Package ‘Matrix’

January 11, 2024

Version 1.6-5

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

Date 2024-01-06

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


BugReports https://R-forge.R-project.org/tracker/?atid=294&group_id=61

Contact Matrix-authors@R-project.org

Depends R (>= 3.5.0), methods

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

Suggests MASS, datasets, sfsmisc, tools

Enhances SparseM, graph

LazyData no

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

BuildResaveData no

Encoding UTF-8

R topics documented:

- abIndex-class .................................................. 4
- abIseq .......................................................... 5
- all.equal-methods ............................................. 6
- asUniqueT ...................................................... 7
- atomicVector-class ........................................... 8
- band-methods .................................................. 9
- bandSparse ................................................... 11
<table>
<thead>
<tr>
<th>R topics documented</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bdiag</td>
<td>12</td>
</tr>
<tr>
<td>boolmatmult-methods</td>
<td>14</td>
</tr>
<tr>
<td>BunchKaufman-class</td>
<td>15</td>
</tr>
<tr>
<td>BunchKaufman-methods</td>
<td>17</td>
</tr>
<tr>
<td>CAex</td>
<td>19</td>
</tr>
<tr>
<td>chind2-methods</td>
<td>20</td>
</tr>
<tr>
<td>CHMfactor-class</td>
<td>21</td>
</tr>
<tr>
<td>chol-methods</td>
<td>25</td>
</tr>
<tr>
<td>chol2inv-methods</td>
<td>28</td>
</tr>
<tr>
<td>Cholesky-class</td>
<td>29</td>
</tr>
<tr>
<td>Cholesky-methods</td>
<td>32</td>
</tr>
<tr>
<td>coerce-methods-graph</td>
<td>37</td>
</tr>
<tr>
<td>coerce-methods-SparseM</td>
<td>38</td>
</tr>
<tr>
<td>colSums-methods</td>
<td>39</td>
</tr>
<tr>
<td>compMatrix-class</td>
<td>41</td>
</tr>
<tr>
<td>condest</td>
<td>42</td>
</tr>
<tr>
<td>CsparseMatrix-class</td>
<td>44</td>
</tr>
<tr>
<td>ddenseMatrix-class</td>
<td>45</td>
</tr>
<tr>
<td>ddiMatrix-class</td>
<td>46</td>
</tr>
<tr>
<td>denseLU-class</td>
<td>47</td>
</tr>
<tr>
<td>denseMatrix-class</td>
<td>49</td>
</tr>
<tr>
<td>dgCMatrix-class</td>
<td>49</td>
</tr>
<tr>
<td>dgeMatrix-class</td>
<td>50</td>
</tr>
<tr>
<td>dgRMatrix-class</td>
<td>51</td>
</tr>
<tr>
<td>dgTMatrix-class</td>
<td>52</td>
</tr>
<tr>
<td>Diagonal</td>
<td>53</td>
</tr>
<tr>
<td>diagonalMatrix-class</td>
<td>55</td>
</tr>
<tr>
<td>diagU2N</td>
<td>57</td>
</tr>
<tr>
<td>dimScale</td>
<td>58</td>
</tr>
<tr>
<td>dMatrix-class</td>
<td>59</td>
</tr>
<tr>
<td>dmperm</td>
<td>60</td>
</tr>
<tr>
<td>dpoMatrix-class</td>
<td>62</td>
</tr>
<tr>
<td>drop0</td>
<td>64</td>
</tr>
<tr>
<td>dsCMatrix-class</td>
<td>65</td>
</tr>
<tr>
<td>dsparseMatrix-class</td>
<td>66</td>
</tr>
<tr>
<td>dsRMatrix-class</td>
<td>67</td>
</tr>
<tr>
<td>dsyMatrix-class</td>
<td>68</td>
</tr>
<tr>
<td>dtCMatrix-class</td>
<td>70</td>
</tr>
<tr>
<td>dtMatrix-class</td>
<td>71</td>
</tr>
<tr>
<td>dtrMatrix-class</td>
<td>73</td>
</tr>
<tr>
<td>expand-methods</td>
<td>75</td>
</tr>
<tr>
<td>expm-methods</td>
<td>77</td>
</tr>
<tr>
<td>externalFormats</td>
<td>79</td>
</tr>
<tr>
<td>facmul-methods</td>
<td>80</td>
</tr>
<tr>
<td>fastMisc</td>
<td>81</td>
</tr>
<tr>
<td>forceSymmetric-methods</td>
<td>85</td>
</tr>
<tr>
<td>formatSparseM</td>
<td>86</td>
</tr>
<tr>
<td>generalMatrix-class</td>
<td>87</td>
</tr>
<tr>
<td>Hilbert</td>
<td>88</td>
</tr>
<tr>
<td>image-methods</td>
<td>88</td>
</tr>
<tr>
<td>index-class</td>
<td>91</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>indMatrix-class</td>
<td>91</td>
</tr>
<tr>
<td>invertPerm</td>
<td>94</td>
</tr>
<tr>
<td>is.na-methods</td>
<td>95</td>
</tr>
<tr>
<td>is.null.DN</td>
<td>96</td>
</tr>
<tr>
<td>isSymmetric-methods</td>
<td>98</td>
</tr>
<tr>
<td>isTriangular-methods</td>
<td>99</td>
</tr>
<tr>
<td>KhatriRao</td>
<td>100</td>
</tr>
<tr>
<td>KNex</td>
<td>102</td>
</tr>
<tr>
<td>kronecker-methods</td>
<td>103</td>
</tr>
<tr>
<td>ldenseMatrix-class</td>
<td>104</td>
</tr>
<tr>
<td>ldMatrix-class</td>
<td>104</td>
</tr>
<tr>
<td>lgeMatrix-class</td>
<td>105</td>
</tr>
<tr>
<td>lspMatrix-methods</td>
<td>106</td>
</tr>
<tr>
<td>lsyMatrix-class</td>
<td>108</td>
</tr>
<tr>
<td>ltrMatrix-class</td>
<td>109</td>
</tr>
<tr>
<td>lu-methods</td>
<td>110</td>
</tr>
<tr>
<td>mat2 triplet</td>
<td>112</td>
</tr>
<tr>
<td>matmult-methods</td>
<td>113</td>
</tr>
<tr>
<td>Matrix</td>
<td>115</td>
</tr>
<tr>
<td>Matrix-class</td>
<td>116</td>
</tr>
<tr>
<td>Matrix-notyet</td>
<td>118</td>
</tr>
<tr>
<td>MatrixClass</td>
<td>119</td>
</tr>
<tr>
<td>MatrixFactorization-class</td>
<td>120</td>
</tr>
<tr>
<td>ndenseMatrix-class</td>
<td>121</td>
</tr>
<tr>
<td>nearPD</td>
<td>122</td>
</tr>
<tr>
<td>ngeMatrix-class</td>
<td>125</td>
</tr>
<tr>
<td>nMatrix-class</td>
<td>126</td>
</tr>
<tr>
<td>nnzero-methods</td>
<td>127</td>
</tr>
<tr>
<td>norm-methods</td>
<td>128</td>
</tr>
<tr>
<td>nspMatrix-class</td>
<td>129</td>
</tr>
<tr>
<td>ntrMatrix-class</td>
<td>132</td>
</tr>
<tr>
<td>number-class</td>
<td>133</td>
</tr>
<tr>
<td>pack</td>
<td>133</td>
</tr>
<tr>
<td>packedMatrix-class</td>
<td>135</td>
</tr>
<tr>
<td>pMatrix-class</td>
<td>136</td>
</tr>
<tr>
<td>printSpMatrix</td>
<td>137</td>
</tr>
<tr>
<td>qr-methods</td>
<td>140</td>
</tr>
<tr>
<td>rankMatrix</td>
<td>143</td>
</tr>
<tr>
<td>rcond-methods</td>
<td>146</td>
</tr>
<tr>
<td>rep2abI</td>
<td>148</td>
</tr>
<tr>
<td>replValue-class</td>
<td>148</td>
</tr>
<tr>
<td>rleDiff-class</td>
<td>149</td>
</tr>
<tr>
<td>rsparsematrix</td>
<td>150</td>
</tr>
<tr>
<td>RsparseMatrix-class</td>
<td>151</td>
</tr>
<tr>
<td>Schur-class</td>
<td>152</td>
</tr>
<tr>
<td>Schur-methods</td>
<td>153</td>
</tr>
<tr>
<td>solve-methods</td>
<td>155</td>
</tr>
<tr>
<td>sparse.model.matrix</td>
<td>157</td>
</tr>
<tr>
<td>sparseLU-class</td>
<td>160</td>
</tr>
<tr>
<td>sparseMatrix</td>
<td>162</td>
</tr>
<tr>
<td>sparseMatrix-class</td>
<td>165</td>
</tr>
</tbody>
</table>
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"): ...  
**length**  signature(x = "abIndex"): ...  

**Ops**  signature(e1 = "abIndex", 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

```r
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")))
```

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

from, to  the starting and (maximal) end value of the sequence.
by  number: increment of the sequence.
length.out  desired length of the sequence. A non-negative number, which for seq and
    seq.int will be rounded up if fractional.
along.with  take the length from the length of this argument.
...  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)
  )

Description

Methods for function \texttt{all.equal()} (from R package \texttt{base}) are defined for all \texttt{Matrix} classes.

Methods

\begin{verbatim}
target = "Matrix", current = "Matrix" \
target = "ANY", current = "Matrix" \
target = "Matrix", current = "ANY"
\end{verbatim}

these three methods are simply using \texttt{all.equal.numeric}

\begin{verbatim}
directly and work via \texttt{as.vector()}.\end{verbatim}

There are more methods, notably also for “\texttt{sparseVector}”’s, see \texttt{showMethods("all.equal")}.\end{verbatim}
asUniqueT

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

Standardize a Sparse Matrix in Triplet Format

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

```r
eexample("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 == L2@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(tol(10L, which(is.na(m.0N))), !anyNA(m.1N)))
symnum(m.0N)
symnum(m.1N)
```

### atomicVector-class

**Virtual Class "atomicVector" of Atomic Vectors**

---

**Description**

The `class "atomicVector"` is a *virtual* class containing all atomic vector classes of base `R`, as also implicitly defined via `is.atomic`.

**Objects from the Class**

A virtual Class: No objects may be created from it.
Methods

In the Matrix package, the "atomicVector" is used in signatures where typically "old-style" "matrix" objects can be used and can be substituted by simple vectors.

Extends

The atomic classes "logical", "integer", "double", "numeric", "complex", "raw" and "character" are extended directly. Note that "numeric" already contains "integer" and "double", but we want all of them to be direct subclasses of "atomicVector".

Author(s)

Martin Maechler

See Also

is.atomic, integer, numeric, complex, etc.

Examples

showClass("atomicVector")

---

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

- **x**: a matrix-like object
- **k, k1, k2**: integers specifying the diagonals that are not set to zero, **k1 <= 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.
- **...**: 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 \texttt{triangularMatrix} where appropriate. It inherits from \texttt{sparseMatrix} if and only if \( x \) does.

Methods

- \( x = \text{"CsparseMatrix"} \) method for compressed, sparse, column-oriented matrices.
- \( x = \text{"RsparseMatrix"} \) method for compressed, sparse, row-oriented matrices.
- \( x = \text{"TsparseMatrix"} \) method for sparse matrices in triplet format.
- \( x = \text{"diagonalMatrix"} \) method for diagonal matrices.
- \( x = \text{"denseMatrix"} \) method for dense matrices in packed or unpacked format.
- \( x = \text{"matrix"} \) method for traditional matrices of implicit class \texttt{matrix}.

See Also

\texttt{bandSparse} for the \textit{construction} of a banded sparse matrix directly from its non-zero diagonals.

Examples

```r
## 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"
```

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"))

Arguments

n, m the matrix dimension \((n, m) = (nrow, ncol)\).

k integer vector of “diagonal numbers”, with identical meaning as in \texttt{band(*, k)}, 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 \texttt{nMatrix}.

diagonals can also be \(n'^d\) matrix, where \(d <- length(k)\) 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 \texttt{symmetricMatrix}) and only the upper or lower triangle must be specified (via \(k\) and diagonals).

repr character string, one of \texttt{"C"}, \texttt{"T"}, or \texttt{"R"}, specifying the sparse representation to be used for the result, i.e., one from the super classes \texttt{CsparseMatrix}, \texttt{TsparseMatrix}, or \texttt{RsparseMatrix}.

giveCsparse \texttt{(deprecated, replaced with \texttt{repr})}: logical indicating if the result should be a \texttt{CsparseMatrix} or a \texttt{TsparseMatrix}, where the default was \texttt{TRUE}, and now is determined from \texttt{repr}; very often \texttt{Csparse} matrices are more efficient subsequently, but not always.

Value

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

See Also

\texttt{band}, for \texttt{extraction} of matrix bands; \texttt{bdiag}, \texttt{diag}, \texttt{sparseMatrix}, \texttt{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(...)  
.bdiag(lst)

Arguments

... individual matrices or a list of matrices.  
lst non-empty list of matrices.

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 × 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))))

ml <- c(as(matrix((1:24)%% 11 == 0, 6,4),"nMatrix"),
        rep(list(Diagonal(2, x=TRUE)), 3))
mln <- c(ml, Diagonal(x = 1:3))
stopifnot(is(bdiag(ml), "lsparseMatrix"),
          is(bdiag(mln),"dsparseMatrix"))

## 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{"dgCMatrix"}

bdiag_m <- function(lmat) {
  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)
  new("dgCMatrix", Dim = c(M,M))
}

Boolean Arithmetic Matrix Products: %&% and Methods

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 sparseVectors 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@x 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 != 0 (or M != 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

\%\% is equivalent to crossprod(), or tcrossprod(), for (regular) matrix product methods.

Examples

```r
set.seed(7)
L <- Matrix(rnorm(20) > 1, 4, 5)
(N <- as(l, "nMatrix"))
L <- L; L.[1:2,1] <- TRUE; L.8x[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-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 \(b_L\) \(1 \times 1\) or \(2 \times 2\) 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 \(nn\) (BunchKaufman) or \(n(n+1)/2\) (pBunchKaufman), where \(n=\text{Dim}[1]\).

The details of the representation are specified by the manual for LAPACK routines dsytrf and dsprf.

- **perm** an integer vector of length \(n=\text{Dim}[1]\) specifying row and column interchanges as described in the manual for LAPACK routines dsytrf and dsprf.
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

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

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

determinant signature(from = "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.


See Also

Class `dsyMatrix` and its packed counterpart.

Generic functions `BunchKaufman`, `expand1`, and `expand2`.
Examples

```r
class <- "BunchKaufman"
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 2b+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"), ...)  
ea2 <- 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)))
```

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

\[
A = U D_U U' = LD_L L'
\]

where \( D_U \) and \( D_L \) are symmetric, block diagonal matrices composed of \( b_U \) and \( b_L \) \( 1 \times 1 \) or \( 2 \times 2 \) 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 \texttt{dsytrf} and \texttt{dsptrf}.
Usage

BunchKaufman(x, ...)  
## S4 method for signature 'dsyMatrix'  
BunchKaufman(x, warnSing = TRUE, ...)  
## S4 method for signature 'dspMatrix'  
BunchKaufman(x, warnSing = TRUE, ...)  
## S4 method for signature 'matrix'  
BunchKaufman(x, uplo = "U", ...)  

Arguments

- **x**: a finite symmetric matrix or Matrix to be factorized. If x is square but not symmetric, then it will be treated 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.
- ... 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](https://netlib.org/lapack/double/dsytrf.f) and [https://netlib.org/lapack/double/dsptrf.f](https://netlib.org/lapack/double/dsptrf.f).


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

```r
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 DU' 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")))

### Description

Albers' example Matrix with "Difficult" Eigen Factorization

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](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

```r
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"))
```
cbind2-methods

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

Description

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

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

Usage

\begin{verbatim}
## 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, ...)
\end{verbatim}

Arguments

\begin{itemize}
\item \ldots for \texttt{[cr]bind}, vector- or matrix-like \texttt{R} objects to be bound together; for \texttt{[cr]bind2}, further arguments passed to or from methods; see \texttt{cbind} and \texttt{cbind2}.
\item \texttt{deparse.level} integer controlling the construction of labels in the case of non-matrix-like arguments; see \texttt{cbind}.
\item \texttt{x, y} vector- or matrix-like \texttt{R} objects to be bound together.
\end{itemize}

Value

typically a ‘matrix-like’ object of a similar \texttt{class} as the first argument in \ldots.

Note that sometimes by default, the result is a \texttt{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 \texttt{sparse} in \texttt{Matrix()}).

Author(s)

Martin Maechler

See Also

cbind, \texttt{cbind2}.

Our class definition help pages mentioning \texttt{cbind2()} and \texttt{rbind2()} methods: "denseMatrix", "diagonalMatrix", "indMatrix".
Examples

(a <- matrix(c(2:1,1:2), 2,2)) # a traditional matrix

(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))

CHMfactor-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

\[
P_1 A P_1' = L_1 D L_1' \quad \begin{cases} \quad D_{jj} \geq 0 \end{cases} \quad LL'
\]

or (equivalently)

\[
A = P_1' L_1 D L_1' P_1 \quad \begin{cases} \quad D_{jj} \geq 0 \end{cases} \quad P_1' LL' P_1
\]

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 \( \text{Dim}[1] \) 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.

type an integer vector of length 6 specifying details of the factorization. The elements correspond to members ordering, is_ll, is_super, is_monotonic, maxsize, and maxesize of the original cholmod_factor_struct. Simplicial and supernodal factorizations are distinguished by is_super. Simplicial factorizations do not use maxsize or maxesize. Supernodal factorizations do not use is_ll or is_monotonic.

Of \text{CHMsimpl} (all unused by \text{nCHMsimpl}):

\( \text{nz} \) an integer vector of length \( \text{Dim}[1] \) 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.

\( \text{p} \) an integer vector of length \( \text{Dim}[1]+1 \). Row indices of nonzero entries in column \( j \) of the lower triangular Cholesky factor are obtained as \( i[p[j]+\text{seq_len(nz[j])}] \).

\( \text{i} \) an integer vector of length greater than or equal to \( \text{sum(nz)} \) 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 \( \text{nz}[j] \).

\( \text{prv}, \text{nxt} \) integer vectors of length \( \text{Dim}[1]+2 \) indicating the order in which the columns of the lower triangular Cholesky factor are stored in \( i \) and \( x \). Starting from \( j <- \text{Dim}[1]+2 \), the recursion \( j <- \text{nxt}[j+1]+1 \) traverses the columns in forward order and terminates when \( \text{nxt}[j+1] = -1 \). Starting from \( j <- \text{Dim}[1]+1 \), the recursion \( j <- \text{prv}[j+1]+1 \) traverses the columns in backward order and terminates when \( \text{prv}[j+1] = -1 \).

Of \text{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 \text{CHMsuper}:

\( \text{super}, \pi, \text{px} \) integer vectors of length \( n\text{super}+1 \), where \( n\text{super} \) is the number of supernodes. \( \text{super}[j]+1 \) is the index of the leftmost column of supernode \( j \). The row indices of supernode \( j \) are obtained as \( s[\pi[j]+\text{seq_len(\pi[j+1]-\pi[j])}] \). The numeric entries of supernode \( j \) are obtained as \( x[\text{px}[j]+\text{seq_len(\text{px}[j+1]-\text{px}[j])}] \) (if slot \( x \) is available).

\( s \) an integer vector of length greater than or equal to \( \text{Dim}[1] \) containing the row indices of the supernodes. \( s \) may contain duplicates, but not within a supernode, where the row indices must be increasing.

Of \text{dCHMsuper}:

\( x \) a numeric vector of length less than or equal to \( \text{prod(Dim)} \) containing the numeric entries of the supernodes.

\textbf{Extends}

\textbf{Class} \text{MatrixFactorization}, directly.
CHMfactor-class

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 = "dCHMsimpl", 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 = "dCHMsuper", 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(from = "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 $L = L_1\sqrt{D}$ 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. 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 = "CHMfactor"): see expand2-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 $F(parent) + mult[1] * I$, i.e., $F(parent)$ plus $mult[1]$ times the identity matrix, where $F = \text{identity}$ for symmetric parent and $F = \text{tcrossprod}$ for other parent. The nonzero pattern of $F(parent)$ must match that of $S$ if object = Cholesky(S, ...).

updown signature(update = ., C = ., object = "CHMfactor"): see updown-methods.
The CHOLMOD source code; see https://github.com/DrTimothyAldenDavis/SuiteSparse, notably the header file ‘CHOLMOD/Include/cholmod.h’ defining cholmod_factor_struct.


See Also

- Class dsCMatrix.
- Generic functions Cholesky, updown, expand1 and expand2.

Examples

```r
showClass("dCHMsimpl")
showClass("dCHMsuper")
set.seed(2)

m <- 1000L
n <- 200L
M <- rsparsematrix(m, n, 0.01)
A <- crossprod(M)

# Cholesky
(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"]]),
    e.ch.A[["L1."]], t(e.ch.A[["L1"]]),
    e.ch.A[["L."]], t(E.ch.A[["L"]]],
    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(ch.A, P1. %*% L1 %*% D %*% L1. %*% 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(identical(det(A), det(ch.A, sqrt = FALSE)),
         identical(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))

---

**chol-methods**

Compute the Cholesky Factor of a 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

\[
P_1 A P_1' = LL'
\]

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

```r
chol(x, ...)
```

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

#### S4 method for signature 'dspMatrix'
```r
chol(x, ...)
```

#### S4 method for signature 'dsCMatrix'
```r
chol(x, pivot = FALSE, ...)
```

#### S4 method for signature 'ddiMatrix'
```r
chol(x, ...)
```

#### S4 method for signature 'generalMatrix'
```r
chol(x, uplo = "U", ...)
```

#### S4 method for signature 'triangularMatrix'
```r
chol(x, uplo = "U", ...)
```

### Arguments

- x  
  
  a finite, symmetric, positive semidefinite matrix or **Matrix** to be factorized. If \( x \) is square but not symmetric, then it will be treated as symmetric; see `uplo`. Methods for dense \( x \) require positive definiteness when `pivot = FALSE`. Methods for sparse (but not diagonal) \( x \) require positive definiteness unconditionally.
pivot

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.

tol

A finite numeric tolerance, used only if pivot = TRUE. The factorization algorithm stops if the pivot is less than or equal to tol. Negative tol is equivalent to \( \text{nrow}(x) \times .\text{Machine}^\text{\$double\_eps} \times \max(\text{diag}(x)) \).

uplo

A string, either "U" or "L", indicating which triangle of \( x \) should be used to compute the factorization. The default is "U", even for lower triangular \( x \), to be consistent with chol from base.

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


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")
\( (A4 \leftarrow \text{toeplitz(as(c(10, 0, 1, 0, 3), \text{"sparseVector"}))) \)
\( (\text{ch.A4.nopivot} \leftarrow \text{Cholesky}(A4, \text{perm} = \text{FALSE}, \text{LDL} = \text{FALSE}, \text{super} = \text{FALSE})) \)
\( (\text{ch.A4} \leftarrow \text{Cholesky}(A4, \text{perm} = \text{TRUE}, \text{LDL} = \text{FALSE}, \text{super} = \text{FALSE})) \)
\( (R4.nopivot \leftarrow \text{chol}(A4)) \)
\( (R4 \leftarrow \text{chol}(A4, \text{pivot} = \text{TRUE})) \)

\[ \text{det4} \leftarrow \text{det}(A4) \]
\[ b4 \leftarrow \text{rnorm}(5L) \]
\[ x4 \leftarrow \text{solve}(A4, b4) \]

\texttt{stopifnot(exprs = {}}
\texttt{ identical(R4.nopivot, expand1(ch.A4.nopivot, \text{"L."}))}
\texttt{ identical(R4, expand1(ch.A4, \text{"L."}))}
\texttt{ all.equal(A4, \text{crossprod}(R4.nopivot))}
\texttt{ all.equal(A4[ch.A4@perm + 1L, ch.A4@perm + 1L], \text{crossprod}(R4))}
\texttt{ all.equal(diag(R4.nopivot), \text{sqrt(diag(ch.A4.nopivot))))}
\texttt{ all.equal(diag(R4), \text{sqrt(diag(ch.A4))))}
\texttt{ all.equal(sqrt(det4), \text{det}(R4.nopivot))}
\texttt{ all.equal(sqrt(det4), \text{det}(R4))}
\texttt{ all.equal(det4, \text{det}(ch.A4.nopivot, \text{sqrt} = \text{FALSE}))}
\texttt{ all.equal(det4, \text{det}(ch.A4, \text{sqrt} = \text{FALSE}))}
\texttt{ all.equal(x4, \text{solve}(R4.nopivot, \text{solve(t(R4.nopivot), b4))})
\texttt{ all.equal(x4, \text{solve}(ch.A4.nopivot, b4))}
\texttt{ all.equal(x4, \text{solve}(ch.A4, b4))}
\texttt{ }) \)

---

### 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**

\[ \text{chol2inv}(x, ...) \]

\# S4 method for signature 'dtrMatrix'
\text{chol2inv}(x, ...)

\# S4 method for signature 'dtCMatrix'
\text{chol2inv}(x, ...)

\# S4 method for signature 'generalMatrix'
\text{chol2inv}(x, uplo = "U", ...)

**Arguments**

- \( x \) a square matrix or \texttt{Matrix}, typically the result of a call to \texttt{chol}. If \( x \) is square but not (formally) triangular, then only the upper or lower triangle is considered, depending on optional argument \texttt{uplo} if \( x \) is a \texttt{Matrix}.  

---

28
Cholesky-class

uplo a string, either "U" or "L", indicating which triangle of x contains the Cholesky factor. The default is "U", to be consistent with chol2inv from base.

... 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

```r
(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-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

$$P_1 A P_1' = L_1 D L_1' = LL'$$

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.
Cholesky-class

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. \( \text{perm} \) 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**(from = "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


See Also

Class CholMfactor for sparse Cholesky factorizations.

Classes dpoMatrix and dppMatrix.

Generic functions Cholesky, expand1 and expand2.

Examples

```r
showClass("Cholesky")
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"), t(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
## A ~ P1'  L1  D  L1'  P1 ~ P1'  L  L'  P1

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(NULL, names(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, L %*% sort(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.))
})
```
## Factorization handled as factorized matrix

```r
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

```r
(ch.A.nopivot <- Cholesky(A, perm = FALSE))
stopifnot(identical(det(A), det(ch.A.nopivot)),
          identical(solve(A, b), solve(ch.A.nopivot, b)))
```

---

### Description

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

$$P_1AP_1' = L_1DL_1' + \geq 0LL'$$

or (equivalently)

$$A = P_1'LL_1'P_1 + \geq D_1'LL_1'P_1$$

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

```r
Cholesky(A, ...)
```

## S4 method for signature 'dsyMatrix'

```r
Cholesky(A, perm = TRUE, tol = -1, ...)
```

## S4 method for signature 'dspMatrix'

```r
Cholesky(A, ...)
```

## S4 method for signature 'dsCMatrix'

```r
Cholesky(A, perm = TRUE, LDL = !super, super = FALSE,
         Imult = 0, ...)
```

## S4 method for signature 'ddiMatrix'

```r
Cholesky(A, ...)
```

## S4 method for signature 'generalMatrix'

```r
Cholesky(A, uplo = "U", ...)
```

## S4 method for signature 'triangularMatrix'

```r
Cholesky(A, uplo = "U", ...)
```

## S4 method for signature 'matrix'

```r
Cholesky(A, uplo = "U", ...)
```
Arguments

A  a finite, symmetric matrix or Matrix to be factorized. If A is square but not symmetric, then it will be treated 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, diagonal 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 nrow(A) * .Machine$double.eps * max(diag(A)).

LDL  a logical indicating if the simplicial factorization should be computed as $P_1' L_1 L_1' P_1$, such that the result stores the lower triangular entries of $L_1 - I + D$. The alternative is $P_1' LL' P_1$, 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 $A + Imult \times diag(nrow(A))$, i.e., A plus Imult times the identity matrix. This argument is useful for symmetric, indefinite A, as Imult > max(rowSums(abs(A)) - diag(abs(A))) ensures that $A + Imult \times diag(nrow(A))$ is diagonally dominant. (Symmetric, diagonally dominant matrices are positive definite.)

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


**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**

```r
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))
```

```r
```
Cholesky-methods

```r
(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 ~ P1' L D L' P1 = P1' L L' P1 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") # 7.678858e-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))

## ---- 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(simp11 = 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)

simp11 <- ch.A4[[c("pivoted", "simpl1")]]
stopifnot(exprs = {
  length(simp11@perm) == ncol(A4)
  isPerm(simp11@perm, 0L)
  is.unsorted(simp11@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(simp11)
stopifnot(min(diag(simp11)) >= 0)
str(e.ch.A4 <- expand2(simp11, LDL = TRUE), max.level = 2L) # default
str(E.ch.A4 <- expand2(simp11, 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), "symmetricMatrix"))
  all.equal(A4, as(Reduce("%^%", E.ch.A4), "symmetricMatrix"))
})

## 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 $L' = \sqrt{D}$ $L_1'$ 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) # 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 $L = L_1' \sqrt{D}$
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"]

---

**Description**

Since 2005, package *Matrix* has supported coercions to and from class *graph* from package *graph*. 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 graph2T(), an R object of class "graph";
   for T2graph(), 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 NULL, "directed", or "undirected". The default NULL looks if the
   matrix is symmetric and assumes "undirected" 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,

\[
\text{as(from, Class)}
\]

Methods

- \text{from} = "matrix.csr", to = "dgRMatrix"
- \text{from} = "matrix.csc", to = "dgCMatrix"
- \text{from} = "matrix.coo", to = "dgTMatrix"
- \text{from} = "dgRMatrix", to = "matrix.csr"
- \text{from} = "dgCMatrix", to = "matrix.csc"
- \text{from} = "dgTMatrix", to = "matrix.coo"
- \text{from} = "Matrix", to = "matrix.csr"
- \text{from} = "matrix.csr", to = "dgCMatrix"
- \text{from} = "matrix.coo", to = "dgCMatrix"
- \text{from} = "matrix.csr", to = "Matrix"
- \text{from} = "matrix.csc", to = "Matrix"
- \text{from} = "matrix.coo", to = "Matrix"

See Also

The documentation in CRAN package SparseM, such as SparseM.ontology, and one important class, \text{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 (\text{sparseVector}), too. Row or column names are kept respectively as for base matrices and \text{colSums} methods, when the result is \text{numeric} vector.

Usage

\[
\begin{align*}
\text{colSums}(x, \text{na.rm} = \text{FALSE}, \text{dims} = 1, \ldots) \\
\text{rowSums}(x, \text{na.rm} = \text{FALSE}, \text{dims} = 1, \ldots) \\
\text{colMeans}(x, \text{na.rm} = \text{FALSE}, \text{dims} = 1, \ldots) \\
\text{rowMeans}(x, \text{na.rm} = \text{FALSE}, \text{dims} = 1, \ldots)
\end{align*}
\]

## S4 method for signature 'CsparseMatrix'
\[
\text{colSums}(x, \text{na.rm} = \text{FALSE}, \text{dims} = 1, \text{sparseResult} = \text{FALSE}, \ldots)
\]

## S4 method for signature 'CsparseMatrix'
\[
\text{rowSums}(x, \text{na.rm} = \text{FALSE}, \text{dims} = 1, \ldots)
\]
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

- **x**
  - a Matrix, i.e., inheriting from \texttt{Matrix}.

- **na.rm**
  - logical. Should missing values (including NaN) be omitted from the calculations?

- **dims**
  - completely ignored by the \texttt{Matrix} methods.

- **...**
  - potentially further arguments, for method \texttt{<-}> generic compatibility.

- **sparseResult**
  - logical indicating if the result should be sparse, i.e., inheriting from class \texttt{sparseVector}.
  - Only applicable when \texttt{x} is inheriting from a \texttt{sparseMatrix} class.

Value

- returns a numeric vector if \texttt{sparseResult} is \texttt{FALSE} as per default. Otherwise, returns a \texttt{sparseVector}.

- \texttt{dimnames(x)} are only kept (as \texttt{names(v)}) when the resulting \texttt{v} is \texttt{numeric}, since \texttt{sparseVectors} do not have names.

See Also

- \texttt{colSums} and the \texttt{sparseVector} classes.

Examples

```r
(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(1,1,6,6,6,6,2), names = colnames(M)))
  all.equal(rowMeans(M),
    structure(c(1,1,4,8,12,3,2)/8, names = paste0("r", 1:7)))
})

### Description

Virtual class of composite matrices; i.e., matrices that can be factorized, typically as a product of simpler matrices.

### Objects from the Class

A virtual Class: No objects may be created from it.

### Slots

- **factors**: Object of class "list" - a list of factorizations of the matrix. Note that this is typically empty, i.e., `list()`, initially and is updated automatically whenever a matrix factorization is computed.

- **Dim**, **Dimnames**: inherited from the Matrix class, see there.

### Extends

Class "Matrix", directly.

### Methods

- **dimnames<-** signature(x = "compMatrix", value = "list"): set the dimnames to a list of length 2, see `dimnames<-`. The factors slot is currently reset to empty, as the factorization dimnames would have to be adapted, too.

### See Also

The matrix factorization classes "MatrixFactorization" and their generators, `lu()`, `qr()`, `chol()` and `Cholesky()`, `BunchKaufman()`, `Schur()`.
**condest**

**Compute Approximate CONDition number and 1-Norm of (Large) Matrices**

**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, $\|A\|_1$, through `onenormest()`.

**Usage**

```r
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 $\|A\|_1$; may be replaced by an estimate.
- **silent**: logical indicating if warning and (by default) convergence messages should be displayed.
- **quiet**: logical indicating if convergence messages should be displayed.
- **A.x, At.x**: when $A$ is missing, these two must be given as functions which compute $A \%\% x$, or $t(A) \%\% x$, respectively.
- **n**: $= nrow(A)$, only needed when $A$ is not specified.
- **iter.max**: maximal number of iterations for the 1-norm estimator.
- **eps**: 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,

- **est**: a number > 0, the estimated (1-norm) condition number $\hat{\kappa}$; when $r := rcond(A)$, $1/\hat{\kappa} \approx r$. 

\( v \)  
the maximal \( Ax \) column, scaled to \( \|v\| = 1 \). Consequently, \( \|Av\| = \text{norm}(A)/\text{est} \); when \( \text{est} \) is large, \( v \) is an approximate null vector.

The function \( \text{onenormest}() \) returns a list with components,

\( \text{est} \)  
a number > 0, the estimated \( \text{norm}(A, \"1\") \).

\( v \)  
0-1 integer vector length \( n \), with an 1 at the index \( j \) with maximal column \( A[,j] \) in \( A \).

\( w \)  
numeric vector, the largest \( Ax \) found.

\( \text{iter} \)  
the number of iterations used.

**Author(s)**

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

**References**


**See Also**

\texttt{norm}, \texttt{rcond}.

**Examples**

```r
data(KNex, package = "Matrix")
m = with(KNex, crossprod(mm))

system.time(ce <- condest(mtm))
sum(abs(ce$v))  ## \| v \|_1 = 1

## Prove that \( \| A v \| = \| A \| / \text{est} \) (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
```


CsparseMatrix-class

Class "CsparseMatrix" of Sparse Matrices in Column-compressed Form

Description

The "CsparseMatrix" class is the virtual class of all sparse matrices coded in sorted compressed column-oriented form. Since it is a virtual class, no objects may be created from it. See showClass("CsparseMatrix") for its subclasses.

Slots

i: Object of class "integer" of length nnzero (number of non-zero elements). These are the 0-based row numbers for each non-zero element in the matrix, i.e., i must be in 0:(nrow(.)-1).

p: integer vector for providing pointers, one for each column, to the initial (zero-based) index of elements in the column. .@p is of length ncol(.) + 1, with p[1] == 0 and p[length(p)] == nnzero, such that in fact, diff(.@p) are the number of non-zero elements for each column.

In other words, m@p[1:ncol(m)] contains the indices of those elements in m@x that are the first elements in the respective column of m.

Dim, Dimnames: inherited from the superclass, see the sparseMatrix class.

Extends

Class "sparseMatrix", directly. Class "Matrix", by class "sparseMatrix".

Methods

matrix products %*%, crossprod() and tcrossprod(), several solve methods, and other matrix methods available:

signature(e1 = "CsparseMatrix", e2 = "numeric"): ...

Arith signature(e1 = "numeric", e2 = "CsparseMatrix"): ...  

Math signature(x = "CsparseMatrix"): ...

band signature(x = "CsparseMatrix"): ...

- signature(e1 = "CsparseMatrix", e2 = "numeric"): ...  
- signature(e1 = "numeric", e2 = "CsparseMatrix"): ...

+ signature(e1 = "CsparseMatrix", e2 = "numeric"): ... 
+ signature(e1 = "numeric", e2 = "CsparseMatrix"): ...

coerce signature(from = "CsparseMatrix", to = "TsparseMatrix"): ...

coerce signature(from = "CsparseMatrix", to = "denseMatrix"): ...

coerce signature(from = "CsparseMatrix", to = "matrix"): ...

coerce signature(from = "TsparseMatrix", to = "CsparseMatrix"): ...

coerce signature(from = "denseMatrix", to = "CsparseMatrix"): ...

diag signature(x = "CsparseMatrix"): ...

gamma signature(x = "CsparseMatrix"): ...

lgamma signature(x = "CsparseMatrix"): ...
**ddenseMatrix-class**

Virtual Class "ddenseMatrix" of Numeric Dense Matrices

**Description**

This is the virtual class of all dense numeric (i.e., double, hence "ddense") S4 matrices. Its most important subclass is the `dgeMatrix` class.

**Extends**

Class "dMatrix" directly; class "Matrix", by the above.

**Slots**

the same slots at its subclass `dgeMatrix`, see there.

**Methods**

Most methods are implemented via `as(*, "generalMatrix")` and are mainly used as "fallbacks" when the subclass doesn't need its own specialized method.

Use `showMethods(class = "ddenseMatrix", where = "package:Matrix")` for an overview.
See Also

The virtual classes Matrix, dMatrix, and dsparseMatrix.

Examples

showClass("ddenseMatrix")

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

---

ddiMatrix-class

Class "ddiMatrix" of Diagonal Numeric Matrices

Description

The class "ddiMatrix" of numerical diagonal matrices.

Note that diagonal matrices now extend sparseMatrix, whereas they did extend dense matrices earlier.

Objects from the Class

Objects can be created by calls of the form new("ddiMatrix", ...) but typically rather via Diagonal.

Slots

x: numeric vector. For an \( n \times n \) matrix, the \( x \) slot is of length \( n \) or 0, depending on the \( \text{diag} \) slot:

diag: "character" string, either "U" or "N" where "U" denotes unit-diagonal, i.e., identity matrices.

Dim, Dimnames: matrix dimension and dimnames, see the Matrix class description.

Extends

Class "diagonalMatrix", directly. Class "dMatrix", directly. Class "sparseMatrix", indirectly, see showClass("ddiMatrix").

Methods

\%\% signature(x = "ddiMatrix", y = "ddiMatrix"): ...

See Also

Class diagonalMatrix and function Diagonal.

Examples

(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)))

---

denseLU-class

Dense LU Factorizations

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 (from = "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](https://netlib.org/lapack/double/dgetrf.f).


## See Also

Class `sparseLU` for sparse LU factorizations.

Class `dgeMatrix`.

