Package 'SparseArray'

November 19, 2026

Title High-performance sparse data representation and manipulation in R

Description The SparseArray package provides array-like containers for efficient in-memory representation of multidimensional sparse data in R (arrays and matrices). The package defines the SparseArray virtual class and two concrete subclasses: COO_SparseArray and SVT_SparseArray. Each subclass uses its own internal representation of the nonzero multidimensional data: the ``COO layout'' and the ``SVT layout'', respectively. SVT_SparseArray objects mimic as much as possible the behavior of ordinary matrix and array objects in base R. In particular, they suppport most of the ``standard matrix and array API'' defined in base R and in the matrixStats package from CRAN.

biocViews Infrastructure, DataRepresentation

URL https://bioconductor.org/packages/SparseArray

BugReports https://github.com/Bioconductor/SparseArray/issues

Version 1.13.5

License Artistic-2.0

Encoding UTF-8

Depends R (>= 4.3.0), methods, Matrix, BiocGenerics (>= 0.43.1), MatrixGenerics (>= 1.11.1), S4Vectors (>= 0.43.2), S4Arrays (>= 1.13.2)

Imports utils, stats, matrixStats, IRanges, XVector

LinkingTo S4Vectors, IRanges, XVector

Suggests HDF5Array, ExperimentHub, testthat, knitr, rmarkdown, BiocStyle

VignetteBuilder knitr

Collate utils.R options.R OPBufTree.R thread-control.R sparseMatrix-utils.R is_nonzero.R SparseArray-class.R COO_SparseArray-class.R SVT_SparseArray-class.R extract_sparse_array.R read_block_as_sparse.R SparseArray-dim-tuning.R SparseArray-aperm.R SparseArray-subsetting.R SparseArray-subassignment.R SparseArray-abind.R SparseArray-summarization.R SparseArray-Arith-methods.R SparseArray-Compare-methods.R SparseArray-Logic-methods.R SparseArray-Math-methods.R

SparseArray-Complex-methods.R SparseArray-misc-methods.R
SparseArray-matrixStats.R rowsum-methods.R SparseMatrix-mult.R
randomSparseArray.R readSparseCSV.R is_nonna.R NaArray-class.R
NaArray-aperm.R NaArray-subsetting.R NaArray-subassignment.R
NaArray-abind.R NaArray-summarization.R NaArray-Arith-methods.R
NaArray-Compare-methods.R NaArray-Logic-methods.R
NaArray-Math-methods.R NaArray-misc-methods.R
NaArray-matrixStats.R zzz.R

git_url https://git.bioconductor.org/packages/SparseArray

git_branch devel

git_last_commit 7f0a66c

git_last_commit_date 2026-11-13

Repository Bioconductor 3.24

Date/Publication 2026-11-18

Author Hervé Pagès [aut, cre] (ORCID: <https://orcid.org/0009-0002-8272-4522>),
Vince Carey [fnd] (ORCID: <https://orcid.org/0000-0003-4046-0063>),
Rafael A. Irizarry [fnd] (ORCID:
<https://orcid.org/0000-0002-3944-4309>),
Jacques Serizay [ctb] (ORCID: <https://orcid.org/0000-0002-4295-0624>)

Maintainer Hervé Pagès <hpages.on.github@gmail.com>

Contents

COO_SparseArray-class . . . 3
extract_sparse_array . . . 5
is_nonna . . . 7
is_nonzero . . . 9
NaArray . . . 12
NaArray-abind . . . 14
NaArray-aperm . . . 15
NaArray-Arith-methods . . . 15
NaArray-Compare-methods . . . 17
NaArray-Logic-methods . . . 18
NaArray-Math-methods . . . 19
NaArray-matrixStats . . . 20
NaArray-misc-methods . . . 22
NaArray-subassignment . . . 23
NaArray-subsetting . . . 24
NaArray-summarization . . . 26
randomSparseArray . . . 27
readSparseCSV . . . 29
read_block_as_sparse . . . 31
rowsum-methods . . . 32
SparseArray . . . 33
SparseArray-abind . . . 37
SparseArray-aperm . . . 38
SparseArray-Arith-methods . . . 39
SparseArray-Compare-methods . . . 41
SparseArray-Complex-methods . . . 42

SparseArray-dim-tuning . . . 42
SparseArray-Logic-methods . . . 43
SparseArray-Math-methods . . . 44
SparseArray-matrixStats . . . 45
SparseArray-misc-methods . . . 48
SparseArray-subassignment . . . 50
SparseArray-subsetting . . . 51
SparseArray-summarization . . . 52
SparseMatrix-mult . . . 54
sparseMatrix-utils . . . 55
SVT_SparseArray-class . . . 55
thread-control . . . 59

Index **61**

`COO_SparseArray-class` *COO_SparseArray objects*

Description

The COO_SparseArray class is a container for efficient in-memory representation of multidimensional sparse arrays. It uses the *COO layout* to represent the nonzero data internally.

A COO_SparseMatrix object is a COO_SparseArray object of 2 dimensions.

IMPORTANT NOTE: COO_SparseArray and COO_SparseMatrix objects are now superseded by the new and more efficient SVT_SparseArray and SVT_SparseMatrix objects.

Usage

```
## Constructor function:
COO_SparseArray(dim, nzcoo=NULL, nzdata=NULL, dimnames=NULL, check=TRUE)

## Getters (in addition to dim(), length(), and dimnames()):
nzcoo(x)
nzdata(x)
```

Arguments

`dim`	The dimensions (supplied as an integer vector) of the COO_SparseArray or COO_SparseMatrix object to construct.
`nzcoo`	A matrix containing the array coordinates of the nonzero elements. This must be an integer matrix of array coordinates like one returned by `base::arrayInd` or `S4Arrays::Lindex2Mindex`, that is, a matrix with `length(dim)` columns and where each row is an n-tuple representing the coordinates of an array element.
`nzdata`	A vector (atomic or list) of length `nrow(nzcoo)` containing the nonzero elements.
`dimnames`	The *dimnames* of the object to construct. Must be `NULL` or a list of length the number of dimensions. Each list element must be either `NULL` or a character vector along the corresponding dimension.
`check`	Should the object be validated upon construction?
`x`	A COO_SparseArray or COO_SparseMatrix object.

Value

- For `COO_SparseArray()`: A COO_SparseArray or COO_SparseMatrix object.
- For `nzcoo()`: A matrix with one column per dimension containing the *array coordinates* of the nonzero elements.
- For `nzdata()`: A vector *parallel* to `nzcoo(x)` (i.e. with one element per row in `nzcoo(x)`) containing the nonzero elements.

See Also

- The new SVT_SparseArray class for a replacement of of the COO_SparseArray class.
- The SparseArray class for the virtual parent class of COO_SparseArray and SVT_SparseArray.
- dgCMatrix-class and lgCMatrix-class in the **Matrix** package, for the de facto standard for sparse matrix representations in the R ecosystem.
- `base::arrayInd` in the **base** package.
- `S4Arrays::Lindex2Mindex` in the **S4Arrays** package for an improved (faster) version of `base::arrayInd`.
- Ordinary array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## EXAMPLE 1
## ---------------------------------------------------------------------
dim1 <- 5:3
nzcoo1 <- Lindex2Mindex(sample(60, 8), 5:3)
nzdata1 <- 11.11 * seq_len(nrow(nzcoo1))
coo1 <- COO_SparseArray(dim1, nzcoo1, nzdata1)
coo1

nzcoo(coo1)
nzdata(coo1)
type(coo1)
sparsity(coo1)

as.array(coo1)  # back to a dense representation

#as.matrix(coo1)  # error!

## ---------------------------------------------------------------------
## EXAMPLE 2
## ---------------------------------------------------------------------
m2 <- matrix(c(5:-2, rep.int(c(0L, 99L), 11)), ncol=6)
coo2 <- as(m2, "COO_SparseArray")
class(coo2)
dim(coo2)
length(coo2)
nzcoo(coo2)
nzdata(coo2)
type(coo2)
sparsity(coo2)

stopifnot(identical(as.matrix(coo2), m2))
```

```
t(coo2)
stopifnot(identical(as.matrix(t(coo2)), t(as.matrix(coo2))))

## ---------------------------------------------------------------------
## COERCION FROM/TO dg[C|R]Matrix OR lg[C|R]Matrix OBJECTS
## ---------------------------------------------------------------------
## dg[C|R]Matrix and lg[C|R]Matrix objects are defined in the Matrix
## package.

## dgCMatrix/dgRMatrix:

M2C <- as(coo2, "dgCMatrix")
stopifnot(identical(M2C, as(m2, "dgCMatrix")))

coo2C <- as(M2C, "COO_SparseArray")
## 'coo2C' is the same as 'coo2' except that 'nzdata(coo2C)' has
## type "double" instead of "integer":
stopifnot(all.equal(coo2, coo2C))
typeof(nzdata(coo2C))  # double
typeof(nzdata(coo2))   # integer

M2R <- as(coo2, "dgRMatrix")
stopifnot(identical(M2R, as(m2, "dgRMatrix")))
coo2R <- as(M2R, "COO_SparseArray")
stopifnot(all.equal(as.matrix(coo2), as.matrix(coo2R)))

## lgCMatrix/lgRMatrix:

m3 <- m2 == 99  # logical matrix
coo3 <- as(m3, "COO_SparseArray")
class(coo3)
type(coo3)

M3C <- as(coo3, "lgCMatrix")
stopifnot(identical(M3C, as(m3, "lgCMatrix")))
coo3C <- as(M3C, "COO_SparseArray")
identical(as.matrix(coo3), as.matrix(coo3C))

M3R <- as(coo3, "lgRMatrix")
#stopifnot(identical(M3R, as(m3, "lgRMatrix")))
coo3R <- as(M3R, "COO_SparseArray")
identical(as.matrix(coo3), as.matrix(coo3R))

## ---------------------------------------------------------------------
## A BIG COO_SparseArray OBJECT
## ---------------------------------------------------------------------
nzcoo4 <- cbind(sample(25000, 600000, replace=TRUE),
                sample(195000, 600000, replace=TRUE))
nzdata4 <- runif(600000)
coo4 <- COO_SparseArray(c(25000, 195000), nzcoo4, nzdata4)
coo4
sparsity(coo4)
```

---

extract_sparse_array *extract_sparse_array*

Description

`extract_sparse_array()` is an internal generic function that is the workhorse behind the default `read_block_as_sparse()` method. It is not intended to be used directly by the end user.

It is similar to the `extract_array()` internal generic function defined in the **S4Arrays** package, with the major difference that, in the case of `extract_sparse_array()`, the extracted array data is returned as a SparseArray object instead of an ordinay array.

Usage

```
extract_sparse_array(x, index)

## S4 method for signature 'ANY'
extract_sparse_array(x, index)
```

Arguments

`x`	An array-like object for which `is_sparse(x)` is `TRUE`.
`index`	An unnamed list of integer vectors, one per dimension in `x`. Each vector is called a *subscript* and can only contain positive integers that are valid 1-based indices along the corresponding dimension in `x`. Empty or missing subscripts are allowed. They must be represented by list elements set to `integer(0)` or `NULL`, respectively. The subscripts cannot contain NAs or non-positive values. Individual subscripts are NOT allowed to contain duplicated indices. This is an important difference with `extract_array`.

Details

`extract_sparse_array()` should *always* be called on an array-like object `x` for which `is_sparse(x)` is `TRUE`. Also it should *never* be called with duplicated indices in the individual list elements of the `index` argument.

For maximum efficiency, `extract_sparse_array()` methods should:

1. NOT check that `is_sparse(x)` is `TRUE`.
2. NOT check that the individual list elements in `index` contain no duplicated indices.
3. NOT try to do anything with the dimnames on `x`.
4. always operate natively on the sparse representation of the data in `x`, that is, they should never *expand* it into a dense representation (e.g. with *as.array()*).

Like for `extract_array()`, `extract_sparse_array()` methods need to support empty or missing subscripts. For example, if `x` is an M x N matrix-like object for which `is_sparse(x)` is `TRUE`, then `extract_sparse_array(x, list(NULL, integer(0)))` must return an M x 0 SparseArray derivative, and `extract_sparse_array(x, list(integer(0), integer(0)))` a 0 x 0 SparseArray derivative.

Value

A SparseArray derivative (COO_SparseArray or SVT_SparseArray) of the same `type()` as `x`. For example, if `x` is an object representing an M x N sparse matrix of complex numbers (i.e. `type(x) == "complex"`), then `extract_sparse_array(x, list(NULL, 2L))` must return the 2nd column in `x` as an M x 1 SparseArray derivative of `type()` `"complex"`.

See Also

- `is_sparse` in the **S4Arrays** package to check whether an object uses a sparse representation of the data or not.
- SparseArray objects.
- `S4Arrays::type` in the **S4Arrays** package to get the type of the elements of an array-like object.
- `read_block_as_sparse` to read array blocks as SparseArray objects.
- `extract_array` in the **S4Arrays** package.
- dgCMatrix-class in the **Matrix** package.

Examples

```
extract_sparse_array
showMethods("extract_sparse_array")

## --- On a dgCMatrix object ---

m <- matrix(0L, nrow=6, ncol=4)
m[c(1:2, 8, 10, 15:17, 24)] <- (1:8)*10L
dgcm <- as(m, "dgCMatrix")
dgcm

extract_sparse_array(dgcm, list(3:6, NULL))
extract_sparse_array(dgcm, list(3:6, 2L))
extract_sparse_array(dgcm, list(3:6, integer(0)))

## --- On a SparseArray object ---

a <- array(0L, dim=5:3, dimnames=list(letters[1:5], NULL, LETTERS[1:3]))
a[c(1:2, 8, 10, 15:17, 20, 24, 40, 56:60)] <- (1:15)*10L
svt <- as(a, "SparseArray")
svt

extract_sparse_array(svt, list(NULL, 4:2, 1L))
extract_sparse_array(svt, list(NULL, 4:2, 2:3))
extract_sparse_array(svt, list(NULL, 4:2, integer(0)))
```

`is_nonna` *is_nonna() and the nna*() functions*

Description

A set of functions for direct manipulation of the non-NA elements of an array-like object.

Note that, for all these functions, a non-NA element is an element for which `is.na()` is FALSE or `is.nan()` is TRUE.

Usage

```
is_nonna(x)

nnacount(x)
nnawhich(x, arr.ind=FALSE)
nnavals(x)
nnavals(x) <- value
```

Arguments

x	Typically (but not necessarily) an array-like object that is *non-NA sparse*, like an NaArray object. However, x can also be an ordinary matrix or array, or any matrix-like or array-like object.
arr.ind	If arr.ind=FALSE (the default), the indices of the non-NA array elements are returned in a numeric vector (a.k.a. *L-index*). Otherwise, they're returned in an ordinary matrix (a.k.a. *M-index*). See ?Lindex in the **S4Arrays** package for more information about *L-index* and *M-index*, and how to convert from one to the other. Note that using arr.ind=TRUE won't work if nnacount(x) is >= .Machine$integer.max (= 2^31), because, in that case, the returned *M-index* would need to be a matrix with more rows than what is supported by base R.
value	A vector, typically of length nnacount(x) (or 1) and type type(x).

Details

nnacount(x) and nnawhich(x) are equivalent to, but typically more efficient than, sum(is_nonna(x)) and which(is_nonna(x)), respectively.

nnavals(x) is equivalent to, but typically more efficient than, x[nnawhich(x)] (or x[is_nonna(x)]).

nnavals(x) <- value replaces the values of the non-NA array elements in x with the supplied values. It's equivalent to, but typically more efficient than, x[nnawhich(x)] <- value.

Note that nnavals(x) <- nnavals(x) is guaranteed to be a no-op.

Value

is_nonna(): An array-like object of type() "logical" and same dimensions as the input object.

nnacount(): The number of non-NA array elements in x.

nnawhich(): The indices of the non-NA array elements in x, either as an *L-index* (if arr.ind is FALSE) or as an *M-index* (if arr.ind is TRUE). Note that the indices are returned sorted in strictly ascending order.

nnavals(): A vector of the same type() as x and containing the values of the non-NA array elements in x. Note that the returned vector is guaranteed to be *parallel* to nnawhich(x).

See Also

- is_nonzero for is_nonzero() and nz*() functions nzcount(), nzwhich(), etc...
- NaArray objects.
- Ordinary array objects in base R.
- base::which in base R.

Examples

```
m <- rbind(c(NA, 0, 3.25, NaN), c(NA, NaN, NA, 1))
is_nonna(m)  # the NaN values are considered non-NA elements

a <- array(NA_integer_, dim=c(5, 12, 2))
a[sample(length(a), 20)] <- (-9):10

is_nonna(a)

## Get the number of non-NA array elements in 'a':
nnacount(a)

## nnawhich() returns the indices of the non-NA array elements in 'a'.
## Either as a "L-index" i.e. an integer (or numeric) vector of
## length 'nnacount(a)' containing "linear indices":
nnaidx <- nnawhich(a)
length(nnaidx)
head(nnaidx)

## Or as an "M-index" i.e. an integer matrix with 'nnacount(a)' rows
## and one column per dimension where the rows represent "array indices"
## (a.k.a. "array coordinates"):
Mnnaidx <- nnawhich(a, arr.ind=TRUE)
dim(Mnnaidx)

## Each row in the matrix is an n-tuple representing the "array
## coordinates" of a non-NA element in 'a':
head(Mnnaidx)
tail(Mnnaidx)

