
- qr, diagonalMatrix-method (qr-methods), [143](#)
- qr, indMatrix-method (qr-methods), [143](#)
- qr, RsparseMatrix-method (qr-methods), [143](#)
- qr, TsparseMatrix-method (qr-methods), [143](#)
- QR-class (MatrixFactorization-class), [122](#)
- qr-methods, [143](#)
- qr.coef, [162](#), [163](#), [180](#)
- qr.coef (qr-methods), [143](#)
- qr.coef, sparseQR, denseMatrix-method (qr-methods), [143](#)
- qr.coef, sparseQR, matrix-method (qr-methods), [143](#)
- qr.coef, sparseQR, sparseMatrix-method (qr-methods), [143](#)
- qr.coef, sparseQR, vector-method (qr-methods), [143](#)
- qr.coef-methods (qr-methods), [143](#)
- qr.default, [143](#)
- qr.fitted, [180](#)
- qr.fitted (qr-methods), [143](#)
- qr.fitted, sparseQR, denseMatrix-method (qr-methods), [143](#)
- qr.fitted, sparseQR, matrix-method (qr-methods), [143](#)
- qr.fitted, sparseQR, sparseMatrix-method (qr-methods), [143](#)
- qr.fitted, sparseQR, vector-method (qr-methods), [143](#)
- qr.Q, [180](#)
- qr.Q (qr-methods), [143](#)
- qr.Q, sparseQR-method (qr-methods), [143](#)
- qr.Q-methods (qr-methods), [143](#)
- qr.qty, [180](#)
- qr.qty (qr-methods), [143](#)
- qr.qty, sparseQR, denseMatrix-method (qr-methods), [143](#)
- qr.qty, sparseQR, matrix-method (qr-methods), [143](#)
- qr.qty, sparseQR, sparseMatrix-method (qr-methods), [143](#)
- qr.qty, sparseQR, vector-method (qr-methods), [143](#)
- qr.qty-methods (qr-methods), [143](#)
- qr.qy, [180](#)
- qr.qy (qr-methods), [143](#)
- qr.qy, sparseQR, denseMatrix-method (qr-methods), [143](#)
- qr.qy, sparseQR, matrix-method (qr-methods), [143](#)
- qr.qy, sparseQR, sparseMatrix-method (qr-methods), [143](#)
- qr.qy, sparseQR, vector-method (qr-methods), [143](#)
- qr.qy-methods (qr-methods), [143](#)
- qr.R, [180](#)
- qr.R (qr-methods), [143](#)
- qr.R, sparseQR-method (qr-methods), [143](#)
- qr.R-methods (qr-methods), [143](#)
- qr.resid, [180](#)
- qr.resid (qr-methods), [143](#)
- qr.resid, sparseQR, denseMatrix-method (qr-methods), [143](#)
- qr.resid, sparseQR, matrix-method (qr-methods), [143](#)
- qr.resid, sparseQR, sparseMatrix-method (qr-methods), [143](#)
- qr.resid, sparseQR, vector-method (qr-methods), [143](#)
- qr.resid-methods (qr-methods), [143](#)
- qr.solve, [180](#)
- qr.X, [180](#)
- qr.X (qr-methods), [143](#)
- qr.X, sparseQR-method (qr-methods), [143](#)

- qr.X-methods (qr-methods), 143
- qr2rankMatrix (rankMatrix), 146
- qrR (qr-methods), 143
- rank, 164, 214
- rankMatrix, 146
- raw, 118
- rbind, 15
- rbind2, 15, 157
- rbind2 (bind-methods), 15
- rbind2, Matrix, Matrix-method (bind-methods), 15
- rbind2, Matrix, matrix-method (bind-methods), 15
- rbind2, matrix, Matrix-method (bind-methods), 15
- rbind2, Matrix, missing-method (bind-methods), 15
- rbind2, Matrix, NULL-method (bind-methods), 15
- rbind2, Matrix, vector-method (bind-methods), 15
- rbind2, NULL, Matrix-method (bind-methods), 15
- rbind2, vector, Matrix-method (bind-methods), 15
- rbind2-methods (bind-methods), 15
- rcond, 36, 162
- rcond (rcond-methods), 149