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

## Examples

```r
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 = c(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

Virtual Class "denseMatrix" of All Dense Matrices

Description

This is the virtual class of all dense (S4) matrices. It partitions into two subclasses \texttt{packedMatrix} and \texttt{unpackedMatrix}. Alternatively into the (currently) three subclasses \texttt{ddenseMatrix}, \texttt{ldenseMatrix}, and \texttt{ndenseMatrix}.

denseMatrix is (hence) the direct superclass of these ($2 + 3 = 5$) classes.

Extends

class "Matrix" directly.

Slots

exactly those of its superclass "Matrix", i.e., "Dim" and "Dimnames".

Methods

Use \texttt{showMethods(class = "denseMatrix", where = "package:Matrix")} for an overview of methods.

Extraction ("\[\]) methods, see \texttt{[<-methods}.

See Also

colSums, kronecker, and other such methods with own help pages.

Its superclass \texttt{Matrix}, and main subclasses, \texttt{ddenseMatrix} and \texttt{sparseMatrix}.

Examples

\texttt{showClass("denseMatrix")}

dgCMatrix-class

Compressed, sparse, column-oriented numeric matrices

Description

The \texttt{dgCMatrix} class is a class of sparse numeric matrices in the compressed, sparse, column-oriented format. In this implementation the non-zero elements in the columns are sorted into increasing row order. \texttt{dgCMatrix} is the "\texttt{standard}" class for sparse numeric matrices in the \texttt{Matrix} package.

Objects from the Class

Objects can be created by calls of the form \texttt{new("dgCMatrix", \ldots)}, more typically via \texttt{as(*, "CsparseMatrix")} or similar. Often however, more easily via \texttt{Matrix(*, sparse = TRUE)}, or most efficiently via \texttt{sparseMatrix()}. 

Slots

x: Object of class "numeric" - the non-zero elements of the matrix.
...
... all other slots are inherited from the superclass "CsparseMatrix".

Methods

Matrix products (e.g., crossprod-methods), and (among other)

coerce signature(from = "matrix", to = "dgCMatrix")
diag signature(x = "dgCMatrix"): returns the diagonal of x
dim signature(x = "dgCMatrix"): returns the dimensions of x
image signature(x = "dgCMatrix"): plots an image of x using the levelplot function
solve signature(a = "dgCMatrix", b = "..."): see solve-methods, notably the extra argument sparse.
lu signature(x = "dgCMatrix"): computes the LU decomposition of a square dgCMatrix object

See Also

classes dsCMatrix, dtCMatrix, lu

Examples

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

The are group methods (see, e.g., \texttt{Arith})

\texttt{Arith} signature(e1 = "dgeMatrix", e2 = "dgeMatrix"): ...
\texttt{Arith} signature(e1 = "dgeMatrix", e2 = "numeric"): ...
\texttt{Arith} signature(e1 = "numeric", e2 = "dgeMatrix"): ...
\texttt{Math} signature(x = "dgeMatrix"): ...
\texttt{Math2} signature(x = "dgeMatrix", digits = "numeric"): ...

matrix products \%*\%, \texttt{crossprod()} and \texttt{tcrossprod()}, several \texttt{solve} methods, and other matrix methods available:

\texttt{Schur} signature(x = "dgeMatrix", vectors = "logical"): ...
\texttt{Schur} signature(x = "dgeMatrix", vectors = "missing"): ...
\texttt{chol} signature(x = "dgeMatrix"): see \texttt{chol}.
\texttt{colMeans} signature(x = "dgeMatrix"): columnwise means (averages)
\texttt{colSums} signature(x = "dgeMatrix"): columnwise sums
\texttt{diag} signature(x = "dgeMatrix"): ...
\texttt{dim} signature(x = "dgeMatrix"): ...
\texttt{dimnames} signature(x = "dgeMatrix"): ...
\texttt{eigen} signature(x = "dgeMatrix", only.values= "logical"): ...
\texttt{eigen} signature(x = "dgeMatrix", only.values= "missing"): ...
\texttt{norm} signature(x = "dgeMatrix", type = "character"): ...
\texttt{norm} signature(x = "dgeMatrix", type = "missing"): ...
\texttt{rcond} signature(x = "dgeMatrix", norm = "character") or norm = "missing": the reciprocal condition number, \texttt{rcond}().
\texttt{rowMeans} signature(x = "dgeMatrix"): rowwise means (averages)
\texttt{rowSums} signature(x = "dgeMatrix"): rowwise sums
\texttt{t} signature(x = "dgeMatrix"): matrix transpose

See Also

Classes \texttt{Matrix}, \texttt{dtrMatrix}, and \texttt{dsyMatrix}.

\begin{verbatim}
dgRMatrix-class  Sparse Compressed, Row-oriented Numeric Matrices
\end{verbatim}

Description

The \texttt{dgRMatrix} class is a class of sparse numeric matrices in the compressed, sparse, row-oriented format. In this implementation the non-zero elements in the rows are sorted into increasing column order.

Note: The column-oriented sparse classes, e.g., \texttt{dgCMatrix}, are preferred and better supported in the \texttt{Matrix} package.
Objects from the Class

Objects can be created by calls of the form `new("dgRMatrix", ...)`. 

Slots

- **j**: Object of class "integer" of length `nnzero` (number of non-zero elements). These are the column numbers for each non-zero element in the matrix.
- **p**: Object of class "integer" of pointers, one for each row, to the initial (zero-based) index of elements in the row.
- **x**: Object of class "numeric" - the non-zero elements of the matrix.
- **Dim**: Object of class "integer" - the dimensions of the matrix.

Methods

- **diag** signature `signature(x = "dgRMatrix")`: returns the diagonal of `x`
- **dim** signature `signature(x = "dgRMatrix")`: returns the dimensions of `x`
- **image** signature `signature(x = "dgRMatrix")`: plots an image of `x` using the `levelplot` function

See Also

the `RsparseMatrix` class, the virtual class of all sparse compressed row-oriented matrices, with its methods. The `dgCMatrix` class (column compressed sparse) is really preferred.

---

dgTMatrix-class

Sparse matrices in triplet form

Description

The "dgTMatrix" class is the class of sparse matrices stored as (possibly redundant) triplets. The internal representation is not at all unique, contrary to the one for class `dgCMatrix`.

Objects from the Class

Objects can be created by calls of the form `new("dgTMatrix", ...)`, but more typically via `spMatrix()` or `sparseMatrix(*, repr = "T")`.

Slots

- **i**: `integer` row indices of non-zero entries in 0-base, i.e., must be in 0:(nrow(.)-1).
- **j**: `integer` column indices of non-zero entries. Must be the same length as slot `i` and 0-based as well, i.e., in 0:(ncol(.)-1).
- **x**: `numeric` vector - the (non-zero) entry at position `(i,j)`. Must be the same length as slot `i`. If an index pair occurs more than once, the corresponding values of slot `x` are added to form the element of the matrix.
- **Dim**: Object of class "integer" of length 2 - the dimensions of the matrix.
Diagonal

Methods

+ signature(e1 = "dgTMatrix", e2 = "dgTMatrix")

image signature(x = "dgTMatrix"): plots an image of x using the levelplot function

t signature(x = "dgTMatrix"): returns the transpose of x

Note

Triplet matrices are a convenient form in which to construct sparse matrices after which they can be coerced to dgCMatrix objects.

Note that both new(.) and spMatrix constructors for "dgTMatrix" (and other "TsparseMatrix" classes) implicitly add x_k's that belong to identical (i_k, j_k) pairs.

However this means that a matrix typically can be stored in more than one possible "TsparseMatrix" representations. Use asUniqueT() in order to ensure uniqueness of the internal representation of such a matrix.

See Also

Class dgCMatrix or the superclasses dsparseMatrix and TsparseMatrix; asUniqueT.

Examples

```r
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)]))
str(mT)
mT[lower.tri(mT, diag=TRUE)] <- 0
mT
```

```r
## Triplett 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)
```

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).
Diagonal(n, x = NULL, names = FALSE)

.sparseDiagonal(n, x = NULL, uplo = "U", shape = "t", unitri = TRUE, kind, cols)
  .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 length(x) is used.

x    numeric or logical vector listing values for the diagonal entries, to be recycled as necessary. If NULL (the default), then the result is a unit diagonal matrix. 
  .sparseDiagonal() and friends ignore non-NULL x when kind = "n".

names    either logical TRUE or FALSE or then a character vector of length n. If true and names(x) is not NULL, use that as both row and column names for the resulting matrix. When a character vector, use it for both dimnames.

uplo    one of c("U", "L"), specifying the uplo slot of the result if the result is formally triangular of symmetric.

shape    one of c("t", "s", "g"), 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 unit triangular, i.e., with diag slot equal to "U" rather than "N".

kind    one of c("d", "l", "n"), 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 x is non-NULL; in that case the mode is determined by typeof(x).

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[, cols+1] rather than D, where D = Diagonal(n, x), and it is always "general" (i.e., shape is ignored).

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
diagonalMatrix-class

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

```r
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))) # sparse symmetric still
stopifnot(is(S, "dsCMatrix"))
(I4 <- .sparseDiagonal(4, shape="t"))# now (2012-10) unitriangular
stopifnot(I4@diag == "U", all(I4 == diag(4)))
```

diagonalMatrix-class  Class "diagonalMatrix" of Diagonal Matrices

Description

Class "diagonalMatrix" is the virtual class of all diagonal matrices.

Objects from the Class

A virtual Class: No objects may be created from it.

Slots

`diag`: character string, either "U" or "N", where "U" means ‘unit-diagonal’.

`Dim`: matrix dimension, and

`Dimnames`: the `dimnames`, a list, see the `Matrix` class description. Typically `list(NULL,NULL)` for diagonal matrices.

Extends

Class "sparseMatrix", directly.
Methods

These are just a subset of the signature for which defined methods. Currently, there are (too) many explicit methods defined in order to ensure efficient methods for diagonal matrices.

- **coerce** signature(from = "matrix", to = "diagonalMatrix"): ...
- **coerce** signature(from = "Matrix", to = "diagonalMatrix"): ...
- **coerce** signature(from = "diagonalMatrix", to = "generalMatrix"): ...
- **coerce** signature(from = "diagonalMatrix", to = "triangularMatrix"): ...
- **coerce** signature(from = "diagonalMatrix", to = "nMatrix"): ...
- **coerce** signature(from = "diagonalMatrix", to = "matrix"): ...
- **coerce** signature(from = "diagonalMatrix", to = "sparseVector"): ...
- **t** signature(x = "diagonalMatrix"): ...
  and many more methods

- **solve** signature(a = "diagonalMatrix", b, ...): is trivially implemented, of course; see also `solve-methods`.

- **which** signature(x = "nMatrix"), semantically equivalent to base function `which(x, arr.ind)`. "Math" signature(x = "diagonalMatrix"): all these group methods return a "diagonalMatrix", apart from `cumsum()` etc which return a vector also for base matrix.

- **signature(e1 = "ddiMatrix", e2="denseMatrix"):** arithmetic and other operators from the `Ops` group have a few dozen explicit method definitions, in order to keep the results diagonal in many cases, including the following:

  - `/ signature(e1 = "ddiMatrix", e2="denseMatrix"): the result is from class `ddiMatrix` which is typically very desirable. Note that when e2 contains off-diagonal zeros or `NA`s, we implicitly use `0/x = 0`, hence differing from traditional R arithmetic (where `0/0` → `NaN`), in order to preserve sparsity.

- **summary** (object = "diagonalMatrix"): Returns an object of S3 class "diagSummary" which is the summary of the vector object@x plus a simple heading, and an appropriate print method.

See Also

`Diagonal()` as constructor of these matrices, and `isDiagonal`, `ddiMatrix` and `ldiMatrix` are "actual" classes extending "diagonalMatrix".

Examples

```r
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))
```
Transform Triangular Matrices from Unit Triangular to General Triangular and Back

Description

Transform a triangular matrix x, i.e., of class triangularMatrix, from (internally!) unit triangular ("unitriangular") to "general" triangular (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 = 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, as per default, 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

"triangularMatrix", "dtCMatrix".
Examples

(T <- Diagonal(7) + triu(rpois(49, 1/4), 7, 7), k = 1))
(uT <- diagN2U(T)) # "unitriangular"
(t.u <- diagN2U(10*T)) # changes the diagonal!
stopifnot(all(T == uT), diag(t.u) == 1,
identical(T, diagU2N(uT))
T[upper.tri(T)] <- 5 # still "dtC"
T <- diagN2U(as(T, "triangularMatrix"))
dT <- as(T, "denseMatrix") # (unitriangular)
dT.n <- diagU2N(dT, checkDense = TRUE)
sT.n <- diagU2N(dT)
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))

dimScale

Scale the Rows and Columns of a Matrix

Description

dimScale, rowScale, and colScale implement $D_1 \times x \times D_2$, $D \times x$, and $x \times D$ for diagonal matrices $D_1$, $D_2$, and $D$ with diagonal entries $d_1$, $d_2$, 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
(mmxx <- D %*% x %*% D) # symmetry and 'dimnames' lost
stopifnot(identical(unname(as(scx, "generalMatrix")), mmxx))

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

---

dMatrix-class

(Virtual) Class "dMatrix" of "double" Matrices

Description

The dMatrix class is a virtual class contained by all actual classes of numeric matrices in the Matrix package. Similarly, all the actual classes of logical matrices inherit from the lMatrix class.

Slots

Common to all matrix object in the package:

Dim: Object of class "integer" - the dimensions of the matrix - must be an integer vector with exactly two non-negative values.

Dimnames: list of length two; each component containing NULL or a character vector length equal the corresponding Dim element.

Methods

There are (relatively simple) group methods (see, e.g., Arith)

Arith signature(e1 = "dMatrix", e2 = "dMatrix"): ...
Arith signature(e1 = "dMatrix", e2 = "numeric"): ...
Arith signature(e1 = "numeric", e2 = "dMatrix"): ...

Math signature(x = "dMatrix"): ...
Math2 signature(x = "dMatrix", digits = "numeric"): this group contains round() and signif().
Compare signature(e1 = "numeric", e2 = "dMatrix"): ...
Compare signature(e1 = "dMatrix", e2 = "numeric"): ...

Summary signature(x = "dMatrix"): The "Summary" group contains the seven functions max(), min(), range(), prod(), sum(), any(), and all().
The following methods are also defined for all double matrices:

- **expm** signature(x = "dMatrix"): computes the “Matrix Exponential”, see `expm`.
- **zapsmall** signature(x = "dMatrix"): ...

The following methods are defined for all logical matrices:

- **which** signature(x = "lsparseMatrix") and many other subclasses of "lMatrix": as the base function `which(x, arr.ind)` returns the indices of the TRUE entries in x; if arr.ind is true, as a 2-column matrix of row and column indices. Since Matrix version 1.2-9, if useNames is true, as by default, with dimnames, the same as base::which.

### See Also

- The nonzero-pattern matrix class **nMatrix**, which can be used to store non-NA logical matrices even more compactly.
- The numeric matrix classes **dgeMatrix**, **dgCMatrix**, and **Matrix**.

### Examples

```r
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))
```

---

**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

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

### Arguments

- **x**: a typically sparse matrix; internally coerced to either "dgCMatrix" or "dtCMatrix".
- **nAns**: an integer specifying the length of the resulting list. Must be 2, 4, or 6.
- **seed**: 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 random permutation (using R’s RNG, seed, etc).
**dmperm**

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

**Author(s)**

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

**References**


**See Also**

Schur, the class of permutation matrices; “pMatrix”.

**Examples**

```r
set.seed(17)
(S9 <- rsparsematrix(9, 9, nnz = 10, symmetric=TRUE)) # dSymMatrix
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
```

```r
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])
```

```r
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))
}

**dpoMatrix-class**

**Positive Semi-definite Dense (Packed \ Non-packed) Numeric Matrices**

**Description**

- The "dpoMatrix" class is the class of positive-semidefinite symmetric matrices in nonpacked storage.
- The "dppMatrix" class is the same except in packed storage. Only the upper triangle or the lower triangle is required to be available.
- The "corMatrix" and "pcorMatrix" classes represent correlation matrices. They extend "dpoMatrix" and "dppMatrix", respectively, with an additional slot sd allowing restoration of the original covariance matrix.

**Objects from the Class**

Objects can be created by calls of the form `new("dpoMatrix", ...)` or from `crossprod` applied to an "dgeMatrix" object.

**Slots**

- uplo: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular.
- x: Object of class "numeric". The numeric values that constitute the matrix, stored in column-major order.
- Dim: Object of class "integer". The dimensions of the matrix which must be a two-element vector of non-negative integers.
- Dimnames: inherited from class "Matrix"
- factors: Object of class "list". A named list of factorizations that have been computed for the matrix.
- sd: (for "corMatrix" and "pcorMatrix") a numeric vector of length n containing the (original) \( \sqrt{\text{var}(.)} \) entries which allow reconstruction of a covariance matrix from the correlation matrix.

**Extends**

Class "dsyMatrix", directly.
Classes "dgeMatrix", "symmetricMatrix", and many more by class "dsyMatrix".
Methods

**chol** signature(x = "dpoMatrix"): Returns (and stores) the Cholesky decomposition of x, see chol.

**determinant** signature(x = "dpoMatrix"): Returns the determinant of x, via chol(x), see above.

**rcond** signature(x = "dpoMatrix", norm = "character"): Returns (and stores) the reciprocal of the condition number of x. The norm can be "O" for the one-norm (the default) or "1" for the infinity-norm. For symmetric matrices the result does not depend on the norm.

**solve** signature(a = "dpoMatrix", b = "....") and

**solve** signature(a = "dppMatrix", b = "....") work via the Cholesky composition, see also the Matrix solve-methods.

**Arith** signature(e1 = "dpoMatrix", e2 = "numeric") (and quite a few other signatures): The result of ("elementwise" defined) arithmetic operations is typically not positive-definite anymore. The only exceptions, currently, are multiplications, divisions or additions with positive length(.) == 1 numbers (or logicals).

Note
Currently the validity methods for these classes such as getValidity(getClass("dpoMatrix")) for efficiency reasons only check the diagonal entries of the matrix – they may not be negative. This is only necessary but not sufficient for a symmetric matrix to be positive semi-definite.

A more reliable (but often more expensive) check for positive semi-definiteness would look at the signs of diag(BunchKaufman(.)) (with some tolerance for very small negative values), and for (strict) positive definiteness at something like !inherits(tryCatch(chol(.), error=identity), "error"). Indeed, when coercing to these classes, a version of Cholesky() or chol() is typically used, e.g., see selectMethod("coerce", c(from="dsyMatrix", to="dpoMatrix")).

See Also
Classes dsyMatrix and dgeMatrix; further, Matrix, rcond, chol, solve, crossprod.

Examples

```r
h6 <- Hilbert(6)
rcond(h6)
str(h6)
h6 * 27720 # is `integer'
solve(h6)
str(hp6 <- pack(h6))
```

```r
### 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$Cholesky),
  all(abs(crossprod(as(chch, "dtrMatrix")) - ch6) < 1e-10))
```
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 as(x, "CsparseMatrix") instead, notably in the tol = 0 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 give.Csparse 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

```r
(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)

---

**dsCMatrix-class**

**Numeric Symmetric Sparse (column compressed) Matrices**

**Description**

The dsCMatrix class is a class of symmetric, sparse numeric matrices in the compressed, column-oriented format. In this implementation the non-zero elements in the columns are sorted into increasing row order.

The dsTMatrix class is the class of symmetric, sparse numeric matrices in triplet format.

**Objects from the Class**

Objects can be created by calls of the form `new("dsCMatrix", ...)` or `new("dsTMatrix", ...)`, or automatically via e.g., `as(*, "symmetricMatrix"),` or (for dsCMatrix) also from `Matrix(.)`.

Creation “from scratch” most efficiently happens via `sparseMatrix(*, symmetric=TRUE).

**Slots**

- uplo: A character object indicating if the upper triangle ("U") or the lower triangle ("L") is stored.
- i: Object of class "integer" of length \(nnZ\) (half number of non-zero elements). These are the row numbers for each non-zero element in the lower triangle of the matrix.
- p: (only in class "dsCMatrix"): an integer vector for providing pointers, one for each column, see the detailed description in `CsparseMatrix`.
- j: (only in class "dsTMatrix"): Object of class "integer" of length \(nnZ\) (as \(i\)). These are the column numbers for each non-zero element in the lower triangle of the matrix.
- x: Object of class "numeric" of length \(nnZ\) – the non-zero elements of the matrix (to be duplicated for full matrix).
- factors: Object of class "list" - a list of factorizations of the matrix.
- Dim: Object of class "integer" - the dimensions of the matrix - must be an integer vector with exactly two non-negative values.

**Extends**

Both classes extend classes and `symmetricMatrix` `dsparseMatrix` directly; dsCMatrix further directly extends `CsparseMatrix`, where dsTMatrix does `TsparseMatrix`. 
dsparseMatrix-class

Methods

solve signature(a = "dsCMatrix", b = "..."): x <- solve(a,b) solves \( Ax = b \) for \( x \); see solve-methods.

chol signature(x = "dsCMatrix", pivot = "logical"): Returns (and stores) the Cholesky decomposition of \( x \), see chol.

Cholesky signature(A = "dsCMatrix", ...): Computes more flexibly Cholesky decompositions, see Cholesky.

determinant signature(x = "dsCMatrix", logarithm = "missing"): Evaluate the determinant of \( x \) on the logarithm scale. This creates and stores the Cholesky factorization.

determinant signature(x = "dsCMatrix", logarithm = "logical"): Evaluate the determinant of \( x \) on the logarithm scale or not, according to the logarithm argument. This creates and stores the Cholesky factorization.

t signature(x = "dsCMatrix"): Transpose. As for all symmetric matrices, a matrix for which the upper triangle is stored produces a matrix for which the lower triangle is stored and vice versa, i.e., the uplo slot is swapped, and the row and column indices are interchanged.

t signature(x = "dsTMatrix"): Transpose. The uplo slot is swapped from "U" to "L" or vice versa, as for a "dsCMatrix", see above.

See Also

Classes dgCMatrix, dgTMatrix, dgeMatrix and those mentioned above.

Examples

```r
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"
```

dsparseMatrix-class

Virtual Class "dsparseMatrix" of Numeric Sparse Matrices

Description

The Class "dsparseMatrix" is the virtual (super) class of all numeric sparse matrices.

Slots

Dim: the matrix dimension, see class "Matrix".
Dimnames: see the "Matrix" class.
x: a numeric vector containing the (non-zero) matrix entries.
dsRMatrix-class

67

Extends

Class "dMatrix" and "sparseMatrix", directly.
Class "Matrix", by the above classes.

See Also

the documentation of the (non virtual) sub classes, see showClass("dsparseMatrix"); in particular, dgTMatrix, dgCMatrix, and dgRMatrix.

Examples

showClass("dsparseMatrix")

--------------------------------

dsRMatrix-class  Symmetric Sparse Compressed Row Matrices
--------------------------------

Description

The dsRMatrix class is a class of symmetric, sparse matrices in the compressed, row-oriented format. In this implementation the non-zero elements in the rows are sorted into increasing column order.

Objects from the Class

These "..RMatrix" classes are currently still mostly unimplemented!
Objects can be created by calls of the form new("dsRMatrix", ...).

Slots

uplo: A character object indicating if the upper triangle ("U") or the lower triangle ("L") is stored.
At present only the lower triangle form is allowed.

j: Object of class "integer" of length nnzero (number of non-zero elements). These are the row numbers for each non-zero element in the matrix.

p: Object of class "integer" of pointers, one for each row, to the initial (zero-based) index of elements in the row.

factors: Object of class "list" - a list of factorizations of the matrix.

x: Object of class "numeric" - the non-zero elements of the matrix.

Dim: Object of class "integer" - the dimensions of the matrix - must be an integer vector with exactly two non-negative values.

Dimnames: List of length two, see Matrix.

Extends

Classes RsparseMatrix, dsparseMatrix and symmetricMatrix, directly.
Class "dMatrix", by class "dsparseMatrix", class "sparseMatrix", by class "dsparseMatrix" or "RsparseMatrix"; class "compMatrix" by class "symmetricMatrix" and of course, class "Matrix".
Methods

forceSymmetric signature(x = "dsRMatrix", uplo = "missing"): a trivial method just returning x.

forceSymmetric signature(x = "dsRMatrix", uplo = "character"): if uplo == x@uplo, this trivially returns x; otherwise t(x).

See Also

the classes \texttt{dgCMatrix}, \texttt{dgTMatrix}, and \texttt{dgeMatrix}.

Examples

\begin{verbatim}
(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
\end{verbatim}

\begin{description}
\item[dsyMatrix-class] Symmetric Dense (Packed or Unpacked) Numeric Matrices
\end{description}

Description

- The "dsyMatrix" class is the class of symmetric, dense matrices in \textit{non-packed} storage and
- "dspMatrix" is the class of symmetric dense matrices in \textit{packed} storage, see \texttt{pack()}. Only the upper triangle or the lower triangle is stored.

Objects from the Class

Objects can be created by calls of the form \texttt{new("dsyMatrix", ...)} or \texttt{new("dspMatrix", ...)}, respectively.

Slots

\begin{itemize}
\item \texttt{uplo}: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular.
\item \texttt{x}: Object of class "numeric". The numeric values that constitute the matrix, stored in column-major order.
\item \texttt{Dim,Dimnames}: The dimension (a length-2 "integer") and corresponding names (or NULL), see the \texttt{Matrix}.
\item \texttt{factors}: Object of class "list". A named list of factorizations that have been computed for the matrix.
\end{itemize}
"dsyMatrix" extends class "dgeMatrix", directly, whereas "dsp2Matrix" extends class "ddenseMatrix", directly.

Both extend class "symmetricMatrix", directly, and class "Matrix" and others, indirectly, use `showClass("dsyMatrix")`, e.g., for details.

Methods

- **norm** signature(x = "dspMatrix", type = "character"), or x = "dsyMatrix" or type = "missing": Computes the matrix norm of the desired type, see, `norm`
- **rcond** signature(x = "dspMatrix", type = "character"), or x = "dsyMatrix" or type = "missing": Computes the reciprocal condition number, `rcond()`
- **solve** signature(a = "dspMatrix", b = "...."), and
  - **solve** signature(a = "dsyMatrix", b = "...."): x <- solve(a, b) solves \(Ax = b\) for \(x\); see `solve-methods`
- **t** signature(x = "dsyMatrix"): Transpose; swaps from upper triangular to lower triangular storage, i.e., the uplo slot from "U" to "L" or vice versa, the same as for all symmetric matrices.

See Also

The positive (Semi-)definite dense (packed or non-packed numeric matrix classes `dpoMatrix`, `dppMatrix` and `corMatrix`,

Classes `dgeMatrix` and `Matrix`; `solve`, `norm`, `rcond`, `t`

Examples

```r
## 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
```

```r
## 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
```
dtCMatrix-class

Triangular, (compressed) sparse column matrices

Description

The "dtCMatrix" class is a class of triangular, sparse matrices in the compressed, column-oriented format. In this implementation the non-zero elements in the columns are sorted into increasing row order.

The "dtTMatrix" class is a class of triangular, sparse matrices in triplet format.

Objects from the Class

Objects can be created by calls of the form \texttt{new("dtCMatrix", ...)} or calls of the form \texttt{new("dtTMatrix", ...)}, but more typically automatically via \texttt{Matrix()} or coercions such as \texttt{as(x, "triangularMatrix")}.

Slots

uplo: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular.

diag: Object of class "character". Must be either "U", for unit triangular (diagonal is all ones), or "N"; see \texttt{triangularMatrix}.

p: (only present in "dtCMatrix";) an \texttt{integer} vector for providing pointers, one for each column, see the detailed description in \texttt{CsparseMatrix}.

i: Object of class "integer" of length nnzero (number of non-zero elements). These are the row numbers for each non-zero element in the matrix.

j: Object of class "integer" of length nnzero (number of non-zero elements). These are the column numbers for each non-zero element in the matrix. (Only present in the dtTMatrix class.)

x: Object of class "numeric" - the non-zero elements of the matrix.

Dim,Dimnames: The dimension (a length-2 "integer") and corresponding names (or NULL), inherited from the \texttt{Matrix}, see there.

Extends

Class "dgCMatrix", directly. Class "triangularMatrix", directly. Class "dMatrix", "sparseMatrix", and more by class "dgCMatrix" etc, see the examples.

Methods

\textbf{solve} signature(a = "dtCMatrix", b = "...."): sparse triangular solve (aka “backsolve” or “forwardsolve”), see \texttt{solve-methods}.

\textbf{t} signature(x = "dtCMatrix"): returns the transpose of \texttt{x}

\textbf{t} signature(x = "dtTMatrix"): returns the transpose of \texttt{x}

See Also

Classes \texttt{dgCMatrix}, \texttt{dgTMatrix}, \texttt{dgeMatrix}, and \texttt{dtrMatrix}.
Examples

```r
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))
tt. <- solve(it1)
all(abs(tt - 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)))
)
```

dtpMatrix-class

Packed Triangular Dense Matrices - "dtpMatrix"

Description

The "dtpMatrix" class is the class of triangular, dense, numeric matrices in packed storage. The "dtrMatrix" class is the same except in nonpacked storage.

Objects from the Class

Objects can be created by calls of the form `new("dtpMatrix", ...)` or by coercion from other classes of matrices.
Slots

duplo: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular.
diag: Object of class "character". Must be either "U", for unit triangular (diagonal is all ones),
or "N"; see triangularMatrix.
x: Object of class "numeric". The numeric values that constitute the matrix, stored in column-major
order. For a packed square matrix of dimension $d \times d$, length(x) is of length $d(d+1)/2$
(also when diag == "U"!).

Dim, Dimnames: The dimension (a length-2 "integer") and corresponding names (or NULL), inherited
from the Matrix, see there.

Extends

Class "ddenseMatrix", directly. Class "triangularMatrix", directly. Class "dMatrix" and more
by class "ddenseMatrix" etc, see the examples.

Methods

%*% signature(x = "dtpMatrix", y = "dgeMatrix"): Matrix multiplication; ditto for several
other signature combinations, see showMethods("%*%", class = "dtpMatrix").
determinant signature(x = "dtpMatrix", logarithm = "logical"): the determinant(x) trivially is prod(diag(x)), but computed on log scale to prevent over- and underflow.
diag signature(x = "dtpMatrix"): ...

norm signature(x = "dtpMatrix", type = "character"): ...

recond signature(x = "dtpMatrix", norm = "character"): ...
solve signature(a = "dtpMatrix", b = "..."): efficiently using internal backsolve or forward-
solve, see solve-methods.

t signature(x = "dtpMatrix"): t(x) remains a "dtpMatrix", lower triangular if x is upper tri-
angular, and vice versa.

See Also

Class dtrMatrix

Examples

showClass("dtrMatrix")

example("dtrMatrix-class", echo=FALSE)
(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,
idemical(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,
idemical(t(ip1), solve(pp)), is(ip1, "dtpMatrix"),
all.equal(as(solve(p1,p1), "diagonalMatrix"), Diagonal(2)))
Description

The \texttt{dtRMatrix} class is a class of triangular, sparse matrices in the compressed, row-oriented format. In this implementation the non-zero elements in the rows are sorted into increasing column order.

Objects from the Class

This class is currently still mostly unimplemented!

Objects can be created by calls of the form \texttt{new("dtRMatrix", ...)}.

Slots

- \texttt{uplo}: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular. At present only the lower triangle form is allowed.
- \texttt{diag}: Object of class "character". Must be either "U", for unit triangular (diagonal is all ones), or "N"; see \texttt{triangularMatrix}.
- \texttt{j}: Object of class "integer" of length \texttt{nnzero(.)} (number of non-zero elements). These are the row numbers for each non-zero element in the matrix.
- \texttt{p}: Object of class "integer" of pointers, one for each row, to the initial (zero-based) index of elements in the row. (Only present in the \texttt{dsRMatrix} class.)
- \texttt{x}: Object of class "numeric" - the non-zero elements of the matrix.
- \texttt{Dim}: The dimension (a length-2 "integer")
- \texttt{Dimnames}: corresponding names (or \texttt{NULL}), inherited from the \texttt{Matrix}, see there.

Extends

Class "\texttt{dgRMatrix}", directly. Class "\texttt{dsparseMatrix}", by class "\texttt{dgRMatrix}". Class "\texttt{dMatrix}", by class "\texttt{dgRMatrix}". Class "\texttt{sparseMatrix}", by class "\texttt{dgRMatrix}". Class "\texttt{Matrix}", by class "\texttt{dgRMatrix}".

Methods

No methods currently with class "\texttt{dsRMatrix}" in the signature.

See Also

Classes \texttt{dgCMatrix}, \texttt{dgTMatrix}, \texttt{dgeMatrix}

Examples

```r
(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")) # --&gt; dtRMatrix
```
dtrMatrix-class

Triangular, dense, numeric matrices

description

The "dtrMatrix" class is the class of triangular, dense, numeric matrices in nonpacked storage. The "dtpMatrix" class is the same except in packed storage, see pack().

Objects from the Class

Objects can be created by calls of the form new("dtrMatrix", ...).

Slots

uplo: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular.
diag: Object of class "character". Must be either "U", for unit triangular (diagonal is all ones), or "N"; see triangularMatrix.
x: Object of class "numeric". The numeric values that constitute the matrix, stored in column-major order.
Dim: Object of class "integer". The dimensions of the matrix which must be a two-element vector of non-negative integers.

Extends

Class "ddenseMatrix", directly. Class "triangularMatrix", directly. Class "Matrix" and others, by class "ddenseMatrix".

Methods

Among others (such as matrix products, e.g. ?crossprod-methods),

norm signature(x = "dtrMatrix", type = "character"): ..
recondition signature(x = "dtrMatrix", norm = "character"): ..
solve signature(a = "dtrMatrix", b = "... "): efficiently use a "forwardsolve" or backsolve for a lower or upper triangular matrix, respectively, see also solve-methods.

+, -, *, ..., ==, >=,... all the Ops group methods are available. When applied to two triangular matrices, these return a triangular matrix when easily possible.

See Also

Classes ddenseMatrix, dtpMatrix, triangularMatrix

Examples

(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
})

---

**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 \texttt{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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>class(x)</td>
<td>\texttt{which}</td>
</tr>
<tr>
<td>Schur</td>
<td>\texttt{c(&quot;Q&quot;, &quot;T&quot;, &quot;Q.&quot;)}</td>
</tr>
<tr>
<td>denseLU</td>
<td>\texttt{c(&quot;P1&quot;, &quot;P1.&quot;, &quot;L&quot;, &quot;U&quot;)}</td>
</tr>
<tr>
<td>sparseLU</td>
<td>\texttt{c(&quot;P1&quot;, &quot;P1.&quot;, &quot;P2&quot;, &quot;P2.&quot;, &quot;L&quot;, &quot;U&quot;)}</td>
</tr>
<tr>
<td>sparseQR</td>
<td>\texttt{c(&quot;P1&quot;, &quot;P1.&quot;, &quot;P2&quot;, &quot;P2.&quot;, &quot;Q&quot;, &quot;Q1&quot;, &quot;R&quot;, &quot;R1&quot;)}</td>
</tr>
<tr>
<td>BunchKaufman, pBunchKaufman</td>
<td>\texttt{c(&quot;U&quot;, &quot;DU&quot;, &quot;U.&quot;, &quot;L&quot;, &quot;DL&quot;, &quot;L.&quot;})</td>
</tr>
<tr>
<td>Cholesky, pCholesky</td>
<td>\texttt{c(&quot;P1&quot;, &quot;P1.&quot;, &quot;L1&quot;, &quot;D&quot;, &quot;L1.&quot;, &quot;L&quot;, &quot;L.&quot;)}</td>
</tr>
<tr>
<td>CHMsimp1, CHMsimp1</td>
<td>\texttt{c(&quot;P1&quot;, &quot;P1.&quot;, &quot;L1&quot;, &quot;D&quot;, &quot;L1.&quot;, &quot;L&quot;, &quot;L.&quot;)}</td>
</tr>
</tbody>
</table>

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

\texttt{expand2 signature(x = \texttt{"CHMsimp1"}):} as CHMsimp1, but the triangular factors are stored as \texttt{dgCMatrix}.

\texttt{expand2 signature(x = \texttt{"pCholesky"}):} expands the factorization \( A = L_1DL_1^T = LL' \) as \texttt{list(L1, D, L1, P1)} (the default) or as \texttt{list(L, L, P1)}, depending on optional logical argument \texttt{LDL}. \texttt{P1} and \texttt{P1} are \texttt{pMatrix}, \texttt{L1}, \texttt{L1}, and \texttt{L} are \texttt{dtrMatrix} or \texttt{dtbMatrix}, and \texttt{D} is a \texttt{ddiMatrix}.

\texttt{expand2 signature(x = \texttt{"pBunchKaufman"}):} expands the factorization \( A = UD_1U = LD_1L' \) where \( U = \prod_{i=1}^{b_1} P_i U_k \) and \( L = \prod_{i=1}^{b_2} P_i L_k \) as \texttt{list(U, D, U, L)} or \texttt{list(L, D, L, L)}, depending on \texttt{x@uplo}. If optional argument complete is \texttt{TRUE}, then an unnamed list giving the full expansion with \( 2b_U + 1 \) or \( 2b_L + 1 \) matrix factors is returned instead. \texttt{P}_i are represented as \texttt{pMatrix}, \texttt{U}_k and \texttt{L}_k are represented as \texttt{dtCMatrix}, and \texttt{D}_U and \texttt{D}_L are represented as \texttt{dsCMatrix}.

\texttt{expand2 signature(x = \texttt{"Schur"}):} expands the factorization \( A = QTQ' \) as \texttt{list(Q, T, Q., .)}. \texttt{Q} and \texttt{Q} are \texttt{x@Q} and \texttt{t(x@Q)} modulo \texttt{Dimnames}, and \texttt{T} is \texttt{x@T}.

\texttt{expand2 signature(x = \texttt{"sparseLU"}):} expands the factorization \( A = P_1LU \) as \texttt{list(P1, L, U, P2)}. \texttt{P1} and \texttt{P2} are \texttt{pMatrix}, and \texttt{L} and \texttt{U} are \texttt{dtrMatrix} if square and \texttt{dgeMatrix} otherwise.

\texttt{expand2 signature(x = \texttt{"denseLU"}):} expands the factorization \( A = P_1LU \) as \texttt{list(P1, L, U, P2)}. \texttt{P1} is a \texttt{pMatrix}, and \texttt{L} and \texttt{U} are \texttt{dtrMatrix} if square and \texttt{dgeMatrix} otherwise.

\texttt{expand2 signature(x = \texttt{"pCholesky"}):} expands the factorization \( A = P_1QRP' \) as \texttt{list(P1, Q, R, P2)} or \texttt{list(P1, Q1, R1, P2)}. depending on optional logical argument complete. \texttt{P1} and \texttt{P2} are \texttt{pMatrix}, \texttt{Q} and \texttt{Q1} are \texttt{dgeMatrix}, \texttt{R} is a \texttt{dgCMatrix}, and \texttt{R1} is a \texttt{dtrMatrix}.

\texttt{expand signature(x = \texttt{"CHMfactor"}):} as expand2, but returning \texttt{list(P, L)}. expand(x)[["P"]], and expand2(x)[["P1"]]), represent the same permutation matrix \texttt{P}_i 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 compMatrix of factorizable matrices.

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

---

**expm-methods**

**Matrix Exponential**

**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 $\expm(A) = I + A + A^2/2! + A^3/3! + \ldots$ (although this is definitely not the way to compute it). The method for the `dgeMatrix` class uses Ward’s diagonal Padé 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 &lt;A.S.Hodel@Eng.Auburn.EDU&gt;. A bug in there has been fixed by Martin Maechler.

References

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


for historical reference mostly:


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

```r
(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 \texttt{sparseMatrix} objects to one of these formats.

Usage

\begin{verbatim}
readHB(file)
readMM(file)
writeMM(obj, file, ...)
\end{verbatim}

Arguments

\begin{verbatim}
obj a real sparse matrix
file for \texttt{writeMM} - the name of the file to be written. For \texttt{readHB} and \texttt{readMM} 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 ".rua" or ".rsa". Those storing matrices in the MatrixMarket format usually end in ".mtx".
Alternatively, \texttt{readHB} and \texttt{readMM} accept connection objects.
...
optional additional arguments. Currently none are used in any methods.
\end{verbatim}

Value

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

Note

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

Note that these formats do \textit{not} know anything about \texttt{dimnames}, hence these are dropped by \texttt{writeMM()}. A very simple way to export small sparse matrices \texttt{S}, is to use \texttt{summary(S)} which returns a \texttt{data.frame} with columns \texttt{i}, \texttt{j}, and possibly \texttt{x}, see \texttt{summary} in \texttt{sparseMatrix-class}, and an example below.