## Extract the values of the non-NA array elements in 'a' and return
## them in a vector "parallel" to 'nnawhich(a)':
a_nnavals <- nnavals(a)  # equivalent to 'a[nnawhich(a)]'
length(a_nnavals)
head(a_nnavals)

nnavals(a) <- 10 ^ nnavals(a)
a

## Sanity checks:
stopifnot(
  identical(as.matrix(is_nonna(NaArray(m))), is_nonna(m)),
  identical(nnaidx, which(!is.na(a))),
  identical(Mnnaidx, which(!is.na(a), arr.ind=TRUE, useNames=FALSE)),
  identical(nnavals(a), a[nnaidx]),
  identical(nnavals(a), a[Mnnaidx]),
  identical(`nnavals<-`(a, nnavals(a)), a)
)
```

`is_nonzero` *is_nonzero() and the nz*() functions*

Description

A set of functions for direct manipulation of the nonzero elements of an array-like object.

Usage

```
is_nonzero(x)

nzcount(x)
nzwhich(x, arr.ind=FALSE)
nzvals(x)
nzvals(x) <- value

sparsity(x)
```

Arguments

x	Typically (but not necessarily) an array-like object that is sparse, like a SparseArray derivative, or a dg[ClR]Matrix or lg[ClR]Matrix object from the **Matrix** package. However, x can also be an ordinary matrix or array, or any matrix-like or array-like object.
arr.ind	If arr.ind=FALSE (the default), the indices of the nonzero array elements are returned in a numeric vector (a.k.a. *L-index*). Otherwise, they're returned in an ordinary matrix (a.k.a. *M-index*). See ?Lindex in the **S4Arrays** package for more information about *L-index* and *M-index*, and how to convert from one to the other. Note that using arr.ind=TRUE won't work if nzcount(x) is >= .Machine$integer.max (= 2^31), because, in that case, the returned *M-index* would need to be a matrix with more rows than what is supported by base R.
value	A vector, typically of length nzcount(x) (or 1) and type type(x).

Details

nzcount(x) and nzwhich(x) are equivalent to, but typically more efficient than, sum(is_nonzero(x)) and which(is_nonzero(x)), respectively.

nzvals(x) is equivalent to, but typically more efficient than, x[nzwhich(x)] (or x[is_nonzero(x)]).

nzvals(x) <- value replaces the values of the nonzero array elements in x with the supplied values. It's equivalent to, but typically more efficient than, x[nzwhich(x)] <- value.

Note that nzvals(x) <- nzvals(x) is guaranteed to be a no-op.

Value

is_nonzero(): An array-like object of type() "logical" and same dimensions as the input object.

nzcount(): The number of nonzero array elements in x.

nzwhich(): The indices of the nonzero array elements in x, either as an *L-index* (if arr.ind is FALSE) or as an *M-index* (if arr.ind is TRUE). Note that the indices are returned sorted in strictly ascending order.

nzvals(): A vector of the same type() as x and containing the values of the nonzero array elements in x. Note that the returned vector is guaranteed to be *parallel* to nzwhich(x).

sparsity(x): The ratio between the number of zero-valued elements in array-like object x and its total number of elements (length(x) or prod(dim(x))). More precisely, sparsity(x) is 1 - nzcount(x)/length(x).

See Also

- is_nonna for `is_nonna()` and `nna*()` functions `nnacount()`, `nnawhich()`, etc...
- SparseArray objects.
- dgCMatrix-class, lgCMatrix-class, and ngCMatrix-class in the **Matrix** package.
- Ordinary array objects in base R.
- `base::which` in base R.

Examples

```
a <- array(rpois(120, lambda=0.3), dim=c(5, 12, 2))

is_nonzero(a)

## Get the number of nonzero array elements in 'a':
nzcount(a)

## nzwhich() returns the indices of the nonzero array elements in 'a'.
## Either as a "L-index" i.e. an integer (or numeric) vector of
## length 'nzcount(a)' containing "linear indices":
nzidx <- nzwhich(a)
length(nzidx)
head(nzidx)

## Or as an "M-index" i.e. an integer matrix with 'nzcount(a)' rows
## and one column per dimension where the rows represent "array indices"
## (a.k.a. "array coordinates"):
Mnzidx <- nzwhich(a, arr.ind=TRUE)
dim(Mnzidx)

## Each row in the matrix is an n-tuple representing the "array
## coordinates" of a nonzero element in 'a':
head(Mnzidx)
tail(Mnzidx)

## Extract the values of the nonzero array elements in 'a' and return
## them in a vector "parallel" to 'nzwhich(a)':
a_nzvals <- nzvals(a)  # equivalent to 'a[nzwhich(a)]'
length(a_nzvals)
head(a_nzvals)

nzvals(a) <- log1p(nzvals(a))
a

## Sanity checks:
stopifnot(
  identical(nzidx, which(a != 0)),
  identical(Mnzidx, which(a != 0, arr.ind=TRUE, useNames=FALSE)),
  identical(nzvals(a), a[nzidx]),
  identical(nzvals(a), a[Mnzidx]),
  identical(`nzvals<-`(a, nzvals(a)), a)
)
```

NaArray *NaArray objects*

Description

EXPERIMENTAL!!!

Like SVT_SparseArray objects but the background value is NA instead of zero.

Usage

```
## Constructor function:
NaArray(x, dim=NULL, dimnames=NULL, type=NA)
```

Arguments

x	If dim is NULL (the default) then x must be an ordinary matrix or array, or a dgCMatrix/lgCMatrix object, or any matrix-like or array-like object that supports coercion to NaArray. If dim is provided then x can either be missing, a vector (atomic or list), or an array-like object. If missing, then an all-NA NaArray object will be constructed. Otherwise x will be used to fill the returned object (length(x) must be <= prod(dim)). Note that if x is an array-like object then its dimensions are ignored i.e. it's treated as a vector.
dim	NULL or the dimensions (supplied as an integer vector) of the NaArray or NaMatrix object to construct. If NULL (the default) then the returned object will have the dimensions of matrix-like or array-like object x.
dimnames	The *dimnames* of the object to construct. Must be NULL or a list of length the number of dimensions. Each list element must be either NULL or a character vector along the corresponding dimension. If both dim and dimnames are NULL (the default) then the returned object will have the *dimnames* of matrix-like or array-like object x.

type	A single string specifying the requested type of the object.
	By default the NaArray object returned by the constructor function will have the same type() as x. However the user can use the type argument to request a different type. Note that doing:
	`naa <- NaArray(x, type=type)`
	is equivalent to doing:
	`naa <- NaArray(x)`
	`type(naa) <- type`
	but the former is more convenient and will generally be more efficient.
	The supported types for NaArray objects are "integer", "logical", "double", "complex", and "character".

Details

NaArray is a concrete subclass of the Array virtual class (the latter is defined in the **S4Arrays** package). This makes NaArray objects Array derivatives.

Like with SVT_SparseArray objects, the non-NA data in an NaArray object is stored in a *Sparse Vector Tree*. See ?SVT_SparseArray for more information.

Most of the *matrix and array standard API* defined in base R should work on NaArray objects, including `dim()`, `length()`, `dimnames()`, `` `dimnames<-`() ``, `[`, `drop()`, `` `[<-` `` (subassignment), `t()`, `rbind()`, `cbind()`, etc...

NaArray objects also support `type()`, `` `type<-`() ``, `is_nonna()`, `nnacount()`, `nnawhich()`, `nnavals()`, `` `nnavals<-`() ``, `arbind()`, and `acbind()`.

Value

An NaArray or NaMatrix object.

The `type()` of the input object is preserved, except if a different one was requested via the `type` argument.

See Also

- The SVT_SparseArray class.
- is_nonna for `is_nonna()` and `nna*()` functions `nnacount()`, `nnawhich()`, etc...
- NaArray_aperm for permuting the dimensions of an NaArray object (e.g. transposition).
- NaArray_subsetting for subsetting an NaArray object.
- NaArray_subassignment for NaArray subassignment.
- NaArray_abind for combining 2D or multidimensional NaArray objects.
- NaArray_summarization for NaArray summarization methods.
- NaArray_Arith, NaArray_Compare, and NaArray_Logic, for operations from the `Arith`, `Compare`, and `Arith` groups on NaArray objects.
- NaArray_Math for operations from the `Math` and `Math2` groups on NaArray objects.
- NaArray_misc for miscellaneous operations on an NaArray object.
- NaArray_matrixStats for col/row summarization methods for NaArray objects.
- Ordinary array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## Display details of class definition & known subclasses
## ---------------------------------------------------------------------

showClass("NaArray")

## ---------------------------------------------------------------------
## The NaArray() constructor
## ---------------------------------------------------------------------

naa1 <- NaArray(dim=5:3)  # all-NA object
naa1

naa2 <- NaArray(dim=c(35000, 2e6), type="integer")  # all-NA object
```

```
naa2

## Add some non-NA values to 'naa2':
naa2[cbind( 1:99, 2:100)] <-  1L
naa2[cbind(1:100, 1:100)] <-  0L
naa2[cbind(2:100,  1:99)] <- -1L
naa2

## The dimnames can be specified at construction time, or
## added/modified later:
naa3 <- NaArray(c(NA, NA, 1L, NA, 0:7, rep(NA, 4), 12:14, NA),
                dim=4:5, dimnames=list(letters[1:4], LETTERS[1:5]))
naa3

colnames(naa3) <- LETTERS[22:26]
naa3

## Sanity checks:
stopifnot(
  is(naa1, "NaArray"),
  identical(dim(naa1), 5:3),
  identical(as.array(naa1), array(dim=5:3)),
  is(naa2, "NaMatrix"),
  all.equal(dim(naa2), c(35000, 2e6)),
  identical(nnacount(naa2), 298L),
  is(naa3, "NaMatrix"),
  identical(dim(naa3), 4:5),
  identical(nnacount(naa3), 12L)
)
```

`NaArray-abind` *Combine multidimensional NaArray objects*

Description

EXPERIMENTAL!!!

Like ordinary matrices and arrays in base R, NaMatrix objects can be combined by rows or columns, with `rbind()` or `cbind()`, and multidimensional NaArray objects can be bound along any dimension with `abind()`.

Note that `arbind()` can also be used to combine the objects along their first dimension, and `acbind()` can be used to combine them along their second dimension.

See Also

- `cbind` in base R.
- `abind` in the **S4Arrays** package.
- NaArray objects.
- Ordinary array objects in base R.

Examples

```
# COMING SOON...
```

NaArray-aperm *NaArray transposition*

Description

EXPERIMENTAL!!!

Transpose an NaArray object by permuting its dimensions.

Value

COMING SOON...

See Also

- `aperm()` in base R.
- NaArray objects.
- Ordinary array objects in base R.

Examples

```
# COMING SOON...
```

NaArray-Arith-methods *'Arith' operations on NaArray objects*

Description

EXPERIMENTAL!!!

NaArray objects support all operations from the `Arith` group. See `?S4groupGeneric` in the **methods** package for more information about the `Arith` group generic.

Note that NaArray of `type()` `"complex"` don't support `Arith` operations at the moment.

Details

Three forms of 'Arith' operations involving NaArray objects are supported:

1. Between an NaArray object `naa` and an atomic vector `y`:

   ```
   naa op y
   y op naa
   ```

2. Between two NaArray objects `naa1` and `naa2` of same dimensions (a.k.a. *conformable arrays*):

   ```
   naa1 op naa2
   ```

3. Between an NaArray object `naa` and an SVT_SparseArray object `svt` of same dimensions (a.k.a. *conformable arrays*):

   ```
   naa op svt
   svt op naa
   ```

Value

An NaArray object of the same dimensions as the input object(s).

See Also

- S4groupGeneric in the **methods** package.
- NaArray objects.
- SVT_SparseArray objects.
- Ordinary array objects in base R.

Examples

```
nam1 <- NaArray(dim=c(15, 6), type="integer")
nam1[cbind(1:15, 2)] <- 100:114
nam1[cbind(1:15, 5)] <- -(114:100)
nam1

nam1 * -0.01

nam1 * 10   # result is of type "double"
nam1 * 10L  # result is of type "integer"

nam1 * c(10, 5, 0.1)  # right vector recycled along 1st dimension

nam1 / 10L

nam1 ^ 2
nam1 ^ (2:6)

nam1 %% 5L
nam1 %/% 5L

nam2 <- NaArray(dim=dim(nam1), type="double")
nam2[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <- runif(26)
nam2

nam2 + nam1
nam2 - nam1
nam2 * nam1
nam2 / nam1
nam2 ^ nam1
nam2 %% nam1
nam2 %/% nam1

## Sanity checks:
m1 <- as.matrix(nam1)
m2 <- as.matrix(nam2)
stopifnot(
  identical(as.matrix(nam1 * -0.01), m1 * -0.01),
  identical(as.matrix(nam1 * 10), m1 * 10),
  identical(as.matrix(nam1 * 10L), m1 * 10L),
  identical(as.matrix(nam1 / 10L), m1 / 10L),
  identical(as.matrix(nam1 ^ 3.5), m1 ^ 3.5),
  identical(as.matrix(nam1 %% 5L), m1 %% 5L),
  identical(as.matrix(nam1 %/% 5L), m1 %/% 5L),
```

```
  identical(as.matrix(nam2 + nam1), m2 + m1),
  identical(as.matrix(nam2 - nam1), m2 - m1),
  identical(as.matrix(nam2 * nam1), m2 * m1),
  identical(as.matrix(nam2 / nam1), m2 / m1),
  all.equal(as.matrix(nam2 ^ nam1), m2 ^ m1),
  identical(as.matrix(nam2 %% nam1), m2 %% m1),
  identical(as.matrix(nam2 %/% nam1), m2 %/% m1)
)
```

`NaArray-Compare-methods`

'Compare' operations on NaArray objects

Description

EXPERIMENTAL!!!

NaArray objects support all operations from the `Compare` group. See `?S4groupGeneric` in the **methods** package for more information about the `Compare` group generic.

Details

Three forms of 'Compare' operations involving NaArray objects are supported:

1. Between an NaArray object `naa` and a single value `y`:

   ```
   naa op y
   y op naa
   ```

2. Between two NaArray objects `naa1` and `naa2` of same dimensions (a.k.a. *conformable arrays*):

   ```
   naa1 op naa2
   ```

3. Between an NaArray object `naa` and an SVT_SparseArray object `svt` of same dimensions (a.k.a. *conformable arrays*):

   ```
   naa op svt
   svt op naa
   ```

Value

An NaArray object of `type()` `"logical"` and same dimensions as the input object(s).

See Also

- `S4groupGeneric` in the **methods** package.
- NaArray objects.
- SVT_SparseArray objects.
- Ordinary array objects in base R.

Examples

```
nam1 <- NaArray(dim=c(15, 6), type="double")
nam1[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <- runif(26)
nam1

nam1 >= 0.2
nam1 != 0

nam2 <- NaArray(dim=dim(nam1), type="integer")
nam2[cbind(1:15, 2)] <- 100:114
nam2[cbind(1:15, 5)] <- -(114:100)
nam2

nam1 < nam2

## Sanity checks:
m1 <- as.matrix(nam1)
m2 <- as.matrix(nam2)
stopifnot(
  identical(as.matrix(nam1 >= 0.2), m1 >= 0.2),
  identical(as.matrix(nam1 != 0), m1 != 0),
  identical(as.matrix(nam1 < nam2), m1 < m2)
)
```

NaArray-Logic-methods *'Logic' operations on NaArray objects*

Description

EXPERIMENTAL!!!

NaArray objects support operations from the `Logic` group (i.e. `&` and `|`), as well as logical negation (`!`). See `?S4groupGeneric` in the **methods** package for more information about the `Logic` group generic.

Note that in base R, `Logic` operations support input of `type()` `"logical"`, `"integer"`, `"double"`, or `"complex"`. However, the corresponding methods for NaArray objects only support objects of `type()` `"logical"` for now.

Details

Three forms of 'Logic' operations involving NaArray objects are supported:

1. Between an NaArray object `naa` and a single logical value `y`:

   ```
   naa op y
   y op naa
   ```

2. Between two NaArray objects `naa1` and `naa2` of same dimensions (a.k.a. *conformable arrays*):

   ```
   naa1 op naa2
   ```

3. Between an NaArray object `naa` and an SVT_SparseArray object `svt` of same dimensions (a.k.a. *conformable arrays*):

   ```
   naa op svt
   svt op naa
   ```

   Note that, in this case, `|` returns an NaArray object but `&` returns an SVT_SparseArray object.

Value

An NaArray object of `type()` `"logical"` and same dimensions as the input object(s), *except* when `&` is used between an NaArray object and an SVT_SparseArray object in which case an SVT_SparseArray object is returned.

See Also

- `S4groupGeneric` in the **methods** package.
- NaArray objects.
- SVT_SparseArray objects.
- Ordinary array objects in base R.

Examples

```
nam1 <- NaArray(dim=c(15, 6))
nam1[cbind(1:15, 2)] <- c(TRUE, FALSE, NA)
nam1[cbind(1:15, 5)] <- c(TRUE, NA, NA, FALSE, TRUE)
nam1

!nam1
nam1 & NA  # replaces all TRUE's with NA's
nam1 | NA  # replaces all FALSE's with NA's

nam2 <- NaArray(dim=dim(nam1))
nam2[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <- c(TRUE, NA)
nam2

nam1 & nam2
nam1 | nam2

## Sanity checks:
m1 <- as.matrix(nam1)
m2 <- as.matrix(nam2)
stopifnot(
  identical(as.matrix(!nam1), !m1),
  identical(as.matrix(nam1 & NA), m1 & NA),
  identical(as.matrix(nam1 | NA), m1 | NA),
  identical(as.matrix(nam1 & nam2), m1 & m2),
  identical(as.matrix(nam1 | nam2), m1 | m2)
)
```

`NaArray-Math-methods` *'Math' and 'Math2' methods for NaArray objects*

Description

EXPERIMENTAL!!!