- rcond, ANY, missing-method (rcond-methods), 149
- rcond, denseMatrix, character-method (rcond-methods), 149
- rcond, diagonalMatrix, character-method (rcond-methods), 149
- rcond, indMatrix, character-method (rcond-methods), 149
- rcond, pMatrix, character-method (rcond-methods), 149
- rcond, sparseMatrix, character-method (rcond-methods), 149
- rcond-methods, 149
- readHB (externalFormats), 73
- readMM (externalFormats), 73
- rep, 151, 152
- rep, denseMatrix-method (rep-methods), 151
- rep, sparseMatrix-method (rep-methods), 151
- rep, sparseVector-method (rep-methods), 151
- rep-methods, 151
- rep.int, 151, 153
- rep.int, denseMatrix-method (rep-methods), 151
- rep.int, sparseMatrix-method (rep-methods), 151
- rep.int, sparseVector-method (rep-methods), 151
- rep.int-methods (rep-methods), 151
- rep2abI, 7, 153
- rep_len, 151
- rep_len, denseMatrix-method (rep-methods), 151
- rep_len, sparseMatrix-method (rep-methods), 151
- rep_len, sparseVector-method (rep-methods), 151
- rep_len-methods (rep-methods), 151
- rle, 5–7, 153, 154
- rleDiff, 5
- rleDiff-class, 153
- round, 184
- rowMeans (colSums-methods), 33
- rowMeans, CsparseMatrix-method (colSums-methods), 33
- rowMeans, denseMatrix-method (colSums-methods), 33
- rowMeans, diagonalMatrix-method (colSums-methods), 33
- rowMeans, indMatrix-method (colSums-methods), 33
- rowMeans, RsparseMatrix-method (colSums-methods), 33
- rowMeans, TsparseMatrix-method (colSums-methods), 33
- rowMeans-methods (colSums-methods), 33
- rowScale (dimScale), 63
- rowSums, 157
- rowSums (colSums-methods), 33
- rowSums, CsparseMatrix-method (colSums-methods), 33
- rowSums, denseMatrix-method (colSums-methods), 33
- rowSums, diagonalMatrix-method (colSums-methods), 33
- rowSums, indMatrix-method

- (colSums-methods), 33
- rowSums, RsparseMatrix-method
 - (colSums-methods), 33
- rowSums, TsparseMatrix-method
 - (colSums-methods), 33
- rowSums-methods (colSums-methods), 33
- RsparseMatrix, 12, 39, 41, 64, 68, 78, 86, 90, 109, 120, 128, 139, 166, 173–175, 193, 197, 199, 200, 202, 216
- rsparsematrix, 154, 175
- RsparseMatrix-class, 156
- sample.int, 155
- Schur, 19, 27, 51, 52, 66, 71, 73, 86, 112, 123, 144, 163
- Schur (Schur-methods), 158
- Schur, denseMatrix-method
 - (Schur-methods), 158
- Schur, matrix-method (Schur-methods), 158
- Schur, sparseMatrix-method
 - (Schur-methods), 158
- Schur-class
 - (MatrixFactorization-class), 122
- Schur-methods, 158
- seq, 7
- seq.int, 38, 156
- seqMat-class (abIndex-class), 5
- set.seed, 36
- show, 6, 84, 140, 142, 154, 184
- show (show-methods), 159
- show, abIndex-method (abIndex-class), 5
- show, BunchKaufman-method
 - (show-methods), 159
- show, Cholesky-method (show-methods), 159
- show, denseMatrix-method (show-methods), 159
- show, diagonalMatrix-method
 - (show-methods), 159
- show, LU-method (show-methods), 159
- show, MatrixFactorization-method
 - (show-methods), 159
- show, QR-method (show-methods), 159
- show, rleDiff-method (rleDiff-class), 153
- show, Schur-method (show-methods), 159
- show, sparseMatrix-method