References

\begin{verbatim}
https://math.nist.gov/MatrixMarket/
https://sparse.tamu.edu/
\end{verbatim}

Examples

\begin{verbatim}
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
\end{verbatim}
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")
data(CAex, package = "Matrix")

## very simple export - in triplet format - to text file:
writeMM(KNex$mm, file=MMfile)

## 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))

---

facmul-methods

Multiplication by Factors from Matrix Factorizations

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

x  
a MatrixFactorization object.

factor  
a character string indicating a factor in the factorization represented by x, typically an element of names(expand2(x, ...)).

y  
a matrix or vector to be multiplied on the left or right by the factor or its transpose.

trans  
a logical indicating if the transpose of the factor should be used, rather than the factor itself.

left  
a logical indicating if the y should be multiplied on the left by the factor, rather than on the right.

...  
further arguments passed to or from methods.
Details

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

Value

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

Examples

```r
## 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)
```

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, ...)  
.M2tri(from, ...)  
.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")
.ind2dense(from, kind = "n")
.m2dense(from, class = ".ge", uplo = "U", diag = "N", trans = FALSE)

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

.tCRT(x, lazy = TRUE)
.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, matrix, or vector.

kind  
a string ("." , ", " , "n" , "l" , or "d") specifying the "kind" of the result. "."  
indicates that the kind of from should be preserved. ","  
is equivalent to "z"  
if from is complex and to "d" otherwise. "."  
indicates that the result should  
inherit from nMatrix or nsparseVector (and so on).

shape  
a string ("." , "g" , "s" , or "t") specifying the "shape" of the result. "."  
indicates that the shape of from should be preserved. "."  
indicates that the result should  
inherit from generalMatrix (and so on).

repr  
a string ("." , "C" , "R" , or "T") specifying the sparse representation of the result. "."  
is accepted only by .ind2sparse and indicates the most efficient representation, which is "C" ("R") for margin = 2 (1). "."  
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 ("U" or "L") 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.

diag  
a string ("N" or "U") 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.

trans  
a logical indicating if the result should be a 1-row matrix rather than a 1-column matrix 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 "^[.nld](ge|sy|tr|sp|tp)" for \texttt{.m2dense} and "^[.nld][gst][CRT]" for \texttt{.m2sparse}, where "." in the first position is equivalent to "l" for logical arguments and "d" for numeric arguments.

... optional arguments passed to \texttt{isSymmetric} or \texttt{isTriangular}.

lazy

a logical indicating if the transpose should be constructed with minimal allocation, but possibly \textit{without} preserving representation.

Chx

optionally, the \texttt{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 \texttt{lu-methods}.

order

see \texttt{qr-methods}.

check

a logical indicating if the first argument should be tested for inheritance from \texttt{dgCMatrix} and coerced if necessary. Set to \texttt{FALSE} for speed only if it is known to already inherit from \texttt{dgCMatrix}.

object

a Cholesky factorization inheriting from virtual class \texttt{CHMfactor}, almost always the result of a call to generic function \texttt{Cholesky}.

parent

an object of class \texttt{dsCMatrix} or class \texttt{dgCMatrix}.

mult

a numeric vector of positive length. Only the first element is used, and that must be finite.

\section*{Details}

Functions with names of the form \texttt{.<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:

\texttt{M Matrix}

\texttt{V sparseVector}

\texttt{m matrix}

\texttt{v vector}

\texttt{dense denseMatrix}

\texttt{unpacked unpackedMatrix}

\texttt{packed packedMatrix}

\texttt{sparse CsparseMatrix,RsparseMatrix,or TsparseMatrix}

\texttt{C CsparseMatrix}

\texttt{R RsparseMatrix}

\texttt{T TsparseMatrix}

\texttt{gen generalMatrix}

\texttt{sym symmetricMatrix}

\texttt{tri triangularMatrix}

\texttt{diag diagonalMatrix}

\texttt{ind indMatrix}

Abbreviations should be seen as a guide, rather than as an exact description of behaviour. Notably, \texttt{.m2dense}, \texttt{.m2sparse}, and \texttt{.m2V} accept vectors that are not matrices.
.tCRT(x): If lazy = TRUE, then .tCRT constructs the transpose of x using the most efficient representation, which for ‘CRT’ is ‘RCT’. If lazy = FALSE, then .tCRT preserves the representation of x, behaving as the corresponding methods for generic function t.

.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(parent) + mult[1] * diag(nrow(parent)). i.e., F(parent) plus mult[1] times the identity matrix, where F = identity if parent is a dsCMatrix and F = tcrossprod if parent is a dgCMatrix. The nonzero pattern of F(parent) must match that of S if object = Cholesky(S, ...).

Examples

```r
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(.M2tri (D.), as(D., "triangularMatrix"))
  identical(.M2sym (D.), as(D., "symmetricMatrix"))
  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(.diag2dense(D.0, ",", ",", "t", TRUE), "C"),
    dense2sparse(.diag2dense(D.0, ",", ",", "s", FALSE)),
    .sparse2dense(.M2gen(.diag2sparse(D.0, ",", ",", "s", "T"))))
  identical(S.,
    .M2msparse(S., ",sR"))
  identical(S. * lower.tri(S.) + diag(1, 6L),
    .M2m(.m2dense (S., ",tr", ",L", ",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))
})

forceSymmetric-methods

**Force a Matrix to 'symmetricMatrix' Without Symmetry Checks**

**Description**

Force a square matrix x to a **symmetricMatrix**, **without** a symmetry check as it would be applied for as(x, "symmetricMatrix").

**Usage**

```r
forceSymmetric(x, uplo)
```

**Arguments**

- `x`: any square matrix (of numbers), either ""traditional"" (matrix) or inheriting from Matrix.
- `uplo`: optional string, "U" or "L" indicating which "triangle" half of x should determine the result. The default is "U" unless x already has a uplo slot (i.e., when it is symmetricMatrix, or triangularMatrix), where the default will be x@uplo.

**Value**

a square matrix inheriting from class symmetricMatrix.

**See Also**

symmpart for the symmetric part of a matrix, or the coercions as(x, <symmetricMatrix class>).

**Examples**

```r
## 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)

formatSparseM

---

**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**

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

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

**Arguments**

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

```r
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  Class “generalMatrix” of General Matrices

description

Virtual class of “general” matrices; i.e., matrices that do not have a known property such as symmetric, triangular, or diagonal.

Objects from the Class

A virtual Class: No objects may be created from it.

Slots

- factors
- Dim

Dimnames: all slots inherited from `compMatrix`; see its description.

Extends

Class “compMatrix”, directly. Class "Matrix", by class "compMatrix".
See Also

Classes `compMatrix`, and the non-general virtual classes: `symmetricMatrix, triangularMatrix, diagonalMatrix`.

---

**Hilbert**

*Generate a Hilbert matrix*

**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**

```r
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**

```r
Hilbert(6)
```

---

**image-methods**

*Methods for image() in Package 'Matrix'*

**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

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

See Also

\texttt{levelplot}, and \texttt{print.trellis} from package \texttt{lattice}.

Examples

```r
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 = 0.01)
## See \texttt{?print.trellis} --- place two image() plots side by side:
print(image(cCA, main= "Cholesky(crossprod(CAex), Imult = 0.01)"),
      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. \texttt{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(1,100), 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-\texttt{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)

index-class

Virtual Class "index" - Simple Class for Matrix Indices

Description

The class "index" is a virtual class used for indices (in signatures) for matrix indexing and subassignment of Matrix matrices.

In fact, it is currently implemented as a simple class union (setClassUnion) of "numeric", "logical" and "character".

Objects from the Class

Since it is a virtual Class, no objects may be created from it.

See Also

[-methods, and
Subassign-methods, also for examples.

Examples

showClass("index")

indMatrix-class

Index Matrices

Description

The indMatrix class is the class of row and column index matrices, stored as 1-based integer index vectors. A row (column) index matrix is a matrix whose rows (columns) are standard unit vectors. Such matrices are useful when mapping observations to discrete sets of covariate values.

Multiplying a matrix on the left by a row index matrix is equivalent to indexing its rows, i.e., sampling the rows "with replacement". Analogously, multiplying a matrix on the right by a column index matrix is equivalent to indexing its columns. Indeed, such products are implemented in Matrix as indexing operations; see 'Details' below.

A matrix whose rows and columns are standard unit vectors is called a permutation matrix. This special case is designated by the pMatrix class, a direct subclass of indMatrix.
Details

The transpose of an index matrix is an index matrix with identical perm but opposite margin. Hence the transpose of a row index matrix is a column index matrix, and vice versa.

The cross product of a row index matrix R and itself is a diagonal matrix whose diagonal entries are the number of entries in each column of R.

Given a row index matrix R with perm slot p, a column index matrix C with perm slot q, and a matrix M with conformable dimensions, we have

\[
\begin{align*}
RM &= R \cdot M = M[p,] \\
MC &= M \cdot C = M[ , q] \\
C' M &= \text{crossprod}(C, M) = M[q,] \\
MR' &= \text{tcrossprod}(M, R) = M[ , p] \\
R'R &= \text{crossprod}(R) = \text{Diagonal}(x=\text{tabulate}(p, \text{ncol}(R))) \\
CC' &= \text{tcrossprod}(C) = \text{Diagonal}(x=\text{tabulate}(q, \text{nrow}(C)))
\end{align*}
\]

Operations on index matrices that result in index matrices will accordingly return an \texttt{indMatrix}. These include products of two column index matrices and (equivalently) column-indexing of a column index matrix (when dimensions are not dropped). Most other operations on \texttt{indMatrix} treat them as sparse nonzero pattern matrices (i.e., inheriting from virtual class \texttt{nsparseMatrix}). Hence vector-valued subsets of \texttt{indMatrix}, such as those given by \texttt{diag}, are always of type \texttt{"logical"}.

Objects from the Class

Objects can be created explicitly with calls of the form \texttt{new("indMatrix", ...)}, but they are more commonly created by coercing 1-based integer index vectors, with calls of the form \texttt{as(..., "indMatrix")}; see \texttt{Methods} below.

Slots

- margin: an integer, either 1 or 2, specifying whether the matrix is a row (1) or column (2) index.
- perm: a 1-based integer index vector, i.e., a vector of length Dim[margin] with elements taken from 1:Dim[1+margin%%2].
- Dim, Dimnames: inherited from virtual superclass \texttt{Matrix}.

Extends

Classes \texttt{"sparseMatrix"} and \texttt{"generalMatrix"}, directly.

Methods

- \%\% signature(x = "indMatrix", y = "Matrix"): matrix products implemented where appropriate as indexing operations.
- coerce signature(from = "numeric", to = "indMatrix"): supporting typical \texttt{indMatrix} construction from a vector of positive integers. Row indexing is assumed.
- coerce signature(from = "list", to = "indMatrix"): supporting \texttt{indMatrix} construction for row and column indexing, including index vectors of length 0 and index vectors whose maximum is less than the number of rows or columns being indexed.
- coerce signature(from = "indMatrix", to = "matrix"): coercion to a traditional \texttt{matrix} of \texttt{logical} type, with FALSE and TRUE in place of 0 and 1.
indMatrix-class

The indMatrix class represents index matrices, which are matrices with integer values used to index other matrices. The class extends other matrix classes like pMatrix and nMatrix.

**signature**

- `t(x = "indMatrix")`: The transpose of an indMatrix, which is an indMatrix with identical `perm` but opposite margin.
- `rowSums(x = "indMatrix")`, `rowMeans(x = "indMatrix")`: Row and column sums and means.
- `cbind2(x = "indMatrix", y = "indMatrix")`: Row-wise concatenation of two row index matrices with equal numbers of columns and column-wise concatenation of two column index matrices with equal numbers of rows.
- `kronecker(X = "indMatrix", Y = "indMatrix")`: Kronecker product of two row index matrices or two column index matrices, giving the row or column index matrix corresponding to their “interaction”.

**Author(s)**

Fabian Scheipl at `uni-muenchen.de`, building on the existing class pMatrix after a nice hike’s conversation with Martin Maechler. Methods for `crossprod(x, y)` and `kronecker(x, y)` with both arguments inheriting from indMatrix were made considerably faster thanks to a suggestion by Boris Vaillant. Diverse tweaks by Martin Maechler and Mikael Jagan, notably the latter’s implementation of `margin`, prior to which the indMatrix class was designated only for row index matrices.

**See Also**

Subclass pMatrix of permutation matrices, a special case of index matrices; virtual class nMatrix of nonzero pattern matrices, and its subclasses.

**Examples**

```r
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"))

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

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")
```

---

The code snippet above demonstrates how to use the indMatrix class in R. It includes examples of creating indMatrix objects, transposing them, calculating row and column sums and means, row-wise and column-wise concatenation, and the Kronecker product. It also shows how to perform operations like crossprod and cross-tabulation on index matrices.
invertPerm

Utilities for Permutation Vectors

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

\[
\begin{align*}
\text{invertPerm}(p, \text{off} = 1L, \text{ioff} = 1L) \\
\text{signPerm}(p, \text{off} = 1L) \\
\text{isPerm}(p, \text{off} = 1L) \\
\text{asPerm}(\text{pivot}, \text{off} = 1L, \text{ioff} = 1L, n = \text{length}(\text{pivot})) \\
\text{invPerm}(p, \text{zero.p} = \text{FALSE}, \text{zero.res} = \text{FALSE})
\end{align*}
\]

Arguments

\(p\) an integer vector of length \(n\).

pivot an integer vector of length \(m\).

off an integer offset, indicating that \(p\) is a permutation of \(\text{off+0:(n-1)}\) or that \(\text{pivot}\) contains \(m\) values sampled with replacement from \(\text{off+0:(n-1)}\).

ioff an integer offset, indicating that the result should be a permutation of \(\text{ioff+0:(n-1)}\).

\(n\) a integer greater than or equal to \(m\), indicating the length of the result. Transpositions are applied to a permutation vector vector initialized as \(\text{seq\_len}(n)\).

zero.p a logical. Equivalent to \(\text{off}=0\) if \(\text{TRUE}\) and \(\text{off}=1\) if \(\text{FALSE}\).

zero.res a logical. Equivalent to \(\text{ioff}=0\) if \(\text{TRUE}\) and \(\text{ioff}=1\) if \(\text{FALSE}\).

Details

\(\text{invertPerm}(p, \text{off}, \text{ioff}=1)\) is equivalent to \(\text{order}(p)\) or \(\text{sort\_list}(p)\) for all values of \(\text{off}\). For the default value \(\text{off}=1\), it returns the value of \(p\) after \(p[p] \leftarrow \text{seq\_along}(p)\).

\(\text{invPerm}\) is a simple wrapper around \(\text{invertPerm}\), 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

```r
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(ip[p], 1:10)
  identical(ip[ip], 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

Methods for generic functions `anyNA()`, `is.na()`, `is.nan()`, `is.infinite()`, and `is.finite()`, for objects inheriting from virtual class `Matrix` or `sparseVector`. 
is.null.DN

Usage

## S4 method for signature 'denseMatrix'
is.na(x)
## S4 method for signature 'sparseMatrix'
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)))

(inA <- is.na(A))
stopifnot(anyNA(A), sum(inA) == 1 + 1 + 5)
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)
})
isSymmetric-methods

Methods for Function 'isSymmetric' in Package 'Matrix'

Description

isSymmetric tests whether its argument is a symmetric square matrix, by default tolerating some numerical fuzz and requiring symmetric [dD]imnames in addition to symmetry in the mathematical sense. isSymmetric is a generic function in base, which has a method for traditional matrices of implicit class "matrix". Methods are defined here for various proper and virtual classes in Matrix, so that isSymmetric works for all objects inheriting from virtual class "Matrix".

Usage

## S4 method for signature 'denseMatrix'

isSymmetric(object, checkDN = TRUE, ...)

## S4 method for signature 'CsparseMatrix'

isSymmetric(object, checkDN = TRUE, ...)

## S4 method for signature 'RsparseMatrix'

isSymmetric(object, checkDN = TRUE, ...)

## S4 method for signature 'TsparseMatrix'

isSymmetric(object, checkDN = TRUE, ...)

## S4 method for signature 'diagonalMatrix'

isSymmetric(object, checkDN = TRUE, ...)

## S4 method for signature 'indMatrix'

isSymmetric(object, checkDN = TRUE, ...)

## S4 method for signature 'dgeMatrix'

isSymmetric(object, checkDN = TRUE, tol = 100 * .Machine$double.eps, tol1 = 8 * tol, ...)

## S4 method for signature 'dgCMatrix'

isSymmetric(object, checkDN = TRUE, tol = 100 * .Machine$double.eps, ...)

Arguments

object  
a "Matrix".
checkDN  
a logical indicating whether symmetry of the Dimnames slot of object should be checked.

...  
further arguments passed to methods (typically methods for all.equal).

tol, tol1  
numerical tolerances allowing approximate symmetry of numeric (rather than logical) matrices. See also 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 "".

hasNoDimnames(m3); hasNoDimnames(m4)

hasNoDimnames(Matrix(m) -> M)

hasNoDimnames(as(M, "sparseMatrix"))

})
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` and `skewpart`: virtual class "symmetricMatrix" and its subclasses.

**Examples**

```r
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 whether a Matrix is Triangular or Diagonal*

**Description**

`isTriangular` and `isDiagonal` test whether their argument is a triangular or diagonal matrix, respectively. Unlike the analogous `isSymmetric`, these two functions are 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 and diagonal matrices are *square*, i.e., they have the same number of rows and columns.

**Usage**

```r
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.
- `...` further arguments passed to methods (currently unused by `Matrix`).
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 classes "triangularMatrix" and "diagonalMatrix" and their 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

isDiagonal(matrix(c(2,0,0,1), 2,2)) # TRUE

## Look at implementations:
showMethods("isTriangular", includeDefs = TRUE)
showMethods("isDiagonal", 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 \(\text{kronecker}(X[,j], Y[,j])\).

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


See Also

\texttt{kronecker}.

Examples

```r
## 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, m0) == 0), identical(dim(K3), dim(Kdm)),
           all(K4 <- KhatriRao(m, d0) == 0), identical(dim(K4), dim(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))
K6Dd <- KhatriRao(Diagonal(3), t(m01), sparseY=FALSE)
K6d <- KhatriRao(diag(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)
})

---

Koenker-Ng Example Sparse Model Matrix and Response Vector

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

```r
data(KNex)
```

References


Examples

```r
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(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 $0 \times NA$ as $0$, not as $NA$ as usually in R (and as used for the base function `kronecker`).

**Methods**

```r
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**

```r
(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)
N2 <- as(N, "generalMatrix") # (lost "triangularity")
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"))
```
ldenseMatrix-class

Virtual Class "ldenseMatrix" of Dense Logical Matrices

Description

ldenseMatrix is the virtual class of all dense logical (S4) matrices. It extends both denseMatrix and lMatrix directly.

Slots

- **x**: logical vector containing the entries of the matrix.
- **Dim**, **Dimnames**: see Matrix.

Extends

Class "lMatrix", directly. Class "denseMatrix", directly. Class "Matrix", by class "lMatrix". Class "Matrix", by class "denseMatrix".

Methods

- `as.vector` signature(x = "ldenseMatrix", mode = "missing"): ...
- `which` signature(x = "ndenseMatrix"), semantically equivalent to base function `which(x, arr.ind)`: for details, see the lMatrix class documentation.

See Also

Class lgeMatrix and the other subclasses.

Examples

```r
showClass("ldenseMatrix")
as(diag(3) > 0, "ldenseMatrix")
```

ldiMatrix-class

Class “ldiMatrix” of Diagonal Logical Matrices

Description

The class “ldiMatrix” of logical diagonal matrices.

Objects from the Class

Objects can be created by calls of the form `new("ldiMatrix", ...)` but typically rather via Diagonal.

Slots

- **x**: "logical" vector.
- **diag**: "character" string, either "U" or "N", see ddiMatrix.
- **Dim**, **Dimnames**: matrix dimension and dimnames, see the Matrix class description.
Extends

Class "diagonalMatrix" and class "lMatrix", directly.
Class "sparseMatrix", by class "diagonalMatrix".

See Also

Classes ddiMatrix and diagonalMatrix; function Diagonal.

Examples

(lM <- Diagonal(x = c(TRUE,FALSE,FALSE)))
str(lM)#> gory details (slots)
crossprod(lM) # numeric
(nM <- as(lM, "nMatrix"))
crossprod(nM) # pattern sparse

lgeMatrix-class  Class "lgeMatrix" of General Dense Logical Matrices

Description

This is the class of general dense logical matrices.

Slots

x: Object of class "logical". The logical values that constitute the matrix, stored in column-major order.
Dim,Dimnames: The dimension (a length-2 "integer") and corresponding names (or NULL), see the Matrix class.
factors: Object of class "list". A named list of factorizations that have been computed for the matrix.

Extends

Class "ldenseMatrix", directly. Class "lMatrix", by class "ldenseMatrix". Class "denseMatrix", by class "ldenseMatrix". Class "Matrix", by class "ldenseMatrix". Class "Matrix", by class "ldenseMatrix".

Methods

Currently, mainly t() and coercion methods (for as(.)); use, e.g., showMethods(class="lgeMatrix") for details.

See Also

Non-general logical dense matrix classes such as ltrMatrix, or lsyMatrix; sparse logical classes such as lgCMatrix.
Examples

```r
tshowClass("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")
```

Description

The `lsparseMatrix` class is a virtual class of logical sparse matrices, i.e., sparse matrices with entries `TRUE`, `FALSE`, or `NA`. These can be stored in the “triplet” form (class `TsparseMatrix`, subclasses `lgTMatrix, lsTMatrix,` and `ltTMatrix`) or in compressed column-oriented form (class `CsparseMatrix`, subclasses `lgCMatrix, lsCMatrix,` and `ltCMatrix`) or—rarely—in compressed row-oriented form (class `RsparseMatrix,` subclasses `lgRMatrix, lsRMatrix,` and `ltRMatrix`). The second letter in the name of these non-virtual classes indicates general, symmetric, or triangular.

Details

Note that triplet stored (`TsparseMatrix`) matrices such as `lgTMatrix` may contain duplicated pairs of indices \((i, j)\) as for the corresponding numeric class `dgTMatrix` where for such pairs, the corresponding \(x\) slot entries are added. For logical matrices, the \(x\) entries corresponding to duplicated index pairs \((i, j)\) are “added” as well if the addition is defined as logical or, i.e., \("TRUE + TRUE \rightarrow TRUE"\) and \("TRUE + FALSE \rightarrow TRUE"\). Note the use of `asUniqueT()` for getting an internally unique representation without duplicated \((i, j)\) entries.

Objects from the Class

Objects can be created by calls of the form `new("lgCMatrix", ...)` and so on. More frequently objects are created by coercion of a numeric sparse matrix to the logical form, e.g. in an expression \(x != 0\).

The logical form is also used in the symbolic analysis phase of an algorithm involving sparse matrices. Such algorithms often involve two phases: a symbolic phase wherein the positions of the non-zeros in the result are determined and a numeric phase wherein the actual results are calculated. During the symbolic phase only the positions of the non-zero elements in any operands are of interest, hence any numeric sparse matrices can be treated as logical sparse matrices.

Slots

- `x`: Object of class "logical", i.e., either `TRUE`, `NA`, or `FALSE`.
- `uplo`: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular. Present in the triangular and symmetric classes but not in the general class.
- `diag`: Object of class "character". Must be either "U", for unit triangular (diagonal is all ones), or "N" for non-unit. The implicit diagonal elements are not explicitly stored when `diag` is "U". Present in the triangular classes only.
lsparseMatrix-classes

p: Object of class "integer" of pointers, one for each column (row), to the initial (zero-based) index of elements in the column. Present in compressed column-oriented and compressed row-oriented forms only.

i: Object of class "integer" of length nnzero (number of non-zero elements). These are the row numbers for each TRUE element in the matrix. All other elements are FALSE. Present in triplet and compressed column-oriented forms only.

j: Object of class "integer" of length nnzero (number of non-zero elements). These are the column numbers for each TRUE element in the matrix. All other elements are FALSE. Present in triplet and compressed row-oriented forms only.

Dim: Object of class "integer" - the dimensions of the matrix.

Methods

coerce signature(from = "dgCMatrix", to = "lgCMatrix")

\(t\) signature(x = "lgCMatrix"): returns the transpose of \(x\)

which signature(x = "lsparseMatrix"), semantically equivalent to \texttt{base}\ function \texttt{which}(x, arr.ind); for details, see the \texttt{lMatrix} class documentation.

See Also

the class \texttt{dgCMatrix} and \texttt{dgTMatrix}

Examples

\begin{verbatim}
(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))
\end{verbatim}

\begin{verbatim}
data(KNex, package = "Matrix")
str(mmG.1 <- (KNex $ mm) > 0.1)# "lgC..."

table(mmG.10x)# however with many `non-structural zeros'
\# from logical to nz_pattern -- okay when there are no NA's:
mmG.1 <- as(mmG.1, "nMatrix") # <<< has "TRUE" also where mmG.1 had FALSE
\# from logical to 'double'
mmG.1 <- as(mmG.1, "dMatrix") # has '0' and back:
mmG.1 <- as(mmG.1, "lMatrix")
stopifnot(identical(mmG.1, as((KNex $ mm) != 0,"nMatrix")),
  validObject(mmG.1),
  identical(mmG.1, mmG.1))

\end{verbatim}

\begin{verbatim}
class(xnx <- crossprod(nnG.1))# "nsC..."
class(xlx <- crossprod(mmG.1))# "dsC..." : numeric
is0 <- (xlx == 0)
mean(as.vector(is0))# 99.3% zeros: quite sparse, but
\# more than half of the entries are (non-structural!) 0
\# compare xnx and xlx: have the *same* non-structural 0s:
\# sapply(slotNames(xnx),
         function(n) identical(slot(xnx, n), slot(xlx, n)))
\end{verbatim}
Description

The "lsyMatrix" class is the class of symmetric, dense logical matrices in non-packed storage and "lspMatrix" is the class of these in packed storage. In the packed form, only the upper triangle or the lower triangle is stored.

Objects from the Class

Objects can be created by calls of the form `new("lsyMatrix", ...)

Slots

- `uplo`: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular.
- `x`: Object of class "logical". The logical values that constitute the matrix, stored in column-major order.
- `Dim`, `Dimnames`: The dimension (a length-2 "integer") and corresponding names (or NULL), see the Matrix class.
- `factors`: Object of class "list". A named list of factorizations that have been computed for the matrix.

Extends

Both extend classes "ldenseMatrix" and "symmetricMatrix", directly; further, class "Matrix" and others, indirectly. Use `showClass("lsyMatrix")`, e.g., for details.

Methods

Currently, mainly `t()` and coercion methods (for `as(.)`; use, e.g., `showMethods(class="lsyMatrix")` for details.

See Also

- `lgeMatrix`, `Matrix`, `t`

Examples

```r
(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
```
**ltrMatrix-class**

**Triangular Dense Logical Matrices**

**Description**

The "ltrMatrix" class is the class of triangular, dense, logical matrices in nonpacked storage. The "ltpMatrix" class is the same except in packed storage.

**Slots**

- **x**: Object of class "logical". The logical values that constitute the matrix, stored in column-major order.
- **uplo**: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular.
- **diag**: Object of class "character". Must be either "U", for unit triangular (diagonal is all ones), or "N"; see `triangularMatrix`.
- **Dim, Dimnames**: The dimension (a length-2 "integer") and corresponding names (or NULL), see the `Matrix` class.
- **factors**: Object of class "list". A named list of factorizations that have been computed for the matrix.

**Extends**

Both extend classes "ldenseMatrix" and "triangularMatrix", directly; further, class "Matrix", "lMatrix" and others, indirectly. Use `showClass("ltrMatrix")`, e.g., for details.

**Methods**

Currently, mainly `t()` and coercion methods (for `as(.)`; use, e.g., `showMethods(class="ltrMatrix")` for details.

**See Also**

Classes `lgeMatrix, Matrix`; function `t`

**Examples**

```r
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)))
```
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 \texttt{denseMatrix} are built on LAPACK routine \texttt{dgetrf}, which does not permute columns, so that \( P_2 \) is an identity matrix.

Methods for \texttt{sparseMatrix} are built on CSparse routine \texttt{cs_lu}, which requires \( m = n \), so that \( L \) and \( U \) are triangular matrices.

Usage

\[
\text{lu}(x, \ldots)
\]

\[
\text{## S4 method for signature 'dgeMatrix'}
\text{lu}(x, \text{warnSing} = \text{TRUE}, \ldots)
\]

\[
\text{## S4 method for signature 'dgCMatrix'}
\text{lu}(x, \text{errSing} = \text{TRUE}, \text{order} = \text{NA_integer}_-,\text{tol} = 1, \ldots)
\]

\[
\text{## S4 method for signature 'dsyMatrix'}
\text{lu}(x, \text{cache} = \text{TRUE}, \ldots)
\]

\[
\text{## S4 method for signature 'dsCMatrix'}
\text{lu}(x, \text{cache} = \text{TRUE}, \ldots)
\]

\[
\text{## S4 method for signature 'matrix'}
\text{lu}(x, \ldots)
\]

Arguments

- \( x \) a finite matrix or \texttt{Matrix} to be factorized, which must be square if sparse.
- \( \text{warnSing} \) a logical indicating if a \texttt{warning} should be signaled for singular \( x \). Used only by methods for dense matrices.
- \( \text{errSing} \) a logical indicating if an \texttt{error} should be signaled for singular \( x \). Used only by methods for sparse matrices.
- \( \text{order} \) an integer in \( 0:3 \) passed to CSparse routine \texttt{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 \( A + A', \tilde{A}'\tilde{A}, \text{and} A'A, \) where \( \tilde{A} \) is \( A \) with “dense” rows removed. \texttt{NA} (the default) is equivalent to 2 if \( \text{tol} = 1 \) and 1 otherwise. Do not set to 0 unless you know that the column order of \( A \) is already sensible.
- \( \text{tol} \) a number. The original pivot element is used if its absolute value exceeds \( \text{tol} \times a \), where \( a \) is the maximum in absolute value of the other possible pivot elements. Set \( \text{tol} < 1 \) only if you know what you are doing.
cache

a logical indicating if the result should be cached in x@factors[["LU"]]}. Note that caching is experimental and that only methods for classes extending \texttt{compMatrix} will have this argument.

... 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 \texttt{dgeMatrix} completes the factorization, warning if \texttt{warnSing = TRUE} and in any case returning a valid \texttt{denseLU} object. Users of this method can detect singular \(x\) with a suitable warning handler; see \texttt{tryCatch}. In contrast, the method for class \texttt{dgCMatrix} abandons further computation, throwing an error if \texttt{errSing = TRUE} and otherwise returning \texttt{NA}. Users of this method can detect singular \(x\) with an error handler or by setting \texttt{errSing = FALSE} and testing for a formal result with \texttt{is(.,, "sparseLU")}.

### Value

An object representing the factorization, inheriting from virtual class \texttt{LU}. The specific class is \texttt{denseLU} unless \(x\) inherits from virtual class \texttt{sparseMatrix}, in which case it is \texttt{sparseLU}.

### References

The LAPACK source code, including documentation; see \url{https://netlib.org/lapack/double/dgetrf.f}.


### See Also

Classes \texttt{denseLU} and \texttt{sparseLU} and their methods.

Classes \texttt{dgeMatrix} and \texttt{dgCMatrix}.

Generic functions \texttt{expand1} and \texttt{expand2}, for constructing matrix factors from the result.

Generic functions \texttt{Cholesky}, \texttt{BunchKaufman}, \texttt{Schur}, and \texttt{qr}, for computing other factorizations.

### Examples

\begin{verbatim}
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)))
\end{verbatim}
## ---- Sparse ----------------------------------------------------------

A3 <- as(readMM(system.file("external/pores_1.mtx", package = "Matrix")),
"CsparseMatrix")
(\text{lu.A3} \leftarrow \text{lu}(A3))

## A \sim P1' \ L \ U \ P2' in floating point
\text{str(e.lu.A3} \leftarrow \text{expand2(\text{lu.A3), max.level = 2\text{l})}}
\text{stopifnot(all.equal(A3, Reduce("%*%", e.lu.A3)))}

---

### mat2triplet

**Map Matrix to its Triplet Representation**

**Description**

From an \texttt{R} object coercible to "\texttt{TsparseMatrix}"\texttt{, 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**

\texttt{mat2triplet(x, uniqT = FALSE)}

**Arguments**

\texttt{x} \hspace{2cm} \texttt{any R object for which as(x, \texttt{"TsparseMatrix"}) works; typically a matrix of one of the \texttt{Matrix} package matrices.}

\texttt{uniqT} \hspace{2cm} \texttt{logical indicating if the triplet representation should be ‘unique’ in the sense of \texttt{asUniqueT(byrow=FALSE)}.}

**Value**

A \texttt{list}, typically with three components,

\texttt{i} \hspace{2cm} \texttt{vector of row indices for all non-zero entries of x}

\texttt{j} \hspace{2cm} \texttt{vector of columns indices for all non-zero entries of x}

\texttt{x} \hspace{2cm} \texttt{vector of all non-zero entries of x; exists only when as(x, \texttt{"TsparseMatrix"}) is not a \"nspars...\texttt{diagonalMatrix".}

**Note**

The mat2triplet() utility was created to be a more efficient and more predictable substitute for \texttt{summary(<sparseMatrix>)}. UseRs have wrongly expected the latter to return a data frame with columns \texttt{i} and \texttt{j} which however is wrong for a "\texttt{diagonalMatrix}".

**See Also**

The \texttt{summary()} method for "\texttt{sparseMatrix}". \texttt{summary,sparseMatrix-method. mat2triplet()} is conceptually the \texttt{inverse} function of \texttt{spMatrix} and (one case of) \texttt{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))

Description

The basic matrix product, \texttt{\%\%\%} is implemented for all our \texttt{Matrix} and also for \texttt{sparseVector}
classes, fully analogously to R's base matrix and vector objects.

The functions \texttt{crossprod} and \texttt{tcrossprod} are matrix products or "cross products", ideally imple-
mented efficiently without computing \texttt{t(.)}'s unnecessarily. They also return \texttt{symmetricMatrix}
classed matrices when easily detectable, e.g., in \texttt{crossprod(m)}, the one argument case.

tcrossprod() takes the cross-product of the transpose of a matrix. \texttt{tcrossprod(x)} is formally
equivalent to, but faster than, the call \texttt{x \%\%\% t(x)}, and so is \texttt{tcrossprod(x, y)} instead of \texttt{x \%\%\% t(y)}.

\texttt{Boolean} matrix products are computed via either \texttt{\%\&\%} or \texttt{boolArith = TRUE}.

Usage

\texttt{## S4 method for signature 'CsparseMatrix,diagonalMatrix'}
\texttt{x \%\%\% y}

\texttt{## S4 method for signature 'CsparseMatrix,diagonalMatrix'}
\texttt{crossprod(x, y = NULL, boolArith = NA, \ldots)}
\texttt{## ..... and for many more signatures}

\texttt{## S4 method for signature 'TsparseMatrix,missing'}
\texttt{tcrossprod(x, y = NULL, boolArith = NA, \ldots)}
\texttt{## ..... and for many more signatures}

Arguments

\begin{itemize}
  \item \texttt{x} a matrix-like object
  \item \texttt{y} a matrix-like object, or for [\texttt{t}]{crossprod()} \texttt{NULL} (by default); the latter case is formally equivalent to \texttt{y = x}.
  \item \texttt{boolArith} \texttt{logical}, i.e., \texttt{NA}, \texttt{TRUE}, or \texttt{FALSE}. If true the result is (coerced to) a pattern
    matrix, i.e., \texttt{"nMatrix"}, unless there are \texttt{NA} entries and the result will be a
    \texttt{\"lMatrix\"}. If false the result is (coerced to) numeric. When \texttt{NA}, currently the
default, the result is a pattern matrix when \texttt{x} and \texttt{y} are \texttt{\"nsparseMatrix\"} and numeric otherwise.
  \item \ldots potentially more arguments passed to and from methods.
\end{itemize}
**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**

```r
## 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

Construct a Matrix of a class that inherits from Matrix.

#### Usage

Matrix(data=NA, nrow=1, ncol=1, byrow=FALSE, dimnames=NULL, 
sparse = NULL, doDiag = TRUE, forceCheck = FALSE)

#### Arguments

- **data**: an optional numeric data vector or matrix.
- **nrow**: when data is not a matrix, the desired number of rows
- **ncol**: when data is not a matrix, the desired number of columns
- **byrow**: logical. If FALSE (the default) the matrix is filled by columns, otherwise the matrix is filled by rows.
- **dimnames**: a dimnames attribute for the matrix: a list of two character components. They are set if not NULL (as per default).
- **sparse**: logical or NULL, specifying if the result should be sparse or not. By default, it is made sparse when more than half of the entries are 0.
- **doDiag**: logical indicating if a diagonalMatrix object should be returned when the resulting matrix is diagonal (mathematically). As class diagonalMatrix extends sparseMatrix, this is a natural default for all values of sparse. Otherwise, if doDiag is false, a dense or sparse (depending on sparse) symmetric matrix will be returned.
- **forceCheck**: logical indicating if the checks for structure should even happen when data is already a "Matrix" object.

#### 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. Further, Matrix() makes efforts to keep logical matrices logical, i.e., inheriting from class lMatrix, and to determine 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"
asl3, "CsparseMatrix")# "lgCMatrix"

Matrix(1:9, nrow=3,
    dimnames = list(c("a", "b", "c"), c("A", "B", "C")))
(l3 <- Matrix(diag(3)))# identity, i.e., unit "diagonalMatrix"
str(l3) # 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

The Matrix class is a class contained by all actual classes in the Matrix package. It is a “virtual” class.
Matrix-class

Slots

Dim an integer vector of length 2 giving the dimensions of the matrix.

Dimnames a list of length 2. Each element must be NULL or a character vector of length equal to the corresponding element of Dim.

Methods

determinant signature(x = "Matrix", logarithm = "missing"): and
determinant signature(x = "Matrix", logarithm = "logical"): compute the (log) determinant of x. The method chosen depends on the actual Matrix class of x. Note that det also works for all our matrices, calling the appropriate determinant() method. The Matrix::det is an exact copy of base::det, but in the correct namespace, and hence calling the S4-aware version of determinant()
.
diff signature(x = "Matrix"): As diff() for traditional matrices, i.e., applying diff() to each column.
dim signature(x = "Matrix"): extract matrix dimensions dim.
dim<- signature(x = "Matrix", value = "ANY"): where value is integer of length 2. Allows to reshape Matrix objects, but only when prod(value) == prod(dim(x)).
dimnames signature(x = "Matrix"): extract dimnames.
dimnames<- signature(x = "Matrix", value = "list"): set the dimnames to a list of length 2, see dimnames<-
.
length signature(x = "Matrix"): simply defined as prod(dim(x)) (and hence of mode "double").
show signature(object = "Matrix"): show method for printing. For printing sparse matrices, see printSpMatrix.
image signature(object = "Matrix"): draws an image of the matrix entries, using levelplot() from package lattice.
head signature(object = "Matrix"): return only the “head”, i.e., the first few rows.
tail signature(object = "Matrix"): return only the “tail”, i.e., the last few rows of the respective matrix.

as.matrix, as.array signature(x = "Matrix"): the same as as(x, "matrix"); see also the note below.
as.vector signature(x = "Matrix", mode = "missing"): as.vector(m) should be identical to as.vector(as(m, "matrix")), implemented more efficiently for some subclasses.
as(x, "vector"), as(x, "numeric") etc, similarly.

coerce signature(from = "ANY", to = "Matrix"): This relies on a correct as.matrix() method for from.

There are many more methods that (conceptually should) work for all "Matrix" objects, e.g., colSums, rowMeans. Even base functions may work automagically (if they first call as.matrix() on their principal argument), e.g., apply, eigen, svd or kappa all do work via coercion to a “traditional” (dense) matrix.