NaArray objects support all operations from the `Math` and `Math2` groups with a few exceptions. See `?S4groupGeneric` in the **methods** package for more information about the `Math` and `Math2` group generics.

Note that `Math` and `Math2` operations only support NaArray objects of `type()` `"double"` at the moment. NaArray objects of `type()` `"integer"` are not supported yet.

Value

A NaArray derivative of the same dimensions as the input object.

See Also

- S4groupGeneric in the **methods** package.
- NaArray objects.
- Ordinary array objects in base R.

Examples

```
nam <- NaArray(dim=c(15, 6))
nam[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <-
               c(runif(22)*1e4, Inf, -Inf, NA, NaN)

log(nam)
exp(nam)
cos(nam)
lgamma(nam)

## Sanity checks:
m <- as.matrix(nam)
stopifnot(
  suppressWarnings(identical(as.matrix(log(nam)), log(m))),
  identical(as.matrix(exp(nam)), exp(m)),
  suppressWarnings(identical(as.matrix(cos(nam)), cos(m))),
  suppressWarnings(identical(as.matrix(lgamma(nam)), lgamma(m)))
)
```

NaArray-matrixStats *NaArray col/row summarization*

Description

EXPERIMENTAL!!!

The **SparseArray** package provides memory-efficient col/row summarization methods (a.k.a. matrixStats methods) for NaArray objects, like `colSums()`, `rowSums()`, `colMeans()`, `rowMeans()`, etc...

Note that these are *S4 generic functions* defined in the **MatrixGenerics** package, with methods for ordinary matrices defined in the **matrixStats** package. This man page documents the methods defined for NaArray objects.

IMPORTANT NOTE: This is WORK-IN-PROGRESS! All the `col*()` methods listed below are supported. However, among the `row*()` methods, only `rowAnyNAs()`, `rowMins()`, `rowMaxs()`, `rowRanges()`, `rowSums()`, and `rowSums2()` are supported on NaArray objects at the moment.

Usage

```
## N.B.: Showing ONLY the col*() methods (usage of row*() methods is
## the same):

## S4 method for signature 'NaArray'
colAnyNAs(x, rows=NULL, cols=NULL, dims=1, ..., useNames=NA)

## S4 method for signature 'NaArray'
colAnys(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'NaArray'
colAlls(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'NaArray'
colMins(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'NaArray'
colMaxs(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'NaArray'
colRanges(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'NaArray'
colSums(x, na.rm=FALSE, dims=1)

## S4 method for signature 'NaArray'
colProds(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'NaArray'
colMeans(x, na.rm=FALSE, dims=1)

## S4 method for signature 'NaArray'
colSums2(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'NaArray'
colMeans2(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'NaArray'
colVars(x, rows=NULL, cols=NULL, na.rm=FALSE, center=NULL, dims=1,
           ..., useNames=NA)

## S4 method for signature 'NaArray'
colSds(x, rows=NULL, cols=NULL, na.rm=FALSE, center=NULL, dims=1,
          ..., useNames=NA)
```

Arguments

x	An NaMatrix or NaArray object.
rows, cols, ...	Not supported.
na.rm, useNames, center	See man pages of the corresponding generics in the **MatrixGenerics** package (e.g. ?MatrixGenerics::colVars) for a description of these arguments.

Note that, unlike the methods for ordinary matrices defined in the **matrixStats** package, the `center` argument of the `colVars()`, `rowVars()`, `colSds()`, and `rowSds()` methods for SparseArray objects can only be a *single value* (or a `NULL`). In particular, if `x` has more than one column, then `center` cannot be a vector with one value per column in `x`.

`dims` See `?base::colSums` for a description of this argument.

Details

These methods are typically used with `na.rm=TRUE` when called on an NaMatrix or NaArray object.

All these methods operate *natively* on the NaArray internal representation, for maximum efficiency.

Note that more col/row summarization methods might be added in the future.

Value

See man pages of the corresponding generics in the **MatrixGenerics** package (e.g. `?MatrixGenerics::colRanges`) for the value returned by these methods.

Note

Most `col*()` methods for NaArray objects are multithreaded. See `set_SparseArray_nthread` for how to control the number of threads.

See Also

- NaArray objects.
- The man pages of the various generic functions defined in the **MatrixGenerics** package e.g. `MatrixGenerics::colVars` etc...

Examples

```
# COMING SOON...
```

`NaArray-misc-methods` *Miscellaneous operations on a NaArray object*

Description

This man page documents various base array operations that are supported by NaArray derivatives, and that didn't belong to any of the groups of operations documented in the other man pages of the **SparseArray** package.

Usage

```
# --- unary isometric array transformations ---

## S4 method for signature 'NaArray'
is.nan(x)

## S4 method for signature 'NaArray'
is.infinite(x)
```

```
# --- N-ary isometric array transformations ---

# COMING SOON...
```

Arguments

x	An NaArray object.

Details

More operations will be added in the future.

Value

`is.nan()` and `is.infinite()` return a SparseArray object of `type()` `"logical"` and same dimensions as the input object.

See Also

- `base::is.nan` and `base::is.infinite` in base R.
- NaArray objects.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
a <- array(c(NA, 2.77, NaN, Inf, NA, -Inf), dim=5:3)
naa <- NaArray(a)  # NaArray object
naa

is.nan(naa)            # SparseArray object of type "logical"
is.infinite(naa)       # SparseArray object of type "logical"

## Sanity checks:
res <- is.nan(naa)
stopifnot(is(res, "SparseArray"), type(res) == "logical",
          identical(as.array(res), is.nan(a)))
res <- is.infinite(naa)
stopifnot(is(res, "SparseArray"), type(res) == "logical",
          identical(as.array(res), is.infinite(a)))
```

NaArray-subassignment *NaArray subassignment*

Description

Like ordinary arrays in base R, NaArray objects support subassignment via the `[<-` operator.

See Also

- [<- in base R.
- NaArray objects.
- Ordinary array objects in base R.

Examples

```
a <- array(NA, dim=5:3)
a[c(1:2, 8, 10, 15:17, 20, 24, 40, 56:60)] <- 0.01 * (1:15)
naa <- NaArray(a)
naa

naa[ , 1:2, 3] <- - naa[ , 1:2 ,1]

naa[5:3, c(4,2,4), 1:2] <- c(0, NA, 0)

## Sanity checks:
a[ , 1:2, 3] <- -a[ , 1:2 ,1]
a[5:3, c(4,2,4), 1:2] <- c(0, NA, 0)
stopifnot(identical(as.array(naa), a), identical(naa, NaArray(a)))
```

NaArray-subsetting *Subsetting an NaArray object*

Description

EXPERIMENTAL!!!

Like ordinary arrays in base R, NaArray objects support subsetting via the single bracket operator ([).

See Also

- drop in base R to drop the *ineffective dimensions* of an array or array-like object.
- Lindex2Mindex in the **S4Arrays** package for how to convert an *L-index* to an *M-index* and vice-versa.
- NaArray objects.
- [and ordinary array objects in base R.

Examples

```
naa <- NaArray(dim=5:3)
naa[c(1:2, 8, 10, 15:17, 20, 24, 40, 56:60)] <- (1:15)*10L

## ---------------------------------------------------------------------
## N-dimensional subsetting
## ---------------------------------------------------------------------
naa[5:3, c(4,2,4), 2:3]
naa[ , c(4,2,4), 2:3]
naa[ , c(4,2,4), -1]
naa[ , c(4,2,4), 1]
```

```
naa2 <- naa[ , c(4,2,4), 1, drop=FALSE]
naa2

## Ineffective dimensions can always be dropped as a separate step:
drop(naa2)

naa[ , c(4,2,4), integer(0)]

dimnames(naa) <- list(letters[1:5], NULL, LETTERS[1:3])

naa[c("d", "a"), c(4,2,4), "C"]

naa2 <- naa["e", c(4,2,4), , drop=FALSE]
naa2

drop(naa2)

## ---------------------------------------------------------------------
## 1D-style subsetting (a.k.a. linear subsetting)
## ---------------------------------------------------------------------

## Using a numeric vector (L-index):
naa[c(60, 24, 21, 56)]

## Using a matrix subscript (M-index):
m <- rbind(c(5, 4, 3),
           c(4, 1, 2),
           c(1, 1, 2),
           c(1, 4, 3))
naa[m]

## See '?Lindex2Mindex' in the S4Arrays package for how to convert an
## L-index to an M-index and vice-versa.

## ---------------------------------------------------------------------
## Sanity checks
## ---------------------------------------------------------------------
a <- as.array(naa)
naa2 <- naa[5:3, c(4,2,4), 2:3]
a2   <- a  [5:3, c(4,2,4), 2:3]
stopifnot(identical(as.array(naa2), a2), identical(naa2, NaArray(a2)))
naa2 <- naa[ , c(4,2,4), 2:3]
a2   <- a  [ , c(4,2,4), 2:3]
stopifnot(identical(as.array(naa2), a2), identical(naa2, NaArray(a2)))
naa2 <- naa[ , c(4,2,4), -1]
a2   <- a  [ , c(4,2,4), -1]
stopifnot(identical(as.array(naa2), a2), identical(naa2, NaArray(a2)))
naa2 <- naa[ , c(4,2,4), 1]
a2   <- a  [ , c(4,2,4), 1]
stopifnot(identical(as.array(naa2), a2), identical(naa2, NaArray(a2)))
naa2 <- naa[ , c(4,2,4), 1, drop=FALSE]
a2   <- a  [ , c(4,2,4), 1, drop=FALSE]
stopifnot(identical(as.array(naa2), a2), identical(naa2, NaArray(a2)))
naa2 <- drop(naa2)
a2 <- drop(a2)
stopifnot(identical(as.array(naa2), a2), identical(naa2, NaArray(a2)))
naa2 <- naa[ , c(4,2,4), integer(0)]
```

```
a2   <- a  [ , c(4,2,4), integer(0)]
stopifnot(identical(as.array(naa2), a2),
          identical(unname(naa2), unname(NaArray(a2))))
naa2 <- naa[c("d", "a"), c(4,2,4), "C"]
a2   <- a  [c("d", "a"), c(4,2,4), "C"]
stopifnot(identical(as.array(naa2), a2), identical(naa2, NaArray(a2)))
naa2 <- naa["e", c(4,2,4), , drop=FALSE]
a2   <- a  ["e", c(4,2,4), , drop=FALSE]
stopifnot(identical(as.array(naa2), a2), identical(naa2, NaArray(a2)))
naa2 <- drop(naa2)
a2 <- drop(a2)
stopifnot(identical(as.array(naa2), a2), identical(naa2, NaArray(a2)))
stopifnot(identical(naa[c(60, 24, 21, 56)], naa[m]))
```

`NaArray-summarization` *NaArray summarization methods*

Description

EXPERIMENTAL!!!

The **SparseArray** package provides memory-efficient summarization methods for NaArray objects. The following methods are supported at the moment: `anyNA()`, `any()`, `all()`, `min()`, `max()`, `range()`, `sum()`, `prod()`, `mean()`, `var()`, `sd()`.

More might be added in the future.

Note that these are *S4 generic functions* defined in base R and in the **BiocGenerics** package, with default methods defined in base R. This man page documents the methods defined for NaArray objects.

Details

All these methods operate *natively* on the NaArray representation, for maximum efficiency.

Value

See man pages of the corresponding default methods in the **base** package (e.g. `?base::range`, `?base::mean`, etc...) for the value returned by these methods.

See Also

- NaArray objects.
- The man pages of the various default methods defined in the **base** package e.g. `base::range`, `base::mean`, `base::anyNA`, etc...

Examples

```
naa <- NaArray(dim=c(4, 5, 2))
naa[c(1:2, 8, 10, 15:17, 24:26, 28, 40)] <- (1:12)*10L
naa

anyNA(naa)

range(naa, na.rm=TRUE)
```

```
sum(naa, na.rm=TRUE)

sd(naa, na.rm=TRUE)

## Sanity checks:
a0 <- as.array(naa)
stopifnot(
  identical(anyNA(naa), anyNA(a0)),
  identical(range(naa), range(a0)),
  identical(range(naa, na.rm=TRUE), range(a0, na.rm=TRUE)),
  identical(sum(naa), sum(a0)),
  identical(sum(naa, na.rm=TRUE), sum(a0, na.rm=TRUE)),
  all.equal(sd(naa, na.rm=TRUE), sd(a0, na.rm=TRUE))
)
```

`randomSparseArray` *Random SparseArray object*

Description

`randomSparseArray()` and `poissonSparseArray()` can be used to generate a random SparseArray object efficiently.

Usage

```
randomSparseArray(dim, density=0.05, dimnames=NULL)
poissonSparseArray(dim, lambda=-log(0.95), density=NA, dimnames=NULL)

## Convenience wrappers for the 2D case:
randomSparseMatrix(nrow, ncol, density=0.05, dimnames=NULL)
poissonSparseMatrix(nrow, ncol, lambda=-log(0.95), density=NA,
                    dimnames=NULL)
```

Arguments

`dim`	The dimensions (specified as an integer vector) of the SparseArray object to generate.
`density`	The desired density (specified as a number >= 0 and <= 1) of the SparseArray object to generate, that is, the ratio between its number of nonzero elements and its total number of elements. This is `nzcount(x)/length(x)` or `1 - sparsity(x)`. Note that for `poissonSparseArray()` and `poissonSparseMatrix()` `density` must be < 1 and the *actual* density of the returned object won't be exactly as requested but will typically be very close.
`dimnames`	The *dimnames* to put on the object to generate. Must be `NULL` or a list of length the number of dimensions. Each list element must be either `NULL` or a character vector along the corresponding dimension.
`lambda`	The mean of the Poisson distribution. Passed internally to the calls to `rpois()`. Only one of `lambda` and `density` can be specified.

When density is requested, rpois() is called internally with lambda set to -log(1 - density). This is expected to generate Poisson data with the requested density.

Finally note that the default value for lambda corresponds to a requested density of 0.05.

nrow, ncol — Number of rows and columns of the SparseMatrix object to generate.

Details

randomSparseArray() mimics the rsparsematrix() function from the **Matrix** package but returns a SparseArray object instead of a dgCMatrix object.

poissonSparseArray() populates a SparseArray object with Poisson data i.e. it's equivalent to:

```
a <- array(rpois(prod(dim), lambda), dim)
as(a, "SparseArray")
```

but is faster and more memory efficient because intermediate dense array a is never generated.

Value

A SparseArray derivative (of class SVT_SparseArray or SVT_SparseMatrix) with the requested dimensions and density.

The type of the returned object is "double" for randomSparseArray() and randomSparseMatrix(), and "integer" for poissonSparseArray() and poissonSparseMatrix().

Note

Unlike with Matrix::rsparsematrix() there's no limit on the number of nonzero elements that can be contained in the returned SparseArray object.

For example Matrix::rsparsematrix(3e5, 2e4, density=0.5) will fail with an error but randomSparseMatrix(3e5, 2e4, density=0.5) should work (even though it will take some time and the memory footprint of the resulting object will be about 18 Gb).

See Also

- The Matrix::rsparsematrix function in the **Matrix** package.
- The stats::rpois function in the **stats** package.
- SVT_SparseArray objects.

Examples

```
## ---------------------------------------------------------------------
## randomSparseArray() / randomSparseMatrix()
## ---------------------------------------------------------------------
set.seed(123)
dgcm1 <- rsparsematrix(2500, 950, density=0.1)
set.seed(123)
svt1 <- randomSparseMatrix(2500, 950, density=0.1)
svt1
type(svt1)  # "double"

stopifnot(identical(as(svt1, "dgCMatrix"), dgcm1))
```

```
## ------------------------------------------------------------------------
## poissonSparseArray() / poissonSparseMatrix()
## ------------------------------------------------------------------------
svt2 <- poissonSparseMatrix(2500, 950, density=0.1)
svt2
type(svt2)  # "integer"
1 - sparsity(svt2)  # very close to the requested density

set.seed(123)
svt3 <- poissonSparseArray(c(600, 1700, 80), lambda=0.01)
set.seed(123)
a3 <- array(rpois(length(svt3), lambda=0.01), dim(svt3))
stopifnot(identical(svt3, SparseArray(a3)))

## The memory footprint of 'svt3' is 10x smaller than that of 'a3':
object.size(svt3)
object.size(a3)
as.double(object.size(a3) / object.size(svt3))
```

`readSparseCSV` *Read/write a sparse matrix from/to a CSV file*

Description

Read/write a sparse matrix from/to a CSV (comma-separated values) file.

Usage

```
writeSparseCSV(x, filepath, sep=",", transpose=FALSE, write.zeros=FALSE,
                  chunknrow=250)

readSparseCSV(filepath, sep=",", transpose=FALSE)
```

Arguments

`x`	A matrix-like object, typically sparse. IMPORTANT: The object must have rownames and colnames! These will be written to the file. Another requirement is that the object must be subsettable. More precisely: it must support 2D-style subsetting of the kind `x[i, ]` and `x[ , j]` where `i` and `j` are integer vectors of valid row and column indices.
`filepath`	The path (as a single string) to the file where to write the matrix-like object or to read it from. Compressed files are supported. If `""`, `writeSparseCSV()` will write the data to the standard output connection. Note that `filepath` can also be a connection.
`sep`	The field separator character. Values on each line of the file are separated by this character.
`transpose`	`TRUE` or `FALSE`. By default, rows in the matrix-like object correspond to lines in the CSV file. Set `transpose` to `TRUE` to transpose the matrix-like object on-the-fly, that is, to have its columns written to or read from the lines in the CSV file.

Note that using `transpose=TRUE` is semantically equivalent to calling `t()` on the object before writing it or after reading it, but it will tend to be more efficient. Also it will work even if `x` does not support `t()` (not all matrix-like objects are guaranteed to be transposable).

`write.zeros` `TRUE` or `FALSE`. By default, the zero values in `x` are not written to the file. Set `write.zeros` to `TRUE` to write them.

`chunknrow` `writeSparseCSV()` uses a block-processing strategy to try to speed up things. By default blocks of 250 rows (or columns if `transpose=TRUE`) are used. In our experience trying to increase this (e.g. to 500 or more) will generally not produce significant benefits while it will increase memory usage, so use carefully.

Value

`writeSparseCSV` returns an invisible `NULL`.

`readSparseCSV` returns a SparseMatrix object of class SVT_SparseMatrix.

See Also

- SparseArray objects.
- dgCMatrix-class in the **Matrix** package.

Examples

```
## ---------------------------------------------------------------------
## writeSparseCSV()
## ---------------------------------------------------------------------

## Prepare toy matrix 'm0':
rownames0 <- LETTERS[1:6]
colnames0 <- letters[1:4]
m0 <- matrix(0L, nrow=length(rownames0), ncol=length(colnames0),
                 dimnames=list(rownames0, colnames0))
m0[c(1:2, 8, 10, 15:17, 24)] <- (1:8)*10L
m0

## writeSparseCSV():
writeSparseCSV(m0, filepath="", sep="\t")
writeSparseCSV(m0, filepath="", sep="\t", write.zeros=TRUE)
writeSparseCSV(m0, filepath="", sep="\t", transpose=TRUE)

## Note that writeSparseCSV() will automatically (and silently) coerce
## non-integer values to integer by passing them thru as.integer().