 - (show-methods), 159
- show, sparseVector-method
 - (show-methods), 159
- show-methods, 159
- showMethods, 149
- signPerm (invertPerm), 94
- simplicialCholesky-class
 - (sparseCholesky-class), 167
- skewpart, 82, 102, 194, 195
- skewpart (skewpart-methods), 160
- skewpart, CsparseMatrix-method
 - (skewpart-methods), 160
- skewpart, denseMatrix-method
 - (skewpart-methods), 160
- skewpart, diagonalMatrix-method
 - (skewpart-methods), 160
- skewpart, indMatrix-method
 - (skewpart-methods), 160
- skewpart, matrix-method
 - (skewpart-methods), 160
- skewpart, RsparseMatrix-method
 - (skewpart-methods), 160
- skewpart, TsparseMatrix-method
 - (skewpart-methods), 160
- skewpart-methods, 160
- slot, 102
- solve, 25, 150, 161, 163
- solve (solve-methods), 161
- solve, CsparseMatrix, CsparseMatrix-method
 - (solve-methods), 161
- solve, CsparseMatrix, denseMatrix-method
 - (solve-methods), 161
- solve, CsparseMatrix, missing-method
 - (solve-methods), 161
- solve, denseBunchKaufman, denseMatrix-method
 - (solve-methods), 161
- solve, denseBunchKaufman, missing-method
 - (solve-methods), 161
- solve, denseCholesky, denseMatrix-method
 - (solve-methods), 161
- solve, denseCholesky, missing-method
 - (solve-methods), 161
- solve, denseLU, denseMatrix-method
 - (solve-methods), 161
- solve, denseLU, missing-method
 - (solve-methods), 161
- solve, denseMatrix, denseMatrix-method
 - (solve-methods), 161
- solve, denseMatrix, missing-method
 - (solve-methods), 161
- solve, denseSchur, denseMatrix-method

- (solve-methods), [161](#)
- solve,denseSchur,missing-method
(solve-methods), [161](#)
- solve,diagonalMatrix,CsparseMatrix-method
(solve-methods), [161](#)
- solve,diagonalMatrix,denseMatrix-method
(solve-methods), [161](#)
- solve,diagonalMatrix,diagonalMatrix-method
(solve-methods), [161](#)
- solve,diagonalMatrix,indMatrix-method
(solve-methods), [161](#)
- solve,diagonalMatrix,matrix-method
(solve-methods), [161](#)
- solve,diagonalMatrix,missing-method
(solve-methods), [161](#)
- solve,diagonalMatrix,RsparseMatrix-method
(solve-methods), [161](#)
- solve,diagonalMatrix,TsparseMatrix-method
(solve-methods), [161](#)
- solve,diagonalMatrix,vector-method
(solve-methods), [161](#)
- solve,indMatrix,CsparseMatrix-method
(solve-methods), [161](#)
- solve,indMatrix,denseMatrix-method
(solve-methods), [161](#)
- solve,indMatrix,diagonalMatrix-method
(solve-methods), [161](#)
- solve,indMatrix,indMatrix-method
(solve-methods), [161](#)
- solve,indMatrix,matrix-method
(solve-methods), [161](#)
- solve,indMatrix,missing-method
(solve-methods), [161](#)
- solve,indMatrix,RsparseMatrix-method
(solve-methods), [161](#)
- solve,indMatrix,TsparseMatrix-method
(solve-methods), [161](#)
- solve,indMatrix,vector-method
(solve-methods), [161](#)
- solve,Matrix,CsparseMatrix-method
(solve-methods), [161](#)
- solve,Matrix,denseMatrix-method
(solve-methods), [161](#)
- solve,Matrix,diagonalMatrix-method
(solve-methods), [161](#)
- solve,Matrix,indMatrix-method
(solve-methods), [161](#)
- solve,Matrix,matrix-method