Note

Loading the Matrix namespace “overloads” as.matrix and as.array in the base namespace by the equivalent of function(x) as(x, "matrix"). Consequently, as.matrix(m) or as.array(m) will properly work when m inherits from the "Matrix" class — also for functions in package base and other packages. E.g., apply or outer can therefore be applied to "Matrix" matrices.
Author(s)

Douglas Bates <bates@stat.wisc.edu> and Martin Maechler

See Also

the classes \texttt{dgeMatrix}, \texttt{dgCMatrix}, and function \texttt{Matrix} for construction (and examples).

Methods, e.g., for \texttt{kronecker}.

Examples

\begin{verbatim}
slotNames("Matrix")

cl <- getClass("Matrix")
names(cl@subclasses) # more than 40 ..

showClass("Matrix")#> output with slots and all subclasses

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

cm <- M[1:4,] + 10*Diagonal(4)
diff(M)

## can reshape it even :
dim(M) <- c(2, 12)
M

stopifnot(identical(M, Matrix(c(0,1,0,0), 2,12)),
          all.equal(det(cm),
                     determinant(as(cm,"matrix"), log=FALSE)$modulus,
                     check.attributes=FALSE))
\end{verbatim}

Virtual Classes Not Yet Really Implemented and Used

\begin{verbatim}
Matrix-notyet

Description

iMatrix is the virtual class of all integer (S4) matrices. It extends the Matrix class directly.

zMatrix is the virtual class of all \texttt{complex} (S4) matrices. It extends the Matrix class directly.

Examples

showClass("iMatrix")
showClass("zMatrix")
\end{verbatim}
MatrixClass

The Matrix (Super-) Class of a Class

Description

Return the (maybe super-)\texttt{class} of class \texttt{cl} from package \texttt{Matrix}, returning \texttt{character(0)} if there is none.

Usage

\texttt{MatrixClass(cl, cld = getClassDef(cl), \ldots Matrix = TRUE, dropVirtual = TRUE, \ldots)}

Arguments

\begin{itemize}
  \item \texttt{cl} \hspace{1cm} \texttt{string}, class name
  \item \texttt{cld} \hspace{1cm} \texttt{its class definition}
  \item \texttt{\ldots Matrix} \hspace{1cm} \texttt{logical} indicating if the result must be of pattern \texttt{"[dlniz]..Matrix"} where the first letter \texttt{"[dlniz]"} denotes the content kind.
  \item \texttt{dropVirtual} \hspace{1cm} \texttt{logical} indicating if virtual classes are included or not.
  \item \texttt{\ldots} \hspace{1cm} \texttt{further arguments are passed to \texttt{.selectSuperClasses()}.}
\end{itemize}

Value

\texttt{a character} string

Author(s)

Martin Maechler, 24 Mar 2009

See Also

\texttt{Matrix}, the mother of all \texttt{Matrix} classes.

Examples

\begin{verbatim}
mkA <- setClass("A", contains="dgCMatrix") (A <- mkA())
stopifnot(identical(
  MatrixClass("A"),
  "dgCMatrix"))
\end{verbatim}
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
The virtual class `compMatrix` of factorizable matrices.
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")
```
Examples

```r
class = showClass("ndenseMatrix")
as(diag(3) > 0, "ndenseMatrix") # -> "nge"
```

**nearPD**  
*Nearest Positive Definite Matrix*

**Description**

Compute the nearest positive definite matrix to an approximate one, typically a correlation or variance-covariance matrix.

**Usage**

```r
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), `symmpart(x)` is used.

- `corr`  
  logical indicating if the matrix should be a *correlation* matrix.

- `keepDiag`  
  logical, generalizing `corr`: if TRUE, the resulting matrix should have the same diagonal (`diag(x)`) 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 `dpoMatrix`.

- `do2eigen`  
  logical indicating if a `posdefify()` eigen step should be applied to the result of the Higham algorithm.

- `doSym`  
  logical indicating if \( X \leftarrow \frac{1}{2}(X + t(X)) \) should be done, after \( X \leftarrow tcrossprod(Qd, Q) \); 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, `symmpart(x)` is used whenever `isSymmetric(x)` is not true. The user can explicitly set this to TRUE or FALSE, saving the symmetry test. *Beware* however that setting it FALSE for an asymmetric input `x`, is typically nonsense!

- `eig.tol`  
  defines relative positiveness of eigenvalues compared to largest one, \( \lambda_1 \). Eigenvalues \( \lambda_k \) are treated as if zero when \( \lambda_k/\lambda_1 \leq \text{eig.tol} \).

- `conv.tol`  
  convergence tolerance for Higham algorithm.
nearPD

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>posd.tol</td>
<td>tolerance for enforcing positive definiteness (in the final posdefify step when do2eigen is TRUE).</td>
</tr>
<tr>
<td>maxit</td>
<td>maximum number of iterations allowed.</td>
</tr>
<tr>
<td>conv.norm.type</td>
<td>convergence norm type (norm(*, type)) used for Higham algorithm. The default is &quot;I&quot; (infinity), for reasons of speed (and back compatibility); using &quot;F&quot; is more in line with Higham's proposal.</td>
</tr>
<tr>
<td>trace</td>
<td>logical or integer specifying if convergence monitoring should be traced.</td>
</tr>
</tbody>
</table>

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

- mat: a matrix of class dpoMatrix, the computed positive-definite matrix.
- eigenvalues: numeric vector of eigenvalues of mat.
- corr: logical, just the argument corr.
- normF: the Frobenius norm (norm(x-X, "F")) of the difference between the original and the resulting matrix.
- iterations: number of iterations needed.
- converged: logical indicating if iterations converged.

Author(s)

Jens Oehlschlägel donated a first version. Subsequent changes by the Matrix package authors.

References

Appl., 19, 1097–1110.


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")]
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$mat)), tolerance = 1e-15) # seen rel.d.= 1.46e-16
})
set.seed(27)
n.A.m <- nearPD(A, corr=TRUE, base.matrix=TRUE)$mat
stopifnot(exprs = {
  all.equal(n.A.m[1,2], 0.760689917)
  all.equal(n.A.m$normF, 0.52779033, tolerance=1e-9)
  all.equal(n.A.m, unname(as.matrix(n.A$mat)), tolerance = 1e-15)
})

m <- matrix(round(rnorm(25),2), 5, 5)
m <- m + t(m)
diag(m) <- pmax(0, diag(m)) + 1
(m <- round(cov2cor(m), 2))

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.$iterations # 2
c.1 <- nearPD(pr, conv.tol = 1e-7, corr = TRUE)
c.1$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)
cnr.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:
round(ncr$mat[1,2], 9)
norm(pr - cov2cor(ncr$mat)) # = 0.08805
norm(pr - ncr.1$mat) # = 0.08846 / 0.08805
### 3) a real data example from a 'systemfit' model (3 eq.):

```r
(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)
```

---

#### ngeMatrix-class

Class "ngeMatrix" of General Dense Nonzero-pattern Matrices

---

**Description**

This is the class of general dense nonzero-pattern matrices, see `nMatrix`.

**Slots**

- **x**: Object of class "logical". The logical values that constitute the matrix, stored in column-major order.
- **Dim,Dimnames**: The dimension (a length-2 "integer") and corresponding names (or NULL), see the `Matrix` class.
- **factors**: Object of class "list". A named list of factorizations that have been computed for the matrix.

**Extends**

Class "ndenseMatrix", directly. Class "lMatrix", by class "ndenseMatrix". Class "denseMatrix", by class "ndenseMatrix". Class "Matrix", by class "ndenseMatrix".

**Methods**

Currently, mainly t() and coercion methods (for as(.)); use, e.g., `showMethods(class="ngeMatrix")` for details.

**See Also**

Non-general logical dense matrix classes such as `ntrMatrix`, or `nsyMatrix`; sparse logical classes such as `ngCMatrix`.

**Examples**

`showClass("ngeMatrix")`

```
## "lgeMatrix" is really more relevant
```
Class "nMatrix" of Non-zero Pattern Matrices

Description

The nMatrix class is the virtual “mother” class of all non-zero pattern (or simply pattern) matrices in the Matrix package.

Slots

Common to all matrix object in the package:

- **Dim**: Object of class "integer" - the dimensions of the matrix - must be an integer vector with exactly two non-negative values.
- **Dimnames**: list of length two: each component containing NULL or a character vector length equal the corresponding Dim element.

Methods

**coerce** signature(from = "matrix", to = "nMatrix"): Note that these coercions (must) coerce NAs to non-zero, hence conceptually TRUE. This is particularly important when sparseMatrix objects are coerced to "nMatrix" and hence to nsparseMatrix.

Additional methods contain group methods, such as

- **Ops** signature(e1 = "nMatrix", e2 = "...").
- **Arith** signature(e1 = "nMatrix", e2 = "...").
- **Compare** signature(e1 = "nMatrix", e2 = "...").
- **Logic** signature(e1 = "nMatrix", e2 = "...").
- **Summary** signature(x = "nMatrix", "...").

See Also

The classes lMatrix, nsparseMatrix, and the mother class, Matrix.

Examples

```r
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"
```
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 \texttt{Matrix}.

Usage

\[
\text{nnzero}(x, \text{na.counted} = \text{NA})
\]

Arguments

- \( x \): an R object, typically inheriting from class \texttt{Matrix} or \texttt{numeric}.
- \( \text{na.counted} \): a \texttt{logical} describing how NAs should be counted. There are three possible settings for \( \text{na.counted} \):
  - \texttt{TRUE}: NAs are counted as non-zero (since “they are not zero”).
  - \texttt{NA} (default): the result will be \texttt{NA} if there are NA’s in \( x \) (since “NA’s are not known, i.e., may be zero”).
  - \texttt{FALSE}: NAs are omitted from \( x \) before the non-zero entries are counted.

For sparse matrices, you may often want to use \( \text{na.counted} = \text{TRUE} \).

Value

the number of non zero entries in \( x \) (typically \texttt{integer}).

Note that for a \texttt{symmetric} sparse matrix \( S \) (i.e., inheriting from class \texttt{symmetricMatrix}), \texttt{nnzero(S)} is typically \texttt{twice} the \texttt{length(S@x)}.

Methods

- \texttt{signature(x = "ANY")}: the default method for non-\texttt{Matrix} class objects, simply counts the number of zeros in \( x \), counting NA’s depending on the \( \text{na.counted} \) argument, see above.
- \texttt{signature(x = "denseMatrix")}: conceptually the same as for traditional \texttt{matrix} objects, care has to be taken for "\texttt{symmetricMatrix}" objects.
- \texttt{signature(x = "diagonalMatrix"), and signature(x = "indMatrix")}: fast simple methods for these special "\texttt{sparseMatrix}" classes.
- \texttt{signature(x = "sparseMatrix")}: typically, the most interesting method, also carefully taking "\texttt{symmetricMatrix}" objects into account.

See Also

The \texttt{Matrix} class also has a \texttt{length} method; typically, \texttt{length(M)} is much larger than \texttt{nnzero(M)} for a sparse matrix \( M \), and the latter is a better indication of the \texttt{size} of \( M \).

\texttt{drop0, zapsmall}. 
Examples

```r
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"))
nzero(mT)
(S <- crossprod(mT))
nzero(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)))
nzero(M) # more relevant than length
## the above are also visible from
str(M)
```

---

### Matrix Norms

**Description**

Computes a matrix norm of \( x \), using Lapack for dense matrices. The norm can be the one ("O", 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**

```r
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" or "f" 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


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)

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

Description

The nsparseMatrix class is a virtual class of sparse “pattern” matrices, i.e., binary matrices conceptually with TRUE/FALSE entries. Only the positions of the elements that are TRUE are stored.

These can be stored in the “triplet” form (TsparseMatrix, subclasses ngTMatrix, nsTMatrix, and ntTMatrix which really contain pairs, not triplets) or in compressed column-oriented form (class CsparseMatrix, subclasses ngCMatrix, nsCMatrix, and ntCMatrix) or—rarely—in compressed row-oriented form (class RsparseMatrix, subclasses ngRMatrix, nsRMatrix, and ntRMatrix). The second letter in the name of these non-virtual classes indicates general, symmetric, or triangular.
Objects from the Class

Objects can be created by calls of the form `new("ngCMatrix", ...)` and so on. More frequently objects are created by coercion of a numeric sparse matrix to the pattern form for use in the symbolic analysis phase of an algorithm involving sparse matrices. Such algorithms often involve two phases: a symbolic phase wherein the positions of the non-zeros in the result are determined and a numeric phase wherein the actual results are calculated. During the symbolic phase only the positions of the non-zero elements in any operands are of interest, hence numeric sparse matrices can be treated as sparse pattern matrices.

Slots

`uplo`: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular. Present in the triangular and symmetric classes but not in the general class.

`diag`: Object of class "character". Must be either "U", for unit triangular (diagonal is all ones), or "N" for non-unit. The implicit diagonal elements are not explicitly stored when `diag` is "U". Present in the triangular classes only.

`p`: Object of class "integer" of pointers, one for each column (row), to the initial (zero-based) index of elements in the column. Present in compressed column-oriented and compressed row-oriented forms only.

`i`: Object of class "integer" of length `nnz` (number of non-zero elements). These are the row numbers for each TRUE element in the matrix. All other elements are FALSE. Present in triplet and compressed column-oriented forms only.

`j`: Object of class "integer" of length `nnz` (number of non-zero elements). These are the column numbers for each TRUE element in the matrix. All other elements are FALSE. Present in triplet and compressed row-oriented forms only.

`Dim`: Object of class "integer" - the dimensions of the matrix.

Methods

`coerce` signature(from = "dgCMatrix", to = "ngCMatrix"), and many similar ones; typically you should coerce to "nsparseMatrix" (or "nMatrix"). Note that coercion to a sparse pattern matrix records all the potential non-zero entries, i.e., explicit ("non-structural") zeroes are coerced to TRUE, not FALSE, see the example.

`t` signature(x = "ngCMatrix"): returns the transpose of x

`which` signature(x = "lsparseMatrix"). semantically equivalent to base function `which(x, arr.ind)`; for details, see the `lMatrix` class documentation.

See Also

the class `dgCMatrix`

Examples

```r
(m <- Matrix(c(0,0,2:0), 3,5, dimnames=list(LETTERS[1:3],NULL)))
## `extract the nonzero-pattern of (m) into an nMatrix`:
mm <- as(m, "nsparseMatrix") ## -> will be a "ngCMatrix"
str(mm) # no 'x' slot
nnm <- !mm # no longer sparse
## consistency check:
stopifnot(xor(as( mm, "matrix"), as(nnm, "matrix")))
```
## low-level way of adding "non-structural zeros" :

```r
nnm <- as(nnm, "lgCMatrix")
nnm$x[2:4] <- c(FALSE, NA, NA)
nnm

as(nnm, "nMatrix") # NAs *and* non-structural 0 |---› 'TRUE'
```

```r
data(KNex, package = "Matrix")
nmm <- as(KNex$mm, "nMatrix")
str(xlx <- crossprod(nmm))# "nsCMATRIX"
stopifnot(isSymmetric(xlx))
image(xlx, main=paste("crossprod(nmm) : Sparse", class(xlx)))
```

---

### nsyMatrix-class

**Symmetric Dense Nonzero-Pattern Matrices**

**Description**

The "nsyMatrix" class is the class of symmetric, dense nonzero-pattern matrices in non-packed storage and "nspMatrix" is the class of these in packed storage. Only the upper triangle or the lower triangle is stored.

**Objects from the Class**

Objects can be created by calls of the form `new("nsyMatrix", ...).

**Slots**

- **uplo**: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular.
- **x**: Object of class "logical". The logical values that constitute the matrix, stored in column-major order.
- **Dim,Dimnames**: The dimension (a length-2 "integer") and corresponding names (or NULL), see the [Matrix] class.
- **factors**: Object of class "list". A named list of factorizations that have been computed for the matrix.

**Extends**

"nsyMatrix" extends class "ngeMatrix", directly, whereas
"nspMatrix" extends class "ndenseMatrix", directly.

Both extend class "symmetricMatrix", directly, and class "Matrix" and others, indirectly, use `showClass("nsyMatrix")`, e.g., for details.

**Methods**

Currently, mainly `t()` and coercion methods (for `as(.)`; use, e.g., `showMethods(class="nsyMatrix")` for details.

**See Also**

`ngeMatrix`, `Matrix`, `t`
Examples

```r
(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
```

ntrMatrix-class

Triangular Dense Logical Matrices

Description

The "ntrMatrix" class is the class of triangular, dense, logical matrices in nonpacked storage. The "ntpMatrix" class is the same except in packed storage.

Slots

- **x**: Object of class "logical". The logical values that constitute the matrix, stored in column-major order.
- **uplo**: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular.
- **diag**: Object of class "character". Must be either "U", for unit triangular (diagonal is all ones), or "N"; see `triangularMatrix`.
- **Dim,Dimnames**: The dimension (a length-2 "integer") and corresponding names (or NULL), see the `Matrix` class.
- **factors**: Object of class "list". A named list of factorizations that have been computed for the matrix.

Extends

"ntrMatrix" extends class "ngeMatrix", directly, whereas "ntpMatrix" extends class "ndenseMatrix", directly.

Both extend Class "triangularMatrix", directly, and class "denseMatrix", "lMatrix" and others, indirectly, use `showClass("nsyMatrix")`, e.g., for details.

Methods

Currently, mainly `t()` and coercion methods (for `as(.)`; use, e.g., `showMethods(class="ntrMatrix")` for details.

See Also

Classes `ngeMatrix`, `Matrix`; function `t`
Examples

```r
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)))
```

number-class

Class “number” of Possibly Complex Numbers

Description

The class "number" is a virtual class, currently used for vectors of eigen values which can be "numeric" or "complex".

It is a simple class union (setClassUnion) of "numeric" and "complex".

Objects from the Class

Since it is a virtual Class, no objects may be created from it.

Examples

```r
showClass("number")

stopifnot( is(1i, "number"), is(pi, "number"), is(1:3, "number") )
```

pack

Representation of Packed and Unpacked Dense Matrices

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

```r
pack(x, ...)
## S4 method for signature 'dgeMatrix'
pack(x, symmetric = NA, upperTri = NA, ...)
## S4 method for signature 'lgeMatrix'
pack(x, symmetric = NA, upperTri = NA, ...)
## S4 method for signature 'ngeMatrix'
pack(x, symmetric = NA, upperTri = NA, ...)
## S4 method for signature 'matrix'
pack(x, symmetric = NA, upperTri = NA, ...)

unpack(x, ...)
```
Arguments

x  A dense symmetric or dense triangular matrix.

For `pack()`: typically an "unpackedMatrix" or a standard "matrix", though "packedMatrix" are allowed and returned unchanged.

For `unpack()`: typically a "packedMatrix", though "unpackedMatrix" are allowed and returned unchanged.

symmetric  logical (including NA) optionally indicating whether x is symmetric (or triangular).

upperTri  (for triangular x only) logical (including NA) indicating whether x is upper (or lower) triangular.

... 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

```r
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
```
Virtual Class \texttt{packedMatrix} of Packed Dense Matrices

Description

Class \texttt{packedMatrix} is the virtual class of dense symmetric or triangular matrices in "packed" format, storing only the choose(n+1,2) = n*(n+1)/2 elements of the upper or lower triangle of an n-by-n matrix. It is used to define common methods for efficient subsetting, transposing, etc. of its proper subclasses: currently \texttt{[dln]spMatrix} (packed symmetric), \texttt{[dln]tpMatrix} (packed triangular), and subclasses of these, such as \texttt{dppMatrix}, \texttt{pCholesky}, and \texttt{pBunchKaufman}.

Slots

- \texttt{uplo}: \texttt{character}; either \texttt{"U"}, for upper triangular, and \texttt{"L"}, for lower.
- \texttt{Dim, Dimnames}: as all \texttt{Matrix} objects.

Extends

Class \texttt{"denseMatrix"}, directly. Class \texttt{"Matrix"}, by class \texttt{"denseMatrix"}, distance 2. Class \texttt{"replValueSp"}, by class \texttt{"Matrix"}, distance 3.

Methods

- \texttt{pack} signature(\texttt{x = "packedMatrix"}): ...
- \texttt{unpack} signature(\texttt{x = "packedMatrix"}): ...
- \texttt{isSymmetric} signature(\texttt{object = "packedMatrix"}): ...
- \texttt{isTriangular} signature(\texttt{object = "packedMatrix"}): ...
- \texttt{isDiagonal} signature(\texttt{object = "packedMatrix"}): ...
- \texttt{t} signature(\texttt{x = "packedMatrix"}): ...
- \texttt{diag} signature(\texttt{x = "packedMatrix"}): ...
- \texttt{diag} signature(\texttt{x = "packedMatrix"}): ...

Author(s)

Mikael Jagan

See Also

- \texttt{pack} and \texttt{unpack}; its virtual "complement" \texttt{"unpackedMatrix"}; its proper subclasses \texttt{"dspMatrix"}, \texttt{"ltpMatrix"}, etc.

Examples

- \texttt{showClass("packedMatrix")}
- \texttt{showMethods(classes = "packedMatrix")}
The \texttt{pMatrix} class is the class of permutation matrices, stored as 1-based integer permutation vectors. A permutation matrix is a square matrix whose rows and columns are all standard unit vectors. It follows that permutation matrices are a special case of index matrices (hence \texttt{pMatrix} is defined as a direct subclass of \texttt{indMatrix}).

Multiplying a matrix on the left by a permutation matrix is equivalent to permuting its rows. Analogously, multiplying a matrix on the right by a permutation matrix is equivalent to permuting its columns. Indeed, such products are implemented in \texttt{Matrix} as indexing operations; see ‘Details’ below.

**Details**

By definition, a permutation matrix is both a row index matrix and a column index matrix. However, the \texttt{perm} slot of a \texttt{pMatrix} cannot be used interchangeably as a row index vector and column index vector. If \texttt{margin=1}, then \texttt{perm} is a row index vector, and the corresponding column index vector can be computed as \texttt{invPerm}(\texttt{perm}), i.e., by inverting the permutation. Analogously, if \texttt{margin=2}, then \texttt{perm} and \texttt{invPerm}(\texttt{perm}) are column and row index vectors, respectively.

Given an $n$-by-$n$ row permutation matrix $P$ with \texttt{perm} slot $p$ and a matrix $M$ with conformable dimensions, we have

\[
PM = P %*% M = M[p, ]
\]
\[
MP = M %*% P = M[, i(p)]
\]
\[
P'M = \text{crossprod}(P, M) = M[i(p), ]
\]
\[
MP' = \text{tcrossprod}(M, P) = M[, p]
\]
\[
P'P = \text{crossprod}(P) = \text{Diagonal}(n)
\]
\[
P P' = \text{tcrossprod}(P) = \text{Diagonal}(n)
\]

where $i := \text{invPerm}$.

**Objects from the Class**

Objects can be created explicitly with calls of the form \texttt{new(“pMatrix”, \ldots)}, but they are more commonly created by coercing 1-based integer index vectors, with calls of the form \texttt{as(., “pMatrix”)}; see ‘Methods’ below.

**Slots**

\texttt{margin,perm} inherited from superclass \texttt{indMatrix}. Here, \texttt{perm} is an integer vector of length \texttt{Dim[1]} and a permutation of \texttt{1:Dim[1]}.

\texttt{Dim,Dimnames} inherited from virtual superclass \texttt{Matrix}.

**Extends**

Class ”\texttt{indMatrix}”, directly.
Methods

%*% signature(x = "pMatrix", y = "Matrix") and others listed by showMethods("%*%", classes = "pMatrix"): matrix products implemented where appropriate as indexing operations.

coerce signature(from = "numeric", to = "pMatrix"): supporting typical pMatrix construction from a vector of positive integers, specifically a permutation of 1:n. Row permutation is assumed.

t signature(x = "pMatrix"): the transpose, which is a pMatrix with identical perm but opposite margin. Coincides with the inverse, as permutation matrices are orthogonal.

solve signature(a = "pMatrix", b = "missing"): the inverse permutation matrix, which is a pMatrix with identical perm but opposite margin. Coincides with the transpose, as permutation matrices are orthogonal. See showMethods("solve", classes = "pMatrix") for more signatures.

determinant signature(x = "pMatrix", logarithm = "logical"): always returning 1 or -1, as permutation matrices are orthogonal. In fact, the result is exactly the sign of the permutation.

See Also

Superclass indMatrix of index matrices, for many inherited methods; invPerm, for computing inverse permutation vectors.

Examples

(pml <- as(as.integer(c(2,3,1)), "pMatrix"))
t(pml) # is the same as solve(pml)
pml %*% t(pml) # check that the transpose is the inverse
stopifnot(all(diag(3) == as(pml %*% t(pml), "matrix")),
           is.logical(as(pml, "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 %*% pml
pml %*% mm
try(as(as.integer(c(3,3,1)), "pMatrix"))# Error: not a permutation

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

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

Format and print sparse matrices flexibly. These are the "workhorses" used by the \texttt{format}, \texttt{show} and \texttt{print} methods for sparse matrices. If \( x \) is large, \texttt{printSpMatrix2(x)} calls \texttt{printSpMatrix()} twice, namely, for the first and the last few rows, suppressing those in between, and also suppresses columns when \( x \) is too wide.

\texttt{printSpMatrix()} basically prints the result of \texttt{formatSpMatrix()}.

Usage

\begin{verbatim}
printSpMatrix(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)
\end{verbatim}

Arguments

\begin{description}
\item[\texttt{x}] an \texttt{R} object inheriting from class \texttt{sparseMatrix}.
\item[\texttt{digits}] significant digits to use for printing, see \texttt{print.default}, the default, \texttt{NULL}, corresponds to using \texttt{getOption("digits")}.
\item[\texttt{maxp}] integer, default from \texttt{options(max.print)}, 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.
\item[\texttt{cld}] the class definition of \texttt{x}; must be equivalent to \texttt{getClassDef(class(x))} and exists mainly for possible speedup.
\item[\texttt{zero.print}] character which should be printed for structural zeroes. The default "." may occasionally be replaced by " " (blank); using "0" would look almost like \texttt{print()}ing of non-sparse matrices.
\item[\texttt{col.names}] logical or string specifying if and how column names of \texttt{x} should be printed, possibly abbreviated. The default is taken from \texttt{options("sparse.colnames")} if that is set, otherwise \texttt{FALSE} unless there are less than ten columns. When \texttt{TRUE} the full column names are printed. When \texttt{col.names} is a string beginning with "\texttt{abb}" or "\texttt{sub}" and ending with an integer \( n \) (i.e., of the form ",abb... <n>"), the column names are \texttt{abbreviate()}d or \texttt{substring()}ed to (target) length \( n \), see the examples.
\item[\texttt{note.dropping.colnames}] logical specifying, when \texttt{col.names} is \texttt{FALSE} if the dropping of the column names should be noted, \texttt{TRUE} by default.
\end{description}
uniDiag 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").

col.trailer a string to be appended to the right of each column; this is typically made use of by show(<sparseMatrix>) only, when suppressing columns.

suppRows, suppCols logicals or NULL, for printSpMatrix2() specifying if rows or columns should be suppressed in printing. If NULL, sensible defaults are determined from dim(x) and options(c("width", "max.print")); Setting both to FALSE may be a very bad idea.

align a string specifying how the zero.print codes should be aligned, i.e., padded as strings. The default, "fancy", takes some effort to align the typical zero.print = "." with the position of 0, i.e., the first decimal (one left of decimal point) of the numbers printed, whereas align = "right" just makes use of print(*, right = TRUE).

width number, a positive integer, indicating the approximately desired (line) width of the output, see also fitWidth.

fitWidth logical indicating if some effort should be made to match the desired width or temporarily enlarge that if deemed necessary.

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 x 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(gl(12, 3, labels = paste(LETTERS[1:12], "c", sep=".")), "sparseMatrix")

## 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

\[
P_1 AP_2 = QR
\]

or (equivalently)

\[
A = P_1'QR P_2'
\]

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 \code{base}, namely \code{qr.default}. It is built on LINPACK routine \code{dqrdc} and LAPACK routine \code{dgeqp3}, which do not pivot rows, so that \( P_1 \) is an identity matrix.

Methods for \code{sparseMatrix} are built on CSparse routines \code{cs_sqr} and \code{cs_qr}, which require \( m \geq n \).

Usage

\footnotesize
\begin{verbatim}
qr(x, ...)  
## S4 method for signature 'dgCMatrix'
qr(x, order = 3L, ...)  
\end{verbatim}

Arguments

\begin{itemize}
  \item \code{x} a finite matrix or \code{Matrix} to be factorized, satisfying \code{nrow(x) >= ncol(x)} if sparse.
  \item \code{order} an integer in \( 0:3 \) passed to CSparse routine \code{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 \( A + \tilde{A}' \tilde{A}, \tilde{A}'A, \) and \( A' \tilde{A}, \) 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.
  \item \ldots further arguments passed to or from methods.
\end{itemize}
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:

\[
P_1 A P_2 = P_1 \begin{bmatrix} A_0 \\ 0 \end{bmatrix} P_2 = QR
\]

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 \texttt{QR} or S3 class \texttt{qr}. The specific class is \texttt{qr} unless \( x \) inherits from virtual class \texttt{sparseMatrix}, in which case it is \texttt{sparseQR}.

References


See Also

Class \texttt{sparseQR} and its methods.

Class \texttt{dgCMatrix}.

Generic function \texttt{qr} from \texttt{base}, whose default method \texttt{qr.default} “defines” the S3 class \texttt{qr} of dense QR factorizations.

Generic functions \texttt{expand1} and \texttt{expand2}, for constructing matrix factors from the result.

Generic functions \texttt{Cholesky}, \texttt{BunchKaufman}, \texttt{Schur}, and \texttt{lu}, for computing other factorizations.

Examples

```r
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 A x = b exists
## and is unique _in exact arithmetic_

(A1 <- M[, -c(3L, 6L)])
(qr.A1 <- qr(A1))
```

\[
\text{stopifnot(exprs = }
\]
\[
\begin{array}{l}
\text{rankMatrix}(A1) == ncol(A1) \\
\{ \text{d1} \leftarrow \text{abs(diag(qr.A1@R))}; \text{sum}(\text{d1} < \text{max(d1)} \times \text{eps}) == 0L \} \\
\text{rcond(crossprod(A1))} \geq \text{eps} \\
\text{all.equal(qr.coef(qr.A1, b), drop(solve(crossprod(A1), crossprod(A1, b))))} \\
\text{all.equal(qr.fitted(qr.A1, b) + qr.resid(qr.A1, b), b)}
\end{array}
\]
\]

<< \ldots [2] numerically rank deficient with full structural rank .......
<< ====> a least squares solution of \( A x = b \) does not
<< exist or is not unique _in exact arithmetic_.
\[
(A2 \leftarrow M) \\
(qr.A2 \leftarrow qr(A2))
\]

\[
\text{stopifnot(exprs = }
\]
\[
\begin{array}{l}
\text{rankMatrix}(A2) == ncol(A2) - 2L \\
\{ \text{d2} \leftarrow \text{abs(diag(qr.A2@R))}; \text{sum}(\text{d2} < \text{max(d2)} \times \text{eps}) == 2L \} \\
\text{rcond(crossprod(A2))} < \text{eps}
\end{array}
\]
\]

<< 'qr.coef' computes unique least squares solution of "nearby" problem
<< Z \( x = b \) for some full rank \( Z \sim A \), currently without warning {FIXME}!
\>
\>
\[
\text{tryCatch}({ \text{qr.coef(qr.A2, b); TRUE }}, \text{condition = function(x) FALSE})
\]

<< all.equal(qr.fitted(qr.A2, b) + qr.resid(qr.A2, b), b)
\]
\]

<< \ldots [3] numerically and structurally rank deficient ...............
<< ====> factorization of _augmented_ matrix with
<< full structural rank proceeds as in [2]
\[
\text{NB: implementation details are subject to change; see (*) below}
\]
\[
A3 \leftarrow M \\
A3[, c(3L, 6L)] \leftarrow 0 \\
A3
\]
\[
(qr.A3 \leftarrow qr(A3)) \# with a warning ... "additional 2 row(s) of zeros"
\]
\[
\text{stopifnot(exprs = }
\]
\[
\begin{array}{l}
\text{rankMatrix}(A3) == ncol(A3) - 2L \\
\{ \text{d3} \leftarrow \text{abs(diag(qr.A3@R))}; \text{sum}(\text{d3} < \text{max(d3)} \times \text{eps}) == 2L \} \\
\text{rcond(crossprod(A3))} < \text{eps}
\end{array}
\]
\]

<< Auxiliary functions accept and return a vector or matrix
<< with dimensions corresponding to the unaugmented matrix (*),
<< in all cases with a warning
\>
\[
\text{qr.coef (qr.A3, b)} \\
\text{qr.fitted(qr.A3, b)} \\
\text{qr.resid (qr.A3, b)}
\]
By disabling column pivoting, one gets the "vanilla" factorization

\[ A = Q R, \quad Q \text{ is orthogonal because } P I = Q \]

```r
(qr.A1.pp <- qr(A1, order = 0L)) # partial pivoting
```

```r
ea1 <- function(a, b, ...) all.equal(as(a, "matrix"), as(b, "matrix"), ...)
ea2 <- function(a, b, ...) ae1(unname(a), unname(b), ...)
```

```r
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))
})
```

---

## 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 \( A \), \( \text{rk}(A) \), is the maximal number of linearly independent columns (or rows); hence \( \text{rk}(A) \leq \min(n, m) \).

**Usage**

```r
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)
```

```r
qr2rankMatrix(qr, tol = NULL, isBqr = is.qr(qr), do.warn = TRUE)
```

**Arguments**

- `x` numeric matrix, of dimension \( n \times m \), say.
- `tol` nonnegative number specifying a (relative, “scalefree”) tolerance for testing of “practically zero” with specific meaning depending on `method`; by default, \( \max(\text{dim}(x)) \times \text{\$double\text{eps}} \) is according to Matlab’s default (for its only method which is our method="tolNorm2").
- `method` a character string specifying the computational method for the rank, can be abbreviated:
  - "tolNorm2": the number of singular values \( \geq \text{tol} \times \text{\$max(sval)} \);
  - "qrLINPACK": for a dense matrix, this is the rank of \( \text{qr}(x, \text{tol}, \text{LAPACK=}\text{FALSE}) \) (which is \( \text{qr}(...)\text{\$rank} \));
  - This ("qr\*", dense) version used to be the recommended way to compute a matrix rank for a while in the past.
  - For sparse \( x \), this is equivalent to "qr.R".
"qr.R": this is the rank of triangular matrix \( R \), where \( \text{qr()} \) uses LAPACK or a "sparseQR" method (see \( \text{qr-methods} \)) 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 \( |d_i| \geq \text{tol} \cdot \max(|d_i|) \).

"qr": is for back compatibility; for dense \( x \), it corresponds to "qrLINPACK", whereas for sparse \( x \), it uses "qr.R".

For all the "qr*" methods, singular values \( \text{sval} \) are not used, which may be crucially important for a large sparse matrix \( x \), as in that case, when \( \text{sval} \) is not specified, the default, computing \( \text{svd()} \) currently coerces \( x \) to a dense matrix.

"useGrad": considering the “gradient” of the (decreasing) singular values, the index of the smallest gap.

"maybeGrad": choosing method "useGrad" only when that seems reasonable; otherwise using "tolNorm2".

\( \text{sval} \)

numeric vector of non-increasing singular values of \( x \); typically unspecified and computed from \( x \) when needed, i.e., unless \( \text{method} = \text{"qr"} \).

\( \text{warn.t} \)

logical indicating if \( \text{rankMatrix()} \) should warn when it needs \( t(x) \) instead of \( x \). Currently, for \( \text{method} = \text{"qr"} \) only, gives a warning by default because the caller often could have passed \( t(x) \) directly, more efficiently.

\( \text{warn.qr} \)

in the \( QR \) cases (i.e., if \( \text{method} \) starts with "\( \text{qr} \)"), \( \text{rankMatrix()} \) calls \( \text{qr2rankMatrix(} \ldots \text{, do.warn = warn.qr)} \), see below.

\( \text{qr} \)

an \( R \) object resulting from \( \text{qr}(x, \ldots) \), i.e., typically inheriting from \text{class} "\( \text{qr} \)" or "\( \text{sparseQR} \)".

\( \text{isBqr} \)

\text{logical} indicating if \( \text{qr} \) is resulting from \text{base} \( \text{qr}() \). (Otherwise, it is typically from \text{Matrix} package sparse \( \text{qr} \).)

\( \text{do.warn} \)

\text{logical}; if true, warn about non-finite diagonal entries in the \( R \) matrix of the \( QR \) decomposition. Do not change lightly!

\text{Details}

\( \text{qr2rankMatrix()} \) is typically called from \( \text{rankMatrix()} \) for the "qr*" methods, but can be used directly - much more efficiently in case the \( QR \)-decomposition is available anyway.

\text{Value}

If \( x \) is a matrix of all 0 (or of zero dimension), the rank is zero; otherwise, typically a positive integer in \( 1:\text{min(dim(x))} \) 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 \( \text{NaN} \) entries. Then, a warning is signalled (unless \( \text{warn.qr} / \text{do.warn} \) is not true) and \( \text{NA} \) (specifically, \( \text{NA_integer}_{-} \)) is returned.

\text{Note}

For large sparse matrices \( x \), unless you can specify \( \text{sval} \) yourself, currently \( \text{method} = \text{"qr"} \) may be the only feasible one, as the others need \( \text{sval} \) and call \( \text{svd()} \) which currently coerces \( x \) to a \text{denseMatrix} which may be very slow or impossible, depending on the matrix dimensions.

Note that in the case of sparse \( x \), \text{method} = "\( \text{qr} \)". all non-strictly zero diagonal entries \( d_i \) where counted, up to including \text{Matrix} version 1.1-0, i.e., that method implicitly used \( \text{tol} = 0 \), see also the \text{set.seed(42)} example below.
rankMatrix

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 12 12 12 (x86_64)
sapply(5:15, rMQR, M = 1000 * H12) # not identical unfortunately
## 7 7 8 10 11 11 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)))
(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)
Estimate the Reciprocal Condition Number

rcond(x, norm, ...)

### S4 method for signature 'sparseMatrix,character'

rcond(x, norm, useInv=FALSE, ...)

**Arguments**

- **x**: an R object that inherits from the `Matrix` class.
- **norm**: character string indicating the type of norm to be used in the estimate. The default is "O" for the 1-norm ("O" is equivalent to "1"). For sparse matrices, when `useInv=TRUE`, `norm` can be any of the kinds allowed for `norm`; otherwise, the other possible value is "I" for the infinity norm, see also `norm`.
- **useInv**: logical (or "Matrix" containing `solve(x)`). If not false, compute the reciprocal condition number as $1/\|x\| \cdot \|x^{-1}\|$, where $x^{-1}$ is the inverse of $x$, `solve(x)`. This may be an efficient alternative (only) in situations where `solve(x)` is fast (or known), e.g., for (very) sparse or triangular matrices. Note that the result may differ depending on `useInv`, as per default, when it is false, an approximation is computed.
- **...**: 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


See Also

\( \text{norm, kappa()} \) from package \texttt{base} computes an approximate condition number of a “traditional” matrix, even non-square ones, with respect to the \( p = 2 \) (Euclidean) \( \text{norm} \). \texttt{solve}.

\( \text{condest} \), a newer approximate estimate of the (1-norm) condition number, particularly efficient for large sparse matrices.

Examples

```r
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))))
```

```r
## 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 \( \text{solve}(M3) \)

system.time(r. <- rcond(M3, useInv=TRUE))
if(interactive()) ## the values are not the same
  c(r, r.) # 0.05555 0.013888
```
## for all 4 norms available for sparseMatrix:
cbind(rr <- sapply(c("1","I","F","M"),
    function(N) rcond(M3, norm=N, useInv=TRUE)))

rep2abI

Replicate Vectors into 'abIndex' Result

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)))

replValue-class  
Virtual Class "replValue" - Simple Class for Subassignment Values

Description
The class "replValue" is a virtual class used for values in signatures for sub-assignment of Matrix matrices.

In fact, it is a simple class union (setClassUnion) of "numeric" and "logical" (and maybe "complex" in the future).

Objects from the Class
Since it is a virtual Class, no objects may be created from it.
Description

Class "rleDiff" is for compactly storing long vectors which mainly consist of linear stretches. For such a vector \( x \), \( \text{diff}(x) \) consists of constant stretches and is hence well compressable via \( \text{rle()} \).

Objects from the Class

Objects can be created by calls of the form \( \text{new("rleDiff", ...)} \).

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 \( \text{rle()} \). As this is used to encode potentially huge index vectors, lengths may be of type \text{double} here.

Methods

There is a simple \text{show} method only.

Note

This is currently an experimental auxiliary class for the class \text{abIndex}, see there.

See Also

\text{rle}, \text{abIndex}.

Examples

\begin{verbatim}
showClass("rleDiff")
ab <- c(abIseq(2, 100), abIseq(20, -2))
ab@rleD  # is "rleDiff"
\end{verbatim}
rsparsematrix  

Random Sparse Matrix

Description

Generate a random sparse matrix efficiently. The default has rounded gaussian non-zero entries, and \texttt{rand.x = NULL} generates random pattern matrices, i.e. inheriting from \texttt{nsparseMatrix}.

Usage

\begin{verbatim}
rsparsematrix(nrow, ncol, density, nnz = round(density * maxE),
              symmetric = FALSE,
              rand.x = function(n) signif(rnorm(n), 2), ...)
\end{verbatim}

Arguments

- \texttt{nrow, ncol}  
  number of rows and columns, i.e., the matrix dimension (\texttt{dim}).
- \texttt{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 \texttt{nnz}, otherwise \texttt{nnz} needs to be specified.
- \texttt{nnz}  
  number of non-zero entries, for a sparse matrix typically considerably smaller than \texttt{nrow*ncol}. Must be specified if \texttt{density} is not.
- \texttt{symmetric}  
  logical indicating if result should be a matrix of class \texttt{symmetricMatrix}. Note that in the symmetric case, \texttt{nnz} denotes the number of non zero entries of the upper (or lower) part of the matrix, including the diagonal.
- \texttt{rand.x}  
  \texttt{NULL} or the random number generator for the \texttt{x} slot, a \texttt{function} such that \texttt{rand.x(n)} generates a numeric vector of length \texttt{n}. Typical examples are \texttt{rand.x = rnorm}, or \texttt{rand.x = runif}; the default is nice for didactical purposes.
- \texttt{...}  
  optionally further arguments passed to \texttt{sparseMatrix()}, notably \texttt{repr}.

Details

The algorithm first samples “encoded” $(i, j)$s without replacement, via one dimensional indices, if not symmetric \texttt{sample.int(nrow*ncol, nnz)}, then—if \texttt{rand.x} is not \texttt{NULL}—gets \texttt{x <- rand.x(nnz)} and calls \texttt{sparseMatrix(i=i, j=j, x=x, ...)}. When \texttt{rand.x=NULL}, \texttt{sparseMatrix(i=i, j=j, \ldots)} will return a pattern matrix (i.e., inheriting from \texttt{nsparseMatrix}).

Value

A \texttt{sparseMatrix}, say \texttt{M} of dimension \texttt{(nrow, ncol)}, i.e., with \texttt{dim(M) == c(nrow, ncol)}, if \texttt{symmetric} is not true, with \texttt{nzM <= nnzero(M)} fulfilling \texttt{nzM <= nnz} and typically, \texttt{nzM == nnz}.

Author(s)

Martin Maechler
Examples

```r
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

Class “RsparseMatrix” of Sparse Matrices in Row-compressed Form

Description

The “RsparseMatrix” class is the virtual class of all sparse matrices coded in sorted compressed row-oriented form. Since it is a virtual class, no objects may be created from it. See `showClass("RsparseMatrix")` for its subclasses.