## Example where type(x) is "double":
m1 <- m0 * runif(length(m0))
m1
type(m1)
writeSparseCSV(m1, filepath="", sep="\t")

## Example where type(x) is "logical":
writeSparseCSV(m0 != 0, filepath="", sep="\t")

## Example where type(x) is "raw":
m2 <- m0
type(m2) <- "raw"
```

```
m2
writeSparseCSV(m2, filepath="", sep="\t")

## ---------------------------------------------------------------------
## readSparseCSV()
## ---------------------------------------------------------------------

csv_file <- tempfile()
writeSparseCSV(m0, csv_file)

svt1 <- readSparseCSV(csv_file)
svt1

svt2 <- readSparseCSV(csv_file, transpose=TRUE)
svt2

## If you need the sparse data as a dgCMatrix object, just coerce the
## returned object:
as(svt1, "dgCMatrix")
as(svt2, "dgCMatrix")

## Sanity checks:
stopifnot(identical(m0, as.matrix(svt1)))
stopifnot(identical(t(m0), as.matrix(svt2)))
```

`read_block_as_sparse` *read_block_as_sparse*

Description

`read_block_as_sparse()` is an internal generic function used by `S4Arrays::read_block()` when `is_sparse(x)` is `TRUE`.

Usage

```
read_block_as_sparse(x, viewport)

## S4 method for signature 'ANY'
read_block_as_sparse(x, viewport)
```

Arguments

`x`	An array-like object for which `is_sparse(x)` is `TRUE`.
`viewport`	An ArrayViewport object compatible with `x`, that is, such that `refdim(viewport)` is identical to `dim(x)`.

Details

Like `read_block_as_dense()` in the **S4Arrays** package, `read_block_as_sparse()` is not meant to be called directly by the end user. The end user should always call the higher-level user-facing `read_block()` function instead. See `?read_block` in the **S4Arrays** package for more information.

Also, like `extract_sparse_array()`, `read_block_as_sparse()` should *always* be called on an array-like object `x` for which `is_sparse(x)` is `TRUE`.

For maximum efficiency, `read_block_as_sparse()` methods should:

1. NOT check that `is_sparse(x)` is `TRUE`.
2. NOT try to do anything with the dimnames on `x` (`read_block()` takes care of that).
3. always operate natively on the sparse representation of the data in `x`, that is, they should never *expand* it into a dense representation (e.g. with *as.array()*).

Value

A block of data as a SparseArray derivative (COO_SparseArray or SVT_SparseArray) of the same `type()` as `x`.

See Also

- `read_block` in the **S4Arrays** package for the higher-level user-facing function for reading array blocks.
- ArrayGrid in the **S4Arrays** package for ArrayGrid and ArrayViewport objects.
- `is_sparse` in the **S4Arrays** package to check whether an object uses a sparse representation of the data or not.
- SparseArray objects.
- `S4Arrays::type` in the **S4Arrays** package to get the type of the elements of an array-like object.
- `extract_sparse_array` for the workhorse behind the default `read_block_as_sparse()` method.
- dgCMatrix-class in the **Matrix** package.

`rowsum-methods` *rowsum() methods for sparse matrices*

Description

The **SparseArray** package provides memory-efficient `rowsum()` and `colsum()` methods for SparseMatrix and dsparseMatrix derivatives.

Usage

```
## S4 method for signature 'SparseMatrix'
rowsum(x, group, reorder=TRUE, ...)

## S4 method for signature 'dsparseMatrix'
rowsum(x, group, reorder=TRUE, ...)

## S4 method for signature 'SparseMatrix'
colsum(x, group, reorder=TRUE, ...)

## S4 method for signature 'dsparseMatrix'
colsum(x, group, reorder=TRUE, ...)
```

Arguments

x	A SparseMatrix derivative (e.g. SVT_SparseMatrix or COO_SparseMatrix object), or dsparseMatrix derivative (e.g. dgCMatrix or dgTMatrix object).
group, reorder	See ?base::rowsum for a description of these arguments.
...	Like the default S3 rowsum() method defined in the **base** package, the methods documented in this man page support additional argument na.rm, set to FALSE by default. If TRUE, missing values (NA or NaN) are omitted from the calculations.

Value

An *ordinary* matrix, like the default rowsum() method. See ?base::rowsum for how the matrix returned by the default rowsum() method is obtained.

See Also

- rowsum in base R.
- S4Arrays::rowsum in the **S4Arrays** package for the rowsum() and colsum() S4 generic functions.
- SparseMatrix objects.
- dgCMatrix-class in the **Matrix** package.

Examples

```
svt0 <- randomSparseMatrix(7e5, 100, density=0.15)
dgcm0 <- as(svt0, "dgCMatrix")
m0 <- as.matrix(svt0)

group <- sample(10, nrow(m0), replace=TRUE)

## Calling rowsum() on the sparse representations is usually faster
## than on the dense representation:
rs1 <- rowsum(m0, group)
rs2 <- rowsum(svt0, group)   # about 3x faster
rs3 <- rowsum(dgcm0, group)  # also about 3x faster

## Sanity checks:
stopifnot(identical(rs1, rs2), identical(rs1, rs3))
```

SparseArray *SparseArray objects*

Description

The **SparseArray** package defines the SparseArray virtual class whose purpose is to be extended by other S4 classes that aim at representing in-memory multidimensional sparse arrays.

It has currently two concrete subclasses, COO_SparseArray and SVT_SparseArray, both also defined in this package. Each subclass uses its own internal representation for the nonzero multidimensional data, the *COO layout* for COO_SparseArray, and the *SVT layout* for SVT_SparseArray. The two layouts are described in the COO_SparseArray and SVT_SparseArray man pages, respectively.

Finally, the package also defines the SparseMatrix virtual class, as a subclass of the SparseArray class, for the specific 2D case.

Usage

```
## Constructor function:
SparseArray(x, type=NA)
```

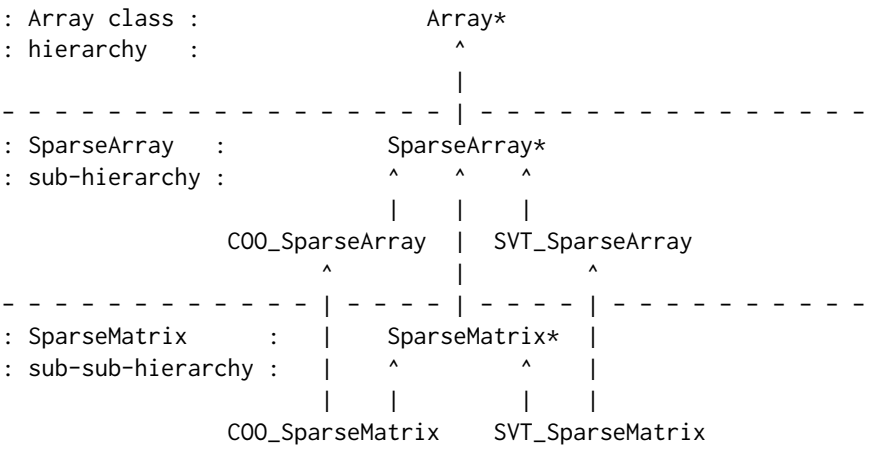
Arguments

x	An ordinary matrix or array, or a dg[C|R]Matrix object, or an lg[C|R]Matrix object, or any matrix-like or array-like object that supports coercion to SVT_SparseArray.

type	A single string specifying the requested type of the object.
	By default, the SparseArray object returned by the constructor function has the same `type()` as `x`. However the user can use the `type` argument to request a different type. Note that doing:
	`sa <- SparseArray(x, type=type)`
	is equivalent to doing:
	`sa <- SparseArray(x)`
	`type(sa) <- type`
	but the former is more convenient and will generally be more efficient.
	Supported types are all R atomic types plus `"list"`.

Details

The SparseArray class extends the Array virtual class defined in the **S4Arrays** package. Here is the full SparseArray sub-hierarchy as defined in the **SparseArray** package (virtual classes are marked with an asterisk):



Any object that belongs to a class that extends SparseArray e.g. (a SVT_SparseArray or SVT_SparseMatrix object) is called a *SparseArray derivative*.

Most of the *matrix and array standard API* defined in base R should work on SparseArray derivatives, including `dim()`, `length()`, `dimnames()`, `` `dimnames<-`() ``, `[`, `drop()`, `` `[<-` `` (subassignment), `t()`, `rbind()`, `cbind()`, etc...

SparseArray derivatives also support `type()`, `` `type<-`() ``, `is_sparse()`, `is_nonzero()`, `nzcount()`, `nzwhich()`, `nzvals()`, `` `nzvals<-`() ``, `sparsity()`, `arbind()`, and `acbind()`.

Value

A *SparseArray derivative*, that is a SVT_SparseArray, COO_SparseArray, SVT_SparseMatrix, or COO_SparseMatrix object.

The `type()` of the input object is preserved, except if a different one was requested via the `type` argument.

What is considered a zero depends on the `type()`:

- `"logical"` zero is `FALSE`;
- `"integer"` zero is `0L`;
- `"double"` zero is `0`;
- `"complex"` zero is `0+0i`;
- `"raw"` zero is `raw(1)`;
- `"character"` zero is `""` (empty string);
- `"list"` zero is `NULL`.

See Also

- The COO_SparseArray and SVT_SparseArray classes.
- is_nonzero for `is_nonzero()` and `nz*()` functions `nzcount()`, `nzwhich()`, etc...
- SparseArray_aperm for permuting the dimensions of a SparseArray object (e.g. transposition).
- SparseArray_subsetting for subsetting a SparseArray object.
- SparseArray_subassignment for SparseArray subassignment.
- SparseArray_abind for combining 2D or multidimensional SparseArray objects.
- SparseArray_summarization for SparseArray summarization methods.
- SparseArray_Arith, SparseArray_Compare, and SparseArray_Logic, for operations from the `Arith`, `Compare`, and `Logic` groups on SparseArray objects.
- SparseArray_Math for operations from the `Math` and `Math2` groups on SparseArray objects.
- SparseArray_Complex for operations from the `Complex` group on SparseArray objects.
- SparseArray_misc for miscellaneous operations on a SparseArray object.
- SparseArray_matrixStats for col/row summarization methods for SparseArray objects.
- rowsum_methods for `rowsum()` methods for sparse matrices.
- SparseMatrix_mult for SparseMatrix multiplication and cross-product.
- `randomSparseArray` to generate a random SparseArray object.
- `readSparseCSV` to read/write a sparse matrix from/to a CSV (comma-separated values) file.
- dgCMatrix-class, dgRMatrix-class, and lgCMatrix-class in the **Matrix** package, for the de facto standard for sparse matrix representations in the R ecosystem.
- `is_sparse` in the **S4Arrays** package.
- The Array class defined in the **S4Arrays** package.
- Ordinary array objects in base R.
- `base::which` in base R.

Examples

```
## ---------------------------------------------------------------------
## Display details of class definition & known subclasses
## ---------------------------------------------------------------------

showClass("SparseArray")

## ---------------------------------------------------------------------
## The SparseArray() constructor
## ---------------------------------------------------------------------

a <- array(rpois(9e6, lambda=0.3), dim=c(500, 3000, 6))
SparseArray(a)    # an SVT_SparseArray object

m <- matrix(rpois(9e6, lambda=0.3), ncol=500)
SparseArray(m)    # an SVT_SparseMatrix object

dgc <- sparseMatrix(i=c(4:1, 2:4, 9:12, 11:9), j=c(1:7, 1:7),
                    x=runif(14), dims=c(12, 7))
class(dgc)
SparseArray(dgc)  # an SVT_SparseMatrix object

dgr <- as(dgc, "RsparseMatrix")
class(dgr)
SparseArray(dgr)  # a COO_SparseMatrix object

## ---------------------------------------------------------------------
## nzcount(), nzwhich(), nzvals(), `nzvals<-`()
## ---------------------------------------------------------------------
x <- SparseArray(a)

## Get the number of nonzero array elements in 'x':
nzcount(x)

## nzwhich() returns the indices of the nonzero array elements in 'x'.
## Either as an integer (or numeric) vector of length 'nzcount(x)'
## containing "linear indices":
nzidx <- nzwhich(x)
length(nzidx)
head(nzidx)

## Or as an integer matrix with 'nzcount(x)' rows and one column per
## dimension where the rows represent "array indices" (a.k.a. "array
## coordinates"):
Mnzidx <- nzwhich(x, arr.ind=TRUE)
dim(Mnzidx)

## Each row in the matrix is an n-tuple representing the "array
## coordinates" of a nonzero element in 'x':
head(Mnzidx)
tail(Mnzidx)

## Extract the values of the nonzero array elements in 'x' and return
## them in a vector "parallel" to 'nzwhich(x)':
x_nzvals <- nzvals(x)  # equivalent to 'x[nzwhich(x)]'
length(x_nzvals)
```

```
head(x_nzvals)

nzvals(x) <- log1p(nzvals(x))
x

## Sanity checks:
stopifnot(identical(nzidx, which(a != 0)))
stopifnot(identical(Mnzidx, which(a != 0, arr.ind=TRUE, useNames=FALSE)))
stopifnot(identical(x_nzvals, a[nzidx]))
stopifnot(identical(x_nzvals, a[Mnzidx]))
stopifnot(identical(`nzvals<-`(x, nzvals(x)), x))
```

SparseArray-abind *Bind SparseArray objects with abind()*

Description

Like ordinary matrices and arrays in base R, SparseMatrix derivatives can be combined by rows or columns, with `rbind()` or `cbind()`, and multidimensional SparseArray derivatives can be bound along any dimension with `abind()`.

Note that `arbind()` can also be used to combine multidimensional array-like objects along their first dimension, and `acbind()` can be used to combine them along their second dimension.

See Also

- `cbind` in base R.
- `abind` in the **S4Arrays** package.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## Binding SparseMatrix objects with rbind() or cbind()
## ---------------------------------------------------------------------

m1 <- matrix(1:8, ncol=4,
             dimnames=list(NULL, paste0("M1y", 1:4)))
m2 <- matrix(101:120, ncol=4,
             dimnames=list(paste0("M2x", 1:5), NULL))
sm1 <- SparseArray(m1)
sm2 <- SparseArray(m2)

rbind(sm1, sm2)

m3 <- matrix(0.5 - 1:28, ncol=4,
             dimnames=list(paste0("M3x", 1:7), paste0("M3y", 1:4)))
sm3 <- SparseArray(m3)

rbind(sm2, sm3, sm1)

cbind(sm3, rbind(sm2, sm1))
```

```
## ---------------------------------------------------------------------
## Binding 3D SparseArray objects with abind()
## ---------------------------------------------------------------------

a1 <- array(1:105, dim=c(5, 7, 3),
            dimnames=list(NULL, paste0("A1y", 1:7), NULL))
a2 <- array(1001:1105, dim=c(5, 7, 3),
            dimnames=list(paste0("A2x", 1:5), paste0("A2y", 1:7), NULL))
sa1 <- SparseArray(a1)
sa2 <- SparseArray(a2)

abind(sa1, sa2)               # same as 'abind(sa1, sa2, along=3)'
abind(sa1, sa2, rev.along=0)  # same as 'abind(sa1, sa2, along=4)'

a3 <- array(1:60, dim=c(3, 5, 4),
            dimnames=list(NULL, paste0("A3y", 1:5), NULL))
a4 <- array(101:240, dim=c(7, 5, 4),
            dimnames=list(paste0("A4x", 1:7), paste0("A4y", 1:5), NULL))
sa3 <- SparseArray(a3)
sa4 <- SparseArray(a4)

arbind(sa3, sa4)  # same as 'abind(sa3, sa4, along=1)'

## ---------------------------------------------------------------------
## Sanity checks
## ---------------------------------------------------------------------

stopifnot(
    identical(as.matrix(rbind(sm1, sm2)),
              rbind(m1, m2)),
    identical(as.matrix(rbind(sm2, sm3, sm1)),
              rbind(m2, m3, m1)),
    identical(as.matrix(cbind(sm3, rbind(sm2, sm1))),
              cbind(m3, rbind(m2, m1))),
    identical(as.array(abind(sa1, sa2)),
              abind(a1, a2)),
    identical(as.array(abind(sa1, sa2, rev.along=0)),
              abind(a1, a2, rev.along=0)),
    identical(as.array(arbind(sa3, sa4)),
              arbind(a3, a4))
)
```

SparseArray-aperm *SparseArray transposition*

Description

EXPERIMENTAL!!!