(solve-methods), [161](#)
- solve,matrix,Matrix-method
(solve-methods), [161](#)
- solve,Matrix,RsparseMatrix-method
(solve-methods), [161](#)
- solve,Matrix,sparseVector-method
(solve-methods), [161](#)
- solve,matrix,sparseVector-method
(solve-methods), [161](#)
- solve,Matrix,TsparseMatrix-method
(solve-methods), [161](#)
- solve,Matrix,vector-method
(solve-methods), [161](#)
- solve,MatrixFactorization,CsparseMatrix-method
(solve-methods), [161](#)
- solve,MatrixFactorization,denseMatrix-method
(solve-methods), [161](#)
- solve,MatrixFactorization,diagonalMatrix-method
(solve-methods), [161](#)
- solve,MatrixFactorization,indMatrix-method
(solve-methods), [161](#)
- solve,MatrixFactorization,matrix-method
(solve-methods), [161](#)
- solve,MatrixFactorization,RsparseMatrix-method
(solve-methods), [161](#)
- solve,MatrixFactorization,sparseVector-method
(solve-methods), [161](#)
- solve,MatrixFactorization,TsparseMatrix-method
(solve-methods), [161](#)
- solve,MatrixFactorization,vector-method
(solve-methods), [161](#)
- solve,RsparseMatrix,CsparseMatrix-method
(solve-methods), [161](#)
- solve,RsparseMatrix,denseMatrix-method
(solve-methods), [161](#)
- solve,RsparseMatrix,missing-method
(solve-methods), [161](#)
- solve,sparseCholesky,CsparseMatrix-method
(solve-methods), [161](#)
- solve,sparseCholesky,denseMatrix-method
(solve-methods), [161](#)
- solve,sparseCholesky,missing-method
(solve-methods), [161](#)
- solve,sparseLU,CsparseMatrix-method
(solve-methods), [161](#)
- solve,sparseLU,denseMatrix-method
(solve-methods), [161](#)
- solve,sparseLU,missing-method

- (solve-methods), 161
- solve,sparseQR,CsparseMatrix-method
(solve-methods), 161
- solve,sparseQR,denseMatrix-method
(solve-methods), 161
- solve,sparseQR,missing-method
(solve-methods), 161
- solve,TsparseMatrix,CsparseMatrix-method
(solve-methods), 161
- solve,TsparseMatrix,denseMatrix-method
(solve-methods), 161
- solve,TsparseMatrix,missing-method
(solve-methods), 161
- solve-methods, 161
- sort, 164, 214
- sort,denseMatrix-method (sort-methods),
164
- sort,sparseMatrix-method
(sort-methods), 164
- sort,sparseVector-method
(sort-methods), 164
- sort-methods, 164
- sort.int, 164
- sort.list, 95
- sparse.model.matrix, 165, 175, 178
- sparseCholesky, 207–210
- sparseCholesky-class, 167
- sparseLU, 49, 70, 111, 112, 123, 162
- sparseLU-class, 171
- SparseM.ontology, 33
- sparseMatrix, 10, 12, 16, 21, 25, 27, 33, 34,
37, 50, 64, 68, 73, 77, 84, 90, 92,
104, 109–111, 113, 117, 118, 120,
127, 141–144, 155, 156, 162, 165,
166, 169, 172, 173, 178, 179, 183,
184, 186, 187, 190, 198, 200, 208,
215
- sparseMatrix-class, 177
- sparseQR, 70, 123, 144, 147, 162
- sparseQR-class, 178
- sparseVector, 8, 17, 33, 34, 41, 60, 67, 78,
96, 108, 115, 133, 152, 162, 164,
182, 182–184, 189, 190, 192, 196,
198, 212, 214, 215
- sparseVector-class, 183
- spMatrix, 113, 142, 186
- stop, 188
- str, 123
- subassign-methods, 187
- subset-methods, 189
- substring, 141
- Summary, 184, 191
- summary, 113
- Summary,abIndex-method (abIndex-class),