Slots

- **j**: Object of class "integer" of length `nnzero` (number of non-zero elements). These are the row numbers for each non-zero element in the matrix.
- **p**: Object of class "integer" of pointers, one for each row, to the initial (zero-based) index of elements in the row.

`Dim`, `Dimnames`: inherited from the superclass, see `sparseMatrix`.

Extends

Class "sparseMatrix", directly. Class "Matrix", by class "sparseMatrix".

Methods

Originally, few methods were defined on purpose, as we rather use the `CsparseMatrix` in `Matrix`. Then, more methods were added but beware that these typically do not return "RsparseMatrix" results, but rather Csparse* or Tsparse* ones; e.g., `R[i, j] <- v` for an "RsparseMatrix" `R` works, but after the assignment, `R` is a (triplet) "TsparseMatrix".

- `t` signature(`x = "RsparseMatrix"`): ...
- `coerce` signature(`from = "RsparseMatrix", to = "CsparseMatrix"`): ...
- `coerce` signature(`from = "RsparseMatrix", to = "TsparseMatrix"`): ...
See Also

its superclass, \code{sparseMatrix}, and, e.g., class \code{dgRMatrix} for the links to other classes.

Examples

\code{showClass("RsparseMatrix")}

Schur-class  \hspace{1cm} Schur Factorizations

Description

\texttt{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 $1 \times 1$ or $2 \times 2$ 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 \textit{similar} and thus have the same spectrum. The eigenvalues are computed trivially as the eigenvalues of the diagonal blocks of \( T \).

Slots

- \texttt{Dim, Dimnames} inherited from virtual class \code{MatrixFactorization}.
- \texttt{Q} an orthogonal matrix, inheriting from virtual class \code{Matrix}.
- \texttt{T} a block upper triangular matrix, inheriting from virtual class \code{Matrix}. The diagonal blocks have dimensions 1-by-1 or 2-by-2.
- \texttt{EValues} a numeric or complex vector containing the eigenvalues of the diagonal blocks of \( T \), which are the eigenvalues of \( T \) and consequently of the factorized matrix.

Extends

Class \code{SchurFactorization}, directly. Class \code{MatrixFactorization}, by class \code{SchurFactorization}, distance 2.

Instantiation

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

Methods

- \texttt{determinant signature(from = "Schur", logarithm = "logical")}: computes the determinant of the factorized matrix \( A \) or its logarithm.
- \texttt{expand1 signature(x = "Schur")}: see \code{expand1-methods}.
- \texttt{expand2 signature(x = "Schur")}: see \code{expand2-methods}.
- \texttt{solve signature(a = "Schur", b = .)}: see \code{solve-methods}.
Schur-methods

References

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


See Also

Class dgeMatrix.

Generic functions Schur, expand1 and expand2.

Examples

```r
showClass("Schur")
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 \sim 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)))

## One of the non-general cases:
Schur(Diagonal(6L))
```

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 $1 \times 1$ and $2 \times 2$ 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.
Schur-methods

Usage

Schur(x, vectors = TRUE, ...)  

Arguments

- **x**: a finite square matrix or `Matrix` to be factorized.
- **vectors**: a logical. If `TRUE` (the default), then Schur vectors are computed in addition to the Schur form.
- **...**: further arguments passed to or from methods.

Value

An object representing the factorization, inheriting from virtual class `SchurFactorization` if `vectors = TRUE`. Currently, the specific class is always `Schur` in that case.

An exception is if `x` is a traditional matrix, in which case the result is a named list containing `Q`, `T`, and `EValues` slots of the `Schur` object.

If `vectors = FALSE`, then the result is the same named list but without `Q`.

References

The LAPACK source code, including documentation; see [https://netlib.org/lapack/](https://netlib.org/lapack/)


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

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

Schur(Hilbert(9L)) # real eigenvalues

(A <- Matrix(round(rnorm(25L, sd = 100)), 5L, 5L))
(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(sch.A$T, only.values = TRUE)$values)
(e2 <- eigen(A, only.values = TRUE)$values)
(e3 <- sch.A$EValues)

stopifnot(exprs = {
  all.equal(e1, e2, tolerance = 1e-13)
})
```

---

**Usage**

Schur(x, vectors = TRUE, ...)  

**Arguments**

- **x**: a finite square matrix or `Matrix` to be factorized.
- **vectors**: a logical. If `TRUE` (the default), then Schur vectors are computed in addition to the Schur form.
- **...**: further arguments passed to or from methods.

**Value**

An object representing the factorization, inheriting from virtual class `SchurFactorization` if `vectors = TRUE`. Currently, the specific class is always `Schur` in that case.

An exception is if `x` is a traditional matrix, in which case the result is a named list containing `Q`, `T`, and `EValues` slots of the `Schur` object.

If `vectors = FALSE`, then the result is the same named list but without `Q`.

**References**

The LAPACK source code, including documentation; see [https://netlib.org/lapack/](https://netlib.org/lapack/)


**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**

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

Schur(Hilbert(9L)) # real eigenvalues

(A <- Matrix(round(rnorm(25L, sd = 100)), 5L, 5L))
(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(sch.A$T, only.values = TRUE)$values)
(e2 <- eigen(A, only.values = TRUE)$values)
(e3 <- sch.A$EValues)

stopifnot(exprs = {
  all.equal(e1, e2, tolerance = 1e-13)
})
```
Methods for generic function `solve` for solving linear systems of equations, i.e., for \(AX = B\), where \(A\) is a square matrix and \(X\) and \(B\) are matrices with dimensions consistent with \(A\).

### Usage

```r
solve(a, b, ...) 
```

---

### 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.
b a vector, \texttt{sparseVector}, matrix, or \texttt{Matrix} satisfying \texttt{NROW(b) == nrow(a)}, giving the right-hand side(s) of the linear system. Vectors \( b \) are treated as \texttt{length(b)-by-1} matrices. If \( b \) is missing, then methods take \( b \) to be an identity matrix.

tol a non-negative number. For a inheriting from \texttt{denseMatrix}, an error is signaled if the reciprocal one-norm condition number (see \texttt{rcond}) of \( a \) is less than \( \texttt{tol} \), indicating that \( a \) is near-singular. For a of class \texttt{sparseLU}, an error is signaled if the ratio \( \min(d)/\max(d) \) is less than \( \texttt{tol} \), where \( d = \text{abs(diag(a@U))} \). (Interpret with care, as this ratio is a cheap heuristic and not in general equal to or even proportional to the reciprocal one-norm condition number.) Setting \( \texttt{tol} = 0 \) disables the test.

sparse a logical indicating if the result should be formally sparse, i.e., if the result should inherit from virtual class \texttt{sparseMatrix}. Only methods for sparse \( a \) and missing or matrix \( b \) have this argument. Methods for missing or sparse \( b \) use \( \texttt{sparse} = \texttt{TRUE} \) by default. Methods for dense \( b \) use \( \texttt{sparse} = \texttt{FALSE} \) by default.

system a string specifying a linear system to be solved. Only methods for a inheriting from \texttt{CHMfactor} have this argument. See ‘Details’.

... 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 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 \texttt{d..Matrix}, where \( .. = \text{(ge|tr|tp|sy|sp|po|pp)} \), uses routines \texttt{d..tri} and \texttt{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 CSparse routines \texttt{cs_lsolve}, \texttt{cs_usolve}, and \texttt{cs_spsolve} and CHOLMOD routines \texttt{cholmod_solve} and \texttt{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 \texttt{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 \texttt{sparseQR} are simple wrappers around \texttt{qr.coef}, giving the least squares solution in overdetermined cases.

Methods for a inheriting from \texttt{CHMfactor} can solve systems other than the default one \( AX = B \). The correspondence between its \texttt{system} argument the system actually solved is outlined in the table below. See \texttt{CHMfactor-class} for a definition of notation.

<table>
<thead>
<tr>
<th>system</th>
<th>isLDL(a)=TRUE</th>
<th>isLDL(a)=FALSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
<td>( AX = B )</td>
<td>( AX = B )</td>
</tr>
<tr>
<td>&quot;LDLt&quot;</td>
<td>( L_1 D L_1^T X = B )</td>
<td>( LL' X = B )</td>
</tr>
<tr>
<td>&quot;LD&quot;</td>
<td>( L_1 D X = B )</td>
<td>( LX = B )</td>
</tr>
</tbody>
</table>
sparse.model.matrix

Construct Sparse Design / Model Matrices

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:

```r
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(XXt <- kronecker(XXt, Diagonal(x=c(4,1))))
image(a <- 2*Diagonal(n) + ZZ %x% Diagonal(x=c(10, rep(1, n-1))))

isSymmetric(a) # FALSE
image(drop0(sparse.model.matrix(~ a))) # checker board, dense [but really, a is singular!]

try(solve(a, tol = 1e-19))##-> error [ TODO: assertError ]
image(drop0(sparse.model.matrix(~ a))) # no error

if(R.version$arch == "x86_64")
  stopifnot(all.equal(as.matrix(ia0), as.matrix(ia0)))

a <- a + Diagonal(n)

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))
)
```
sparse.model.matrix

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 model.frame. If another sort of object, model.frame is called first.
contrasts.arg for sparse.model.matrix(): A list, whose entries are contrasts suitable for input to the contrasts replacement function and whose names are the names of columns of data containing factors.
for fac2Sparse(): character string or NULL or (coercable to) "sparseMatrix", specifying the contrasts to be applied to the factor levels.
xlev to be used as argument of model.frame if data has no “terms” attribute.
transpose logical indicating if the transpose should be returned; if the transposed is used anyway, setting transpose = TRUE is more efficient.
drop.unused.levels should factors have unused levels dropped? The default for sparse.model.matrix has been changed to FALSE, 2010-07, for compatibility with R’s standard (dense) model.matrix().
row.names logical indicating if row names should be used.
sep character string passed to paste() 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 representation 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 \texttt{fac2sparse()}.

**Value**

a sparse matrix, extending \texttt{CsparseMatrix} (for \texttt{fac2sparse()} if \texttt{repr = "C"} as per default; a \texttt{TsparseMatrix} or \texttt{RsparseMatrix}, otherwise).

For \texttt{fac2Sparse()}, a \texttt{list} of length two, both components with the corresponding transposed model matrix, where the corresponding \texttt{factorPatt12} is true.

\texttt{fac2sparse()}, the basic workhorse of \texttt{sparse.model.matrix()}, returns the \texttt{transpose} (\texttt{t}) of the model matrix.

**Note**

\texttt{model.Matrix(sparse = TRUE)} from package \texttt{MatrixModels} may be nowadays be preferable to \texttt{sparse.model.matrix}, as \texttt{model.Matrix} returns an object of class \texttt{modelMatrix} with additional slots \texttt{assign} and \texttt{contrasts} relating to the model variables.

**Author(s)**

Doug Bates and Martin Maechler, with initial suggestions from Tim Hesterberg.

**See Also**

\texttt{model.matrix} in package \texttt{stats}, part of base \texttt{R}.
\texttt{model.Matrix} in package \texttt{MatrixModels}; see ‘Note’.

\texttt{as(f, "sparseMatrix")} (see \texttt{coerce(from = "factor", \ldots)} in the class doc \texttt{sparsesMatrix}) produces the \texttt{transposed} sparse model matrix for a single factor \texttt{f} (and \texttt{no} contrasts).

**Examples**

```r
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 = list(a="contr.sum"))
sparse.model.matrix(~ a + b, dd, contrasts = 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)
)

<table>
<thead>
<tr>
<th>sparseLU-class</th>
<th>Sparse LU Factorizations</th>
</tr>
</thead>
</table>

**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(from = "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


See Also

Class `denseLU` for dense LU factorizations.

Class `dgCMatrix`.

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

Examples

```r
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 \texttt{CsparseMatrix}, \texttt{RsparseMatrix}, or \texttt{TsparseMatrix}) from the positions and values of their nonzero entries.

This interface is recommended over direct construction via calls such as \texttt{new("...[CRT]Matrix", ...)}.

Usage

\begin{verbatim}
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)
\end{verbatim}

Arguments

- \texttt{i}, \texttt{j} integer vectors of equal length specifying the positions (row and column indices) of the nonzero (or non-\texttt{TRUE}) entries of the matrix. Note that, when \texttt{x} is non-missing, the \texttt{x}_{k} corresponding to repeated pairs \((i_k, j_k)\) are added, for consistency with the definition of class \texttt{TsparseMatrix}, unless \texttt{use.last.ij} is \texttt{TRUE}, in which case only the last such \texttt{x}_{k} is used.
- \texttt{p} 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 \texttt{i}, \texttt{j}, and \texttt{p} must be missing.
- \texttt{x} optional, typically nonzero values for the matrix entries. If specified, then the length must equal that of \texttt{i} (or \texttt{j}) or equal 1, in which case \texttt{x} is recycled as necessary. If missing, then the result is a nonzero pattern matrix, i.e., inheriting from class \texttt{nsparseMatrix}.
- \texttt{dims} optional length-2 integer vector of matrix dimensions. If missing, then \texttt{index1+c(max(i),max(j))} is used.
- \texttt{dimnames} optional list of \texttt{dimnames}; if missing, then \texttt{NULL} ones are used.
- \texttt{symmetric} logical indicating if the resulting matrix should be symmetric. In that case, \((i, j, p)\) should specify only one triangle (upper or lower).
- \texttt{triangular} logical indicating if the resulting matrix should be triangular. In that case, \((i, j, p)\) should specify only one triangle (upper or lower).
- \texttt{index1} logical. If \texttt{TRUE} (the default), then \texttt{i} and \texttt{j} are interpreted as 1-based indices, following the \texttt{R} convention. That is, counting of rows and columns starts at 1. If \texttt{FALSE}, then they are interpreted as 0-based indices.
- \texttt{repr} \texttt{character} string, one of \texttt{"C"}, \texttt{"R"}, and \texttt{"T"}, specifying the \texttt{representation} of the sparse matrix result, i.e., specifying one of the virtual classes \texttt{CsparseMatrix}, \texttt{RsparseMatrix}, and \texttt{TsparseMatrix}.
- \texttt{giveCsparse} \texttt{(deprecated, replaced by \texttt{repr})} logical indicating if the result should inherit from \texttt{CsparseMatrix} or \texttt{TsparseMatrix}. Note that operations involving \texttt{CsparseMatrix} are very often (but not always) more efficient.
check 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!

use.last.ij 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 \texttt{TsparseMatrix}.

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 \(p\), \texttt{rep(seq_along(dp),dp)} where \(dp <- \text{diff(p)}\), is used as the (1-based) row or column indices.

You cannot set both \texttt{singular} and \texttt{triangular} to true; rather use \texttt{Diagonal()} (or its alternatives, see there).

The values of \(i\), \(j\), \(p\) and \texttt{index1} are used to create 1-based index vectors \(i\) and \(j\) from which a \texttt{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 \texttt{TsparseMatrix} class, unless \texttt{use.last.ij} is set to true.

By default, when \texttt{repr = "C"}, the \texttt{CsparseMatrix} derived from this triplet form is returned, where \texttt{repr = "R"} now allows to directly get an \texttt{RsparseMatrix} and \texttt{repr = "T"} leaves the result as \texttt{TsparseMatrix}.

The reason for returning a \texttt{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 \texttt{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 \texttt{CsparseMatrix} and \texttt{generalMatrix}.

Note

You do need to use \texttt{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

\texttt{Matrix}(\(*\), \texttt{sparse=TRUE}) for the constructor of such matrices from a \texttt{dense} matrix. That is easier in small sample, but much less efficient (or impossible) for large matrices, where something like \texttt{sparsesMatrix()} is needed. Further \texttt{bdiag} and \texttt{Diagonal} for (block-)diagonal and \texttt{bandSparse} for banded sparse matrix constructors.

Random sparse matrices via \texttt{rsparsematrix()}.

The standard \texttt{R xtabs}(\(*\), \texttt{sparse=TRUE}), for sparse tables and \texttt{sparses.model.matrix()} for building sparse model matrices.

Consider \texttt{CsparseMatrix} and similar class definition help files.
Examples

```r
## 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) # 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)
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' --> pattern 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,
```
sparseMatrix-class

Virtual Class "sparseMatrix" — Mother of Sparse Matrices

Description

Virtual Mother Class of All Sparse Matrices

Slots

Dim: Object of class "integer" - the dimensions of the matrix - must be an integer vector with exactly two non-negative values.

Dimnames: a list of length two - inherited from class Matrix, see Matrix.

Extends

Class "Matrix", directly.

Methods

show (object = "sparseMatrix"): The show method for sparse matrices prints "structural" zeroes as "." using printSpMatrix() which allows further customization.

print signature(x = "sparseMatrix"). ....

The print method for sparse matrices by default is the same as show() but can be called with extra optional arguments, see printSpMatrix().

format signature(x = "sparseMatrix"). ....

The format method for sparse matrices, see formatSpMatrix() for details such as the extra optional arguments.

```
--snip--
```
**summary** (object = "sparseMatrix", uniqT=FALSE): Returns an object of S3 class "sparseSummary" which is basically a data.frame with columns (i,j,x) (or just (i,j) for nsparseMatrix class objects) with the stored (typically non-zero) entries. The print method resembles Mat-lab’s way of printing sparse matrices, and also the MatrixMarket format, see writeMM.

cbind2 (x = *, y = *): several methods for binding matrices together, column-wise, see the basic cbind and rbind functions.

Note that the result will typically be sparse, even when one argument is dense and larger than the sparse one.

dim< signature(x = "sparseMatrix", value = "ANY"): allows to reshape a sparse matrix to a sparse matrix with the same entries but different dimensions. value must be of length two and fulfill prod(value) == prod(dim(x)).

core signature(from = "factor", to = "sparseMatrix"): Coercion of a factor to "sparseMatrix" produces the matrix of indicator rows stored as an object of class "dgCMatrix". To obtain columns representing the interaction of the factor and a numeric covariate, replace the "x" slot of the result by the numeric covariate then take the transpose. Missing values (NA) from the factor are translated to columns of all 0s.

See also colSums, norm, ... for methods with separate help pages.

**Note**

In method selection for multiplication operations (i.e. %*% and the two-argument form of crossprod) the sparseMatrix class takes precedence in the sense that if one operand is a sparse matrix and the other is any type of dense matrix then the dense matrix is coerced to a dgeMatrix and the appropriate sparse matrix method is used.

**See Also**

sparseMatrix, and its references, such as xtabs(*, sparse=TRUE), or sparse.model.matrix(), for constructing sparse matrices.

T2graph for conversion of "graph" objects (package graph) to and from sparse matrices.

**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

T2graph for conversion of "graph" objects (package graph) to and from sparse matrices.
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

\[
P_1AP_2 = QR = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1R_1
\]

or (equivalently)

\[
A = P'_1QRP'_2 = P'_1 \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} P'_2 = P'_1Q_1R_1P'_2
\]

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).

Usage

\[
\text{qrR(qr, complete = FALSE, backPermute = TRUE, row.names = TRUE)}
\]

Arguments

- **qr** an object of class \texttt{sparseQR}, almost always the result of a call to generic function \texttt{qr} with sparse \texttt{x}.
- **complete** a logical indicating if \( R \) should be returned instead of \( R_1 \).
- **backPermute** a logical indicating if \( R \) or \( R_1 \) should be multiplied on the right by \( P'_2 \).
- **row.names** a logical indicating if \( \text{dimnames(qr)[1]} \) should be propagated unpermuted to the result. If \( \text{complete} = \text{FALSE} \), then only the first \( n \) names are kept.

Details

The method for \texttt{qr.Q} does not return \( Q \) but rather the (also orthogonal) product \( P'_1Q \). This behaviour is algebraically consistent with the base implementation (see \texttt{qr}), which can be seen by noting that \texttt{qr.default} in base does not pivot rows, constraining \( P_1 \) to be an identity matrix. It follows that \( \text{qr.Q(qr.default(x))} \) also returns \( P'_1Q \).

Similarly, the methods for \texttt{qr.qy} and \texttt{qr.qty} multiply on the left by \( P'_1Q \) and \( Q'P_1 \) rather than \( Q \) and \( Q' \).

It is wrong to expect the values of \texttt{qr.Q} (or \texttt{qr.R}, \texttt{qr.qy}, \texttt{qr.qty}) computed from “equivalent” sparse and dense factorizations (say, \texttt{qr(x)} and \texttt{qr(as(x, "matrix"))}) for \texttt{x} of class \texttt{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 \( \text{qr.X} \), \( \text{qr.coef} \), \( \text{qr.fitted} \), and \( \text{qr.resid} \) are well-defined, and in those cases the sparse and dense computations should compare equal (within some tolerance).

The method for \( \text{qr.R} \) is a simple wrapper around \( \text{qrR} \), but not back-permuting by default and never giving row names. It did not support \texttt{backPermute = TRUE} until \texttt{Matrix 1.6-0}, hence code needing the back-permuted result should call \( \text{qrR} \) if \texttt{Matrix >= 1.6-0} is not known.

### Slots

- **Dim, Dimnames** inherited from virtual class \texttt{MatrixFactorization}.
- \texttt{beta} a numeric vector of length \texttt{Dim[2]}, used to construct Householder matrices; see \texttt{V} below.
- \texttt{V} an object of class \texttt{dgCMatrix} with \texttt{Dim[2]} columns. The number of rows \texttt{nrow(V)} is at least \texttt{Dim[1]} and at most \texttt{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 \( \text{diag(Dim[1])} - \text{beta[j]} \times \text{tcrossprod(V[, j])} \).
- \texttt{R} an object of class \texttt{dgCMatrix} with \texttt{nrow(V)} rows and \texttt{Dim[2]} columns. \( R \) is the upper trapezoidal \( R \) factor.
- \texttt{p, q} 0-based integer vectors of length \texttt{nrow(V)} and \texttt{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 \texttt{R} syntax, the matrix \( P_1^T A P_2 \) is precisely \( A[p+1, q+1] \) (\( A[p+1, \] when \( q \) has length 0).

### Extends

Class \texttt{QR}, directly. Class \texttt{MatrixFactorization}, by class \texttt{QR}, distance 2.

### Instantiation

Objects can be generated directly by calls of the form \texttt{new("sparseQR", ...)}, but they are more typically obtained as the value of \texttt{qr(x)} for \( x \) inheriting from \texttt{sparseMatrix} (often \texttt{dgCMatrix}).

### Methods

- \texttt{determinant} signature (\texttt{from = "sparseQR", logarithm = "logical"}): computes the determinant of the factorized matrix \( A \) or its logarithm.
- \texttt{expand1} signature (\texttt{x = "sparseQR"): see \texttt{expand1-methods}.
- \texttt{expand2} signature (\texttt{x = "sparseQR"): see \texttt{expand2-methods}.
- \texttt{qr.Q} signature (\texttt{qr = "sparseQR"): returns as a \texttt{dgeMatrix} either \( P_1^T \) or \( P_1^T Q_1 \), depending on optional argument \texttt{complete}. The default is \texttt{FALSE}, indicating \( P_1^T Q_1 \).
- \texttt{qr.R} signature (\texttt{qr = "sparseQR"): \texttt{qrR} returns \( R \), \( R_1 \), \( RP_2 \), or \( R_1 P_2 \), depending on optional arguments \texttt{complete} and \texttt{backPermute}. The default in both cases is \texttt{FALSE}, indicating \( R_1 \), for compatibility with \texttt{base}. The class of the result in that case is \texttt{dtCMatrix}. In the other three cases, it is \texttt{dgCMatrix}.
- \texttt{qr.X} signature (\texttt{qr = "sparseQR"): returns \( A \) as a \texttt{dgeMatrix}, by default. If \( m > n \) and optional argument \texttt{ncol} is greater than \( n \), then the result is augmented with \( P_1^T Q_1 J \), where \( J \) is composed of columns \((n+1)\) through \texttt{ncol} of the \( m \times m \) identity matrix.
- \texttt{qr.coef} signature (\texttt{qr = "sparseQR", y = .}): returns as a \texttt{dgeMatrix} or vector the result of multiplying \( y \) on the left by \( P_2 R_1^{-1} Q_1 P_1 \).
- \texttt{qr.fitted} signature (\texttt{qr = "sparseQR", y = .}): returns as a \texttt{dgeMatrix} or vector the result of multiplying \( y \) on the left by \( P_1^T Q_1 Q_1^T P_1 \).
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


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.


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
    }
  }
}

slowQ(A, c(1:3))

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

ae2 <- function(a, b, ...) ae1(unname(a), unname(b), ...)

## A ~ P1' Q R P2' ~ 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."]$pMatrix$, 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."]$pMatrix$, 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 - qrf)
  ae2(qrq <- qr.qy(qr.A, b), with(E.qr.A, P1. %*% Q %*% b))
  ae2(qrr, qr.resid (qr.Am, 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**
sparseVector

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(0,0,0,3.2, 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))))
Description

Sparse Vector Classes: The virtual mother class "sparseVector" has the five actual daughter classes "dsparseVector", "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 >= 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 × n Toeplitz matrix from x, where n = 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.

[ signature(x = "atomicVector", i = ...): not only can you subset (aka “index into”) sparseVectors x[i] using sparseVectors i, but we also support efficient subsetting of traditional vectors x by logical sparse vectors (i.e., i of class "nsparseVector" or "lsparseVector").

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.

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 "isparsiveVector" (!)
(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:
## 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[,1]), !any(lis[,1]), length(nn@i) == 0, !any(nn),
  all(!isn),
  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**

```r
spMatrix(nrow, ncol, i = integer(0L), j = integer(0L), x = double(0L))
```
spMatrix

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, ..., n, where n = 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) # 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))
}
Subassign-methods

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**

code Snippet:
```r
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

Methods for "[", i.e., extraction or subsetting mostly of matrices, in package `Matrix`.

### Methods

There are more than these:

- `x = "Matrix", i = "missing", j = "missing", drop= "ANY"` ...
- `x = "Matrix", i = "numeric", j = "missing", drop= "missing"` ...
- `x = "Matrix", i = "missing", j = "numeric", drop= "missing"` ...
- `x = "dsparseMatrix", i = "missing", j = "numeric", drop= "logical"` ...
- `x = "dsparseMatrix", i = "numeric", j = "missing", drop= "logical"` ...
- `x = "dsparseMatrix", i = "numeric", j = "numeric", drop= "logical"` ...

### See Also

`[<--methods` for subassignment to "Matrix" objects. `Extract` about the standard extraction.

### Examples

```r
str(m <- Matrix(round(rnorm(7*4),2), nrow = 7))
stopifnot(identical(m, m[]))
m[2, 3] # simple number
m[2, 3:4] # simple numeric of length 2
m[2, 3:4, drop=FALSE] # sub matrix of class 'dgeMatrix'
## rows or columns only:
m[1,] # first row, as simple numeric vector
m[,1:2] # sub matrix of first two columns
showMethods("[", inherited = FALSE)
```
Virtual Class of Symmetric Matrices in Package Matrix

Description

The virtual class of symmetric matrices, "symmetricMatrix", from the package Matrix contains numeric and logical, dense and sparse matrices, e.g., see the examples with the “actual” subclasses.

The main use is in methods (and C functions) that can deal with all symmetric matrices, and in as(*, "symmetricMatrix").

Slots

- **uplo**: Object of class "character". Must be either "U", for upper triangular, and "L", for lower triangular.
- **Dim, Dimnames**: The dimension (a length-2 "integer") and corresponding names (or NULL), inherited from the Matrix, see there. See below, about storing only one of the two Dimnames components.
- **factors**: a list of matrix factorizations, also from the Matrix class.

Extends

Class "Matrix", directly.

Methods

- **dimnames** signature(object = "symmetricMatrix"): returns symmetric dimnames, even when the Dimnames slot only has row or column names. This allows to save storage for large (typically sparse) symmetric matrices.
- **isSymmetric** signature(object = "symmetricMatrix"): returns TRUE trivially.

There’s a C function symmetricMatrix_validate() called by the internal validity checking functions, and also from getValidity(getClass("symmetricMatrix")).

Validity and dimnames

The validity checks do not require a symmetric Dimnames slot, so it can be list(NULL, <character>), e.g., for efficiency. However, dimnames() and other functions and methods should behave as if the dimnames were symmetric, i.e., with both list components identical.

See Also

isSymmetric which has efficient methods (isSymmetric-methods) for the Matrix classes. Classes triangularMatrix, and, e.g., dsyMatrix for numeric dense matrices, or lsCMatrix for a logical sparse matrix class.
Examples

```r
## 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
dimnames(sy) # both parts - as sy *is* symmetrical
```

```r
showClass("symmetricMatrix")
```

```r
## The names of direct subclasses:
scl <- getClass("symmetricMatrix")@subclasses
directly <- sapply(lapply(scl, slot, "by"), length) == 0
names(scl)[directly]
```

```r
## Methods -- applicable to all subclasses above:
showMethods(classes = "symmetricMatrix")
```

---

**symmpart-methods**

**Symmetric Part and Skew(symmetric) Part of a Matrix**

**Description**

`symmpart(x)` computes the symmetric part \( (x + t(x))/2 \) and `skewpart(x)` the skew symmetric part \( (x - t(x))/2 \) of a square matrix \( x \), more efficiently for specific Matrix classes.

Note that \( x == \text{symmpart}(x) + \text{skewpart}(x) \) for all square matrices – apart from extraneous NA values in the RHS.

**Usage**

```r
symmpart(x)
skewpart(x)
```

**Arguments**

- `x` - a square matrix; either “traditional” of class "matrix", or typically, inheriting from the `Matrix` class.

**Details**

These are generic functions with several methods for different matrix classes, use e.g., `showMethods(symmpart)` to see them.

If the row and column names differ, the result will use the column names unless they are (partly) NULL where the row names are non-NULL (see also the examples).

**Value**

- `symmpart(x)` returns a symmetric matrix, inheriting from `symmetricMatrix` or `diagonalMatrix` if \( x \) inherits from `Matrix`.
- `skewpart(x)` returns a skew-symmetric matrix, inheriting from `generalMatrix`, `symmetricMatrix` or `diagonalMatrix` if \( x \) inherits from `Matrix`. 
triangularMatrix-class

Virtual Class of Triangular Matrices in Package Matrix

Description

The virtual class of triangular matrices, "triangularMatrix", the package Matrix contains square (nrow == ncol) numeric and logical, dense and sparse matrices, e.g., see the examples. A main use of the virtual class is in methods (and C functions) that can deal with all triangular matrices.

Slots

uplo: String (of class "character"). Must be either "U", for upper triangular, and "L", for lower triangular.
diag: String (of class "character"). Must be either "U", for unit triangular (diagonal is all ones), or "N" for non-unit. The diagonal elements are not accessed internally when diag is "U". For denseMatrix classes, they need to be allocated though, such that the length of the x slot does not depend on diag.
Dim, Dimnames: The dimension (a length-2 "integer") and corresponding names (or NULL), inherited from the Matrix, see there.

Extends

Class "Matrix", directly.

Methods

There’s a C function triangularMatrix_validity() called by the internal validity checking functions.
Currently, Schur, isSymmetric and as() (i.e. coerce) have methods with triangularMatrix in their signature.
See Also

`isTriangular()` for testing any matrix for triangularity; classes `symmetricMatrix`, and, e.g.,
`dtrMatrix` for numeric `dense` matrices, or `ltCMatrix` for a logical `sparse` matrix subclass of
"triangularMatrix".

Examples

```r
showClass("triangularMatrix")

## The names of direct subclasses:
scf <- getClass("triangularMatrix")@subclasses
directly <- sapply(lapply(scf, slot, "by"), length) == 0
names(scf)[directly]

(m <- matrix(c(5,1,0,3), 2))
as(m, "triangularMatrix")
```

### TsparseMatrix-class

**Class "TsparseMatrix" of Sparse Matrices in Triplet Form**

**Description**

The "TsparseMatrix" class is the virtual class of all sparse matrices coded in triplet form. Since
it is a virtual class, no objects may be created from it. See `showClass("TsparseMatrix")` for its
subclasses.

**Slots**

- `Dim, Dimnames`: from the "Matrix" class,
- `i`: Object of class "integer" - the row indices of non-zero entries in 0-base, i.e., must be in
  `0:(nrow(.)-1)`.
- `j`: Object of class "integer" - the column indices of non-zero entries. Must be the same length
  as slot `i` and 0-based as well, i.e., in `0:(ncol(.)-1)`. For numeric Tsparse matrices, `(i,j)`
pairs can occur more than once, see `dgTMatrix`.

**Extends**

Class "sparseMatrix", directly. Class "Matrix", by class "sparseMatrix".

**Methods**

Extraction ("\[\"") methods, see `[-methods`.

**Note**

Most operations with sparse matrices are performed using the compressed, column-oriented or
`CsparseMatrix` representation. The triplet representation is convenient for creating a sparse ma-
trix or for reading and writing such matrices. Once it is created, however, the matrix is generally
coerced to a `CsparseMatrix` for further operations.

Note that all `new(.), spMatrix` and `sparseMatrix(*, repr="T")` constructors for "TsparseMatrix"
classes implicitly add (i.e., “sum up”) `x_k`'s that belong to identical `(i_k, j_k)` pairs, see, the example
below, or also "dgTMatrix".
For convenience, methods for some operations such as `%*%` and `crossprod` are defined for `TsparseMatrix` objects. These methods simply coerce the `TsparseMatrix` object to a `CsparseMatrix` object then perform the operation.

See Also
its superclass, `sparseMatrix`, and the `dgTMatrix` class, for the links to other classes.

Examples

```r
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 !
```

unpackedMatrix-class  Virtual Class "unpackedMatrix" of Unpacked Dense Matrices

Description

Class "unpackedMatrix" is the virtual class of dense matrices in "unpacked" format, storing all \(m \times n\) elements of an \(m\)-by-\(n\) matrix. It is used to define common methods for efficient subsetting, transposing, etc. of its proper subclasses: currently "[dln]geMatrix" (unpacked general), "[dln]syMatrix" (unpacked symmetric), "[dln]trMatrix" (unpacked triangular), and subclasses of these, such as "dpoMatrix", "Cholesky", and "BunchKaufman".

Slots

- `Dim, Dimnames`: as all `Matrix` objects.

Extends


Methods

- `pack` signature(x = "unpackedMatrix"): ...
- `unpack` signature(x = "unpackedMatrix"): ...
- `isSymmetric` signature(object = "unpackedMatrix"): ...
- `isTriangular` signature(object = "unpackedMatrix"): ...
- `isDiagonal` signature(object = "unpackedMatrix"): ...
- `t` signature(x = "unpackedMatrix"): ...
- `diag` signature(x = "unpackedMatrix"): ...
- `diag<-` signature(x = "unpackedMatrix"): ...
updown-methods

Author(s)
Mikael Jagan

See Also
pack and unpack; its virtual "complement" "packedMatrix"; its proper subclasses "dsyMatrix", "ltrMatrix", etc.

Examples
showClass("unpackedMatrix")
showMethods(classes = "unpackedMatrix")

Description
Computes a rank-$k$ update or downdate of a sparse Cholesky factorization

$$P_1 A P_1' = L_1 D L_1' = LL'$$

which for some $k$-column matrix $C$ is the factorization

$$P_1 (A + sCC') P_1' = \tilde{L}_1 \tilde{D} \tilde{L}_1' = \tilde{L}\tilde{L}'$$

Here, $s = 1$ for an update and $s = -1$ for a downdate.

Usage
updown(update, C, L)

Arguments
update a logical (TRUE or FALSE) or character ("+" or "-") indicating if L should be updated (or otherwise downdated).
C a finite matrix or Matrix such that tcrossprod(C) has the dimensions of L.
L an object of class dCHMsimpl or dCHMsuper specifying a sparse Cholesky factorization.

Value
A sparse Cholesky factorization with dimensions matching L, typically of class dCHMsimpl.

Author(s)
Initial implementation by Nicholas Nagle, University of Tennessee.

References
See Also

Classes dCHMsimpl and dCHMsuper and their methods, notably for generic function update, which is not equivalent to updown(update = TRUE).

Generic function Cholesky.

Examples