Transpose a SparseArray object by permuting its dimensions.

Value

COMING SOON...

See Also

- aperm() in base R.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
# COMING SOON...
```

SparseArray-Arith-methods

'Arith' operations on SparseArray objects

Description

SparseArray derivatives support operations from the Arith group, with some restrictions. See ?S4groupGeneric in the **methods** package for more information about the Arith group generic.

IMPORTANT NOTES:

- Only SVT_SparseArray objects are supported at the moment. Support for COO_SparseArray objects might be added in the future.
- SVT_SparseArray of type() "complex" don't support Arith operations at the moment.

Details

Two forms of 'Arith' operations are supported:

1. Between an SVT_SparseArray object svt and an atomic vector y:

   ```
   svt op y
   y op svt
   ```

   The Arith operations that support this form are: *, /, ^, %%,%/%. Note that, except for * (for which both svt * y and y * svt are supported), atomic vector y must be on the right e.g. svt ^ 3.

2. Between two SVT_SparseArray objects svt1 and svt2 of same dimensions (a.k.a. *conformable arrays*):

   ```
   svt1 op svt2
   ```

   The Arith operations that support this form are: +, -, *.

Value

A SparseArray derivative of the same dimensions as the input object(s).

See Also

- S4groupGeneric in the **methods** package.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## Basic examples
## ---------------------------------------------------------------------

svt1 <- SVT_SparseArray(dim=c(15, 6), type="integer")
svt1[cbind(1:15, 2)] <- 100:114
svt1[cbind(1:15, 5)] <- -(114:100)
svt1

svt1 * -0.01

svt1 * 10   # result is of type "double"
svt1 * 10L  # result is of type "integer"

svt1 * c(10, 5, 0.1)  # right vector recycled along 1st dimension

svt1 / 10L

svt1 ^ 2
svt1 ^ (2:6)

svt1 %% 5L
svt1 %/% 5L

svt2 <- SVT_SparseArray(dim=dim(svt1), type="double")
svt2[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <- runif(26)
svt2

svt1 + svt2
svt1 - svt2
svt1 * svt2

svt2 * (0.1 * svt1 - svt2 ^ 2) + svt1 / sum(svt2)

## Sanity checks:
m1 <- as.matrix(svt1)
m2 <- as.matrix(svt2)
stopifnot(
  identical(as.matrix(svt1 * -0.01), m1 * -0.01),
  identical(as.matrix(svt1 * 10), m1 * 10),
  identical(as.matrix(svt1 * 10L), m1 * 10L),
  identical(as.matrix(svt1 / 10L), m1 / 10L),
  identical(as.matrix(svt1 ^ 3.5), m1 ^ 3.5),
  identical(as.matrix(svt1 %% 5L), m1 %% 5L),
  identical(as.matrix(svt1 %/% 5L), m1 %/% 5L),
  identical(as.matrix(svt1 + svt2), m1 + m2),
  identical(as.matrix(svt1 - svt2), m1 - m2),
  identical(as.matrix(svt1 * svt2), m1 * m2),
  all.equal(as.matrix(svt2 * (0.1 * svt1 - svt2 ^ 2) + svt1 / sum(svt2)),
            m2 * (0.1 * m1 - m2 ^ 2) + m1 / sum(m2))
)

## ---------------------------------------------------------------------
## An example combining operations from the 'Arith', 'Compare',
## and 'Logic' groups
```

```
## ------------------------------------------------------------------

m3 <- matrix(0L, nrow=15, ncol=6)
m3[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <- 101:126
svt3 <- SparseArray(m3)

## Can be 5x or 10x faster than with a dgCMatrix object on a big
## SVT_SparseMatrix object!
svt4 <- (svt3^1.5 + svt3) %% 100 - 0.2 * svt3 > 0
svt4

## Sanity check:
m4 <- (m3^1.5 + m3) %% 100 - 0.2 * m3 > 0
stopifnot(identical(as.matrix(svt4), m4))
```

`SparseArray-Compare-methods`
'Compare' operations on SparseArray objects

Description

SparseArray derivatives support operations from the `Compare` group, with some restrictions. See `?S4groupGeneric` in the **methods** package for more information about the `Compare` group generic.

IMPORTANT NOTE: Only SVT_SparseArray objects are supported at the moment. Support for COO_SparseArray objects might be added in the future.

Details

Two forms of 'Compare' operations are supported:

1. Between an SVT_SparseArray object `svt` and a single value `y`:

   ```
   svt op y
   y op svt
   ```

   All operations from the `Compare` group support this form, with single value `y` either on the left or the right. However, there are some operation-dependent restrictions on the value of `y`.

2. Between two SVT_SparseArray objects `svt1` and `svt2` of same dimensions (a.k.a. *conformable arrays*):

   ```
   svt1 op svt2
   ```

   The `Compare` operations that support this form are: `!=`, `<`, `>`.

Value

A SparseArray derivative of `type()` `"logical"` and same dimensions as the input object(s).

See Also

- `S4groupGeneric` in the **methods** package.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
svt1 <- SVT_SparseArray(dim=c(15, 6), type="double")
svt1[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <- runif(26)
svt1

svt1 >= 0.2
svt1 != 0

## Sanity checks:
m1 <- as.matrix(svt1)
stopifnot(
  identical(as.matrix(svt1 >= 0.2), m1 >= 0.2),
  identical(as.matrix(svt1 != 0), m1 != 0)
)
```

SparseArray-Complex-methods

'Complex' methods for SparseArray objects

Description

WORK-IN-PROGRESS

Value

COMING SOON...

See Also

- S4groupGeneric in the **methods** package.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
# COMING SOON...
```

SparseArray-dim-tuning

Internal "dim tuning" utilities for SparseArray objects

Description

Internal "dim tuning" utilities for SparseArray objects not meant to be used directly by the end user.

SparseArray-Logic-methods
'Logic' operations on SparseArray objects

Description

SparseArray derivatives support operations from the `Logic` group (i.e. `&` and `|`), with some restrictions. See `?S4groupGeneric` in the **methods** package for more information about the `Logic` group generic.

IMPORTANT NOTES:

- Only SVT_SparseArray objects are supported at the moment. Support for COO_SparseArray objects might be added in the future.
- In base R, `Logic` operations support input of `type()` `"logical"`, `"integer"`, `"double"`, or `"complex"`. However, the corresponding methods for SVT_SparseArray objects only support objects of `type()` `"logical"` for now.

Value

A SparseArray derivative of `type()` `"logical"` and same dimensions as the input object(s).

See Also

- `S4groupGeneric` in the **methods** package.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
svt1 <- svt2 <- SVT_SparseArray(dim=c(15, 6))
svt1[cbind(1:15, 2)] <- c(TRUE, FALSE, NA)
svt1[cbind(1:15, 5)] <- c(TRUE, NA, NA, FALSE, TRUE)
svt2[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <- c(TRUE, NA)

svt1 & svt2
svt1 | svt2

## Sanity checks:
m1 <- as.matrix(svt1)
m2 <- as.matrix(svt2)
stopifnot(
  identical(as.matrix(svt1 & svt2), m1 & m2),
  identical(as.matrix(svt1 | svt2), m1 | m2)
)
```

SparseArray-Math-methods *'Math' and 'Math2' methods for SparseArray objects*

Description

SparseArray derivatives support a *subset* of operations from the `Math` and `Math2` groups. See `?S4groupGeneric` in the **methods** package for more information about the `Math` and `Math2` group generics.

IMPORTANT NOTES:

- Only operations from these groups that preserve sparsity are supported. For example, `sqrt()`, `trunc()`, `log1p()`, and `sin()` are supported, but `cumsum()`, `log()`, `cos()`, or `gamma()` are not.
- Only SVT_SparseArray objects are supported at the moment. Support for COO_SparseArray objects might be added in the future.
- `Math` and `Math2` operations only support SVT_SparseArray objects of `type()` `"double"` at the moment.

Value

A SparseArray derivative of the same dimensions as the input object.

See Also

- `S4groupGeneric` in the **methods** package.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
m <- matrix(0, nrow=15, ncol=6)
m[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <-
    c(runif(22)*1e4, Inf, -Inf, NA, NaN)
svt <- SparseArray(m)

svt2 <- trunc(sqrt(svt))
svt2

## Sanity check:
m2 <- suppressWarnings(trunc(sqrt(m)))
stopifnot(identical(as.matrix(svt2), m2))
```

SparseArray-matrixStats

SparseArray col/row summarization

Description

The **SparseArray** package provides memory-efficient col/row summarization methods (a.k.a. matrixStats methods) for SparseArray objects, like `colSums()`, `rowSums()`, `colMedians()`, `rowMedians()`, `colVars()`, `rowVars()`, etc...

Note that these are *S4 generic functions* defined in the **MatrixGenerics** package, with methods for ordinary matrices defined in the **matrixStats** package. This man page documents the methods defined for SparseArray objects.

Usage

```
## N.B.: Showing ONLY the col*() methods (usage of row*() methods is
## the same):

## S4 method for signature 'SparseArray'
colAnyNAs(x, rows=NULL, cols=NULL, dims=1, ..., useNames=NA)

## S4 method for signature 'SparseArray'
colAnys(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'SparseArray'
colAlls(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'SparseArray'
colMins(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'SparseArray'
colMaxs(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'SparseArray'
colRanges(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'SparseArray'
colSums(x, na.rm=FALSE, dims=1)

## S4 method for signature 'SparseArray'
colProds(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'SparseArray'
colMeans(x, na.rm=FALSE, dims=1)

## S4 method for signature 'SparseArray'
colSums2(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'SparseArray'
colMeans2(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)
```

```
## S4 method for signature 'SparseArray'
colVars(x, rows=NULL, cols=NULL, na.rm=FALSE, center=NULL, dims=1,
           ..., useNames=NA)

## S4 method for signature 'SparseArray'
colSds(x, rows=NULL, cols=NULL, na.rm=FALSE, center=NULL, dims=1,
          ..., useNames=NA)

## S4 method for signature 'SparseArray'
colMedians(x, rows=NULL, cols=NULL, na.rm=FALSE, ..., useNames=NA)
```

Arguments

x	A SparseMatrix or SparseArray object. Note that the `colMedians()` and `rowMedians()` methods only support 2D objects (i.e. SparseMatrix objects) at the moment.
rows, cols, ...	Not supported.
na.rm, useNames, center	See man pages of the corresponding generics in the **MatrixGenerics** package (e.g. `?MatrixGenerics::colVars`) for a description of these arguments. Note that, unlike the methods for ordinary matrices defined in the **matrixStats** package, the `center` argument of the `colVars()`, `rowVars()`, `colSds()`, and `rowSds()` methods for SparseArray objects can only be a *single value* (or a `NULL`). In particular, if `x` has more than one column, then `center` cannot be a vector with one value per column in `x`.
dims	See `?base::colSums` for a description of this argument. Note that all the methods above support it, except `colMedians()` and `rowMedians()`.

Details

All these methods operate *natively* on the SVT_SparseArray internal representation, for maximum efficiency.

Note that more col/row summarization methods might be added in the future.

Value

See man pages of the corresponding generics in the **MatrixGenerics** package (e.g. `?MatrixGenerics::colRanges`) for the value returned by these methods.

Note

Most `col*()` methods for SparseArray objects are multithreaded. See `set_SparseArray_nthread` for how to control the number of threads.

See Also

- SparseArray objects.
- The man pages of the various generic functions defined in the **MatrixGenerics** package e.g. `MatrixGenerics::colVars` etc...

Examples

```
## ---------------------------------------------------------------------
## 2D CASE
## ---------------------------------------------------------------------
m0 <- matrix(0L, nrow=6, ncol=4, dimnames=list(letters[1:6], LETTERS[1:4]))
m0[c(1:2, 8, 10, 15:17, 24)] <- (1:8)*10L
m0["e", "B"] <- NA
svt0 <- SparseArray(m0)
svt0

colSums(svt0)
colSums(svt0, na.rm=TRUE)

rowSums(svt0)
rowSums(svt0, na.rm=TRUE)

colMeans(svt0)
colMeans(svt0, na.rm=TRUE)

colRanges(svt0)
colRanges(svt0, useNames=FALSE)
colRanges(svt0, na.rm=TRUE)
colRanges(svt0, na.rm=TRUE, useNames=FALSE)

colVars(svt0)
colVars(svt0, useNames=FALSE)

## Sanity checks:
stopifnot(
  identical(colSums(svt0), colSums(m0)),
  identical(colSums(svt0, na.rm=TRUE), colSums(m0, na.rm=TRUE)),
  identical(rowSums(svt0), rowSums(m0)),
  identical(rowSums(svt0, na.rm=TRUE), rowSums(m0, na.rm=TRUE)),
  identical(colMeans(svt0), colMeans(m0)),
  identical(colMeans(svt0, na.rm=TRUE), colMeans(m0, na.rm=TRUE)),
  identical(colRanges(svt0), colRanges(m0, useNames=TRUE)),
  identical(colRanges(svt0, useNames=FALSE), colRanges(m0, useNames=FALSE)),
  identical(colRanges(svt0, na.rm=TRUE),
            colRanges(m0, na.rm=TRUE, useNames=TRUE)),
  identical(colVars(svt0), colVars(m0, useNames=TRUE)),
  identical(colVars(svt0, na.rm=TRUE),
            colVars(m0, na.rm=TRUE, useNames=TRUE))
)

## ---------------------------------------------------------------------
## 3D CASE (AND ARBITRARY NUMBER OF DIMENSIONS)
## ---------------------------------------------------------------------
set.seed(2009)
svt <- 6L * (poissonSparseArray(5:3, density=0.35) -
             poissonSparseArray(5:3, density=0.35))
dimnames(svt) <- list(NULL, letters[1:4], LETTERS[1:3])

cs1 <- colSums(svt)
cs1  # cs1[j , k] is equal to sum(svt[ , j, k])

cs2 <- colSums(svt, dims=2)
```

```
cs2  # cv2[k] is equal to sum(svt[ , , k])

cv1 <- colVars(svt)
cv1  # cv1[j , k] is equal to var(svt[ , j, k])

cv2 <- colVars(svt, dims=2)
cv2  # cv2[k] is equal to var(svt[ , , k])

## Sanity checks:
k_idx <- setNames(seq_len(dim(svt)[3]), dimnames(svt)[[3]])
j_idx <- setNames(seq_len(dim(svt)[2]), dimnames(svt)[[2]])
cv1b <- sapply(k_idx, function(k)
                      sapply(j_idx, function(j) var(svt[ , j, k, drop=FALSE])))
cv2b <- sapply(k_idx, function(k) var(svt[ , , k]))
stopifnot(
  identical(colSums(svt), colSums(as.array(svt))),
  identical(colSums(svt, dims=2), colSums(as.array(svt), dims=2)),
  identical(cv1, cv1b),
  identical(cv2, cv2b)
)
```

SparseArray-misc-methods

Miscellaneous operations on a SparseArray object

Description

This man page documents various base array operations that are supported by SparseArray derivatives, and that didn't belong to any of the groups of operations documented in the other man pages of the **SparseArray** package.

Usage

```
# --- unary isometric array transformations ---

## S4 method for signature 'COO_SparseArray'
is.na(x)
## S4 method for signature 'SVT_SparseArray'
is.na(x)

## S4 method for signature 'COO_SparseArray'
is.nan(x)
## S4 method for signature 'SVT_SparseArray'
is.nan(x)

## S4 method for signature 'COO_SparseArray'
is.infinite(x)
## S4 method for signature 'SVT_SparseArray'
is.infinite(x)

## S4 method for signature 'COO_SparseArray'
tolower(x)
```

```
## S4 method for signature 'COO_SparseArray'
toupper(x)

## S4 method for signature 'COO_SparseArray'
nchar(x, type="chars", allowNA=FALSE, keepNA=NA)

# --- N-ary isometric array transformations ---

## S4 method for signature 'SparseArray'
pmin(..., na.rm=FALSE)
## S4 method for signature 'SparseArray'
pmax(..., na.rm=FALSE)
```

Arguments

x	A SparseArray derivative.
type, allowNA, keepNA	See ?base::nchar for a description of these arguments.
...	SparseArray derivatives.
na.rm	See ?base::pmin for a description of this argument.

Details

More operations will be added in the future.

Value

See man pages of the corresponding base functions (e.g. ?base::is.na, ?base::nchar, ?base::pmin, etc...) for the value returned by these methods.

Note that, like the base functions, the methods documented in this man page are *endomorphisms* i.e. they return an array-like object of the same class as the input.