5
- Summary,CsparseMatrix-method
(Summary-methods), 191
- Summary,denseMatrix-method
(Summary-methods), 191
- Summary,diagonalMatrix-method
(Summary-methods), 191
- summary,diagonalMatrix-method
(show-methods), 159
- Summary,indMatrix-method
(Summary-methods), 191
- Summary,RsparseMatrix-method
(Summary-methods), 191
- summary,sparseMatrix-method
(show-methods), 159
- Summary,sparseVector-method
(Summary-methods), 191
- Summary,TsparseMatrix-method
(Summary-methods), 191
- Summary-methods, 191
- supernodalCholesky-class
(sparseCholesky-class), 167
- svd, 120, 131, 147, 148
- symmetricMatrix, 12, 24, 38, 41, 54, 56,
62–64, 78, 82, 86, 90, 92, 102, 109,
115, 116, 118, 120, 128–130, 134,
136, 139, 140, 155, 156, 190, 197,
200, 201, 204, 216
- symmetricMatrix-class, 192
- symmpart, 82, 102, 124, 160, 194
- symmpart (symmpart-methods), 194
- symmpart,CsparseMatrix-method
(symmpart-methods), 194
- symmpart,denseMatrix-method
(symmpart-methods), 194
- symmpart,diagonalMatrix-method
(symmpart-methods), 194
- symmpart,indMatrix-method
(symmpart-methods), 194
- symmpart,matrix-method
(symmpart-methods), 194
- symmpart,RsparseMatrix-method

- (symmpart-methods), 194
- symmpart, TsparseMatrix-method
 - (symmpart-methods), 194
- symmpart-methods, 194
- t, 41, 42, 115, 147, 166, 196
- t, CsparseMatrix-method (t-methods), 196
- t, denseMatrix-method (t-methods), 196
- t, diagonalMatrix-method (t-methods), 196
- t, indMatrix-method (t-methods), 196
- t, pMatrix-method (t-methods), 196
- t, RsparseMatrix-method (t-methods), 196
- t, sparseVector-method (t-methods), 196
- t, TsparseMatrix-method (t-methods), 196
- t-methods, 196
- T2graph (coerce-methods-graph), 31
- table, 100
- tail (head-methods), 86
- tail, Matrix-method (head-methods), 86
- tail, sparseVector-method
 - (head-methods), 86
- tail-methods (head-methods), 86
- tcrossprod, 17, 115, 116, 140, 170, 208, 209
- tcrossprod (matmult-methods), 115
- tcrossprod, ANY, Matrix-method
 - (matmult-methods), 115
- tcrossprod, ANY, sparseVector-method
 - (matmult-methods), 115
- tcrossprod, CsparseMatrix, CsparseMatrix-method
 - (matmult-methods), 115
- tcrossprod, CsparseMatrix, denseMatrix-method
 - (matmult-methods), 115
- tcrossprod, CsparseMatrix, diagonalMatrix-method
 - (matmult-methods), 115
- tcrossprod, CsparseMatrix, matrix-method
 - (matmult-methods), 115
- tcrossprod, CsparseMatrix, missing-method
 - (matmult-methods), 115
- tcrossprod, CsparseMatrix, RsparseMatrix-method
 - (matmult-methods), 115
- tcrossprod, CsparseMatrix, TsparseMatrix-method
 - (matmult-methods), 115
- tcrossprod, CsparseMatrix, vector-method
 - (matmult-methods), 115
- tcrossprod, denseMatrix, CsparseMatrix-method
 - (matmult-methods), 115
- tcrossprod, denseMatrix, denseMatrix-method
 - (matmult-methods), 115
- tcrossprod, denseMatrix, diagonalMatrix-method
 - (matmult-methods), 115
- tcrossprod, denseMatrix, matrix-method
 - (matmult-methods), 115
- tcrossprod, denseMatrix, missing-method
 - (matmult-methods), 115
- tcrossprod, denseMatrix, RsparseMatrix-method