```r
m <- sparseMatrix(i = c(3, 1, 3:2, 2:1), p = c(0:2, 4, 4, 6), x = 1:6, 
    dimnames = list(LETTERS[1:3], letters[1:5]))
uc0 <- Cholesky(A <- crossprod(m) + Diagonal(5))
uc1 <- updown("+", Diagonal(5, 1), uc0)
uc2 <- updown("-", Diagonal(5, 1), uc1)
stopifnot(all.equal(uc0, uc2))
```

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

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))

---

**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


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)

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))
Index

![,Matrix-method (Matrix-class), 116
![,indMatrix-method (indMatrix-class), 91
![,ldenseMatrix-method (ldenseMatrix-class), 104
![,ldiMatrix-method (ldiMatrix-class), 104
![,lsparseMatrix-method (lsparseMatrix-classes), 106
![,lsparseVector-method (sparseVector-class), 172
![,ndenseMatrix-method (ndenseMatrix-class), 121
![,ndiMatrix-method (ldiMatrix-class), 104
![,nsparseMatrix-method (nsparseMatrix-classes), 129
![,nsparseVector-method (sparseVector-class), 172

* Choleski
  Cholesky-methods, 32
* NA
  is.na-methods, 95

* algebra
  boolmatmult-methods, 14
  BunchKaufman-class, 15
  BunchKaufman-methods, 17
  CHMfactor-class, 21
  chol-methods, 25
  chol2inv-methods, 28
  Cholesky-class, 29
  Cholesky-methods, 32
  colSums-methods, 39
  compMatrix-class, 41
  condest, 42
  denseLU-class, 47
  dimScale, 58
  dmperm, 60
  expand-methods, 75
  KhatriRao, 100
  kronecker-methods, 103
  lu-methods, 110
  matmult-methods, 113
  MatrixFactorization-class, 120
  nearPD, 122
  norm-methods, 128
  qr-methods, 140
  rankMatrix, 143
  rcond-methods, 146
  Schur-class, 152
  Schur-methods, 153
  solve-methods, 155
  sparseLU-class, 160
  sparseQR-class, 167
  symmpart-methods, 179
  updown-methods, 183

* arith
  all.equal-methods, 6
  colSums-methods, 39
  dimScale, 58
  facmul-methods, 80
  KhatriRao, 100
  kronecker-methods, 103
  matmult-methods, 113
  symmpart-methods, 179

* array
  asUniqueT, 7
  band-methods, 9
  bandSparse, 11
  bdiag, 12
  boolmatmult-methods, 14
  BunchKaufman-class, 15
  BunchKaufman-methods, 17
  cbind2-methods, 20
  CHMfactor-class, 21
  chol-methods, 25
  chol2inv-methods, 28
  Cholesky-class, 29
  Cholesky-methods, 32
  colSums-methods, 39
  compMatrix-class, 41
  CsparseMatrix-class, 44
  ddenseMatrix-class, 45
  denseLU-class, 47

187
denseMatrix-class, 49
dgCMatrix-class, 49
dgeMatrix-class, 50
dgRMatrix-class, 51
dgTMatrix-class, 52
Diagonal, 53
diagonalMatrix-class, 55
diagU2N, 57
dimScale, 58
dMatrix-class, 59
dmperm, 60
dpOMatrix-class, 62
drop0, 64
dsCMatrix-class, 65
dsparsetMatrix-class, 66
dsymMatrix-class, 67
dsysMatrix-class, 68
dtCMatrix-class, 70
dtpMatrix-class, 71
dtRMatrix-class, 73
dtrMatrix-class, 74
expand-methods, 75
expm-methods, 77
facmul-methods, 80
forceSymmetric-methods, 85
generalMatrix-class, 87
Hilbert, 88
indMatrix-class, 91
is.null.DN, 96
isSymmetric-methods, 98
isTriangular-methods, 99
KhatriRao, 100
kronecker-methods, 103
ldenseMatrix-class, 104
ldiMatrix-class, 104
lgeMatrix-class, 105
lsparseMatrix-classes, 106
lsyMatrix-class, 108
ltreMatrix-class, 109
lu-methods, 110
mat2triplet, 112
matmult-methods, 113
Matrix, 115
Matrix-notyet, 118
MatrixFactorization-class, 120
ndenseMatrix-class, 121
nearPD, 122
ngeMatrix-class, 125
mMatrix-class, 126
nnzero-methods, 127
nsparsmatrices, 129
nsyMatrix-class, 131
ntrMatrix-class, 132
pack, 133
packedMatrix-class, 135
pMatrix-class, 136
qr-methods, 140
rsparsematrix, 150
RsparseMatrix-class, 151
Schur-class, 152
Schur-methods, 153
solve-methods, 155
sparse.model.matrix, 157
sparsLU-class, 160
sparseMatrix, 162
sparseMatrix-class, 165
sparseQR-class, 167
spMatrix, 174
Subassign-methods, 176
Subscript-methods, 177
symmetricMatrix-class, 178
symmpart-methods, 179
triangularMatrix-class, 180
tsparsetMatrix-class, 181
unpackedMatrix-class, 182
updown-methods, 183
* attribute
diagU2N, 57
is.null.DN, 96
* character
formatSparseM, 86
printSpMatrix, 137
* classes
abIndex-class, 4
atomicVector-class, 8
BunchKaufman-class, 15
CHMfactor-class, 21
Cholesky-class, 29
compMatrix-class, 41
CsparseMatrix-class, 44
ddenseMatrix-class, 45
ddiMatrix-class, 46
denseLU-class, 47
denseMatrix-class, 49
dgCMatrix-class, 49
dgeMatrix-class, 50
dgRMatrix-class, 51
dgTMatrix-class, 52
diagonalMatrix-class, 55
dMatrix-class, 59
dpOMatrix-class, 62
dsCMatrix-class, 65
dsparsetMatrix-class, 66
INDEX

<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsRMatrix-class</td>
<td>67</td>
</tr>
<tr>
<td>dsyMatrix-class</td>
<td>68</td>
</tr>
<tr>
<td>dtCMatrix-class</td>
<td>70</td>
</tr>
<tr>
<td>dtpMatrix-class</td>
<td>71</td>
</tr>
<tr>
<td>dtRMatrix-class</td>
<td>73</td>
</tr>
<tr>
<td>dtrMatrix-class</td>
<td>74</td>
</tr>
<tr>
<td>generalMatrix-class</td>
<td>87</td>
</tr>
<tr>
<td>index-class</td>
<td>91</td>
</tr>
<tr>
<td>indMatrix-class</td>
<td>91</td>
</tr>
<tr>
<td>ldenseMatrix-class</td>
<td>104</td>
</tr>
<tr>
<td>ldiMatrix-class</td>
<td>104</td>
</tr>
<tr>
<td>lgeMatrix-class</td>
<td>105</td>
</tr>
<tr>
<td>lsparseMatrix-classes</td>
<td>106</td>
</tr>
<tr>
<td>lsyMatrix-class</td>
<td>108</td>
</tr>
<tr>
<td>ltrMatrix-class</td>
<td>109</td>
</tr>
<tr>
<td>Matrix-class</td>
<td>116</td>
</tr>
<tr>
<td>Matrix-notyet</td>
<td>118</td>
</tr>
<tr>
<td>MatrixFactorization-class</td>
<td>120</td>
</tr>
<tr>
<td>ndenseMatrix-class</td>
<td>121</td>
</tr>
<tr>
<td>nMatrix-class</td>
<td>126</td>
</tr>
<tr>
<td>nsparseMatrix-classes</td>
<td>129</td>
</tr>
<tr>
<td>nsyMatrix-class</td>
<td>131</td>
</tr>
<tr>
<td>ntrMatrix-class</td>
<td>132</td>
</tr>
<tr>
<td>number-class</td>
<td>133</td>
</tr>
<tr>
<td>packedMatrix-class</td>
<td>135</td>
</tr>
<tr>
<td>pMatrix-class</td>
<td>136</td>
</tr>
<tr>
<td>replValue-class</td>
<td>148</td>
</tr>
<tr>
<td>rleDiff-class</td>
<td>149</td>
</tr>
<tr>
<td>RsparseMatrix-class</td>
<td>151</td>
</tr>
<tr>
<td>Schur-class</td>
<td>152</td>
</tr>
<tr>
<td>sparseLU-class</td>
<td>160</td>
</tr>
<tr>
<td>sparseMatrix-class</td>
<td>165</td>
</tr>
<tr>
<td>sparseQR-class</td>
<td>167</td>
</tr>
<tr>
<td>sparseVector-class</td>
<td>172</td>
</tr>
<tr>
<td>symmetricMatrix-class</td>
<td>178</td>
</tr>
<tr>
<td>triangularMatrix-class</td>
<td>180</td>
</tr>
<tr>
<td>TsparseMatrix-class</td>
<td>181</td>
</tr>
<tr>
<td>unpackedMatrix-class</td>
<td>182</td>
</tr>
</tbody>
</table>

* connection
  - externalFormats, 79

* datasets
  - CAex, 19
  - KNex, 102
  - USCounties, 184
  - wrld_1deg, 185

* distribution
  - rsparsematrix, 150

* file
  - externalFormats, 79

* hplot
  - image-methods, 88

* logic
  - all.equal-methods, 6
  - asUniqueT, 7
  - boolmatmult-methods, 14
  - nnzero-methods, 127

* manip
  - abIseq, 5
  - asUniqueT, 7
  - cbind2-methods, 20
  - drop0, 64
  - rep2abI, 148
  - sparseVector-class, 172

* math
  - condest, 42
  - expm-methods, 77
  - is.na-methods, 95
  - norm-methods, 128
  - rcond-methods, 146

* methods
  - all.equal-methods, 6
  - band-methods, 9
  - boolmatmult-methods, 14
  - BunchKaufman-methods, 17
  - cbind2-methods, 20
  - chol-methods, 25
  - chol2inv-methods, 28
  - Cholesky-methods, 32
  - coerce-methods-graph, 37
  - coerce-methods-SparseM, 38
  - colSums-methods, 39
  - expand-methods, 75
  - expm-methods, 77
  - externalFormats, 79
  - facmul-methods, 80
  - forceSymmetric-methods, 85
  - image-methods, 88
  - is.na-methods, 95
  - isSymmetric-methods, 98
  - isTriangular-methods, 99
  - kronecker-methods, 103
  - lu-methods, 110
  - nnzero-methods, 127
  - norm-methods, 128
  - pack, 133
  - qr-methods, 140
  - rcond-methods, 146
  - Schur-methods, 153
  - solve-methods, 155
  - Subassign-methods, 176
  - Subscript-methods, 177
  - symmpart-methods, 179
  - updown-methods, 183
INDEX

* models
  sparse.model.matrix, 157
  *,Matrix,ddiMatrix-method
    (Matrix-class), 116
  *,ddenseMatrix,ddiMatrix-method
    (ddenseMatrix-class), 45
  *,ndiMatrix,ddiMatrix-method
    (ndiMatrix-class), 45
  *,ddiMatrix,ddiMatrix-method
    (ddiMatrix-class), 46
  *,ldiMatrix,ddiMatrix-method
    (ldiMatrix-class), 46
  *,ndiMatrix,ldiMatrix-method
    (ndiMatrix-class), 104
  *,ldiMatrix,ndiMatrix-method
    (ldiMatrix-class), 104

* print
  formatSparseM, 86
  printSpMatrix, 137
  printSparseM, 86
  printSpMatrix, 137
  formatSparseM, 86
  printSpMatrix, 137
  formatSparseM, 86
  printSpMatrix, 137

* programming
  all.equal-methods, 6
  CHMfactor-class, 21
  is.na-methods, 95
  is.null.DN, 96
  isSymmetric-methods, 98
  isTriangular-methods, 99
  CHMfactor-class, 21
  is.na-methods, 95
  is.null.DN, 96
  isSymmetric-methods, 98
  isTriangular-methods, 99

* utilities
  abIseq, 5
  asUniqueT, 7
  bandSparse, 11
  bdiag, 12
  CHMfactor-class, 21
  coerce-methods-graph, 37
  condens, 42
  Diagonal, 53
  diagU2N, 57
  dimScale, 58
  dmperm, 60
  externalFormats, 79
  fastMisc, 81
  formatSparseM, 86
  Hilbert, 88
  invertPerm, 94
  is.null.DN, 96
  KhatriRao, 100
  mat2triplet, 112
  Matrix, 115
  MatrixClass, 119
  nearPD, 122
  printSpMatrix, 137
  rankMatrix, 143
  rep2abI, 148
  rsparsematrix, 150
  sparse.model.matrix, 157
  sparseMatrix, 162
  sparseQR-class, 167
  sparseVector, 170
  spMatrix, 174
  *,Matrix,ddiMatrix-method
    (Matrix-class), 116
  *,Matrix,ldiMatrix-method
    (Matrix-class), 116
  *,Matrix,ndiMatrix-method
    (Matrix-class), 116
  *,ddenseMatrix,ddiMatrix-method
    (dddenseMatrix-class), 45
  *,ddenseMatrix,ldiMatrix-method
    (dddenseMatrix-class), 45
  *,ddenseMatrix,ndiMatrix-method
    (dddenseMatrix-class), 45
  *,ddiMatrix,ldiMatrix-method
    (ddiMatrix-class), 46
  *,ldiMatrix,ddiMatrix-method
    (ldiMatrix-class), 46
  *,ldiMatrix,ndiMatrix-method
    (ldiMatrix-class), 104
  *,ndiMatrix,ddiMatrix-method
    (ndiMatrix-class), 121
  *,ndiMatrix,ldiMatrix-method
    (ndiMatrix-class), 121
  *,ndiMatrix,ndiMatrix-method
    (ndiMatrix-class), 121
  *,ldiMatrix,Matrix-method
    (ldiMatrix-class), 104
  *,ndiMatrix,ldiMatrix-method
    (ndiMatrix-class), 104
  *,ndiMatrix,ndiMatrix-method
    (ndiMatrix-class), 121
  *,Matrix,missing-method (Matrix-class), 116
  *,dgTMatrix,dgTMatrix-method
    (dgTMatrix-class), 52
  -,Matrix,missing-method (Matrix-class), 116
  -,denseMatrix,missing-method
    (denseMatrix-class), 49
  -,diagonalMatrix,missing-method
    (diagonalMatrix-class), 55
  -,dsparseVector,missing-method
INDEX

...(-, indMatrix, missing-method)
... (indMatrix-class), 91
...(-, nsparsMat, missing-method)
... (nsparsMat-class), 129
...(-, sparseMat, missing-method)
... (sparseMat-class), 165
... .4
... .CR2RC (fastMisc), 81
... .CR2T (fastMisc), 81
... .M2C (fastMisc), 81
... .M2R (fastMisc), 81
... .M2T (fastMisc), 81
... .M2V (fastMisc), 81
... .M2diag (fastMisc), 81
... .M2gen (fastMisc), 81
... .M2kind (fastMisc), 81
... .M2m (fastMisc), 81
... .M2packed (fastMisc), 81
... .M2sym (fastMisc), 81
... .M2tri (fastMisc), 81
... .M2unpacked (fastMisc), 81
... .M2v (fastMisc), 81
... .Machine, 143, 172
... .T2CR (fastMisc), 81
... .bdig (bdig), 12
... .dense2g (fastMisc), 81
... .dense2kind (fastMisc), 81
... .dense2m (fastMisc), 81
... .dense2sparse (fastMisc), 81
... .dense2v (fastMisc), 81
... .diag.dsc (fastMisc), 81
... .diag2dense (fastMisc), 81
... .diag2sparse (fastMisc), 81
... .diag2u2n (diag2u2n), 57
... .diagu2n (diag2u2n), 57
... .formatSparseSimple, 139
... .formatSparseSimple (formatSparseM), 86
... .ind2dense (fastMisc), 81
... .ind2sparse (fastMisc), 81
... .m2V (fastMisc), 81
... .m2dense (fastMisc), 81
... .m2sparse (fastMisc), 81
... .selectSuperClasses, 119
... .solve.dgC.chol (fastMisc), 81
... .solve.dgC.lu (fastMisc), 81
... .solve.dgC.qr (fastMisc), 81
... .sparse2dense (fastMisc), 81
... .sparse2g (fastMisc), 81
... .sparse2kind (fastMisc), 81
... .sparse2m (fastMisc), 81
... .sparse2v (fastMisc), 81
... .sparseDiagonal (Diagonal), 53
... .symDiagonal (Diagonal), 53
... .tcr2rc (fastMisc), 81
... .tcrt (fastMisc), 81
... .trDiagonal (Diagonal), 53
... .updateCHMfactor (fastMisc), 81
... .validateCsparse (CsparseMat-class), 44
... .ddiMatrix, Matrix-method
... (.ddiMatrix-class), 46
... .ddiMatrix, denseMatrix-method
... (.ddiMatrix-class), 46
... .ddiMatrix, lenseMatrix-method
... (.ddiMatrix-class), 46
... .ddiMatrix, ndenseMatrix-method
... (.ddiMatrix-class), 46
... .ldiMatrix, Matrix-method
... (.ldiMatrix-class), 104
... .ndiMatrix, denseMatrix-method
... (.ndiMatrix-class), 104
... .ndiMatrix, lenseMatrix-method
... (.ndiMatrix-class), 104
... .ndiMatrix, ndenseMatrix-method
... (.ndiMatrix-class), 104
... [ (Subscript-methods), 177
... [, Matrix, ANY, NULL, ANY-method
... (Subscript-methods), 177
... [, Matrix, NULL, ANY, ANY-method
... (Subscript-methods), 177
... [, Matrix, NULL, NULL, ANY-method
... (Subscript-methods), 177
... [, Matrix, index, index, logical-method
... (Subscript-methods), 177
... [, Matrix, index, index, missing-method
... (Subscript-methods), 177
... [, Matrix, index, missing, logical-method
... (Subscript-methods), 177
... [, Matrix, index, missing, missing-method
... (Subscript-methods), 177
... [, Matrix, Matrix, missing, missing-method
... (Subscript-methods), 177
... [, Matrix, missing, index, logical-method
<-, diagonalMatrix, index, index, sparseMatrix-method
(Subassign-methods), 176
<-, diagonalMatrix, index, index, sparseVector-method
(Subassign-methods), 176
<-, diagonalMatrix, index, missing, replValue-method
(CsparseMatrix, matrix-method)
(Subassign-methods), 176
<-, diagonalMatrix, index, missing, sparseMatrix-method
(Subassign-methods), 176
<-, diagonalMatrix, index, missing, sparseVector-method
(Subassign-methods), 176
<-, diagonalMatrix, matrix, missing, replValue-method
(Subassign-methods), 176
<-, diagonalMatrix, missing, index, replValue-method
(indMatrix-method, ANY-method)
(Subassign-methods), 176
<-, diagonalMatrix, missing, index, sparseMatrix-method
(Subassign-methods), 176
<-, diagonalMatrix, missing, index, sparseVector-method
(Subassign-methods), 176
<-, diagonalMatrix, missing, missing, ANY-method
(Subassign-methods), 176
<-, indMatrix, index, ANY-method
(matmult-methods), 113
<-, indMatrix, index, missing, ANY-method
(matmult-methods), 113
<-, indMatrix, missing, index, ANY-method
(matmult-methods), 113
<-, indMatrix, missing, missing, ANY-method
(matmult-methods), 113
<-, sparseMatrix, ANY, sparseMatrix-method
(matmult-methods), 113
<-, sparseMatrix, ANY, sparseVector-method
(matmult-methods), 113
<-, sparseMatrix, missing, ANY-method
(matmult-methods), 113
<-, sparseVector, index, replValueSp-method
(denseMatrix-method)
(Subassign-methods), 176
<-, sparseVector, index, sparseVector-method
(denseMatrix-method)
(Subassign-methods), 176
%*%, ANY, Matrix-method
(matmult-methods), 113
%*%, ANY, sparseVector-method
(matmult-methods), 113
%*%, CsparseMatrix, CsparseMatrix-method
(matmult-methods), 113
%*%, CsparseMatrix, RsparseMatrix-method
(matmult-methods), 113
%*%, denseMatrix, CsparseMatrix-method
(matmult-methods), 113
%*%, denseMatrix, RsparseMatrix-method
(matmult-methods), 113
%*%, denseMatrix, denseMatrix-method
(matmult-methods), 113
%*%, sparseMatrix, matrix-method
(matmult-methods), 113
%*%, sparseMatrix, vector-method
(matmult-methods), 113
%*%, TsparseMatrix, matrix-method
(matmult-methods), 113
%*%, TsparseMatrix, vector-method
(matmult-methods), 113
%*%, TsparseMatrix, CsparseMatrix-method
(matmult-methods), 113
%*%, TsparseMatrix, RsparseMatrix-method
(matmult-methods), 113
%*%, TsparseMatrix, denseMatrix-method
(matmult-methods), 113
%*%, TsparseMatrix, diagonalMatrix-method
(matmult-methods), 113
%*%, TspMatrix, matrix-method
(matmult-methods), 113
%*%, TspMatrix, vector-method
(matmult-methods), 113
%*%, TspMatrix, TMatrix-method
(matmult-methods), 113
%*%, TspMatrix, denseMatrix-method
(matmult-methods), 113
%*%, TspMatrix, diagonalMatrix-method
(matmult-methods), 113
%*%, denseMatrix, diagonalMatrix-method (matmult-methods), 113
%*%, denseMatrix, matrix-method (matmult-methods), 113
%*%, denseMatrix, vector-method (matmult-methods), 113
%*%, diagonalMatrix, CsparseMatrix-method (matmult-methods), 113
%*%, diagonalMatrix, RsparseMatrix-method (matmult-methods), 113
%*%, diagonalMatrix, TsparseMatrix-method (matmult-methods), 113
%*%, diagonalMatrix, denseMatrix-method (matmult-methods), 113
%*%, diagonalMatrix, diagonalMatrix-method (matmult-methods), 113
%*%, diagonalMatrix, matrix-method (matmult-methods), 113
%*%, diagonalMatrix, vector-method (matmult-methods), 113
%*%, indMatrix, Matrix-method (matmult-methods), 113
%*%, indMatrix, indMatrix-method (matmult-methods), 113
%*%, indMatrix, matrix-method (matmult-methods), 113
%*%, indMatrix, pMatrix-method (matmult-methods), 113
%*%, indMatrix, vector-method (matmult-methods), 113
%*%, matrix, CsparseMatrix-method (matmult-methods), 113
%*%, matrix, RsparseMatrix-method (matmult-methods), 113
%*%, matrix, TsparseMatrix-method (matmult-methods), 113
%*%, matrix, denseMatrix-method (matmult-methods), 113
%*%, matrix, diagonalMatrix-method (matmult-methods), 113
%*%, matrix, indMatrix-method (matmult-methods), 113
%*%, matrix, pMatrix-method (matmult-methods), 113
%*%, matrix, sparseVector-method (matmult-methods), 113
%*%, pMatrix, Matrix-method (matmult-methods), 113
%*%, pMatrix, indMatrix-method (matmult-methods), 113
%*%, pMatrix, matrix-method (matmult-methods), 113
%*%, pMatrix, pMatrix-method (matmult-methods), 113
%*%, pMatrix, vector-method (matmult-methods), 113
%*%, sparseVector, ANY-method (matmult-methods), 113
%*%, sparseVector, Matrix-method (matmult-methods), 113
%*%, sparseVector, sparseVector-method (matmult-methods), 113
%*%, vector, CsparseMatrix-method (matmult-methods), 113
%*%, vector, RsparseMatrix-method (matmult-methods), 113
%*%, vector, TsparseMatrix-method (matmult-methods), 113
%*%, vector, denseMatrix-method (matmult-methods), 113
%*%, vector, diagonalMatrix-method (matmult-methods), 113
%*%, vector, indMatrix-method (matmult-methods), 113
%*%, vector, pMatrix-method (matmult-methods), 113
%*%, vector, sparseVector-method (matmult-methods), 113
%*%-methods (matmult-methods), 113
%/%, ddiMatrix, Matrix-method (ddiMatrix-class), 46
%/%, ddiMatrix, ddenseMatrix-method (ddiMatrix-class), 46
%/%, ddiMatrix, ldenseMatrix-method (ddiMatrix-class), 46
%/%, ddiMatrix, ndenseMatrix-method (ddiMatrix-class), 46
%/%, ldiMatrix, Matrix-method (ldiMatrix-class), 104
%/%, ldiMatrix, ddenseMatrix-method (ldiMatrix-class), 104
%/%, ldiMatrix, ldenseMatrix-method (ldiMatrix-class), 104
%/%, ldiMatrix, ndenseMatrix-method (ldiMatrix-class), 104
%/%, ndiMatrix, Matrix-method (ldiMatrix-class), 104
%/%, ndiMatrix, ddenseMatrix-method (ldiMatrix-class), 104
%/%, ndiMatrix, ldenseMatrix-method (ldiMatrix-class), 104
%/%, ndiMatrix, ndenseMatrix-method (ldiMatrix-class), 104
%/%, ndiMatrix, ndenseMatrix-method (ldiMatrix-class), 104
%\%,ndiMatrix,ldenseMatrix-method (ldiMatrix-class), 104
%\%,ndiMatrix,ndenseMatrix-method (ldiMatrix-class), 104
%\%,ddiMatrix,Matrix-method (ddiMatrix-class), 46
%\%,ddiMatrix,ldenseMatrix-method (ddiMatrix-class), 46
%\%,ddiMatrix,ndenseMatrix-method (ddiMatrix-class), 46
%\%,ldiMatrix,Matrix-method (ldiMatrix-class), 104
%\%,ldiMatrix,ddenseMatrix-method (ldiMatrix-class), 104
%\%,ldiMatrix,ldenseMatrix-method (ldiMatrix-class), 104
%\%,ldiMatrix,ndenseMatrix-method (ldiMatrix-class), 104
%\%,ndiMatrix,Matrix-method (ldiMatrix-class), 104
%\%,ndiMatrix,ddenseMatrix-method (ldiMatrix-class), 104
%\%,ndiMatrix,ldenseMatrix-method (ldiMatrix-class), 104
%\%,ndiMatrix,ndenseMatrix-method (ldiMatrix-class), 104
%\%,ANY,Matrix-method (boolmatmult-methods), 14
%\%,ANY,ANY-method (boolmatmult-methods), 14
%\%,ANY,matrix-method (boolmatmult-methods), 14
%\%,ANY,sparseVector-method (boolmatmult-methods), 14
%\%,ANY,vector-method (boolmatmult-methods), 14
%\%,CsparseMatrix,CsparseMatrix-method (boolmatmult-methods), 14
%\%,CsparseMatrix,RsparseMatrix-method (boolmatmult-methods), 14
%\%,CsparseMatrix,TsparseMatrix-method (boolmatmult-methods), 14
%\%,CsparseMatrix,denseMatrix-method (boolmatmult-methods), 14
%\%,CsparseMatrix,diagonalMatrix-method (boolmatmult-methods), 14
%\%,CsparseMatrix,matrix-method (boolmatmult-methods), 14
%\%,CsparseMatrix,vector-method (boolmatmult-methods), 14
%\%,Matrix,ANY-method (boolmatmult-methods), 14
%\%,Matrix,ndiMatrix-method (boolmatmult-methods), 14
%\%,Matrix,ndiMatrix-method (boolmatmult-methods), 14
%\%,Matrix,pMatrix-method (boolmatmult-methods), 14
%\%,Matrix,spmatrix-method (boolmatmult-methods), 14
%\%,RsparseMatrix,CsparseMatrix-method (boolmatmult-methods), 14
%\%,RsparseMatrix,RsparseMatrix-method (boolmatmult-methods), 14
%\%,RsparseMatrix,TsparseMatrix-method (boolmatmult-methods), 14
%\%,RsparseMatrix,ldenseMatrix-method (boolmatmult-methods), 14
%\%,RsparseMatrix,ldenseMatrix-method (boolmatmult-methods), 14
%\%,RsparseMatrix,matrix-method (boolmatmult-methods), 14
%\%,RsparseMatrix,vector-method (boolmatmult-methods), 14
%\%,TsparseMatrix,CsparseMatrix-method (boolmatmult-methods), 14
%\%,TsparseMatrix,RsparseMatrix-method (boolmatmult-methods), 14
%\%,TsparseMatrix,TsparseMatrix-method (boolmatmult-methods), 14
%\%,TsparseMatrix,ldenseMatrix-method (boolmatmult-methods), 14
%\%,TsparseMatrix,matrix-method (boolmatmult-methods), 14
%\%,TsparseMatrix,vector-method (boolmatmult-methods), 14
%\%,denseMatrix,CsparseMatrix-method (boolmatmult-methods), 14
%\%,denseMatrix,RsparseMatrix-method (boolmatmult-methods), 14
%\%,denseMatrix,TsparseMatrix-method (boolmatmult-methods), 14
%\%,denseMatrix,ldenseMatrix-method (boolmatmult-methods), 14
%\%,denseMatrix,matrix-method (boolmatmult-methods), 14
%\%,denseMatrix,vector-method (boolmatmult-methods), 14
%\%,diagonalMatrix,CsparseMatrix-method (boolmatmult-methods), 14
%&%, boolmatmult-methods, 14
%&%, diagonalMatrix, RsparseMatrix-method (boolmatmult-methods), 14
%&%, diagonalMatrix, TsparseMatrix-method (boolmatmult-methods), 14
%&%, diagonalMatrix, denseMatrix-method (boolmatmult-methods), 14
%&%, diagonalMatrix, diagonalMatrix-method (boolmatmult-methods), 14
%&%, diagonalMatrix, matrix-method (boolmatmult-methods), 14
%&%, indMatrix, Matrix-method (boolmatmult-methods), 14
%&%, indMatrix, indMatrix-method (boolmatmult-methods), 14
%&%, indMatrix, matrix-method (boolmatmult-methods), 14
%&%, indMatrix, pMatrix-method (boolmatmult-methods), 14
%&%, indMatrix, vector-method (boolmatmult-methods), 14
%&%, matrix, ANY-method (boolmatmult-methods), 14
%&%, matrix, CsparseMatrix-method (boolmatmult-methods), 14
%&%, matrix, RsparseMatrix-method (boolmatmult-methods), 14
%&%, matrix, TsparseMatrix-method (boolmatmult-methods), 14
%&%, matrix, denseMatrix-method (boolmatmult-methods), 14
%&%, matrix, diagonalMatrix-method (boolmatmult-methods), 14
%&%, matrix, indMatrix-method (boolmatmult-methods), 14
%&%, matrix, pMatrix-method (boolmatmult-methods), 14
%&%, pMatrix, Matrix-method (boolmatmult-methods), 14
%&%, pMatrix, vector-method (boolmatmult-methods), 14
%&%, sparseVector, ANY-method (boolmatmult-methods), 14
%&%, sparseVector, CsparseMatrix-method (boolmatmult-methods), 14
%&%, sparseVector, RsparseMatrix-method (boolmatmult-methods), 14
%&%, sparseVector, TsparseMatrix-method (boolmatmult-methods), 14
%&%, sparseVector, denseMatrix-method (boolmatmult-methods), 14
%&%, vector, ANY-method (boolmatmult-methods), 14
%&%, vector, CsparseMatrix-method (boolmatmult-methods), 14
%&%, vector, RsparseMatrix-method (boolmatmult-methods), 14
%&%, vector, TsparseMatrix-method (boolmatmult-methods), 14
%&%, vector, sparseVector-method (boolmatmult-methods), 14
%&%, vector, matrix-method (boolmatmult-methods), 14
%&%, vector, pMatrix-method (boolmatmult-methods), 14
%&%, vector, sparseVector-method (boolmatmult-methods), 14
%&%, vector, matrix-method (boolmatmult-methods), 14
%&%-methods (boolmatmult-methods), 14
&%&, Matrix, ddiMatrix-method (Matrix-class), 116
&%&, Matrix, ldiMatrix-method (Matrix-class), 116
&%&, Matrix, ndiMatrix-method (Matrix-class), 116
&%&, ddenseMatrix, ddiMatrix-method (ddenseMatrix-class), 45
&%&, ddenseMatrix, ldiMatrix-method (ddenseMatrix-class), 45
&%&, ddenseMatrix, ndiMatrix-method (ddenseMatrix-class), 45
&%&, ddiMatrix, Matrix-method (ddiMatrix-class), 46
&%&, ddiMatrix, ddenseMatrix-method (ddiMatrix-class), 46
\&, ddiMatrix, ldenseMatrix-method (ldenseMatrix-class), 104
\&, ldenseMatrix, ddiMatrix-method (ldenseMatrix-class), 104
\&, ddiMatrix, ndiMatrix-method (ndenseMatrix-class), 121
\&, ndenseMatrix, ddiMatrix-method (ddenseMatrix-class), 45
\&, ddenseMatrix, ldiiMatrix-method (ldenseMatrix-class), 104
\&, ldiiMatrix, ddenseMatrix-method (ldenseMatrix-class), 104
\&, ldiiMatrix, ddiMatrix-method (ldiiMatrix-class), 104
\&, ddiMatrix, ldiiMatrix-method (ldiiMatrix-class), 104
\&, ddiMatrix, ndiiMatrix-method (ndiiMatrix-class), 121
\&, ndiiMatrix, ddiMatrix-method (ndiiMatrix-class), 121
\&, ndiiMatrix, ndiiMatrix-method (ndiiMatrix-class), 121
\&\%\%, Matrix, ddiMatrix-method (Matrix-class), 116
\&\%\%, Matrix, ldiiMatrix-method (Matrix-class), 116
\&\%\%, Matrix, ndiiMatrix-method (Matrix-class), 116
^, ddenseMatrix, ddiMatrix-method (ddenseMatrix-class), 45
^, ddenseMatrix, ldiiMatrix-method (ldenseMatrix-class), 45
^, ddenseMatrix, ndiiMatrix-method (ddenseMatrix-class), 45
^, ldiiMatrix, ddiMatrix-method (ldiiMatrix-class), 104
^, ldiiMatrix, ldiiMatrix-method (ldiiMatrix-class), 104
^, ldiiMatrix, ndiiMatrix-method (ldiiMatrix-class), 104
^, ndenseMatrix, ddiMatrix-method (ndenseMatrix-class), 121
^, ndenseMatrix, ldiiMatrix-method (ndenseMatrix-class), 121
^, ndenseMatrix, ndiiMatrix-method (ndenseMatrix-class), 121
abbreviate, 138
abIndex, 5, 6, 148, 149
abIndex-class, 4
abIseq, 4, 5, 148
abIseq1 (abIseq), 5
abs, 89
aggregateT (asUniqueT), 7
all, 59
all.equal, 6, 98
all.equal (all.equal-methods), 6
all.equal, abIndex, abIndex-method (all.equal-methods), 6
all.equal, abIndex, numLike-method (all.equal-methods), 6
all.equal, Matrix, Matrix-method (all.equal-methods), 6
all.equal, Matrix, sparseVector-method (all.equal-methods), 6
all.equal, Matrix, vector-method (all.equal-methods), 6
all.equal, numLike, abIndex-method (all.equal-methods), 6
all.equal, sparseVector, Matrix-method (all.equal-methods), 6
all.equal, sparseVector, sparseVector-method (all.equal-methods), 6
all.equal, sparseVector, vector-method (all.equal-methods), 6
all.equal, vector, Matrix-method (all.equal-methods), 6
all.equal, vector, sparseVector-method (all.equal-methods), 6
all.equal-methods, 6
all.equal.numeric, 6
any, 59
anyDuplicated, 7
anyDuplicatedT (asUniqueT), 7
anyNA, 95
anyNA (is.na-methods), 95
anyNA, denseMatrix-method (is.na-methods), 95
anyNA, diagonalMatrix-method (is.na-methods), 95
anyNA, indMatrix-method (is.na-methods), 95
anyNA, sparseMatrix-method (is.na-methods), 95
anyNA, sparseVector-method (is.na-methods), 95
anyNA-methods (is.na-methods), 95
apply, 117
Arith, abIndex, abIndex-method (abIndex-class), 4
Arith, abIndex, numLike-method (abIndex-class), 4
Arith, CsparseMatrix, CsparseMatrix-method (CsparseMatrix-class), 44
Arith, CsparseMatrix, numeric-method (CsparseMatrix-class), 44
Arith, ddenseMatrix, logical-method (ddenseMatrix-class), 45
Arith, ddenseMatrix, numeric-method (ddenseMatrix-class), 45
Arith, ddenseMatrix, sparseVector-method (ddenseMatrix-class), 45
Arith, ddiMatrix, logical-method (ddiMatrix-class), 46
Arith, ddiMatrix, numeric-method (ddiMatrix-class), 46
Arith, dgCMatrix, dgCMatrix-method (dgCMatrix-class), 49
Arith, dgCMatrix, logical-method (dgCMatrix-class), 49
Arith, dgCMatrix, numeric-method (dgCMatrix-class), 49
Arith, dgeMatrix, dgeMatrix-method (dgeMatrix-class), 50
Arith, dgeMatrix, logical-method (dgeMatrix-class), 50
Arith, dgeMatrix, numeric-method (dgeMatrix-class), 50
Arith, dgeMatrix, sparseVector-method (dgeMatrix-class), 50
Arith, dpoMatrix, logical-method (dpoMatrix-class), 62
Arith, dpoMatrix, numeric-method (dpoMatrix-class), 62
Arith, dppMatrix, logical-method (dppMatrix-class), 62
Arith, dppMatrix, numeric-method (dppMatrix-class), 62
Arith, dsCMatrix, dsCMatrix-method (dsCMatrix-class), 65
Arith, dsparseMatrix, logical-method (dsparseMatrix-class), 66
Arith, dsparseMatrix, nsparseMatrix-method (nsparseMatrix-classes), 129
Arith, dsparseMatrix, numeric-method (dsparseMatrix-class), 66
Arith, dsparseVector, dsparseVector-method (sparseVector-class), 172
Arith, dtCMatrix, dtCMatrix-method (dtCMatrix-class), 70
Arith, ldiMatrix, logical-method (ldiMatrix-class), 104
Arith, ldiMatrix, numeric-method (ldiMatrix-class), 104
Arith, lgCMatrix, lgCMatrix-method (lsparseMatrix-classes), 106
Arith, lgeMatrix, lgeMatrix-method (lgeMatrix-class), 105
Arith, lgTMatrix, lgTMatrix-method (lsparseMatrix-classes), 106
Arith, lMatrix, logical-method (lMatrix-class), 59
Arith, lMatrix, numeric-method (lMatrix-class), 59
Arith, logical, ddenseMatrix-method (ddenseMatrix-class), 45
Arith, logical, ddiMatrix-method (ddiMatrix-class), 46
Arith, logical, dgCMatrix-method (dgCMatrix-class), 49
Arith, logical, dgeMatrix-method (dgeMatrix-class), 50
Arith, logical, dpoMatrix-method (dpoMatrix-class), 62
Arith, logical, dppMatrix-method (dppMatrix-class), 62
Arith, logical, dsparseMatrix-method (dsparseMatrix-class), 66
Arith, logical, ldiMatrix-method (ldiMatrix-class), 104
Arith, logical, lMatrix-method (lMatrix-class), 59
Arith, logical, nMatrix-method (nMatrix-class), 126
Arith, lsparseMatrix, Matrix-method (lsparseMatrix-classes), 106
Arith, lsparseMatrix, nsparseMatrix-method (nsparseMatrix-classes), 129
Arith, Matrix, lsparseMatrix-method (Matrix-class), 116
Arith, Matrix, Matrix-method (Matrix-class), 116
Arith, Matrix, nsparseMatrix-method (Matrix-class), 116
Arith, ngeMatrix, ngeMatrix-method
BunchKaufman, 16, 18, 34, 41, 63, 76, 77, 111, 121, 141, 154, 157, 182
BunchKaufman (BunchKaufman-methods), 17
BunchKaufman, dspMatrix-method (BunchKaufman-methods), 17
BunchKaufman, dsysMatrix-method (BunchKaufman-methods), 17
BunchKaufman, matrix-method (BunchKaufman-methods), 17
BunchKaufman-class, 15
BunchKaufman-methods, 17
BunchKaufmanFactorization, 16, 18
BunchKaufmanFactorization-class (MatrixFactorization-class), 120
c, 4
c.abIndex (abIseq), 5
c.Matrix (Matrix-class), 116
c.sparseVector (sparseVector-class), 172
CAex, 19
cbind, 20, 166
cbind2, 20
cbind2 (cbind2-methods), 20
cbind2, Matrix, Matrix-method (cbind2-methods), 20
cbind2, Matrix, matrix-method (cbind2-methods), 20
cbind2, matrix, Matrix-method (cbind2-methods), 20
cbind2, matrix, missing-method (cbind2-methods), 20
cbind2, Matrix, NULL-method (cbind2-methods), 20
cbind2, Matrix, vector-method (cbind2-methods), 20
cbind2, NULL, Matrix-method (cbind2-methods), 20
cbind2, vector, Matrix-method (cbind2-methods), 20
cbind2-methods, 20
character, 4, 11, 54, 55, 59, 119, 126, 158, 162
CMHfactor, 31, 121, 156
CMHfactor-class, 21
CHMsimp1, 76
CHMsimpl-class (CHMfactor-class), 21
CHMsimpl-class (CMHfactor-class), 21
chol, 26, 28, 29, 33, 34, 41, 51, 63, 66
chol (chol-methods), 25
chol, ddiMatrix-method (chol-methods), 25
chol, diagonalMatrix-method (chol-methods), 25
chol, dsCMMatrix-method (chol-methods), 25
chol, dspMatrix-method (chol-methods), 25
chol, dsRMatrix-method (chol-methods), 25
chol, dsTMatrix-method (chol-methods), 25
chol, dsysMatrix-method (chol-methods), 25
chol, generalMatrix-method (chol-methods), 25
chol, symmetricMatrix-method (chol-methods), 25
chol, triangularMatrix-method (chol-methods), 25
chol-methods, 25
chol2inv, 29
chol2inv (chol2inv-methods), 28
chol2inv, ANY-method (chol2inv-methods), 28
chol2inv, ddiMatrix-method (chol2inv-methods), 28
chol2inv, diagonalMatrix-method (chol2inv-methods), 28
chol2inv, dtCMATRIX-method (chol2inv-methods), 28
chol2inv, dtPMatrix-method (chol2inv-methods), 28
chol2inv, dtRMatrix-method (chol2inv-methods), 28
chol2inv, dtTMatrix-method (chol2inv-methods), 28
chol2inv, generalMatrix-method (chol2inv-methods), 28
chol2inv, symmetricMatrix-method (chol2inv-methods), 28
chol2inv, triangularMatrix-method (chol2inv-methods), 28
chol2inv-methods, 28
Cholesky, 18, 21, 23, 24, 26, 30, 31, 34, 41, 63, 66, 76, 77, 83, 111, 121, 141, 154, 157, 166, 182, 184
Cholesky (Cholesky-methods), 32
Cholesky, ddiMatrix-method (Cholesky-methods), 32