See Also

- base::is.na and base::is.infinite in base R.
- base::tolower in base R.
- base::nchar in base R.
- base::pmin in base R.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
a <- array(c(0, 2.77, NA, 0, NaN, -Inf), dim=5:3)
svt <- SparseArray(a)  # SVT_SparseArray object
class(svt)

is.na(svt)             # SVT_SparseArray object of type "logical"
is.nan(svt)            # SVT_SparseArray object of type "logical"
is.infinite(svt)       # SVT_SparseArray object of type "logical"
```

```
svt1 <- poissonSparseMatrix(500, 20, density=0.2)
svt2 <- poissonSparseMatrix(500, 20, density=0.25) * 0.77
pmin(svt1, svt2)
pmax(svt1, svt2)

## Sanity checks:
res <- is.na(svt)
stopifnot(is(res, "SVT_SparseArray"), type(res) == "logical",
          identical(as.array(res), is.na(a)))
res <- is.nan(svt)
stopifnot(is(res, "SVT_SparseArray"), type(res) == "logical",
          identical(as.array(res), is.nan(a)))
res <- is.infinite(svt)
stopifnot(is(res, "SVT_SparseArray"), type(res) == "logical",
          identical(as.array(res), is.infinite(a)))
res <- pmin(svt1, svt2)
stopifnot(is(res, "SVT_SparseArray"),
          identical(as.array(res), pmin(as.array(svt1), as.array(svt2))))
res <- pmax(svt1, svt2)
stopifnot(is(res, "SVT_SparseArray"),
          identical(as.array(res), pmax(as.array(svt1), as.array(svt2))))
```

`SparseArray-subassignment`

SparseArray subassignment

Description

Like ordinary arrays in base R, SparseArray derivatives support subassignment via the `[<-` operator.

See Also

- `[<-` in base R.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
a <- array(0L, dim=5:3)
a[c(1:2, 8, 10, 15:17, 20, 24, 40, 56:60)] <- (1:15)*10L
svt <- SparseArray(a)
svt

svt[ , 1:2, 3] <- - svt[ , 1:2 ,1]

svt[5:3, c(4,2,4), 1:2] <- 999L

## Sanity checks:
a[ , 1:2, 3] <- -a[ , 1:2 ,1]
a[5:3, c(4,2,4), 1:2] <- 999L
stopifnot(identical(as.array(svt), a), identical(svt, SparseArray(a)))
```

SparseArray-subsetting

Subsetting a SparseArray object

Description

Like ordinary arrays in base R, SparseArray derivatives support subsetting via the single bracket operator (`[`).

See Also

- `drop` in base R to drop the *ineffective dimensions* of an array or array-like object.
- `Lindex2Mindex` in the **S4Arrays** package for how to convert an *L-index* to an *M-index* and vice-versa.
- SparseArray objects.
- `[` and ordinary array objects in base R.

Examples

```
a <- array(0L, dim=5:3)
a[c(1:2, 8, 10, 15:17, 20, 24, 40, 56:60)] <- (1:15)*10L
svt <- SparseArray(a)
svt

## ---------------------------------------------------------------------
## N-dimensional subsetting
## ---------------------------------------------------------------------
svt[5:3, c(4,2,4), 2:3]
svt[ , c(4,2,4), 2:3]
svt[ , c(4,2,4), -1]
svt[ , c(4,2,4), 1]

svt2 <- svt[ , c(4,2,4), 1, drop=FALSE]
svt2

## Ineffective dimensions can always be dropped as a separate step:
drop(svt2)

svt[ , c(4,2,4), integer(0)]

dimnames(a) <- list(letters[1:5], NULL, LETTERS[1:3])
svt <- SparseArray(a)

svt[c("d", "a"), c(4,2,4), "C"]

svt2 <- svt["e", c(4,2,4), , drop=FALSE]
svt2

drop(svt2)

## ---------------------------------------------------------------------
## 1D-style subsetting (a.k.a. linear subsetting)
## ---------------------------------------------------------------------
```

```
## Using a numeric vector (L-index):
svt[c(60, 24, 21, 56)]

## Using a matrix subscript (M-index):
m <- rbind(c(5, 4, 3),
           c(4, 1, 2),
           c(1, 1, 2),
           c(1, 4, 3))
svt[m]

## See '?Lindex2Mindex' in the S4Arrays package for how to convert an
## L-index to an M-index and vice-versa.

## ---------------------------------------------------------------------
## Sanity checks
## ---------------------------------------------------------------------
svt2 <- svt[5:3, c(4,2,4), 2:3]
a2   <- a  [5:3, c(4,2,4), 2:3]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt[ , c(4,2,4), 2:3]
a2   <- a  [ , c(4,2,4), 2:3]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt[ , c(4,2,4), -1]
a2   <- a  [ , c(4,2,4), -1]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt[ , c(4,2,4), 1]
a2   <- a  [ , c(4,2,4), 1]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt[ , c(4,2,4), 1, drop=FALSE]
a2   <- a  [ , c(4,2,4), 1, drop=FALSE]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- drop(svt2)
a2 <- drop(a2)
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt[ , c(4,2,4), integer(0)]
a2   <- a  [ , c(4,2,4), integer(0)]
stopifnot(identical(as.array(svt2), a2),
          identical(unname(svt2), unname(SparseArray(a2))))
svt2 <- svt[c("d", "a"), c(4,2,4), "C"]
a2   <- a  [c("d", "a"), c(4,2,4), "C"]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt["e", c(4,2,4), , drop=FALSE]
a2   <- a  ["e", c(4,2,4), , drop=FALSE]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- drop(svt2)
a2 <- drop(a2)
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
stopifnot(identical(svt[c(60, 24, 21, 56)], svt[m]))
```

SparseArray-summarization

SparseArray summarization methods

Description

The **SparseArray** package provides memory-efficient summarization methods for SparseArray objects. The following methods are supported at the moment: `anyNA()`, `any()`, `all()`, `min()`, `max()`, `range()`, `sum()`, `prod()`, `mean()`, `var()`, `sd()`.

More might be added in the future.

Note that these are *S4 generic functions* defined in base R and in the **BiocGenerics** package, with default methods defined in base R. This man page documents the methods defined for SparseArray objects.

Details

All these methods operate *natively* on the COO_SparseArray or SVT_SparseArray representation, for maximum efficiency.

Value

See man pages of the corresponding default methods in the **base** package (e.g. `?base::range`, `?base::mean`, etc...) for the value returned by these methods.

See Also

- SparseArray objects.
- The man pages of the various default methods defined in the **base** package e.g. `base::range`, `base::mean`, `base::anyNA`, etc...

Examples

```
svt0 <- SVT_SparseArray(dim=c(4, 5, 2))
svt0[c(1:2, 8, 10, 15:17, 24:26, 28, 40)] <- (1:12)*10L
svt0[4, 3, 1] <- NA
svt0

anyNA(svt0)

range(svt0)

range(svt0, na.rm=TRUE)

sum(svt0, na.rm=TRUE)

sd(svt0, na.rm=TRUE)

## Sanity checks:
a0 <- as.array(svt0)
stopifnot(
  identical(anyNA(svt0), anyNA(a0)),
  identical(range(svt0), range(a0)),
  identical(range(svt0, na.rm=TRUE), range(a0, na.rm=TRUE)),
  identical(sum(svt0), sum(a0)),
  identical(sum(svt0, na.rm=TRUE), sum(a0, na.rm=TRUE)),
  all.equal(sd(svt0, na.rm=TRUE), sd(a0, na.rm=TRUE))
)
```

SparseMatrix-mult *SparseMatrix multiplication and cross-product*

Description

Like ordinary matrices in base R, SparseMatrix derivatives can be multiplied with the `%*%` operator. They also support `crossprod()` and `tcrossprod()`.

Value

The `%*%`, `crossprod()` and `tcrossprod()` methods for SparseMatrix objects always return an *ordinary* matrix of `type()` `"double"`.

Note

Matrix multiplication and cross-product of SparseMatrix derivatives are multithreaded. See `set_SparseArray_nthread` for how to control the number of threads.

See Also

- `%*%` and `crossprod` in base R.
- SparseMatrix objects.
- `S4Arrays::type` in the **S4Arrays** package to get the type of the elements of an array-like object.
- Ordinary matrix objects in base R.

Examples

```
m1 <- matrix(0L, nrow=15, ncol=6)
m1[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <- 101:126
svt1 <- as(m1, "SVT_SparseMatrix")

set.seed(333)
svt2 <- poissonSparseMatrix(nrow=6, ncol=7, density=0.2)

svt1 %*% svt2
m1   %*% svt2

## Unary crossprod() and tcrossprod():
crossprod(svt1)               # same as t(svt1) %*% svt1
tcrossprod(svt1)              # same as svt1 %*% t(svt1)

## Binary crossprod() and tcrossprod():
crossprod(svt1[1:6, ], svt2)  # same as t(svt1[1:6, ]) %*% svt2
tcrossprod(svt1, t(svt2))     # same as svt1 %*% svt2

## Sanity checks:
m12 <- m1 %*% as.matrix(svt2)
stopifnot(
  identical(svt1 %*% svt2, m12),
  identical(m1 %*% svt2, m12),
  identical(crossprod(svt1), t(svt1) %*% svt1),
```

```
  identical(tcrossprod(svt1), svt1 %*% t(svt1)),
  identical(crossprod(svt1[1:6, ], svt2), t(svt1[1:6, ]) %*% svt2),
  identical(tcrossprod(svt1, t(svt2)), m12)
)
```

`sparseMatrix-utils` *Internal utilities to handle sparseMatrix derivatives*

Description

The **SparseArray** package defines some utilities to handle sparseMatrix derivatives (e.g. dgCMatrix and lgCMatrix objects) from the **Matrix** package. These are for internal use only.

See Also

- dgCMatrix-class in the **Matrix** package.

`SVT_SparseArray-class` *SVT_SparseArray objects*

Description

The SVT_SparseArray class is a new container for efficient in-memory representation of multidimensional sparse arrays. It uses the *SVT layout* to represent the nonzero multidimensional data internally.

An SVT_SparseMatrix object is an SVT_SparseArray object of 2 dimensions.

Note that SVT_SparseArray and SVT_SparseMatrix objects replace the older and less efficient COO_SparseArray and COO_SparseMatrix objects.

Usage

```
## Constructor function:
SVT_SparseArray(x, dim=NULL, dimnames=NULL, type=NA)
```

Arguments

`x`	If `dim` is `NULL` (the default) then `x` must be an ordinary matrix or array, or a dgCMatrix/lgCMatrix object, or any matrix-like or array-like object that supports coercion to SVT_SparseArray. If `dim` is provided then `x` can either be missing, a vector (atomic or list), or an array-like object. If missing, then an allzero SVT_SparseArray object will be constructed. Otherwise `x` will be used to fill the returned object (`length(x)` must be <= `prod(dim)`). Note that if `x` is an array-like object then its dimensions are ignored i.e. it's treated as a vector.
`dim`	`NULL` or the dimensions (supplied as an integer vector) of the SVT_SparseArray or SVT_SparseMatrix object to construct. If `NULL` (the default) then the returned object will have the dimensions of matrix-like or array-like object `x`.

`dimnames` The *dimnames* of the object to construct. Must be `NULL` or a list of length the number of dimensions. Each list element must be either `NULL` or a character vector along the corresponding dimension. If both `dim` and `dimnames` are `NULL` (the default) then the returned object will have the *dimnames* of matrix-like or array-like object `x`.

`type` A single string specifying the requested type of the object.

Normally, the SVT_SparseArray object returned by the constructor function has the same `type()` as `x` but the user can use the `type` argument to request a different type. Note that doing:

```
svt <- SVT_SparseArray(x, type=type)
```

is equivalent to doing:

```
svt <- SVT_SparseArray(x)
type(svt) <- type
```

but the former is more convenient and will generally be more efficient.

Supported types are all R atomic types plus `"list"`.

Details

SVT_SparseArray is a concrete subclass of the SparseArray virtual class. This makes SVT_SparseArray objects SparseArray derivatives.

The nonzero data in a SVT_SparseArray object is stored in a *Sparse Vector Tree*. We'll refer to this internal data representation as the *SVT layout*. See the "SVT layout" section below for more information.

The SVT layout is similar to the CSC layout (compressed, sparse, column-oriented format) used by CsparseMatrix derivatives from the **Matrix** package, like dgCMatrix or lgCMatrix objects, but with the following improvements:

- The SVT layout supports sparse arrays of arbitrary dimensions.
- With the SVT layout, the sparse data can be of any type. Whereas CsparseMatrix derivatives only support sparse data of type `"double"` or `"logical"` at the moment.
- The SVT layout imposes no limit on the number of nonzero elements that can be stored. With dgCMatrix/lgCMatrix objects, this number must be < 2^31.
- Overall, the SVT layout allows more efficient operations on SVT_SparseArray objects.

Value

An SVT_SparseArray or SVT_SparseMatrix object.

SVT layout

An SVT (Sparse Vector Tree) is a tree of depth N - 1 where N is the number of dimensions of the sparse array.

The leaves in the tree can only be of two kinds: NULL or *leaf vector*. Leaves that are leaf vectors can only be found at the deepest level in the tree (i.e. at depth N - 1). All leaves found at a lower depth must be NULLs.

A leaf vector represents a sparse vector along the first dimension (a.k.a. innermost or fastest moving dimension) of the sparse array. It contains a collection of offset/value pairs sorted by strictly ascending offset. More precisely, a leaf vector is represented by an ordinary list of 2 parallel dense vectors:

1. nzvals: a vector (atomic or list) of nonzero values (zeros are not allowed);
2. nzoffs: an integer vector of offsets (i.e. 0-based positions).

The 1st vector determines the type of the leaf vector i.e. `"double"`, `"integer"`, `"logical"`, etc... All the leaf vectors in the SVT must have the same type as the sparse array.

It's useful to realize that a leaf vector simply represents a 1D SVT.

In **SparseArray** 1.5.4 a new type of leaf vector was introduced called *lacunar leaf*. A lacunar leaf is a non-empty leaf vector where the nzvals component is set to `NULL`. In this case the nonzero values are implicit: they're all considered to be equal to one.

Examples:

- An SVT_SparseArray object with 1 dimension has its nonzero data stored in an SVT of depth 0. Such SVT is represented by a single *leaf vector*.
- An SVT_SparseArray object with 2 dimensions has its nonzero data stored in an SVT of depth 1. Such SVT is represented by a list of length the extend of the 2nd dimension (number of columns). Each list element is an SVT of depth 0 (as described above), or a NULL if the corresponding column is empty (i.e. has no nonzero data).

  For example, the nonzero data of an 8-column sparse matrix will be stored in an SVT that looks like this:

```
    .-----------------list-of-length-8----------------.
   /       /       /      |       |      \       \       \
  |       |       |       |       |       |       |       |
 leaf    leaf    NULL    leaf    leaf    leaf    leaf    NULL
vector  vector          vector  vector  vector  vector
```

  The NULL leaves represent the empty columns (i.e. the columns with no nonzero elements).
- An SVT_SparseArray object with 3 dimensions has its nonzero data stored in an SVT of depth 2. Such SVT is represented by a list of length the extend of the 3rd dimension. Each list element must be an SVT of depth 1 (as described above) that stores the nonzero data of the corresponding 2D slice, or a NULL if the 2D slice is empty (i.e. has no nonzero data).
- And so on...

NOTE TO DEVELOPERS: Please refer to src/leaf_utils.h in the package for additional details about SVT leaf representation.