 - (matmult-methods), 115
- tcrossprod, denseMatrix, TsparseMatrix-method
 - (matmult-methods), 115
- tcrossprod, denseMatrix, vector-method
 - (matmult-methods), 115
- tcrossprod, diagonalMatrix, CsparseMatrix-method
 - (matmult-methods), 115
- tcrossprod, diagonalMatrix, denseMatrix-method
 - (matmult-methods), 115
- tcrossprod, diagonalMatrix, diagonalMatrix-method
 - (matmult-methods), 115
- tcrossprod, diagonalMatrix, matrix-method
 - (matmult-methods), 115
- tcrossprod, diagonalMatrix, missing-method
 - (matmult-methods), 115
- tcrossprod, diagonalMatrix, RsparseMatrix-method
 - (matmult-methods), 115
- tcrossprod, diagonalMatrix, TsparseMatrix-method
 - (matmult-methods), 115
- tcrossprod, diagonalMatrix, vector-method
 - (matmult-methods), 115
- tcrossprod, indMatrix, Matrix-method
 - (matmult-methods), 115
- tcrossprod, indMatrix, matrix-method
 - (matmult-methods), 115
- tcrossprod, indMatrix, missing-method
 - (matmult-methods), 115
- tcrossprod, indMatrix, vector-method
 - (matmult-methods), 115
- tcrossprod, Matrix, ANY-method
 - (matmult-methods), 115
- tcrossprod, matrix, CsparseMatrix-method
 - (matmult-methods), 115
- tcrossprod, matrix, denseMatrix-method
 - (matmult-methods), 115
- tcrossprod, matrix, diagonalMatrix-method
 - (matmult-methods), 115
- tcrossprod, Matrix, indMatrix-method
 - (matmult-methods), 115
- tcrossprod, matrix, indMatrix-method
 - (matmult-methods), 115

- tcrossprod,matrix,RsparseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,Matrix,sparseVector-method
(matmult-methods), [115](#)
- tcrossprod,matrix,sparseVector-method
(matmult-methods), [115](#)
- tcrossprod,matrix,TsparseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,pMatrix,Matrix-method
(matmult-methods), [115](#)
- tcrossprod,pMatrix,matrix-method
(matmult-methods), [115](#)
- tcrossprod,pMatrix,missing-method
(matmult-methods), [115](#)
- tcrossprod,pMatrix,pMatrix-method
(matmult-methods), [115](#)
- tcrossprod,pMatrix,vector-method
(matmult-methods), [115](#)
- tcrossprod,RsparseMatrix,CsparseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,RsparseMatrix,denseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,RsparseMatrix,diagonalMatrix-method
(matmult-methods), [115](#)
- tcrossprod,RsparseMatrix,matrix-method
(matmult-methods), [115](#)
- tcrossprod,RsparseMatrix,missing-method
(matmult-methods), [115](#)
- tcrossprod,RsparseMatrix,RsparseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,RsparseMatrix,TsparseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,RsparseMatrix,vector-method
(matmult-methods), [115](#)
- tcrossprod,sparseVector,ANY-method
(matmult-methods), [115](#)
- tcrossprod,sparseVector,Matrix-method
(matmult-methods), [115](#)
- tcrossprod,sparseVector,matrix-method
(matmult-methods), [115](#)
- tcrossprod,sparseVector,missing-method
(matmult-methods), [115](#)
- tcrossprod,sparseVector,sparseVector-method
(matmult-methods), [115](#)
- tcrossprod,sparseVector,vector-method
(matmult-methods), [115](#)