Cholesky, diagonalMatrix-method (Cholesky-methods), 32
Cholesky, dsCMMatrix-method (Cholesky-methods), 32
Cholesky, dspMatrix-method (Cholesky-methods), 32
Cholesky, dsRMatrix-method (Cholesky-methods), 32
Cholesky, dsTMatrix-method (Cholesky-methods), 32
Cholesky, dsysMatrix-method (Cholesky-methods), 32
Cholesky, dsTMatrix-method (Cholesky-methods), 32
Cholesky-methods, 32

Cholesky,dsyMatrix-method
Cholesky-methods, 32

Cholesky,generalMatrix-method
(Cholesky-methods), 32

Cholesky,matrix-method
(Cholesky-methods), 32

Cholesky,symmetricMatrix-method
(Cholesky-methods), 32

Cholesky,triangularMatrix-method
(Cholesky-methods), 32

Cholesky-class, 29

Cholesky-methods, 32

CholeskyFactorization, 30, 33, 34

CholeskyFactorization-class
(MatrixFactorization-class), 120

coerce, 180

coerce,abIndex,integer-method
(abIndex-class), 4

coerce,abIndex,numeric-method
(abIndex-class), 4

coerce,abIndex,seqMat-method
(abIndex-class), 4

coerce,abIndex,vector-method
(abIndex-class), 4

coerce,ANY,denseMatrix-method
(denseMatrix-class), 49

coerce,ANY,Matrix-method
(Matrix-class), 116

coerce,ANY,sparseMatrix-method
(sparseMatrix-class), 165

coerce,ANY,sparseVector-method
(sparseVector-class), 172

coerce,atomicVector,dspMatrix-method
(atomicVector-class), 8

coerce,atomicVector,spMatrix-method
(atomicVector-class), 8

coerce,BunchKaufman,dtrMatrix-method
(BunchKaufman-class), 15

coerce,CMsimpl,dtMatrix-method
(CMHfactor-class), 21

coerce,CMsuper,dgCMMatrix-method
(CMHfactor-class), 21

coerce,Cholesky,dtrMatrix-method
(Cholesky-class), 29

coerce,corMatrix,pcorMatrix-method
(dpMatrix-class), 62

coerce,denseLU,dgeMatrix-method
(denseLU-class), 47

coerce,dgCMMatrix,matrix.csc-method
(coerce-methods-SparseM), 38

coerce,dgRMMatrix,matrix.csr-method
(coerce-methods-SparseM), 38

coerce,dgTMatrix,matrix.coo-method
(coerce-methods-SparseM), 38

coerce,diagonalMatrix,symmetricMatrix-method
(diagonalMatrix-class), 55

coerce,diagonalMatrix,triangularMatrix-method
(diagonalMatrix-class), 55

coerce,dpMatrix,corMatrix-method
(dpMatrix-class), 62

coerce,dpMatrix,dppMatrix-method
(dpMatrix-class), 62

coerce,dppMatrix,dpMatrix-method
(dpMatrix-class), 62

coerce,dppMatrix,pcorMatrix-method
(dpMatrix-class), 62

coerce,dsyMatrix,corMatrix-method
(dsyMatrix-class), 68

coerce,dsyMatrix,dpMatrix-method
(dsyMatrix-class), 68

coerce,dsyMatrix,dpMatrix-method
(dpMatrix-class), 68

coerce,dsyMatrix,dpMatrix-method
(sparseMatrix-class), 165

coerce,generalMatrix,packedMatrix-method
(generalMatrix-class), 87

coerce,graph,CsparseMatrix-method
(graph-methods), 37

coerce,graph,Matrix-method
(graph-methods), 37

coerce,graph,RsparseMatrix-method
(graph-methods), 37

coerce,graph,sparseMatrix-method
(graph-methods), 37

coerce,graph,sparseMatrix-method
(sparseMatrix-class), 37

coerce,graph,spMatrix-method
(sparseMatrix-class), 37

coerce,graph,spMatrix-method
(sparseMatrix-class), 37

coerce,graphAM,spMatrix-method
(sparseMatrix-class), 37

coerce,graphAM,sparseMatrix-method
(graph-methods), 37

coerce,indMatrix,pMatrix-method
(indMatrix-class), 91

coerce,indicMatrix,sparseMatrix-method
(indMatrix-class), 91

coerce,logical,abIndex-method
(abIndex-class), 4

coerce,Matrix,corMatrix-method
(Matrix-class), 116
coerce, matrix, corMatrix-method
(dpoMatrix-class), 62
coerce, Matrix, CsparseMatrix-method
(CsparseMatrix-class), 116
coerce, matrix, CsparseMatrix-method
(denseMatrix-class), 44
coerce, Matrix, ddenseMatrix-method
(Matrix-class), 116
coerce, matrix, ddenseMatrix-method
(ddenseMatrix-class), 45
coerce, Matrix, denseMatrix-method
(Matrix-class), 116
coerce, matrix, denseMatrix-method
(denseMatrix-class), 49
coerce, matrix, dgCMatrix-method
(dgCMatrix-class), 49
coerce, Matrix, diagonalMatrix-method
(Matrix-class), 116
coerce, matrix, diagonalMatrix-method
(diagonalMatrix-class), 55
coerce, Matrix, dMatrix-method
(Matrix-class), 116
coerce, matrix, dMatrix-method
(dMatrix-class), 59
coerce, Matrix, dpoMatrix-method
(dpoMatrix-class), 116
coerce, matrix, dpoMatrix-method
(dpoMatrix-class), 62
coerce, Matrix, dppMatrix-method
(dppMatrix-class), 116
coerce, matrix, dppMatrix-method
(dppMatrix-class), 62
coerce, Matrix, dsparseMatrix-method
(dsparseMatrix-class), 66
coerce, Matrix, generalMatrix-method
(Matrix-class), 116
coerce, matrix, generalMatrix-method
(generalMatrix-class), 87
coerce, Matrix, graph-method
(coerce-methods-graph), 37
coerce, Matrix, graphNEL-method
(coerce-methods-graph), 37
coerce, Matrix, indMatrix-method
(Matrix-class), 116
coerce, matrix, indMatrix-method
(indMatrix-class), 91
coerce, Matrix, ldenseMatrix-method
(Matrix-class), 116
coerce, matrix, ldenseMatrix-method
(ldenseMatrix-class), 104
coerce, Matrix, lMatrix-method
(Matrix-class), 116
coerce, matrix, lMatrix-method
(lMatrix-class), 99
coerce, Matrix, lMatrix-method
(dMatrix-class), 59
coerce, Matrix, lMatrix-method
(lMatrix-class), 116
coerce, matrix, lMatrix-method
(lMatrix-class), 106
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, matrix-method
(Matrix-class), 116
coerce, Matrix, RsparseMatrix-method
(RsparseMatrix-class), 151
coerce, matrix, RsparseMatrix-method
(RsparseMatrix-class), 116
coerce, Matrix, sparseMatrix-method
(sparseMatrix-class), 116
coerce, matrix, sparseMatrix-method
(sparseMatrix-class), 116
coerce, Matrix, sparseMatrix-method
(sparseMatrix-class), 116
coerce, Matrix, sparseVector-method
(Matrix-class), 116
INDEX

coerce, matrix, sparseVector-method (sparseVector-class), 172
   (Matrix-class), 116
coerce, Matrix, symmetricMatrix-method (symmetricMatrix-class), 178
coerce, Matrix, triangularMatrix-method (triangularMatrix-class), 180
coerce, Matrix, TsparseMatrix-method (TsparseMatrix-class), 181
coerce, matrix, unpackedMatrix-method (unpackedMatrix-class), 182
coerce, matrix.coo, CsparseMatrix-method (coerce-methods-SparseM), 38
coerce, matrix.coo, dgCMatrix-method (coerce-methods-SparseM), 38
coerce, matrix.coo, dgTMatrix-method (coerce-methods-SparseM), 38
coerce, matrix.csr, CsparseMatrix-method (coerce-methods-SparseM), 38
coerce, matrix.csr, dgCMatrix-method (coerce-methods-SparseM), 38
coerce, matrix.csr, dgRMatrix-method (coerce-methods-SparseM), 38
coerce, matrix.csr, Matrix-method (coerce-methods-SparseM), 38
coerce, matrix.csr, RsparseMatrix-method (coerce-methods-SparseM), 38
coerce, matrix.csr, sparseMatrix-method (coerce-methods-SparseM), 38
coerce, matrix.csr, TsparseMatrix-method (coerce-methods-SparseM), 38
coerce, nsparseMatrix, indMatrix-method (nsparseMatrix-classes), 129
coerce, nsparseMatrix, pMatrix-method (nsparseMatrix-classes), 129
coerce, numeric, abIndex-method (abIndex-class), 4
coerce, numeric, IndMatrix-method (indMatrix-class), 91
coerce, numeric, pMatrix-method (pMatrix-class), 136
coerce, numeric, seqMat-method (seqMat-class), 4
coerce, pBunchKaufman, dtpMatrix-method (BunchKaufman-class), 15
coerce, pCholesky, dtpMatrix-method (Cholesky-class), 29
coerce, pcovMatrix, corMatrix-method (dpoMatrix-class), 62
coerce, seqMat, abIndex-method (abIndex-class), 4
coerce, seqMat, numeric-method (abIndex-class), 4
coerce, sparseVector, CsparseMatrix-method (sparseVector-class), 172
coerce, sparseVector, denseMatrix-method (sparseVector-class), 172
coerce, sparseVector, dsparseVector-method (sparseVector-class), 172
coerce, sparseVector, generalMatrix-method (sparseVector-class), 172
coerce, sparseVector, isparseVector-method (sparseVector-class), 172
coerce, sparseVector, lsparseVector-method (sparseVector-class), 172
coerce, sparseVector, Matrix-method (sparseVector-class), 172
coerce, sparseVector, nsparseVector-method (sparseVector-class), 172
coerce, sparseVector, RsparseMatrix-method (sparseVector-class), 172
coerce, sparseVector, sparseMatrix-method (sparseVector-class), 172
coerce, sparseVector, TsparseMatrix-method (sparseVector-class), 172
coerce, sparseVector, unpackedMatrix-method (sparseVector-class), 172
coerce, sparseVector, zsparseVector-method
  (sparseVector-class), 172
coerce, TsparseMatrix, graphNEL-method
  (coerce-methods-graph), 37
coerce, vector, CsparseMatrix-method
  (CsparseMatrix-class), 44
coerce, vector, ddenseMatrix-method
  (ddenseMatrix-class), 45
coerce, vector, denseMatrix-method
  (denseMatrix-class), 49
coerce, vector, dMatrix-method
  (dMatrix-class), 59
coerce, vector, dsparseMatrix-method
  (dsparseMatrix-class), 66
coerce, vector, dsparseVector-method
  (sparseVector-class), 172
coerce, vector, generalMatrix-method
  (generalMatrix-class), 87
coerce, vector, isparseVector-method
  (sparseVector-class), 172
coerce, vector, ldenseMatrix-method
  (ldenseMatrix-class), 104
coerce, vector, lMatrix-method
  (lMatrix-class), 116
coerce, vector, ndenseMatrix-method
  (ndenseMatrix-class), 121
coerce, vector, nMatrix-method
  (nMatrix-class), 126
coerce, vector, nSparseMatrix-method
  (nSparseMatrix-classes), 129
coerce, vector, RsparseMatrix-method
  (RsparseMatrix-class), 151
coerce, vector, sparseMatrix-method
  (sparseMatrix-class), 165
coerce, vector, sparseVector-method
  (sparseVector-class), 172
coerce, vector, TsparseMatrix-method
  (TsparseMatrix-class), 181
coerce, vector, unpackedMatrix-method
  (unpackedMatrix-class), 182
coerce, vector, zsparseVector-method
  (sparseVector-class), 172
coerce-methods-graph, 37
coerce-methods-SparseM, 38

colMeans, colSums-methods, 39
colMeans, CsparseMatrix-method
  (colSums-methods), 39
colMeans, denseMatrix-method
  (colSums-methods), 39
colMeans, diagonalMatrix-method
  (colSums-methods), 39
colMeans, indMatrix-method
  (colSums-methods), 39
colMeans, RsparseMatrix-method
  (colSums-methods), 39
colMeans, TsparseMatrix-method
  (colSums-methods), 39
colMeans-methods (colSums-methods), 39
colScale (dimScale), 58
colSums, 39, 40, 45, 49, 117, 166
colSums (colSums-methods), 39
colSums, CsparseMatrix-method
  (colSums-methods), 39
colSums, denseMatrix-method
  (colSums-methods), 39
colSums, diagonalMatrix-method
  (colSums-methods), 39
colSums, indMatrix-method
  (colSums-methods), 39
colSums, RsparseMatrix-method
  (colSums-methods), 39
colSums, TsparseMatrix-method
  (colSums-methods), 39
colSums-methods, 39
Compare, CsparseMatrix, CsparseMatrix-method
  (CsparseMatrix-class), 44
Compare, dMatrix, logical-method
  (dMatrix-class), 59
Compare, dMatrix, numeric-method
  (dMatrix-class), 59
Compare, lgeMatrix, lgeMatrix-method
  (lgeMatrix-class), 105
Compare, lMatrix, logical-method
  (lMatrix-class), 59
Compare, lMatrix, numeric-method
  (lMatrix-class), 59
Compare, logical, dMatrix-method
  (dMatrix-class), 59
Compare, logical, lMatrix-method
  (lMatrix-class), 59
Compare, logical, nMatrix-method
  (nMatrix-class), 126
Compare, ngeMatrix, ngeMatrix-method
  (ngeMatrix-class), 125
Compare, nMatrix, logical-method
  (nMatrix-class), 126
<table>
<thead>
<tr>
<th>Function</th>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare,nMatrix,nMatrix-method (nMatrix-class)</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>Compare,nMatrix,numeric-method (nMatrix-class)</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>Compare,numeric,dMatrix-method (dMatrix-class)</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>Compare,numeric,lMatrix-method (dMatrix-class)</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>Compare,numeric,nMatrix-method (nMatrix-class)</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>Compare,triangularMatrix,diagonalMatrix-method (triangularMatrix-class)</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>complex</td>
<td>9, 118</td>
<td></td>
</tr>
<tr>
<td>compMatrix</td>
<td>77, 87, 88, 111, 121</td>
<td></td>
</tr>
<tr>
<td>compMatrix-class</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>condest</td>
<td>42, 42, 147</td>
<td></td>
</tr>
<tr>
<td>contrasts</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>corMatrix</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>corMatrix-class (dpoMatrix-class)</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>cov2cor</td>
<td>58, 59</td>
<td></td>
</tr>
<tr>
<td>cov2cor,packedMatrix-method (packedMatrix-class)</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>cov2cor,sparseMatrix-method (sparseMatrix-class)</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>cov2cor,unpackedMatrix-method (unpackedMatrix-class)</td>
<td>182</td>
<td></td>
</tr>
<tr>
<td>crossprod</td>
<td>44, 51, 63, 93, 113, 114, 166</td>
<td></td>
</tr>
<tr>
<td>crossprod,(ANY,Matrix-method (matmult-methods))</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,ANY,sparseVector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,CsparseMatrix,CsparseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,CsparseMatrix,denseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,CsparseMatrix,diagonalMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,CsparseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,CsparseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,TsparseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,vector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,CsparseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,denseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,diagonalMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,pMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,diagonalMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,TsparseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,vector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,vector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,CsparseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,denseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,diagonalMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,pMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,diagonalMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,TsparseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,vector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,vector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,CsparseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,denseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,diagonalMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,pMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,diagonalMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,vector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,vector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,CsparseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,denseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,diagonalMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,pMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,diagonalMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,vector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,vector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,CsparseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,denseMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,diagonalMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,Matrix,pMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,diagonalMatrix-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,denseMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,vector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,missing-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>crossprod,diagonalMatrix,vector-method (matmult-methods)</td>
<td>113</td>
<td></td>
</tr>
</tbody>
</table>
Crossprod, matrix, RsparseMatrix-method (matmult-methods), 113

Crossprod, matrix, sparseVector-method (matmult-methods), 113

Crossprod, matrix, TsparseMatrix-method (matmult-methods), 113

Crossprod, matrix, T sparseMatrix-method (matmult-methods), 113

Crossprod, pMatrix, missing-method (matmult-methods), 113

Crossprod, pMatrix, pMatrix-method (matmult-methods), 113

Crossprod, RsparseMatrix, C sparseMatrix-method (matmult-methods), 113

Crossprod, RsparseMatrix, diagonalMatrix-method (matmult-methods), 113

Crossprod, RsparseMatrix, matrix-method (matmult-methods), 113

Crossprod, RsparseMatrix, missing-method (matmult-methods), 113

Crossprod, RsparseMatrix, RsparseMatrix-method (matmult-methods), 113

Crossprod, RsparseMatrix, vector-method (matmult-methods), 113

Crossprod, sparseVector, ANY-method (matmult-methods), 113

Crossprod, sparseVector, Matrix-method (matmult-methods), 113

Crossprod, sparseVector, matrix-method (matmult-methods), 113

Crossprod, sparseVector, missing-method (matmult-methods), 113

Crossprod, sparseVector, sparseVector-method (matmult-methods), 113

Crossprod, sparseVector, vector-method (matmult-methods), 113

Crossprod, T sparseMatrix, C sparseMatrix-method (matmult-methods), 113

data.frame, 79, 166

dCHMsimp1, 34, 183, 184
dCHMsimp1-class (CHMfactor-class), 21
dCHMSuper, 34, 183, 184
dCHMSuper-class (CHMfactor-class), 21
dDenseMatrix, 49, 74
dDenseMatrix-class, 45
dDiMatrix, 56, 76, 104, 105
dDiMatrix-class, 46
denseLU, 76, 111, 121, 161
denseLU-class, 47
denseMatrix, 20, 25, 32, 47, 57, 64, 82, 83, 104, 110, 121, 135, 140, 144, 156, 180, 182
determinant, 63, 72, 166
determinant, BunchKaufman, logical-method (BunchKaufman-class), 15
determinant, CHMfactor, logical-method (CHMfactor-class), 21
determinant, Cholesky, logical-method (Cholesky-class), 29

crossprod, TsparseMatrix, TsparseMatrix-method (matmult-methods), 113

crossprod, vector, vector-method (matmult-methods), 113

crossprod, vector, sparseVector-method (matmult-methods), 113

crossprod, vector, T sparseMatrix-method (matmult-methods), 113

crossprod, vector, sparseVector-method (matmult-methods), 113

crossprod, vector, TsparseMatrix-method (matmult-methods), 113

crossprod, vector, indMatrix-method (matmult-methods), 113

crossprod, vector, RsparseMatrix-method (matmult-methods), 113

crossprod, vector, SparseMatrix-method (matmult-methods), 113

crossprod, vector, denseMatrix-method (matmult-methods), 113

crossprod, vector, diagonalMatrix-method (matmult-methods), 113

crossprod, vector, sparseVector-method (matmult-methods), 113

CsparseMatrix, 11, 12, 50, 53, 54, 64, 65, 70, 82, 83, 101, 106, 114, 129, 151, 158, 159, 162, 163, 175, 181, 182
cumsum, 56
INDEX

- determinant, denseLU, logical-method (denseLU-class), 47
- determinant, dgCMatrix, logical-method (dgCMatrix-class), 49
- determinant, dgeMatrix, logical-method (dgeMatrix-class), 50
- determinant, dgRMatrix, logical-method (dgRMatrix-class), 51
- determinant, dgTMatrix, logical-method (dgTMatrix-class), 52
- determinant, diagonalMatrix, logical-method (diagonalMatrix-class), 55
- determinant, dpoMatrix, logical-method (dpoMatrix-class), 62
- determinant, dppMatrix, logical-method (dppMatrix-class), 62
- determinant, dsCMatrix, logical-method (dsCMatrix-class), 65
- determinant, dspMatrix, logical-method (dspMatrix-class), 68
- determinant, dsRMatrix, logical-method (dsRMatrix-class), 67
- determinant, dsTMatrix, logical-method (dsTMatrix-class), 65
- determinant, dsyMatrix, logical-method (dsyMatrix-class), 68
- determinant, indMatrix, logical-method (indMatrix-class), 91
- determinant, Matrix, logical-method (Matrix-class), 116
- determinant, Matrix, missing-method (Matrix-class), 116
- determinant, MatrixFactorization, missing-method (MatrixFactorization-class), 120
- determinant, pBunchKaufman, logical-method (pBunchKaufman-class), 15
- determinant, pCholesky, logical-method (pCholesky-class), 29
- determinant, pMatrix, logical-method (pMatrix-class), 136
- determinant, Schur, logical-method (Schur-class), 152
- determinant, sparseLU, logical-method (sparseLU-class), 160
- determinant, sparseQR, logical-method (sparseQR-class), 167
- determinant, triangularMatrix, logical-method (triangularMatrix-class), 180


- dgCMatric-class, 49
- dgeMatrix, 45, 47, 48, 60, 63, 66, 68–70, 73, 76, 111, 118, 152–154, 168, 169
- dgeMatrix-class, 50
- dgRMatrix, 67, 152
- dgRMatrix-class, 51
- dgTMatrix, 38, 66–68, 70, 73, 90, 106, 107, 181, 182
- dgTMatrix-class, 52
- diag, 11, 55, 92, 121, 122
- diag, CHMfactor-method (CHMfactor-class), 21

- diag, Cholesky-method (Cholesky-class), 29
- diag, CsparseMatrix-method (CsparseMatrix-class), 44
- diag, denseMatrix-method (denseMatrix-class), 49
- diag, diagonalMatrix-method (diagonalMatrix-class), 55
- diag, indMatrix-method (indMatrix-class), 91
- diag, pCholesky-method (Cholesky-class), 29
- diag, RsparseMatrix-method (RsparseMatrix-class), 151
- diag, TsparseMatrix-method (TsparseMatrix-class), 181
- diag<-, CsparseMatrix-method (CsparseMatrix-class), 44
- diag<-, denseMatrix-method (denseMatrix-class), 49
- diag<-, indMatrix-method (indMatrix-class), 91
- diag<-, RsparseMatrix-method (RsparseMatrix-class), 151
- diag<-, TsparseMatrix-method (TsparseMatrix-class), 181
- diagN2U (diagU2N), 57

- Diagonal, 13, 46, 53, 56, 104, 105, 116, 163, 175
- diagonalMatrix-class, 55
- diagU2N, 57
- diff, 117, 149
- diff, denseMatrix-method (denseMatrix-class), 49
- diff, sparseMatrix-method (sparseMatrix-class), 165
diff, sparseVector-method
   (sparseVector-class), 172
dim, 117, 139, 150
dim, Matrix-method (Matrix-class), 116
dim, MatrixFactorization-method
   (MatrixFactorization-class), 120
dim<-, denseMatrix-method
   (denseMatrix-class), 49
dim<-, sparseMatrix-method
   (sparseMatrix-class), 165
dim<-, sparseVector-method
   (sparseVector-class), 172
dimnames, 40, 46, 55, 60, 79, 86, 97, 100, 104, 115, 117, 162, 178
dimnames, Matrix-method (Matrix-class), 116
dimnames, MatrixFactorization-method
   (MatrixFactorization-class), 120
dimnames, symmetricMatrix-method
   (symmetricMatrix-class), 178
dimnames<-, compMatrix, list-method
   (compMatrix-class), 41
dimnames<-, compMatrix, NULL-method
   (compMatrix-class), 41
dimnames<-, Matrix, list-method
   (Matrix-class), 116
dimnames<-, Matrix, NULL-method
   (Matrix-class), 116
dimnames<-, MatrixFactorization, list-method
   (MatrixFactorization-class), 120
dimnames<-, MatrixFactorization, NULL-method
   (MatrixFactorization-class), 120
dimScale, 58
dMatrix, 14, 46, 58, 78
dMatrix-class, 59
dperm, 60
double, 149, 158
dpoMatrix, 30, 31, 34, 69, 88, 122, 123, 182
dpoMatrix-class, 62
dppMatrix, 30, 31, 34, 69, 135
dppMatrix-class (dpoMatrix-class), 62
drop, abIndex-method (abIndex-class), 4
drop, Matrix-method (Matrix-class), 116
drop0, 14, 60, 64, 127, 171
dscMatrix, 23, 24, 34, 50, 76, 83, 184, 185
dscMatrix-class, 65
dsparseMatrix, 46, 53, 54, 65, 67
dsparseMatrix-class, 66
dspMatrix, 16, 18, 30, 135
dspMatrix-class (dsyMatrix-class), 68
dspMatrix-class (dsyMatrix-class), 68
dspMatrix-class (dsyMatrix-class), 68
dsparseVector-class
   (sparseVector-class), 172
dspMatrix, 16, 18, 30, 135
dspMatrix-class (dsyMatrix-class), 68
dspMatrix-class (dsyMatrix-class), 68
dspMatrix-class (dsyMatrix-class), 68
dtCMatrix, 23, 30, 57, 60, 76, 160, 168
dtCMatrix-class, 70
dtpMatrix, 16, 30, 74, 76
dtpMatrix-class, 71
dtrMatrix, 16, 30, 51, 70, 72, 76, 181
dtrMatrix-class, 73
dtrMatrix-class, 74
dtTMatrix-class (dtCMatrix-class), 70
eigen, 19, 117
error, 110
expand (expand-methods), 75
expand, CHMfactor-method
   (expand-methods), 75
expand, denseLU-method (expand-methods), 75
expand, sparseLU-method
   (expand-methods), 75
expand-methods, 75
expand1, 16, 18, 24, 31, 33, 34, 48, 111, 121, 141, 153, 154, 161, 169
expand1 (expand-methods), 75
expand1, BunchKaufman-method
   (expand-methods), 75
expand1, CHMsimpl-method
   (expand-methods), 75
expand1, CHMsuper-method
   (expand-methods), 75
expand1, Cholesky-method
   (expand-methods), 75
expand1, denseLU-method
   (expand-methods), 75
expand1, pBunchKaufman-method
   (expand-methods), 75
expand1, pCholesky-method
   (expand-methods), 75
expand1, Schur-method (expand-methods), 75
expand1, sparseLU-method
   (expand-methods), 75
expand1, sparseQR-method
   (expand-methods), 75
expand1-methods (expand-methods), 75
INDEX

expand2, 16, 18, 24, 31, 34, 48, 80, 111, 121, 141, 153, 154, 161, 169
expand2 (expand-methods), 75
expand2, BunchKaufman-method (expand-methods), 75
expand2, CHMsimpl-method (expand-methods), 75
expand2, CHMsuper-method (expand-methods), 75
expand2, Cholesky-method (expand-methods), 75
expand2, denseLU-method (expand-methods), 75
expand2, pBunchKaufman-method (expand-methods), 75
expand2, pCholesky-method (expand-methods), 75
expand2, Schur-method (expand-methods), 75
expand2, sparseLU-method (expand-methods), 75
expand2, sparseQR-method (expand-methods), 75
expand2-methods (expand-methods), 75
expm, 60
expm (expm-methods), 77
expm, ddiMatrix-method (expm-methods), 77
expm, dgeMatrix-method (expm-methods), 77
expm, dMatrix-method (expm-methods), 77
expm, dspMatrix-method (expm-methods), 77
expm, dtpMatrix-method (expm-methods), 77
expm, dtrMatrix-method (expm-methods), 77
expm, dgeMatrix-method (expm-methods), 77
expm, matrix-method (expm-methods), 77
expm-methods, 77
extends, 115
Extract, 176, 177
fac2sparse (sparse.model.matrix), 157
fac2sparse, 159
facmul (facmul-methods), 80
facmul-methods, 80
factor, 158
FALSE, 97
fastMice, 81
finite, 18, 25, 26, 33, 110, 140, 154, 155, 183
forceSymmetric, 99
forceSymmetric (forceSymmetric-methods), 85
forceSymmetric, CsparseMatrix, character-method (forceSymmetric-methods), 85
forceSymmetric, CsparseMatrix, missing-method (forceSymmetric-methods), 85
forceSymmetric, denseMatrix, character-method (forceSymmetric-methods), 85
forceSymmetric, denseMatrix, missing-method (forceSymmetric-methods), 85
forceSymmetric, diagonalMatrix, character-method (forceSymmetric-methods), 85
forceSymmetric, diagonalMatrix, missing-method (forceSymmetric-methods), 85
forceSymmetric, indMatrix, character-method (forceSymmetric-methods), 85
forceSymmetric, indMatrix, missing-method (forceSymmetric-methods), 85
forceSymmetric, matrix, character-method (forceSymmetric-methods), 85
forceSymmetric, matrix, missing-method (forceSymmetric-methods), 85
forceSymmetric, RsparseMatrix, character-method (forceSymmetric-methods), 85
forceSymmetric, RsparseMatrix, missing-method (forceSymmetric-methods), 85
forceSymmetric, TsparseMatrix, character-method (forceSymmetric-methods), 85
forceSymmetric, TsparseMatrix, missing-method (forceSymmetric-methods), 85
forceSymmetric-methods, 85
format, 86, 87, 138, 165
format, sparseMatrix-method (sparseMatrix-class), 165
formatSparseM, 86, 139
formatSpMatrix, 86, 87, 165
formatSpMatrix (printSpMatrix), 137
function, 100, 150
generalMatrix, 54, 82, 83, 92, 163, 175, 179
generalMatrix-class, 87
get.gpar, 89
classDef, 138
getOption, 138
getValidity, 63, 178
graph, 37
graph2T (coerce-methods-graph), 37
grey, 89
grid.raster, 89
grid.rect, 89
head, 172
head, Matrix-method (Matrix-class), 116
is.null, 97
is.null.DN, 96
isDiagonal, 56
isDiagonal (isTriangular-methods), 99
isDiagonal, CsparseMatrix-method (isTriangular-methods), 99
isDiagonal, diagonalMatrix-method (isTriangular-methods), 99
isDiagonal, indMatrix-method (isTriangular-methods), 99
isDiagonal, matrix-method (isTriangular-methods), 99
isDiagonal, RsparseMatrix-method (isTriangular-methods), 99

is.finite, diagonalMatrix-method (is.na-methods), 95
is.finite, indMatrix-method (is.na-methods), 95
is.finite, sparseMatrix-method (is.na-methods), 95
is.finite-methods (is.na-methods), 95
is.infinite (is.na-methods), 95
is.infinite, abIndex-method (is.na-methods), 95
is.infinite, denseMatrix-method (is.na-methods), 95

is.finite, diagonalMatrix-method (is.na-methods), 95
is.finite, indMatrix-method (is.na-methods), 95
is.finite, sparseMatrix-method (is.na-methods), 95
is.finite-methods (is.na-methods), 95
is.infinite (is.na-methods), 95
is.infinite, abIndex-method (is.na-methods), 95
is.infinite, denseMatrix-method (is.na-methods), 95
isDiagonal,TsparseMatrix-method (isTriangular-methods), 99
isDiagonal-methods (isTriangular-methods), 99
isLDL (CHMfactor-class), 21
isSparseVector-class (sparseVector-class), 172
isPerm (invertPerm), 94
isSymmetric, 83, 99, 100, 122, 134, 178, 180
isSymmetric (isSymmetric-methods), 98
isSymmetric,CsparseMatrix-method (isSymmetric-methods), 98
isSymmetric,denseMatrix-method (isSymmetric-methods), 98
isSymmetric,dgCMatrix-method (isSymmetric-methods), 98
isSymmetric,dgeMatrix-method (isSymmetric-methods), 98
isSymmetric,dgRMatrix-method (isSymmetric-methods), 98
isSymmetric,dgTMatrix-method (isSymmetric-methods), 98
isSymmetric,diagonalMatrix-method (isSymmetric-methods), 98
isSymmetric,dtCMatrix-method (isSymmetric-methods), 98
isSymmetric,dtMatrix-method (isSymmetric-methods), 98
isSymmetric,dtCMatrix-method (isSymmetric-methods), 98
isSymmetric,matrix-method (isSymmetric-methods), 98
isSymmetric,dtTMatrix-method (isSymmetric-methods), 98
isSymmetric,indMatrix-method (isSymmetric-methods), 98
isSymmetric,RsparseMatrix-method (isSymmetric-methods), 98
isSymmetric,spMatrix-method (isSymmetric-methods), 98
isTriangular, 83, 134, 181
isTriangular (isTriangular-methods), 99
isTriangular,CsparseMatrix-method (isTriangular-methods), 99
isTriangular,denseMatrix-method (isTriangular-methods), 99
isTriangular,diagonalMatrix-method (isTriangular-methods), 99
isTriangular,indMatrix-method (isTriangular-methods), 99
isTriangular,matrix-method (isTriangular-methods), 99
isTriangular,RsparseMatrix-method (isTriangular-methods), 99
isTriangular,TsparseMatrix-method (isTriangular-methods), 99
isTriangular-methods (isTriangular-methods), 99
isUniqueT (asUniqueT), 7
kappa, 117, 147
KhatriRao, 100
KNeX, 102
kronecker, 13, 45, 49, 93, 100, 101, 103, 118
kronecker (kronecker-methods), 103
kronecker,CsparseMatrix,CsparseMatrix-method (kronecker-methods), 103
kronecker,CsparseMatrix,diagonalMatrix-method (kronecker-methods), 103
kronecker,CsparseMatrix,Matrix-method (kronecker-methods), 103
kronecker,denseMatrix,Matrix-method (kronecker-methods), 103
kronecker,denseMatrix,denseMatrix-method (kronecker-methods), 103
kronecker,diagonalMatrix,CsparseMatrix-method (kronecker-methods), 103
kronecker,diagonalMatrix,diagonalMatrix-method (kronecker-methods), 103
kronecker,diagonalMatrix,Matrix-method (kronecker-methods), 103
kronecker,diagonalMatrix,indMatrix-method (kronecker-methods), 103
kronecker,diagonalMatrix,matrix-method (kronecker-methods), 103
kronecker,diagonalMatrix,RsparseMatrix-method (kronecker-methods), 103
kronecker,diagonalMatrix,TsparseMatrix-method (kronecker-methods), 103
kronecker,diagonalMatrix,TsparseMatrix-method (kronecker-methods), 103
kronecker,indMatrix,diagonalMatrix-method (kronecker-methods), 103
kronecker,indMatrix,matrix-method (kronecker-methods), 103
kronecker,matrix,matrix-method (kronecker-methods), 103
kronecker,matrix,vector-method (kronecker-methods), 103
kronecker,RsparseMatrix,diagonalMatrix-method (kronecker-methods), 103
kronecker,RsparseMatrix,Matrix-method (kronecker-methods), 103
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>kronecker, RsparseMatrix, RsparseMatrix-method</td>
<td>103</td>
</tr>
<tr>
<td>kronecker, TsparseMatrix, diagonalMatrix-method</td>
<td>103</td>
</tr>
<tr>
<td>kronecker, TsparseMatrix, Matrix-method</td>
<td>103</td>
</tr>
<tr>
<td>kronecker, TsparseMatrix, TsparseMatrix-method</td>
<td>103</td>
</tr>
<tr>
<td>kronecker, vector, Matrix-method</td>
<td>103</td>
</tr>
<tr>
<td>ldenseMatrix</td>
<td>49, 108, 109</td>
</tr>
<tr>
<td>ldenseMatrix-class</td>
<td>104</td>
</tr>
<tr>
<td>ldiMatrix</td>
<td>14, 56</td>
</tr>
<tr>
<td>ldiMatrix-class</td>
<td>104</td>
</tr>
<tr>
<td>length, abIndex-method</td>
<td>4</td>
</tr>
<tr>
<td>length, Matrix-method</td>
<td>116</td>
</tr>
<tr>
<td>length, MatrixFactorization-method</td>
<td>120</td>
</tr>
<tr>
<td>length, sparseVector-method</td>
<td>172</td>
</tr>
<tr>
<td>levelplot</td>
<td>50, 52, 53, 88–90, 117</td>
</tr>
<tr>
<td>lgCMatrix</td>
<td>105</td>
</tr>
<tr>
<td>lgCMatrix-class</td>
<td>106</td>
</tr>
<tr>
<td>lgCMatrix-method</td>
<td>106</td>
</tr>
<tr>
<td>lgMatrix</td>
<td>104, 108, 109</td>
</tr>
<tr>
<td>lgMatrix-class</td>
<td>105</td>
</tr>
<tr>
<td>lgTMatrix</td>
<td>106</td>
</tr>
<tr>
<td>lgTMatrix-class</td>
<td>106</td>
</tr>
<tr>
<td>list</td>
<td>12, 41, 42, 55, 60, 61, 89, 112, 117, 149, 159</td>
</tr>
<tr>
<td>lMatrix</td>
<td>14, 104, 105, 107, 109, 113, 115, 121, 126, 130</td>
</tr>
<tr>
<td>lMatrix-class (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>log, denseMatrix-method</td>