See Also

- The SparseArray class for the virtual parent class of COO_SparseArray and SVT_SparseArray.
- dgCMatrix-class and lgCMatrix-class in the **Matrix** package, for the de facto standard for sparse matrix representations in the R ecosystem.
- CsparseMatrix-class for the parent class of all the classes in the **Matrix** package that use the "CSC layout" (dgCMatrix, lgCMatrix, etc...)
- The `Matrix::rsparsematrix` function in the **Matrix** package.
- Ordinary array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## BASIC CONSTRUCTION
## ---------------------------------------------------------------------
SVT_SparseArray(dim=5:3)  # allzero object
```

```
SVT_SparseArray(dim=c(35000, 2e6), type="raw")  # allzero object

## Use a dgCMatrix object to fill the SVT_SparseArray object to construct:
x <- rsparsematrix(10, 16, density=0.1)  # random dgCMatrix object
SVT_SparseArray(x, dim=c(8, 5, 4))

svt1 <- SVT_SparseArray(dim=c(12, 5, 2))  # allzero object
svt1[cbind(11, 2:5, 2)] <- 22:25
svt1

svt2 <- SVT_SparseArray(dim=c(6, 4), type="integer",
                        dimnames=list(letters[1:6], LETTERS[1:4]))
svt2[c(1:2, 8, 10, 15:17, 24)] <- (1:8)*10L
svt2

## ---------------------------------------------------------------------
## CSC (Compressed Sparse Column) LAYOUT VS SVT LAYOUT
## ---------------------------------------------------------------------

## dgCMatrix objects from the Matrix package use the CSC layout:
dgcm2 <- as(svt2, "dgCMatrix")
dgcm2@x  # nonzero values
dgcm2@i  # row indices of the nonzero values
dgcm2@p  # breakpoints (0 followed by one breakpoint per column)

str(svt2)

m3 <- matrix(rpois(54e6, lambda=0.4), ncol=1200)

## Note that 'SparseArray(m3)' can also be used for this:
svt3 <- SVT_SparseArray(m3)
svt3

dgcm3 <- as(m3, "dgCMatrix")

## Compare type and memory footprint:
type(svt3)
object.size(svt3)
type(dgcm3)
object.size(dgcm3)

## Transpose:
system.time(svt <- t(t(svt3)))
system.time(dgcm <- t(t(dgcm3)))
identical(svt, svt3)
identical(dgcm, dgcm3)

## rbind():
m4 <- matrix(rpois(45e6, lambda=0.4), ncol=1200)
svt4 <- SVT_SparseArray(m4)
dgcm4 <- as(m4, "dgCMatrix")

system.time(rbind(svt3, svt4))
system.time(rbind(dgcm3, dgcm4))
```

thread-control *Number of threads used by SparseArray operations*

Description

Use `get_SparseArray_nthread` or `set_SparseArray_nthread` to get or set the number of threads to use by the multithreaded operations implemented in the **SparseArray** package.

Usage

```
get_SparseArray_nthread()
set_SparseArray_nthread(nthread=NULL)
```

Arguments

`nthread`	The number of threads to use by multithreaded operations implemented in the **SparseArray** package. On systems where OpenMP is available, this must be `NULL` or an integer value >= 1. When `NULL` (the default), a "reasonable" value will be used that never exceeds one third of the number of logical cpus available on the machine. On systems where OpenMP is not available, the supplied `nthread` is ignored and `set_SparseArray_nthread()` is a no-op.

Details

Multithreaded operations in the **SparseArray** package are implemented in C with OpenMP (https://www.openmp.org/).

Note that OpenMP is not available on all systems. On systems where it's available, `get_SparseArray_nthread()` is guaranteed to return a value >= 1. On systems where it's not available (e.g. macOS), `get_SparseArray_nthread()` returns 0 and `set_SparseArray_nthread()` is a no-op.

IMPORTANT: The portable way to disable multithreading is by calling `set_SparseArray_nthread(1)`, NOT `set_SparseArray_nthread(0)` (the latter returns an error on systems where OpenMP is available).

Value

`get_SparseArray_nthread()` returns an integer value >= 1 on systems where OpenMP is available, and 0 on systems where it's not.

`set_SparseArray_nthread()` returns the *previous* `nthread` value, that is, the value returned by `get_SparseArray_nthread()` before the call to `set_SparseArray_nthread()`. Note that the value is returned invisibly.

See Also

- SparseArray_matrixStats for SparseArray col/row summarization methods.
- SparseMatrix_mult for SparseMatrix multiplication and cross-product.
- SparseArray objects.

Examples

```
get_SparseArray_nthread()

if (get_SparseArray_nthread() != 0) {  # multithreading is available
    svt1 <- poissonSparseMatrix(77000L, 15000L, density=0.01)

    ## 'user' time is typically N x 'elapsed' time where N is roughly the
    ## number of threads that was effectively used:
    system.time(cv1 <- colVars(svt1))

    svt2 <- poissonSparseMatrix(77000L, 300L, density=0.3) * 0.77
    system.time(cp12 <- crossprod(svt1, svt2))

    prev_nthread <- set_SparseArray_nthread(1)  # disable multithreading
    system.time(cv1 <- colVars(svt1))
    system.time(cp12 <- crossprod(svt1, svt2))