- tcrossprod,TsparseMatrix,CsparseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,TsparseMatrix,denseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,TsparseMatrix,diagonalMatrix-method
(matmult-methods), [115](#)
- tcrossprod,TsparseMatrix,matrix-method
(matmult-methods), [115](#)
- tcrossprod,TsparseMatrix,missing-method
(matmult-methods), [115](#)
- tcrossprod,TsparseMatrix,RsparseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,TsparseMatrix,TsparseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,TsparseMatrix,vector-method
(matmult-methods), [115](#)
- tcrossprod,vector,CsparseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,vector,denseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,vector,diagonalMatrix-method
(matmult-methods), [115](#)
- tcrossprod,vector,indMatrix-method
(matmult-methods), [115](#)
- tcrossprod,vector,RsparseMatrix-method
(matmult-methods), [115](#)
- tcrossprod,vector,sparseVector-method
(matmult-methods), [115](#)
- tcrossprod,vector,TsparseMatrix-method
(matmult-methods), [115](#)
- tcrossprod-methods (matmult-methods),
[115](#)
- toeplitz, [184](#), [198](#)
- toeplitz,sparseVector-method
(toeplitz-methods), [198](#)
- toeplitz-methods, [198](#)
- triangularMatrix, [10](#), [22](#), [26](#), [38](#), [54](#), [56–58](#),
[63–65](#), [78](#), [86](#), [90–92](#), [103](#), [109](#), [116](#),
[118](#), [120](#), [128](#), [134](#), [136](#), [140](#), [157](#),
[190](#), [194](#), [201](#), [204](#), [216](#)
- triangularMatrix-class, [199](#)
- tril, [200](#)
- tril (band-methods), [10](#)
- tril,CsparseMatrix-method
(band-methods), [10](#)
- tril,denseMatrix-method (band-methods),
[10](#)
- tril,diagonalMatrix-method
(band-methods), [10](#)
- tril,indMatrix-method (band-methods), [10](#)

- tril,matrix-method (band-methods), 10
- tril,RsparseMatrix-method
 - (band-methods), 10
- tril,TsparseMatrix-method
 - (band-methods), 10
- tril-methods (band-methods), 10
- triu, 200
- triu (band-methods), 10
- triu,CsparseMatrix-method
 - (band-methods), 10
- triu,denseMatrix-method (band-methods), 10
- triu,diagonalMatrix-method
 - (band-methods), 10
- triu,indMatrix-method (band-methods), 10
- triu,matrix-method (band-methods), 10
- triu,RsparseMatrix-method
 - (band-methods), 10
- triu,TsparseMatrix-method
 - (band-methods), 10
- triu-methods (band-methods), 10
- TRUE, 97
- trunc, 108
- tryCatch, 111
- TsparseMatrix, 8, 9, 12, 14, 31, 32, 39, 64, 65, 68, 78, 86, 90, 91, 109, 113, 120, 128, 139, 157, 166, 173–175, 186, 187, 194, 200, 216
- TsparseMatrix-class, 200
- type, 188
- typeof, 54

- uniqTsparse (asUniqueT), 8
- unpack, 137
- unpack (pack-methods), 134
- unpack,matrix-method (pack-methods), 134
- unpack,packedMatrix-method
 - (pack-methods), 134
- unpack,sparseMatrix-method
 - (pack-methods), 134
- unpack,unpackedMatrix-method
 - (pack-methods), 134
- unpack-methods (pack-methods), 134
- unpackedMatrix, 27, 50, 64, 77, 78, 85, 90, 109, 120, 128, 134, 137, 139, 177, 193, 199, 216
- unpackedMatrix-class, 204
- update, 207, 210