<td>49</td>
</tr>
<tr>
<td>log, diagonalMatrix-method</td>
<td>55</td>
</tr>
<tr>
<td>log, indMatrix-method</td>
<td>91</td>
</tr>
<tr>
<td>log, sparseMatrix-method</td>
<td>165</td>
</tr>
<tr>
<td>log, sparseVector-method</td>
<td>172</td>
</tr>
<tr>
<td>Logic, ANY, Matrix-method</td>
<td>116</td>
</tr>
<tr>
<td>Logic, CsparseMatrix, CsparseMatrix-method (CsparseMatrix-class)</td>
<td>44</td>
</tr>
<tr>
<td>Logic, dMatrix, logical-method</td>
<td>(dMatrix-class), 59</td>
</tr>
<tr>
<td>Logic, dMatrix, numeric-method</td>
<td>(dMatrix-class), 59</td>
</tr>
<tr>
<td>Logic, ldenseMatrix, lsparseMatrix-method (ldenseMatrix-class)</td>
<td>104</td>
</tr>
<tr>
<td>Logic, lgCMatrix, lgCMatrix-method</td>
<td>(lgCMatrix-classes), 106</td>
</tr>
<tr>
<td>Logic, lgeMatrix, lgeMatrix-method</td>
<td>(lgeMatrix-class), 105</td>
</tr>
<tr>
<td>Logic, lgTMatrix, lgTMatrix-method</td>
<td>(lgTMatrix-classes), 106</td>
</tr>
<tr>
<td>Logic, lMatrix, logical-method</td>
<td>(dMatrix-class), 59</td>
</tr>
<tr>
<td>Logic, lMatrix, numeric-method</td>
<td>(dMatrix-class), 59</td>
</tr>
<tr>
<td>Logic, logical, dMatrix-method</td>
<td>(dMatrix-class), 59</td>
</tr>
<tr>
<td>Logic, logical, lMatrix-method</td>
<td>(dMatrix-class), 59</td>
</tr>
<tr>
<td>Logic, logical, nMatrix-method</td>
<td>(nMatrix-class), 126</td>
</tr>
<tr>
<td>Logic, IsCMatrix, IsCMatrix-method</td>
<td>(IsCMatrix-classes), 106</td>
</tr>
<tr>
<td>Logic, IsparseMatrix, ldenseMatrix-method (IsparseMatrix-classes)</td>
<td>106</td>
</tr>
<tr>
<td>Logic, IsparseMatrix, lsparseMatrix-method (IsparseMatrix-classes)</td>
<td>106</td>
</tr>
<tr>
<td>Logic, IsparseMatrix, lsparseVector-method (IsparseVector-class)</td>
<td>172</td>
</tr>
<tr>
<td>Logic, ltCMatrix, ltCMatrix-method</td>
<td>(ltCMatrix-classes), 106</td>
</tr>
<tr>
<td>Logic, Matrix, ANY-method (Matrix-class)</td>
<td>116</td>
</tr>
<tr>
<td>Logic, Matrix, nMatrix-method</td>
<td>(Matrix-class), 116</td>
</tr>
<tr>
<td>Logic, ngeMatrix, ngeMatrix-method</td>
<td>(ngeMatrix-class), 125</td>
</tr>
<tr>
<td>Logic, nMatrix, logical-method</td>
<td>(nMatrix-class), 126</td>
</tr>
<tr>
<td>Logic, nMatrix, Matrix-method</td>
<td>(nMatrix-class), 126</td>
</tr>
<tr>
<td>Logic, nMatrix, nMatrix-method</td>
<td>(nMatrix-class), 126</td>
</tr>
<tr>
<td>Logic, nMatrix, numeric-method</td>
<td>(nMatrix-class), 126</td>
</tr>
<tr>
<td>Logic, nMatrix, sparseVector-method</td>
<td>(nMatrix-class), 126</td>
</tr>
</tbody>
</table>
INDEX

Logic, numeric, dMatrix-method (dMatrix-class), 59
Logic, numeric, lMatrix-method (lMatrix-class), 59
Logic, numeric, nMatrix-method (nMatrix-class), 126
Logic, sparseVector, dMatrix-method (sparseVector-class), 172
Logic, sparseVector, lMatrix-method (sparseVector-class), 172
Logic, sparseVector, nMatrix-method (sparseVector-class), 172
Logic, sparseVector, sparseVector-method (sparseVector-class), 172
Logic, triangularMatrix, diagonalMatrix-method (triangularMatrix-class), 180
logical, 54, 60, 63, 92, 97–100, 105, 112, 113, 115, 119, 127, 144, 173, 176
lsCMatrix, 178
lsCMatrix-class
(lsparseMatrix-classes), 106
lsparseMatrix, 54
lsparseMatrix-class
(lsparseMatrix-classes), 106
lsparseMatrix-classes, 106
lsparseVector-class (sparseVector-class), 172
lsyMatrix-class (lsyMatrix-class), 108
lsyMatrix-class
(lsparseMatrix-classes), 106
lsyMatrix-class, 106
lsyMatrix-class, 108
lsyMatrix-class
(lsparseMatrix-classes), 106
lsyMatrix-class, 105
ltrMatrix-class (ltrMatrix-class), 109
ltrMatrix-class
(lsparseMatrix-classes), 106
ltrMatrix-class, 109
ltrMatrix-class
(lsparseMatrix-classes), 106
ltrMatrix-class, 109
ltrMatrix-class
(lsparseMatrix-classes), 106
ltrMatrix-class, 109
ltrMatrix-class
(lsparseMatrix-classes), 106
ltrMatrix-class, 109
LU, 47, 111, 160
lu, 18, 34, 41, 42, 47, 48, 50, 77, 121, 141, 154, 157, 160, 161, 166
lu (lu-methods), 110
lu,denseMatrix-method (lu-methods), 110
lu,dgCMatrix-method (lu-methods), 110
lu,dgRMatrix-method (lu-methods), 110
lu,dgTMatrix-method (lu-methods), 110
lu,diagonalMatrix-method (lu-methods), 110
lu,dsCMatrix-method (lu-methods), 110
lu,dsCSPMatrix-method (lu-methods), 110
lu,dsRMMatrix-method (lu-methods), 110
lu,dsTMatrix-method (lu-methods), 110
lu,dsyMatrix-method (lu-methods), 110
lu,dtCMatrix-method (lu-methods), 110
lu,dtMatrix-method (lu-methods), 110
lu,dtRMMatrix-method (lu-methods), 110
lu,dtrMatrix-method (lu-methods), 110
lu,dtrTMatrix-method (lu-methods), 110
lu,matrix-method (lu-methods), 110
lu,sparseMatrix-method (lu-methods), 110
LU-class (MatrixFactorization-class), 120
lu-methods, 110
mat2triplet, 112
Math, denseMatrix-method (denseMatrix-class), 49
Math, diagonalMatrix-method (diagonalMatrix-class), 55
Math, indMatrix-method (indMatrix-class), 91
Math, sparseMatrix-method (sparseMatrix-class), 165
Math, sparseVector-method (sparseVector-class), 172
Math2, Matrix-method (Matrix-class), 116
Math2, sparseVector-method (sparseVector-class), 172
matmult-methods, 113
matrix, 10, 14, 56, 85, 86, 92, 97–99, 112, 115–117, 122, 127, 176
Matrix-class, 116
Matrix-notyet, 118
matrix.csr, 39
Matrix.Version (Matrix-class), 116
MatrixClass, 119
MatrixFactorization, 15, 16, 21, 22, 30, 41, 47, 75, 77, 80, 152, 155, 157, 160, 168
MatrixFactorization-class, 120
max, 59
mean, denseMatrix-method (denseMatrix-class), 49
mean, sparseMatrix-method (sparseMatrix-class), 165
mean, sparseVector-method (sparseVector-class), 172
method, 98
min, 59
model.frame, 158
model.matrix, 158, 159
NA, 56, 60, 96, 99, 100, 103, 106, 126, 127, 166, 173, 176, 179
NA_integer_, 144
names, 40, 54
NaN, 96, 144
nCHMsimpl-class (CHMfactor-class), 21
nCHMsuper-class (CHMfactor-class), 21
ncol, 180
ndenseMatrix, 49
ndenseMatrix-class, 121
diMatrix-class (ldiMatrix-class), 104
nearcor, 123
nearPD, 122
new, 23, 45, 162
ngCMatrix, 125
ngCMatrix-class (nsparseMatrix-classes), 129
ngeMatrix, 121, 131, 132
ngeMatrix-class, 125
ngRMatrix-class (nsparseMatrix-classes), 129
ngTMatrix, 38
ngTMatrix-class (nsparseMatrix-classes), 129
mMatrix, 11, 14, 60, 82, 93, 96, 112, 113, 125
mMatrix-class, 126
nnzero, 64, 73, 150
nnzero, ANY-method (nnzero-methods), 127
nnzero, CHMfactor-method (nnzero-methods), 127
nnzero, denseMatrix-method (nnzero-methods), 127
nnzero, diagonalMatrix-method (nnzero-methods), 127
nnzero, indMatrix-method (nnzero-methods), 127
nnzero, sparseMatrix-method (nnzero-methods), 127
nnzero, vector-method (nnzero-methods), 127
nnzero-methods, 127
norm, 42, 43, 69, 123, 129, 146, 147, 166
norm (norm-methods), 128
norm, ANY, missing-method (norm-methods), 128
norm, denseMatrix, character-method (norm-methods), 128
norm, diagonalMatrix, character-method (norm-methods), 128
norm, indMatrix, character-method (norm-methods), 128
norm, pMatrix, character-method (norm-methods), 128
norm, sparseMatrix, character-method (norm-methods), 128
norm-methods, 128
nrow, 180
nsCMatrix-class (nsparseMatrix-classes), 129
nsCMatrix-class (nsrMatrix-classes), 129
nsCMatrix-class (nsyMatrix-classes), 129
nsRMatrix-class (nsparseMatrix-classes), 129
nsSMatrix-class (nsysMatrix-classes), 131
nsTMatrix-class (nsrMatrix-classes), 129
nsTMatrix-class (nsyMatrix-classes), 129
nsyMatrix, 125
nsyMatrix-class, 131
nsrMatrix-class (nsrMatrix-classes), 129
nsrMatrix-class (nsyMatrix-classes), 129
ntMatrix-class (nsrMatrix-classes), 132
ntMatrix-class (nsyMatrix-classes), 132
ntRMatrix-class (nsrMatrix-classes), 129
ntRMatrix-class (nsyMatrix-classes), 129
NULL, 79, 86, 97, 115, 138, 150, 162
numeric, 4, 5, 9, 39, 40, 52, 62, 66, 102, 127
onenormest, 129
onenormest (condest), 42
Ops, 5, 56, 74, 173
Ops, abIndex, abIndex-method (abIndex-class), 4
<table>
<thead>
<tr>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ops, abIndex, ANY-method (abIndex-class)</td>
<td>4</td>
</tr>
<tr>
<td>Ops, ANY, abIndex-method (abIndex-class)</td>
<td>4</td>
</tr>
<tr>
<td>Ops, ANY, ddiMatrix-method (ddiMatrix-class)</td>
<td>46</td>
</tr>
<tr>
<td>Ops, ANY, ldiMatrix-method (ldiMatrix-class)</td>
<td>104</td>
</tr>
<tr>
<td>Ops, ANY, Matrix-method (Matrix-class)</td>
<td>116</td>
</tr>
<tr>
<td>Ops, ANY, sparseVector-method (sparseVector-class)</td>
<td>172</td>
</tr>
<tr>
<td>Ops, atomicVector, sparseVector-method (atomicVector-class)</td>
<td>8</td>
</tr>
<tr>
<td>Ops, ddiMatrix, ANY-method (ddiMatrix-class)</td>
<td>46</td>
</tr>
<tr>
<td>Ops, ddiMatrix, ddiMatrix-method (ddiMatrix-class)</td>
<td>46</td>
</tr>
<tr>
<td>Ops, ddiMatrix, dMatrix-method (ddiMatrix-class)</td>
<td>46</td>
</tr>
<tr>
<td>Ops, ddiMatrix, ldiMatrix-method (ddiMatrix-class)</td>
<td>46</td>
</tr>
<tr>
<td>Ops, ddiMatrix, logical-method (ddiMatrix-class)</td>
<td>46</td>
</tr>
<tr>
<td>Ops, ddiMatrix, Matrix-method (ddiMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, ddiMatrix, nMatrix-method (ddiMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, dMatrix, ddiMatrix-method (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, dMatrix, dMatrix-method (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, dMatrix, lMatrix-method (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, dMatrix, logical-method (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, dMatrix, numeric-method (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, dMatrix, matrix-method (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, dMatrix, NULL-method (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, dMatrix, sparseMatrix-method (dMatrix-class)</td>
<td>62</td>
</tr>
<tr>
<td>Ops, dpoMatrix, logical-method (dpoMatrix-class)</td>
<td>62</td>
</tr>
<tr>
<td>Ops, dpoMatrix, numeric-method (dpoMatrix-class)</td>
<td>62</td>
</tr>
<tr>
<td>Ops, dppMatrix, logical-method (dppMatrix-class)</td>
<td>62</td>
</tr>
<tr>
<td>Ops, dppMatrix, numeric-method (dppMatrix-class)</td>
<td>62</td>
</tr>
<tr>
<td>Ops, dsparseMatrix, nsparseMatrix-method (dsparseMatrix-class)</td>
<td>66</td>
</tr>
<tr>
<td>Ops, ldenseMatrix, ldenseMatrix-method (ldenseMatrix-class)</td>
<td>104</td>
</tr>
<tr>
<td>Ops, lidiMatrix, ANY-method (lidiMatrix-class)</td>
<td>104</td>
</tr>
<tr>
<td>Ops, lidiMatrix, ddiMatrix-method (lidiMatrix-class)</td>
<td>104</td>
</tr>
<tr>
<td>Ops, lidiMatrix, dMatrix-method (lidiMatrix-class)</td>
<td>104</td>
</tr>
<tr>
<td>Ops, lidiMatrix, ldiMatrix-method (lidiMatrix-class)</td>
<td>104</td>
</tr>
<tr>
<td>Ops, lidiMatrix, logical-method (lidiMatrix-class)</td>
<td>104</td>
</tr>
<tr>
<td>Ops, lidiMatrix, Matrix-method (lidiMatrix-class)</td>
<td>104</td>
</tr>
<tr>
<td>Ops, lidiMatrix, numeric-method (lidiMatrix-class)</td>
<td>104</td>
</tr>
<tr>
<td>Ops, lidiMatrix, sparseMatrix-method (lidiMatrix-class)</td>
<td>104</td>
</tr>
<tr>
<td>Ops, lMatrix, dMatrix-method (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, lMatrix, lMatrix-method (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, lMatrix, nMatrix-method (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, lMatrix, numeric-method (dMatrix-class)</td>
<td>59</td>
</tr>
<tr>
<td>Ops, logical, dpoMatrix-method (dpoMatrix-class)</td>
<td>62</td>
</tr>
<tr>
<td>Ops, logical, dppMatrix-method (dpoMatrix-class)</td>
<td>62</td>
</tr>
<tr>
<td>Ops, lsparseMatrix, lsparseMatrix-method (lsparseMatrix-classes)</td>
<td>106</td>
</tr>
<tr>
<td>Ops, lsparseMatrix, nsparseMatrix-method (lsparseMatrix-classes)</td>
<td>106</td>
</tr>
<tr>
<td>Ops, Matrix, ANY-method (Matrix-class)</td>
<td>116</td>
</tr>
<tr>
<td>Ops, Matrix, ddiMatrix-method (Matrix-class)</td>
<td>116</td>
</tr>
<tr>
<td>Ops, Matrix, ldiMatrix-method (Matrix-class)</td>
<td>116</td>
</tr>
<tr>
<td>Ops, Matrix, matrix-method (Matrix-class)</td>
<td>116</td>
</tr>
<tr>
<td>Ops, matrix, Matrix-method (Matrix-class)</td>
<td>116</td>
</tr>
<tr>
<td>Ops, Matrix, NULL-method (Matrix-class)</td>
<td>116</td>
</tr>
<tr>
<td>Ops, Matrix, sparseVector-method (Matrix-class)</td>
<td>116</td>
</tr>
</tbody>
</table>
Ops, ndenseMatrix, ndenseMatrix-method
(ndeMatrix-class), 121
Ops, ndiMatrix, ddiMatrix-method
(ldiMatrix-class), 104
Ops, ndiMatrix, idiMatrix-method
(ldiMatrix-class), 104
Ops, ndiMatrix, ndiMatrix-method
(ldiMatrix-class), 104
Ops, nMatrix, dMatrix-method
(nMatrix-class), 126
Ops, nMatrix, lMatrix-method
(nMatrix-class), 126
Ops, nMatrix, nMatrix-method
(nMatrix-class), 126
Ops, nMatrix, numeric-method
(nMatrix-class), 126
Ops, nsparseMatrix, dsparseMatrix-method
(nsparseMatrix-classes), 129
Ops, nsparseMatrix, lsparseMatrix-method
(nsparseMatrix-classes), 129
Ops, nsparseMatrix, sparseMatrix-method
(nsparseMatrix-classes), 129
Ops, NULL, Matrix-method (Matrix-class), 116
Ops, numeric, dpoMatrix-method
(dpoMatrix-class), 62
Ops, numeric, dppMatrix-method
(dpoMatrix-class), 62
Ops, numeric, lMatrix-method
(lMatrix-class), 59
Ops, numeric, nMatrix-method
(nMatrix-class), 126
Ops, numeric, sparseMatrix-method
(sparseMatrix-class), 165
Ops, sparseMatrix, ddiMatrix-method
(sparseMatrix-class), 165
Ops, sparseMatrix, idiMatrix-method
(sparseMatrix-class), 165
Ops, sparseMatrix, nMatrix-method
(sparseMatrix-class), 165
Ops, sparseMatrix, numeric-method
(sparseMatrix-class), 165
Ops, sparseMatrix, sparseMatrix-method
(sparseMatrix-class), 165
Ops, sparseVector, ANY-method
(sparseVector-class), 172
Ops, sparseVector, atomicVector-method
(sparseVector-class), 172
Ops, sparseVector, Matrix-method
(sparseVector-class), 172
Ops, sparseVector, sparseVector-method
(sparseVector-class), 172
options, 23, 138, 139, 172
order, 94, 112
outer, 117
pack, 68, 74, 133, 135, 183
pack, dgeMatrix-method (pack), 133
pack, lgeMatrix-method (pack), 133
pack, matrix-method (pack), 133
pack, ngeMatrix-method (pack), 133
pack, packedMatrix-method (pack), 133
pack, sparseMatrix-method (pack), 133
pack, unpackedMatrix-method (pack), 133
pack-methods (pack), 133
packedMatrix, 18, 34, 49, 82, 83, 133, 183
packedMatrix-class, 135
panel.levelplot.raster, 89
paste, 158
pBunchKaufman, 18, 76, 121, 135
pBunchKaufman-class
(BunchKaufman-class), 15
pCholesky, 34, 76, 121, 135
pCholesky-class (Cholesky-class), 29
pcorMatrix-class (dpoMatrix-class), 62
plot.default, 89
pMatrix, 61, 76, 91, 93, 95
pMatrix-class, 136
posdefify, 122, 123
print, 56, 87, 88, 117, 138, 139, 165, 166
print, diagonalMatrix-method
(diagonalMatrix-class), 55
print, sparseMatrix-method
(sparseMatrix-class), 165
print.default, 86, 138
print.sparseMatrix
(sparseMatrix-class), 165
print.trellis, 90
printSpMatrix, 87, 117, 137, 165
printSpMatrix2 (printSpMatrix), 137
prod, 59
QR, 141, 168
qr, 18, 34, 41, 77, 111, 121, 141, 143–145,
154, 157, 167–169
qr (qr-methods), 140
qr, dgCMatrix-method (qr-methods), 140
qr, sparseMatrix-method (qr-methods), 140
QR-class (MatrixFactorization-class),
120
qr-methods, 140
qr.coef, 156, 157, 169
qr.coef, sparseQR, dgeMatrix-method
(sparseQR-class), 167
qr.coef, sparseQR, Matrix-method (sparseQR-class), 167
qr.coef, sparseQR, matrix-method (sparseQR-class), 167
qr.coef, sparseQR, vector-method (sparseQR-class), 167
qr.default, 140
qr.fitted, 169
qr.fitted, sparseQR, dgeMatrix-method (sparseQR-class), 167
qr.fitted, sparseQR, Matrix-method (sparseQR-class), 167
qr.fitted, sparseQR, matrix-method (sparseQR-class), 167
qr.fitted, sparseQR, vector-method (sparseQR-class), 167
qr.Q, 169
qr.Q, sparseQR-method (sparseQR-class), 167
qr.qty, 169
qr.qty, sparseQR, dgeMatrix-method (sparseQR-class), 167
qr.qty, sparseQR, Matrix-method (sparseQR-class), 167
qr.qty, sparseQR, matrix-method (sparseQR-class), 167
qr.qty, sparseQR, vector-method (sparseQR-class), 167
qr.qy, 169
qr.qy, sparseQR, dgeMatrix-method (sparseQR-class), 167
qr.qy, sparseQR, Matrix-method (sparseQR-class), 167
qr.qy, sparseQR, matrix-method (sparseQR-class), 167
qr.qy, sparseQR, vector-method (sparseQR-class), 167
qr.R, 169
qr.R, sparseQR-method (sparseQR-class), 167
qr.resid, 169
qr.resid, sparseQR, dgeMatrix-method (sparseQR-class), 167
qr.resid, sparseQR, Matrix-method (sparseQR-class), 167
qr.resid, sparseQR, matrix-method (sparseQR-class), 167
qr.resid, sparseQR, vector-method (sparseQR-class), 167
qr.solve, 169
qr.X, 169
qr.X, sparseQR-method (sparseQR-class), 167
qr2rankMatrix (rankMatrix), 143
qrR (sparseQR-class), 167
range, 59
rankMatrix, 143
rbind, 20, 166
rbind2, 20
rbind2 (cbind2-methods), 20
rbind2, Matrix, matrix-method (cbind2-methods), 20
rbind2, Matrix, method (cbind2-methods), 20
rbind2, vector, Matrix-method (cbind2-methods), 20
rbind2-methods (cbind2-methods), 20
rcond, 43, 51, 63, 69, 156
rcond (rcond-methods), 146
rcond, ANY, missing-method (rcond-methods), 146
rcond, denseMatrix, character-method (rcond-methods), 146
rcond, diagonalMatrix, character-method (rcond-methods), 146
rcond, indMatrix, character-method (rcond-methods), 146
rcond, pMatrix, character-method (rcond-methods), 146
rcond, sparseMatrix, character-method (rcond-methods), 146
rcond-methods, 146
readHB (externalFormats), 79
readMM (externalFormats), 79
rep, denseMatrix-method (denseMatrix-class), 49
rep, sparseMatrix-method (sparseMatrix-class), 165
rep, sparseVector-method (sparseVector-class), 172
rep.int, 148
rep2abI, 6, 148
replValue-class, 148
rle, 4–6, 149
rleDiff, 4
rleDiff-class, 149
round, 59
rowMeans, 117
rowMeans (colSums-methods), 39
rowMeans, CsparseMatrix-method (colSums-methods), 39
rowMeans, denseMatrix-method (colSums-methods), 39
rowMeans, diagonalMatrix-method (colSums-methods), 39
rowMeans, indMatrix-method (colSums-methods), 39
rowMeans, RsparseMatrix-method (colSums-methods), 39
rowMeans, TsparseMatrix-method (colSums-methods), 39
rowMeans-methods (colSums-methods), 39
rowScale (dimScale), 58
rowSums (colSums-methods), 39
rowSums, CsparseMatrix-method (colSums-methods), 39
rowSums, denseMatrix-method (colSums-methods), 39
rowSums, diagonalMatrix-method (colSums-methods), 39
rowSums, indMatrix-method (colSums-methods), 39
rowSums, RsparseMatrix-method (colSums-methods), 39
rowSums, TsparseMatrix-method (colSums-methods), 39
rowSums-methods (colSums-methods), 39
RsparseMatrix, 11, 52, 64, 67, 83, 106, 129, 158, 159, 162, 163
RsparseMatrix-class, 150, 163
sample.int, 150
Schur, 18, 34, 41, 61, 76–78, 111, 121, 141, 152–154, 157, 180
Schur (Schur-methods), 153
Schur, dgeMatrix-method (Schur-methods), 153
Schur, dsyMatrix-method (Schur-methods), 153
Schur, generalMatrix-method (Schur-methods), 153
Schur, symmetricMatrix-method (Schur-methods), 153

 Schur, triangularMatrix-method (Schur-methods), 153
 Schur-class, 152
 Schur-methods, 153
 SchurFactorization, 152, 154
 SchurFactorization-class (MatrixFactorization-class), 120

 seq, 5
 seqMat-class (abIndex-class), 4
 set.seed, 42
 setClassUnion, 91, 133, 148
 show, 5, 87, 117, 138, 139, 149, 165, 172
 show, abIndex-method (abIndex-class), 4
 show, BunchKaufmanFactorization-method (MatrixFactorization-class), 120
 show, CholeskyFactorization-method (MatrixFactorization-class), 120
 show, denseMatrix-method (denseMatrix-class), 49
 show, diagonalMatrix-method (diagonalMatrix-class), 55
 show, LU-method (MatrixFactorization-class), 120
 show, MatrixFactorization-method (MatrixFactorization-class), 120
 show, QR-method (MatrixFactorization-class), 120
 show, rleDiff-method (rleDiff-class), 149
 show, SchurFactorization-method (MatrixFactorization-class), 120
 show, sparseMatrix-method (sparseMatrix-class), 165
 show, sparseVector-method (sparseVector-class), 172
 showClass, 46, 69, 108, 109, 131, 132
 showMethods, 45, 49, 105, 108, 109, 125, 131, 132, 146, 179
 signif, 59
 signPerm (invertPerm), 94
 skewpart, 99
 skewpart (symmpart-methods), 179
 skewpart, sparseMatrix-method (symmpart-methods), 179
 skewpart, denseMatrix-method (symmpart-methods), 179
skewpart, diagonalMatrix-method (symmpart-methods), 179
skewpart, indMatrix-method (symmpart-methods), 179
skewpart, matrix-method (symmpart-methods), 179
skewpart, RsparseMatrix-method (symmpart-methods), 179
skewpart, TsparseMatrix-method (symmpart-methods), 179
skewpart-methods (symmpart-methods), 179
solve, ANY, ANY-method (solve-methods), 155
solve, BunchKaufman, dgeMatrix-method (solve-methods), 155
solve, BunchKaufman, missing-method (solve-methods), 155
solve, CHMfactor, dgCMatrix-method (solve-methods), 155
solve, CHMfactor, dgeMatrix-method (solve-methods), 155
solve, CHMfactor, missing-method (solve-methods), 155
solve, Cholesky, dgeMatrix-method (solve-methods), 155
solve, Cholesky, missing-method (solve-methods), 155
solve, ddiMatrix, ANY-method (solve-methods), 155
solve, ddiMatrix, matrix-method (solve-methods), 155
solve, ddiMatrix, missing-method (solve-methods), 155
solve, denseLU, dgeMatrix-method (solve-methods), 155
solve, denseLU, missing-method (solve-methods), 155
solve, dgCMatrix, ANY-method (solve-methods), 155
solve, dgCMatrix, matrix-method (solve-methods), 155
solve, dgCMatrix, missing-method (solve-methods), 155
solve, dgeMatrix, ANY-method (solve-methods), 155
solve, diagonalMatrix, ANY-method (solve-methods), 155
solve, dpoMatrix, ANY-method (solve-methods), 155
solve, dppMatrix, ANY-method (solve-methods), 155
solve, dsCMatrix, denseMatrix-method (solve-methods), 155
solve, dsCMatrix, matrix-method (solve-methods), 155
solve, dsCMatrix, missing-method (solve-methods), 155
solve, dpdMatrix, ANY-method (solve-methods), 155
solve, dsyMatrix, ANY-method (solve-methods), 155
solve, dtCMatrix, dgCMatrix-method (solve-methods), 155
solve, dtCMatrix, ANY-method (solve-methods), 155
solve, dtCMatrix, missing-method (solve-methods), 155
solve, dtCMatrix, triangularMatrix-method (solve-methods), 155
solve, dtpMatrix, ANY-method (solve-methods), 155
solve, dtpMatrix, missing-method (solve-methods), 155
solve, dtpMatrix, triangularMatrix-method (solve-methods), 155
solve, dtrMatrix, ANY-method (solve-methods), 155
solve, dtrMatrix, missing-method (solve-methods), 155
solve, dtrMatrix, triangularMatrix-method (solve-methods), 155
solve, indMatrix, ANY-method (solve-methods), 155
solve, matrix, ANY-method (solve-methods), 155
solve, Matrix, sparseVector-method (solve-methods), 155
solve, matrix, sparseVector-method (solve-methods), 155
solve, MatrixFactorization, CsparseMatrix-method (solve-methods), 155
solve, MatrixFactorization, denseMatrix-method (solve-methods), 155
solve, MatrixFactorization, dgCMatrix-method (solve-methods), 155
solve, MatrixFactorization, dgeMatrix-method (solve-methods), 155
solve, MatrixFactorization, diagonalMatrix-method (solve-methods), 155
solve, MatrixFactorization, indMatrix-method (solve-methods), 155
solve, MatrixFactorization, matrix-method (solve-methods), 155
solve, MatrixFactorization, RsparseMatrix-method (solve-methods), 155
solve, MatrixFactorization, sparseVector-method (solve-methods), 155
solve, MatrixFactorization, TsparseMatrix-method (solve-methods), 155
solve, MatrixFactorization, vector-method (solve-methods), 155
solve, pBunchKaufman, dgeMatrix-method (solve-methods), 155
solve, pCholesky, dgeMatrix-method (solve-methods), 155
solve, pMatrix, matrix-method (solve-methods), 155
solve, pMatrix, missing-method (solve-methods), 155
solve, pMatrix, vector-method (solve-methods), 155
solve, RsparseMatrix, ANY-method (solve-methods), 155
solve, Schur, ANY-method (solve-methods), 155
solve, sparseLU, dgCMatrix-method (solve-methods), 155
solve, sparseLU, dgeMatrix-method (solve-methods), 155
solve, sparseLU, missing-method (solve-methods), 155
solve, sparseQR, dgCMatrix-method (solve-methods), 155
solve, sparseQR, dgeMatrix-method (solve-methods), 155
solve, triangularMatrix, CsparseMatrix-method (solve-methods), 155
solve, triangularMatrix, denseMatrix-method (solve-methods), 155
solve, triangularMatrix, dgCMatrix-method (solve-methods), 155
solve, triangularMatrix, dgeMatrix-method (solve-methods), 155
solve, triangularMatrix, diagonalMatrix-method (solve-methods), 155
solve, triangularMatrix, indMatrix-method (solve-methods), 155
solve, triangularMatrix, matrix-method (solve-methods), 155
solve, triangularMatrix, RsparseMatrix-method (solve-methods), 155
solve, triangularMatrix, sparseVector-method (solve-methods), 155
solve, triangularMatrix, TsparseMatrix-method (solve-methods), 155
solve, triangularMatrix, vector-method (solve-methods), 155
solve, TsparseMatrix, ANY-method (solve-methods), 155
sort, sparseVector-method (sparseVector-class), 172
sort.list, 94
sparse.model.matrix, 157, 163, 166
sparseLU, 48, 76, 111, 121, 156
sparseLU-class, 160
SparseM.ontology, 39
sparseMatrix-class, 165
sparseQR, 76, 121, 141, 144, 156, 167
sparseQR-class, 167
sparseVector, 6, 14, 39, 40, 83, 95, 113, 156, 170, 171, 173
sparseVector-class, 172
spMatrix, 52, 53, 112, 139, 174, 181
stop, 176
str, 120
Subassign-methods, 176
Subscript-methods, 177
substring, 138
INDEX

sum, 59
Summary, 173
summary, 112
Summary, abIndex-method (abIndex-class), 4
Summary, denseMatrix-method (denseMatrix-class), 49
Summary, diagonalMatrix-method (diagonalMatrix-class), 55
summary, diagonalMatrix-method (diagonalMatrix-class), 55
Summary, indMatrix-method (indMatrix-class), 91
Summary, sparseMatrix-method (sparseMatrix-class), 165
summary, sparseMatrix-method (sparseMatrix-class), 165
Summary, sparseVector-method (sparseVector-class), 172
svd, 117, 128, 144, 145
symmetricMatrix-class, 178
symmpart, 85, 99, 122
symmpart (symmpart-methods), 179
symmpart, CsparseMatrix-method (symmpart-methods), 179
symmpart, denseMatrix-method (symmpart-methods), 179
symmpart, diagonalMatrix-method (symmpart-methods), 179
symmpart, indMatrix-method (symmpart-methods), 179
symmpart, matrix-method (symmpart-methods), 179
symmpart, RsparseMatrix-method (symmpart-methods), 179
symmpart, TsparseMatrix-method (symmpart-methods), 179
symmpart-methods, 179
t, 69, 105, 108, 109, 113, 125, 131, 132, 144, 159
t,CsparseMatrix-method (CsparseMatrix-class), 44
t,denseMatrix-method (denseMatrix-class), 49
t,diagonalMatrix-method (diagonalMatrix-class), 55
t,indMatrix-method (indMatrix-class), 91
t,pMatrix-method (pMatrix-class), 136
t,RsparseMatrix-method (RsparseMatrix-class), 151
t,sparseVector-method (sparseVector-class), 172
t,TsparseMatrix-method (TsparseMatrix-class), 181
t2graph, 166
tcrossprod, 15, 23, 113, 114, 183
tcrossprod (matmult-methods), 113
tcrossprod,ANY,Matrix-method (matmult-methods), 113
tcrossprod,ANY,sparseVector-method (matmult-methods), 113
tcrossprod,CsparseMatrix,CsparseMatrix-method (matmult-methods), 113
tcrossprod,CsparseMatrix,denseMatrix-method (matmult-methods), 113
tcrossprod,CsparseMatrix,diagonalMatrix-method (matmult-methods), 113
tcrossprod,CsparseMatrix,matrix-method (matmult-methods), 113
tcrossprod,CsparseMatrix,missing-method (matmult-methods), 113
tcrossprod,CsparseMatrix,RsparseMatrix-method (matmult-methods), 113
tcrossprod,CsparseMatrix,TsparseMatrix-method (matmult-methods), 113
tcrossprod,CsparseMatrix,vector-method (matmult-methods), 113
tcrossprod,denseMatrix,CsparseMatrix-method (matmult-methods), 113
tcrossprod,denseMatrix,denseMatrix-method (matmult-methods), 113
tcrossprod,denseMatrix,diagonalMatrix-method (matmult-methods), 113
tcrossprod,denseMatrix,matrix-method (matmult-methods), 113
tcrossprod,denseMatrix,missing-method (matmult-methods), 113
tcrossprod,denseMatrix,RsparseMatrix-method (matmult-methods), 113
tcrossprod,denseMatrix,TsparseMatrix-method (matmult-methods), 113
tcrossprod,denseMatrix,vector-method (matmult-methods), 113
tcrossprod,diagonalMatrix,CsparseMatrix-method (matmult-methods), 113
tcrossprod,diagonalMatrix,denseMatrix-method
tcrossprod, diagonalMatrix, diagonalMatrix-method
(mumatmult-methods), 113

tcrossprod, diagonalMatrix, matrix-method
(mumatmult-methods), 113
tcrossprod, diagonalMatrix, missing-method
(mumatmult-methods), 113
tcrossprod, diagonalMatrix, R sparseMatrix-method
tcrossprod, sparseMatrix, R sparseMatrix-method
(mumatmult-methods), 113
tcrossprod, diagonalMatrix, vector-method
(mumatmult-methods), 113
tcrossprod, indMatrix, Matrix-method
(mumatmult-methods), 113
tcrossprod, indMatrix, matrix-method
(mumatmult-methods), 113
tcrossprod, indMatrix, missing-method
(mumatmult-methods), 113
tcrossprod, indMatrix, vector-method
(mumatmult-methods), 113
tcrossprod, Matrix, ANY-method
(mumatmult-methods), 113
tcrossprod, Matrix, C sparseMatrix-method
tcrossprod, sparseVector, C sparseMatrix-method
(mumatmult-methods), 113
tcrossprod, matrix, denseMatrix-method
(mumatmult-methods), 113
tcrossprod, matrix, diagonalMatrix-method
(mumatmult-methods), 113
tcrossprod, matrix, indMatrix-method
(mumatmult-methods), 113
tcrossprod, matrix, matrix-method
(mumatmult-methods), 113
tcrossprod, matrix, sparseVector-method
(mumatmult-methods), 113
tcrossprod, matrix, sparseVector-method
(mumatmult-methods), 113
tcrossprod, matrix, TsparseMatrix-method
tcrossprod, TsparseMatrix, TsparseMatrix-method
(mumatmult-methods), 113
tcrossprod, pMatrix, Matrix-method
(mumatmult-methods), 113
tcrossprod, pMatrix, matrix-method
(mumatmult-methods), 113
tcrossprod, pMatrix, missing-method
(mumatmult-methods), 113
tcrossprod, pMatrix, pMatrix-method
(mumatmult-methods), 113
tcrossprod, pMatrix, vector-method
(mumatmult-methods), 113
tcrossprod, R sparseMatrix, C sparseMatrix-method
tcrossprod, sparseVector, vector-method
(mumatmult-methods), 113
INDEX

(matmult-methods), 113
tcrossprod.vector, sparseVector-method (matmult-methods), 113
tcrossprod.vector, TsparseMatrix-method (matmult-methods), 113
tcrossprod-methods (matmult-methods), 113
toeplitz, 172
toeplitz, sparseVector-method (sparseVector-class), 172
triangularMatrix, 10, 26, 33, 54, 57, 58, 70, 72–74, 83, 85, 88, 100, 109, 114, 116, 132, 134, 178
triangularMatrix-class, 180
tril (band-methods), 9
tril, CsparseMatrix-method (band-methods), 9
tril, denseMatrix-method (band-methods), 9
tril, diagonalMatrix-method (band-methods), 9
tril, indMatrix-method (band-methods), 9
tril, matrix-method (band-methods), 9
tril, RsparseMatrix-method (band-methods), 9
tril, TsparseMatrix-method (band-methods), 9
tril-methods (band-methods), 9
triu (band-methods), 9
triu, CsparseMatrix-method (band-methods), 9
triu, denseMatrix-method (band-methods), 9
triu, diagonalMatrix-method (band-methods), 9
triu, indMatrix-method (band-methods), 9
triu, matrix-method (band-methods), 9
triu, RsparseMatrix-method (band-methods), 9
triu, TsparseMatrix-method (band-methods), 9
triu-methods (band-methods), 9
TRUE, 60, 97
tryCatch, 111
TsparseMatrix, 7, 11–13, 38, 53, 64, 65, 83, 106, 112, 129, 158, 159, 162, 163, 174, 175, 182
TsparseMatrix-class, 181
type, 176
typeof, 54
uniqTsparse (asUniqueT), 7
unname, Matrix-method (Matrix-class), 116

unnamed.MatrixFactorization-method (MatrixFactorization-class), 120
unpack, 135, 183
unpack (pack), 133
unpack, matrix-method (pack), 133
unpack, packedMatrix-method (pack), 133
unpack, sparseMatrix-method (pack), 133
unpack, unpackedMatrix-method (pack), 133
unpack-methods (pack), 133
unpackMatrix, 34, 49, 82, 83, 133, 135
unpackMatrix-class, 182
update, 184
update, CHMfactor-method (CHMfactor-class), 21
updown, 24
updown (updown-methods), 183
updown, character, ANY, ANY-method (updown-methods), 183
updown, logical, dgCMatrix, CHMfactor-method (updown-methods), 183
updown, logical, dsCMatrix, CHMfactor-method (updown-methods), 183
updown, logical, dtCMatrix, CHMfactor-method (updown-methods), 183
updown, logical, Matrix, CHMfactor-method (updown-methods), 183
updown, logical, matrix, CHMfactor-method (updown-methods), 183
updown-methods, 183
USCounties, 184
validObject, 45

warning, 18, 110
which, 56, 60, 104, 107, 121, 130
which, indMatrix-method (indMatrix-class), 91
which, ldenseMatrix-method (ldenseMatrix-class), 104
which, ldiMatrix-method (ldiMatrix-class), 104
which, lsparseMatrix-method (lsparseMatrix-classes), 106
which, lsparseVector-method (lsparseVector-class), 172
which, ndenseMatrix-method (ndenseMatrix-class), 121
which, n logicalMatrix-method (n logicalMatrix-class), 124
which, nsparseMatrix-method (nsparseMatrix-classes), 129
which, nsparseVector-method
  (sparseVector-class), 172
writeMM, 166
writeMM (externalFormats), 79
writeMM, CsparseMatrix-method
  (externalFormats), 79
writeMM, sparseMatrix-method
  (externalFormats), 79
wrld_1deg, 185
xtabs, 163, 166
zapsmall, 64, 127
zapsmall, Matrix-method (Matrix-class), 116
zapsmall, sparseVector-method
  (sparseVector-class), 172
zMatrix-class (Matrix-notyet), 118
zsparseVector-class
  (sparseVector-class), 172