- update,sparseCholesky-method
 - (update-methods), 207
- update-methods, 207
- updown, 170, 208
- updown (updown-methods), 208
- updown,character,ANY,ANY-method
 - (updown-methods), 208
- updown,logical,Matrix,sparseCholesky-method
 - (updown-methods), 208
- updown,logical,matrix,sparseCholesky-method
 - (updown-methods), 208
- updown-methods, 208
- USCounties, 210

- warning, 19, 111
- which, 211, 212
- which,indMatrix-method (which-methods), 211
- which,ldenseMatrix-method
 - (which-methods), 211
- which,ldiMatrix-method (which-methods), 211
- which,lsparseMatrix-method
 - (which-methods), 211
- which,lsparseVector-method
 - (which-methods), 211
- which,ndenseMatrix-method
 - (which-methods), 211
- which,ndiMatrix-method (which-methods), 211
- which,nsparseMatrix-method
 - (which-methods), 211
- which,nsparseVector-method
 - (which-methods), 211
- which-methods, 211
- writeMM (externalFormats), 73
- writeMM,CsparseMatrix-method
 - (externalFormats), 73
- writeMM,sparseMatrix-method
 - (externalFormats), 73
- wrld_1deg, 212

- xtabs, 175, 178
- xtfrm, 164, 213
- xtfrm,denseMatrix-method
 - (xtfrm-methods), 213
- xtfrm,sparseMatrix-method
 - (xtfrm-methods), 213

- xtfrm, sparseVector-method
(xtfrm-methods), 213
- xtfrm-methods, 213
- zapsmall, 68, 130, 214, 215
- zapsmall, Matrix-method
(zapsmall-methods), 214
- zapsmall, sparseVector-method
(zapsmall-methods), 214
- zapsmall-methods, 214
- zdenseBunchKaufman-class
(denseBunchKaufman-class), 43
- zdenseCholesky-class
(denseCholesky-class), 45
- zdenseLU-class (denseLU-class), 48
- zdenseMatrix-class (zMatrix-class), 215
- zdenseSchur-class (denseSchur-class), 51
- zdiMatrix-class (diagonalMatrix-class),
56
- zgCMatrix-class (CsparseMatrix-class),
37
- zgeMatrix-class (unpackedMatrix-class),
204
- zgRMatrix-class (RsparseMatrix-class),
156
- zgTMatrix-class (TsparseMatrix-class),
200
- zinMatrix-class (indexMatrix-class), 91
- zMatrix, 38, 56, 65, 85, 86, 91, 101, 109, 120,
128, 136, 139, 156, 157, 193, 194,
199–201, 204
- zMatrix-class, 215
- zpCMatrix-class (CsparseMatrix-class),
37
- zpoMatrix-class (unpackedMatrix-class),
204
- zppMatrix-class (packedMatrix-class),
135
- zpRMatrix-class (RsparseMatrix-class),
156
- zpTMatrix-class (TsparseMatrix-class),
200
- zsCMatrix-class (CsparseMatrix-class),
37
- zsimplicialCholesky-class
(sparseCholesky-class), 167
- zsparseCholesky-class
(sparseCholesky-class), 167
- zsparseLU-class (sparseLU-class), 171
- zsparseMatrix-class (zMatrix-class), 215
- zsparseQR-class (sparseQR-class), 178
- zsparseVector-class
(sparseVector-class), 183
- zspMatrix-class (packedMatrix-class),
135
- zsRMatrix-class (RsparseMatrix-class),
156
- zsTMatrix-class (TsparseMatrix-class),
200
- zsupernodalCholesky-class
(sparseCholesky-class), 167
- zsyMatrix-class (unpackedMatrix-class),
204
- ztCMatrix-class (CsparseMatrix-class),
37
- ztpMatrix-class (packedMatrix-class),
135
- ztrMatrix-class (RsparseMatrix-class),
156
- ztrMatrix-class (unpackedMatrix-class),
204
- ztTMatrix-class (TsparseMatrix-class),
200