    ## Restore previous 'nthread' value:
    set_SparseArray_nthread(prev_nthread)
}
```

Index

!,NaArray-method
(NaArray-Logic-methods), 18
!,SparseArray-method
(SparseArray-Logic-methods), 43
* **algebra**
NaArray-Arith-methods, 15
NaArray-Compare-methods, 17
NaArray-Logic-methods, 18
NaArray-matrixStats, 20
NaArray-summarization, 26
rowsum-methods, 32
SparseArray-Arith-methods, 39
SparseArray-Compare-methods, 41
SparseArray-Logic-methods, 43
SparseArray-matrixStats, 45
SparseArray-summarization, 52
* **arith**
NaArray-Arith-methods, 15
NaArray-Math-methods, 19
NaArray-matrixStats, 20
NaArray-summarization, 26
rowsum-methods, 32
SparseArray-Arith-methods, 39
SparseArray-Math-methods, 44
SparseArray-matrixStats, 45
SparseArray-summarization, 52
* **array**
extract_sparse_array, 5
is_nonna, 7
is_nonzero, 9
NaArray-abind, 14
NaArray-aperm, 15
NaArray-Arith-methods, 15
NaArray-Compare-methods, 17
NaArray-Logic-methods, 18
NaArray-Math-methods, 19
NaArray-matrixStats, 20
NaArray-misc-methods, 22
NaArray-subassignment, 23
NaArray-subsetting, 24
NaArray-summarization, 26
read_block_as_sparse, 31
rowsum-methods, 32
SparseArray-abind, 37
SparseArray-aperm, 38
SparseArray-Arith-methods, 39
SparseArray-Compare-methods, 41
SparseArray-Complex-methods, 42
SparseArray-Logic-methods, 43
SparseArray-Math-methods, 44
SparseArray-matrixStats, 45
SparseArray-misc-methods, 48
SparseArray-subassignment, 50
SparseArray-subsetting, 51
SparseArray-summarization, 52
SparseMatrix-mult, 54
sparseMatrix-utils, 55
* **classes**
COO_SparseArray-class, 3
NaArray, 12
SparseArray, 33
SVT_SparseArray-class, 55
* **complex**
SparseArray-Complex-methods, 42
* **internal**
extract_sparse_array, 5
read_block_as_sparse, 31
SparseArray-dim-tuning, 42
sparseMatrix-utils, 55
* **manip**
NaArray-abind, 14
SparseArray-abind, 37
* **methods**
COO_SparseArray-class, 3
extract_sparse_array, 5
is_nonna, 7
is_nonzero, 9
NaArray, 12
NaArray-abind, 14
NaArray-aperm, 15
NaArray-Arith-methods, 15
NaArray-Compare-methods, 17
NaArray-Logic-methods, 18
NaArray-Math-methods, 19
NaArray-matrixStats, 20
NaArray-misc-methods, 22

NaArray-subassignment, 23
NaArray-subsetting, 24
NaArray-summarization, 26
read_block_as_sparse, 31
rowsum-methods, 32
SparseArray, 33
SparseArray-abind, 37
SparseArray-aperm, 38
SparseArray-Arith-methods, 39
SparseArray-Compare-methods, 41
SparseArray-Complex-methods, 42
SparseArray-Logic-methods, 43
SparseArray-Math-methods, 44
SparseArray-matrixStats, 45
SparseArray-misc-methods, 48
SparseArray-subassignment, 50
SparseArray-subsetting, 51
SparseArray-summarization, 52
SparseMatrix-mult, 54
sparseMatrix-utils, 55
SVT_SparseArray-class, 55
∗ **utilities**
randomSparseArray, 27
readSparseCSV, 29
thread-control, 59
+,NaArray,missing-method
(NaArray-Arith-methods), 15
+,SparseArray,missing-method
(SparseArray-Arith-methods), 39
-,NaArray,missing-method
(NaArray-Arith-methods), 15
-,SparseArray,missing-method
(SparseArray-Arith-methods), 39
[, *24*, *51*
%*% (SparseMatrix-mult), 54
%*%,ANY,SparseMatrix-method
(SparseMatrix-mult), 54
%*%,SparseMatrix,ANY-method
(SparseMatrix-mult), 54
%*%,SparseMatrix,SparseMatrix-method
(SparseMatrix-mult), 54
%*%,SparseMatrix,matrix-method
(SparseMatrix-mult), 54
%*%,matrix,SparseMatrix-method
(SparseMatrix-mult), 54
%*%, *54*

abind, *14*, *37*
abind,NaArray-method (NaArray-abind), 14
abind,SparseArray-method
(SparseArray-abind), 37
acbind, *37*
anyNA, *26*, *53*
anyNA,NaArray-method
(NaArray-summarization), 26
anyNA,SparseArray-method
(SparseArray-summarization), 52
aperm, *15*, *39*
aperm,COO_SparseArray-method
(SparseArray-aperm), 38
aperm,NaArray-method (NaArray-aperm), 15
aperm,SVT_SparseArray-method
(SparseArray-aperm), 38
aperm.COO_SparseArray
(SparseArray-aperm), 38
aperm.NaArray (NaArray-aperm), 15
aperm.SVT_SparseArray
(SparseArray-aperm), 38
arbind, *37*
Arith,array,NaArray-method
(NaArray-Arith-methods), 15
Arith,array,SVT_SparseArray-method
(SparseArray-Arith-methods), 39
Arith,NaArray,array-method
(NaArray-Arith-methods), 15
Arith,NaArray,NaArray-method
(NaArray-Arith-methods), 15
Arith,NaArray,SVT_SparseArray-method
(NaArray-Arith-methods), 15
Arith,NaArray,vector-method
(NaArray-Arith-methods), 15
Arith,SVT_SparseArray,array-method
(SparseArray-Arith-methods), 39
Arith,SVT_SparseArray,NaArray-method
(NaArray-Arith-methods), 15
Arith,SVT_SparseArray,SVT_SparseArray-method
(SparseArray-Arith-methods), 39
Arith,SVT_SparseArray,vector-method
(SparseArray-Arith-methods), 39
Arith,vector,NaArray-method
(NaArray-Arith-methods), 15
Arith,vector,SVT_SparseArray-method
(SparseArray-Arith-methods), 39
Array, *13*, *34*, *35*
array, *4*, *8*, *11*, *13–17*, *19*, *20*, *23*, *24*, *35*, *37*, *39*, *41–44*, *49–51*, *57*
ArrayGrid, *32*
arrayInd, *3*, *4*
ArrayViewport, *31*
as.array,COO_SparseArray-method
(COO_SparseArray-class), 3
as.array,NaArray-method (NaArray), 12
as.array,SVT_SparseArray-method
(SVT_SparseArray-class), 55
as.array.COO_SparseArray

(COO_SparseArray-class), 3
as.array.NaArray (NaArray), 12
as.array.SVT_SparseArray
(SVT_SparseArray-class), 55

cbind, *14*, *37*
class:COO_SparseArray
(COO_SparseArray-class), 3
class:COO_SparseMatrix
(COO_SparseArray-class), 3
class:NaArray (NaArray), 12
class:NaMatrix (NaArray), 12
class:NULL_OR_list
(SVT_SparseArray-class), 55
class:SparseArray (SparseArray), 33
class:SparseMatrix (SparseArray), 33
class:SVT_SparseArray
(SVT_SparseArray-class), 55
class:SVT_SparseMatrix
(SVT_SparseArray-class), 55
coerce,ANY,COO_SparseArray-method
(COO_SparseArray-class), 3
coerce,ANY,COO_SparseMatrix-method
(COO_SparseArray-class), 3
coerce,ANY,SparseArray-method
(SVT_SparseArray-class), 55
coerce,ANY,SparseMatrix-method
(SVT_SparseArray-class), 55
coerce,ANY,SVT_SparseArray-method
(SVT_SparseArray-class), 55
coerce,ANY,SVT_SparseMatrix-method
(SVT_SparseArray-class), 55
coerce,Array,CsparseMatrix-method
(sparseMatrix-utils), 55
coerce,Array,dgCMatrix-method
(sparseMatrix-utils), 55
coerce,Array,dgRMatrix-method
(sparseMatrix-utils), 55
coerce,Array,lgCMatrix-method
(sparseMatrix-utils), 55
coerce,Array,lgRMatrix-method
(sparseMatrix-utils), 55
coerce,array,NaArray-method (NaArray),
12
coerce,Array,ngCMatrix-method
(sparseMatrix-utils), 55
coerce,Array,ngRMatrix-method
(sparseMatrix-utils), 55
coerce,Array,RsparseMatrix-method
(sparseMatrix-utils), 55
coerce,array,SparseArray-method
(SVT_SparseArray-class), 55
coerce,Array,sparseMatrix-method
(sparseMatrix-utils), 55
coerce,array,SVT_SparseArray-method
(SVT_SparseArray-class), 55
coerce,Array,TsparseMatrix-method
(sparseMatrix-utils), 55
coerce,COO_SparseArray,COO_SparseMatrix-method
(COO_SparseArray-class), 3
coerce,COO_SparseArray,SVT_SparseArray-method
(SVT_SparseArray-class), 55
coerce,COO_SparseMatrix,COO_SparseArray-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,dgCMatrix-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,dgRMatrix-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,dgTMatrix-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,lgCMatrix-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,lgRMatrix-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,lgTMatrix-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,ngCMatrix-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,ngRMatrix-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,ngTMatrix-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,SparseArray-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,sparseMatrix-method
(COO_SparseArray-class), 3
coerce,COO_SparseMatrix,SVT_SparseMatrix-method
(SVT_SparseArray-class), 55
coerce,CsparseMatrix,ngCMatrix-method
(sparseMatrix-utils), 55
coerce,CsparseMatrix,SVT_SparseMatrix-method
(SVT_SparseArray-class), 55
coerce,dgCMatrix,COO_SparseMatrix-method
(COO_SparseArray-class), 3
coerce,dgCMatrix,ngCMatrix-method
(sparseMatrix-utils), 55
coerce,dgRMatrix,COO_SparseMatrix-method
(COO_SparseArray-class), 3
coerce,lgCMatrix,COO_SparseMatrix-method
(COO_SparseArray-class), 3
coerce,lgRMatrix,COO_SparseMatrix-method
(COO_SparseArray-class), 3
coerce,Matrix,COO_SparseArray-method
(COO_SparseArray-class), 3

coerce,matrix,dgRMatrix-method
(sparseMatrix-utils), 55
coerce,matrix,lgCMatrix-method
(sparseMatrix-utils), 55
coerce,matrix,lgRMatrix-method
(sparseMatrix-utils), 55
coerce,matrix,NaMatrix-method
(NaArray), 12
coerce,matrix,ngCMatrix-method
(sparseMatrix-utils), 55
coerce,matrix,ngRMatrix-method
(sparseMatrix-utils), 55
coerce,Matrix,SparseArray-method
(SVT_SparseArray-class), 55
coerce,matrix,SparseMatrix-method
(SVT_SparseArray-class), 55
coerce,matrix,SVT_SparseMatrix-method
(SVT_SparseArray-class), 55
coerce,NaArray,NaMatrix-method
(NaArray), 12
coerce,NaMatrix,NaArray-method
(NaArray), 12
coerce,ngCMatrix,COO_SparseMatrix-method
(COO_SparseArray-class), 3
coerce,ngRMatrix,COO_SparseMatrix-method
(COO_SparseArray-class), 3
coerce,RsparseMatrix,ngRMatrix-method
(sparseMatrix-utils), 55
coerce,RsparseMatrix,SparseMatrix-method
(COO_SparseArray-class), 3
coerce,sparseMatrix,SparseMatrix-method
(SVT_SparseArray-class), 55
coerce,SVT_SparseArray,COO_SparseArray-method
(SVT_SparseArray-class), 55
coerce,SVT_SparseArray,SVT_SparseMatrix-method
(SVT_SparseArray-class), 55
coerce,SVT_SparseMatrix,COO_SparseMatrix-method
(SVT_SparseArray-class), 55
coerce,SVT_SparseMatrix,dgCMatrix-method
(SVT_SparseArray-class), 55
coerce,SVT_SparseMatrix,dgTMatrix-method
(SVT_SparseArray-class), 55
coerce,SVT_SparseMatrix,lgCMatrix-method
(SVT_SparseArray-class), 55
coerce,SVT_SparseMatrix,lgTMatrix-method
(SVT_SparseArray-class), 55
coerce,SVT_SparseMatrix,ngCMatrix-method
(SVT_SparseArray-class), 55
coerce,SVT_SparseMatrix,ngTMatrix-method
(SVT_SparseArray-class), 55
coerce,SVT_SparseMatrix,SparseArray-method
(SVT_SparseArray-class), 55
coerce,SVT_SparseMatrix,SVT_SparseArray-method
(SVT_SparseArray-class), 55
coerce,TsparseMatrix,COO_SparseMatrix-method
(COO_SparseArray-class), 3
coerce,TsparseMatrix,ngTMatrix-method
(sparseMatrix-utils), 55
coerce,TsparseMatrix,SVT_SparseMatrix-method
(SVT_SparseArray-class), 55
colAlls (SparseArray-matrixStats), 45
colAlls,NaArray-method
(NaArray-matrixStats), 20
colAlls,SparseArray-method
(SparseArray-matrixStats), 45
colAnyNAs (SparseArray-matrixStats), 45
colAnyNAs,NaArray-method
(NaArray-matrixStats), 20
colAnyNAs,SparseArray-method
(SparseArray-matrixStats), 45
colAnys (SparseArray-matrixStats), 45
colAnys,NaArray-method
(NaArray-matrixStats), 20
colAnys,SparseArray-method
(SparseArray-matrixStats), 45
colMaxs (SparseArray-matrixStats), 45
colMaxs,NaArray-method
(NaArray-matrixStats), 20
colMaxs,SparseArray-method
(SparseArray-matrixStats), 45
colMeans (SparseArray-matrixStats), 45
colMeans,NaArray-method
(NaArray-matrixStats), 20
colMeans,SparseArray-method
(SparseArray-matrixStats), 45
colMeans2 (SparseArray-matrixStats), 45
colMeans2,NaArray-method
(NaArray-matrixStats), 20
colMeans2,SparseArray-method
(SparseArray-matrixStats), 45
colMedians (SparseArray-matrixStats), 45
colMedians,SparseArray-method
(SparseArray-matrixStats), 45
colMins (SparseArray-matrixStats), 45
colMins,NaArray-method
(NaArray-matrixStats), 20
colMins,SparseArray-method
(SparseArray-matrixStats), 45
colProds (SparseArray-matrixStats), 45
colProds,NaArray-method
(NaArray-matrixStats), 20
colProds,SparseArray-method
(SparseArray-matrixStats), 45
colRanges, *22*, *46*

colRanges (SparseArray-matrixStats), 45
colRanges,NaArray-method
(NaArray-matrixStats), 20
colRanges,SparseArray-method
(SparseArray-matrixStats), 45
colSds (SparseArray-matrixStats), 45
colSds,NaArray-method
(NaArray-matrixStats), 20
colSds,SparseArray-method
(SparseArray-matrixStats), 45
colsum, *32*
colsum (rowsum-methods), 32
colsum,dsparseMatrix-method
(rowsum-methods), 32
colsum,SparseMatrix-method
(rowsum-methods), 32
colSums, *22*, *46*
colSums (SparseArray-matrixStats), 45
colSums,NaArray-method
(NaArray-matrixStats), 20
colSums,SparseArray-method
(SparseArray-matrixStats), 45
colSums2 (SparseArray-matrixStats), 45
colSums2,NaArray-method
(NaArray-matrixStats), 20
colSums2,SparseArray-method
(SparseArray-matrixStats), 45
colVars, *21*, *22*, *46*
colVars (SparseArray-matrixStats), 45
colVars,NaArray-method
(NaArray-matrixStats), 20
colVars,SparseArray-method
(SparseArray-matrixStats), 45
Compare,array,NaArray-method
(NaArray-Compare-methods), 17
Compare,array,SVT_SparseArray-method
(SparseArray-Compare-methods),
41
Compare,NaArray,array-method
(NaArray-Compare-methods), 17
Compare,NaArray,NaArray-method
(NaArray-Compare-methods), 17
Compare,NaArray,SVT_SparseArray-method
(NaArray-Compare-methods), 17
Compare,NaArray,vector-method
(NaArray-Compare-methods), 17
Compare,SVT_SparseArray,array-method
(SparseArray-Compare-methods),
41
Compare,SVT_SparseArray,NaArray-method
(NaArray-Compare-methods), 17
Compare,SVT_SparseArray,SVT_SparseArray-method
(SparseArray-Compare-methods),
41
Compare,SVT_SparseArray,vector-method
(SparseArray-Compare-methods),
41
Compare,vector,NaArray-method
(NaArray-Compare-methods), 17
Compare,vector,SVT_SparseArray-method
(SparseArray-Compare-methods),
41
Complex,SVT_SparseArray-method
(SparseArray-Complex-methods),
42
COO_SparseArray, *6*, *32*, *33*, *35*, *39*, *41*, *43*,
44, *53*, *55*
COO_SparseArray
(COO_SparseArray-class), 3
COO_SparseArray-class, 3
COO_SparseMatrix, *33*, *35*
COO_SparseMatrix
(COO_SparseArray-class), 3
COO_SparseMatrix-class
(COO_SparseArray-class), 3
crossprod, *54*
crossprod (SparseMatrix-mult), 54
crossprod,ANY,SparseMatrix-method
(SparseMatrix-mult), 54
crossprod,matrix,SparseMatrix-method
(SparseMatrix-mult), 54
crossprod,SparseMatrix,ANY-method
(SparseMatrix-mult), 54
crossprod,SparseMatrix,matrix-method
(SparseMatrix-mult), 54
crossprod,SparseMatrix,missing-method
(SparseMatrix-mult), 54
crossprod,SparseMatrix,SparseMatrix-method
(SparseMatrix-mult), 54
CsparseMatrix-class, *57*

dgCMatrix, *33*
dgCMatrix-class, *4*, *7*, *11*, *30*, *32*, *33*, *35*, *55*,
57
dgRMatrix-class, *35*
dgTMatrix, *33*
dim,NaArray-method (NaArray), 12
dim,SparseArray-method (SparseArray), 33
dimnames,NaArray-method (NaArray), 12
dimnames,SparseArray-method
(SparseArray), 33
dimnames<-,NaArray,ANY-method
(NaArray), 12
dimnames<-,SparseArray,ANY-method
(SparseArray), 33

drop, *24*, *51*
dsparseMatrix, *32*, *33*

extract_array, *6*, *7*
extract_array,COO_SparseArray-method (SparseArray-subsetting), 51
extract_array,NaArray-method (NaArray-subsetting), 24
extract_array,SVT_SparseArray-method (SparseArray-subsetting), 51
extract_na_array (NaArray-subsetting), 24
extract_na_array,NaArray-method (NaArray-subsetting), 24
extract_sparse_array, 5, *31*, *32*
extract_sparse_array,ANY-method (extract_sparse_array), 5
extract_sparse_array,COO_SparseArray-method (SparseArray-subsetting), 51
extract_sparse_array,SVT_SparseArray-method (SparseArray-subsetting), 51

get_SparseArray_nthread (thread-control), 59

is.infinite, *23*, *49*
is.infinite (SparseArray-misc-methods), 48
is.infinite,COO_SparseArray-method (SparseArray-misc-methods), 48
is.infinite,NaArray-method (NaArray-misc-methods), 22
is.infinite,SVT_SparseArray-method (SparseArray-misc-methods), 48
is.na, *49*
is.na (SparseArray-misc-methods), 48
is.na,COO_SparseArray-method (SparseArray-misc-methods), 48
is.na,NaArray-method (NaArray-misc-methods), 22
is.na,SVT_SparseArray-method (SparseArray-misc-methods), 48
is.nan, *23*
is.nan (SparseArray-misc-methods), 48
is.nan,COO_SparseArray-method (SparseArray-misc-methods), 48
is.nan,NaArray-method (NaArray-misc-methods), 22
is.nan,SVT_SparseArray-method (SparseArray-misc-methods), 48
is_nonna, 7, *11*, *13*
is_nonna,ANY-method (is_nonna), 7
is_nonna,NaArray-method (NaArray), 12
is_nonzero, *8*, 9, *35*
is_nonzero,ANY-method (is_nonzero), 9
is_nonzero,COO_SparseArray-method (COO_SparseArray-class), 3
is_nonzero,sparseMatrix-method (is_nonzero), 9
is_nonzero,SVT_SparseArray-method (SVT_SparseArray-class), 55
is_sparse, *6*, *7*, *31*, *32*, *35*
is_sparse,SparseArray-method (SparseArray), 33

lgCMatrix-class, *4*, *11*, *35*, *57*
Lindex, *8*, *10*
Lindex2Mindex, *3*, *4*, *24*, *51*
Logic,array,NaArray-method (NaArray-Logic-methods), 18
Logic,array,SVT_SparseArray-method (SparseArray-Logic-methods), 43
Logic,NaArray,array-method (NaArray-Logic-methods), 18
Logic,NaArray,NaArray-method (NaArray-Logic-methods), 18
Logic,NaArray,SVT_SparseArray-method (NaArray-Logic-methods), 18
Logic,NaArray,vector-method (NaArray-Logic-methods), 18
Logic,SVT_SparseArray,array-method (SparseArray-Logic-methods), 43
Logic,SVT_SparseArray,NaArray-method (NaArray-Logic-methods), 18
Logic,SVT_SparseArray,SVT_SparseArray-method (SparseArray-Logic-methods), 43
Logic,SVT_SparseArray,vector-method (SparseArray-Logic-methods), 43
Logic,vector,NaArray-method (NaArray-Logic-methods), 18
Logic,vector,SVT_SparseArray-method (SparseArray-Logic-methods), 43

Math,NaArray-method (NaArray-Math-methods), 19
Math,SVT_SparseArray-method (SparseArray-Math-methods), 44
Math2,NaArray-method (NaArray-Math-methods), 19
Math2,SVT_SparseArray-method (SparseArray-Math-methods), 44
matrix, *54*
mean, *26*, *53*
mean,NaArray-method (NaArray-summarization), 26

mean,SparseArray-method
(SparseArray-summarization), 52
mean.NaArray (NaArray-summarization), 26
mean.SparseArray
(SparseArray-summarization), 52

NaArray, *8*, 12, *14–24*, *26*
NaArray-abind, 14
NaArray-aperm, 15
NaArray-Arith (NaArray-Arith-methods), 15
NaArray-arith (NaArray-Arith-methods), 15
NaArray-Arith-methods, 15
NaArray-arith-methods
(NaArray-Arith-methods), 15
NaArray-class (NaArray), 12
NaArray-combine (NaArray-abind), 14
NaArray-Compare
(NaArray-Compare-methods), 17
NaArray-compare
(NaArray-Compare-methods), 17
NaArray-Compare-methods, 17
NaArray-compare-methods
(NaArray-Compare-methods), 17
NaArray-Logic (NaArray-Logic-methods), 18
NaArray-logic (NaArray-Logic-methods), 18
NaArray-Logic-methods, 18
NaArray-logic-methods
(NaArray-Logic-methods), 18
NaArray-Math (NaArray-Math-methods), 19
NaArray-math (NaArray-Math-methods), 19
NaArray-Math-methods, 19
NaArray-math-methods
(NaArray-Math-methods), 19
NaArray-Math2 (NaArray-Math-methods), 19
NaArray-math2 (NaArray-Math-methods), 19
NaArray-Math2-methods
(NaArray-Math-methods), 19
NaArray-math2-methods
(NaArray-Math-methods), 19
NaArray-matrixStats, 20
NaArray-misc (NaArray-misc-methods), 22
NaArray-misc-methods, 22
NaArray-subassignment, 23
NaArray-subsetting, 24
NaArray-summarization, 26
NaArray-transposition (NaArray-aperm), 15
NaArray_abind, *13*
NaArray_abind (NaArray-abind), 14
NaArray_aperm, *13*
NaArray_aperm (NaArray-aperm), 15
NaArray_Arith, *13*
NaArray_Arith (NaArray-Arith-methods), 15
NaArray_arith (NaArray-Arith-methods), 15
NaArray_Arith-methods
(NaArray-Arith-methods), 15
NaArray_arith-methods
(NaArray-Arith-methods), 15
NaArray_combine (NaArray-abind), 14
NaArray_Compare, *13*
NaArray_Compare
(NaArray-Compare-methods), 17
NaArray_compare
(NaArray-Compare-methods), 17
NaArray_Compare-methods
(NaArray-Compare-methods), 17
NaArray_compare-methods
(NaArray-Compare-methods), 17
NaArray_Logic, *13*
NaArray_Logic (NaArray-Logic-methods), 18
NaArray_logic (NaArray-Logic-methods), 18
NaArray_Logic-methods
(NaArray-Logic-methods), 18
NaArray_logic-methods
(NaArray-Logic-methods), 18
NaArray_Math, *13*
NaArray_Math (NaArray-Math-methods), 19
NaArray_math (NaArray-Math-methods), 19
NaArray_Math2 (NaArray-Math-methods), 19
NaArray_math2 (NaArray-Math-methods), 19
NaArray_Math2_methods
(NaArray-Math-methods), 19
NaArray_math2_methods
(NaArray-Math-methods), 19
NaArray_Math_methods
(NaArray-Math-methods), 19
NaArray_math_methods
(NaArray-Math-methods), 19
NaArray_matrixStats, *13*
NaArray_matrixStats
(NaArray-matrixStats), 20
NaArray_misc, *13*
NaArray_misc (NaArray-misc-methods), 22
NaArray_misc_methods
(NaArray-misc-methods), 22
NaArray_subassignment, *13*
NaArray_subassignment

(NaArray-subassignment), 23
NaArray_subsetting, *13*
NaArray_subsetting
(NaArray-subsetting), 24
NaArray_summarization, *13*
NaArray_summarization
(NaArray-summarization), 26
NaArray_transposition (NaArray-aperm), 15
NaMatrix, *14*, *21*, *22*
NaMatrix (NaArray), 12
NaMatrix-class (NaArray), 12
nchar, *49*
nchar (SparseArray-misc-methods), 48
nchar,COO_SparseArray-method
(SparseArray-misc-methods), 48
ngCMatrix-class, *11*
nnacount (is_nonna), 7
nnacount,ANY-method (is_nonna), 7
nnacount,NaArray-method (NaArray), 12
nnavals (is_nonna), 7
nnavals,ANY-method (is_nonna), 7
nnavals,NaArray-method (NaArray), 12
nnavals<- (is_nonna), 7
nnavals<-,ANY-method (is_nonna), 7
nnawhich (is_nonna), 7
nnawhich,ANY-method (is_nonna), 7
nnawhich,NaArray-method (NaArray), 12
NULL_OR_list (SVT_SparseArray-class), 55
NULL_OR_list-class
(SVT_SparseArray-class), 55
nzcoo (COO_SparseArray-class), 3
nzcoo,COO_SparseArray-method
(COO_SparseArray-class), 3
nzcount (is_nonzero), 9
nzcount,ANY-method (is_nonzero), 9
nzcount,COO_SparseArray-method
(COO_SparseArray-class), 3
nzcount,CsparseMatrix-method
(is_nonzero), 9
nzcount,RsparseMatrix-method
(is_nonzero), 9
nzcount,SVT_SparseArray-method
(SVT_SparseArray-class), 55
nzcount,TsparseMatrix-method
(is_nonzero), 9
nzdata (COO_SparseArray-class), 3
nzdata,COO_SparseArray-method
(COO_SparseArray-class), 3
nzvals (is_nonzero), 9
nzvals,ANY-method (is_nonzero), 9
nzvals,COO_SparseArray-method
(COO_SparseArray-class), 3
nzvals,dgCMatrix-method (is_nonzero), 9
nzvals,lgCMatrix-method (is_nonzero), 9
nzvals,nMatrix-method (is_nonzero), 9
nzvals,SVT_SparseArray-method
(SVT_SparseArray-class), 55
nzvals<- (is_nonzero), 9
nzvals<-,ANY-method (is_nonzero), 9
nzvals<-,COO_SparseArray-method
(COO_SparseArray-class), 3
nzwhich (is_nonzero), 9
nzwhich,ANY-method (is_nonzero), 9
nzwhich,COO_SparseArray-method
(COO_SparseArray-class), 3
nzwhich,CsparseMatrix-method
(is_nonzero), 9
nzwhich,RsparseMatrix-method
(is_nonzero), 9
nzwhich,SVT_SparseArray-method
(SVT_SparseArray-class), 55

pmax (SparseArray-misc-methods), 48
pmax,SparseArray-method
(SparseArray-misc-methods), 48
pmin, *49*
pmin (SparseArray-misc-methods), 48
pmin,SparseArray-method
(SparseArray-misc-methods), 48
poissonSparseArray (randomSparseArray), 27
poissonSparseMatrix
(randomSparseArray), 27

randomSparseArray, 27, *35*
randomSparseMatrix (randomSparseArray), 27
range, *26*, *53*
range,COO_SparseArray-method
(SparseArray-summarization), 52
range,NaArray-method
(NaArray-summarization), 26
range,SVT_SparseArray-method
(SparseArray-summarization), 52
range.COO_SparseArray
(SparseArray-summarization), 52
range.NaArray (NaArray-summarization), 26
range.SVT_SparseArray
(SparseArray-summarization), 52
read_block, *31*, *32*
read_block_as_dense, *31*
read_block_as_sparse, *6*, *7*, 31

read_block_as_sparse,ANY-method
(read_block_as_sparse), 31
readSparseCSV, 29, *35*
readSparseTable (readSparseCSV), 29
round,NaArray-method
(NaArray-Math-methods), 19
round,SVT_SparseArray-method
(SparseArray-Math-methods), 44
rowAlls (SparseArray-matrixStats), 45
rowAlls,SparseArray-method
(SparseArray-matrixStats), 45
rowAnyNAs (SparseArray-matrixStats), 45
rowAnyNAs,NaArray-method
(NaArray-matrixStats), 20
rowAnyNAs,SparseArray-method
(SparseArray-matrixStats), 45
rowAnys (SparseArray-matrixStats), 45
rowAnys,SparseArray-method
(SparseArray-matrixStats), 45
rowMaxs (SparseArray-matrixStats), 45
rowMaxs,NaArray-method
(NaArray-matrixStats), 20
rowMaxs,SparseArray-method
(SparseArray-matrixStats), 45
rowMeans (SparseArray-matrixStats), 45
rowMeans,SparseArray-method
(SparseArray-matrixStats), 45
rowMeans2 (SparseArray-matrixStats), 45
rowMeans2,SparseArray-method
(SparseArray-matrixStats), 45
rowMedians (SparseArray-matrixStats), 45
rowMedians,SparseArray-method
(SparseArray-matrixStats), 45
rowMins (SparseArray-matrixStats), 45
rowMins,NaArray-method
(NaArray-matrixStats), 20
rowMins,SparseArray-method
(SparseArray-matrixStats), 45
rowProds (SparseArray-matrixStats), 45
rowProds,SparseArray-method
(SparseArray-matrixStats), 45
rowRanges (SparseArray-matrixStats), 45
rowRanges,NaArray-method
(NaArray-matrixStats), 20
rowRanges,SparseArray-method
(SparseArray-matrixStats), 45
rowSds (SparseArray-matrixStats), 45
rowSds,SparseArray-method
(SparseArray-matrixStats), 45
rowsum, *32*, *33*
rowsum (rowsum-methods), 32
rowsum,dsparseMatrix-method
(rowsum-methods), 32
rowsum,SparseMatrix-method
(rowsum-methods), 32
rowsum-methods, 32
rowsum.dsparseMatrix (rowsum-methods), 32
rowsum.SparseMatrix (rowsum-methods), 32
rowsum_methods, *35*
rowsum_methods (rowsum-methods), 32
rowSums (SparseArray-matrixStats), 45
rowSums,NaArray-method
(NaArray-matrixStats), 20
rowSums,SparseArray-method
(SparseArray-matrixStats), 45
rowSums2 (SparseArray-matrixStats), 45
rowSums2,NaArray-method
(NaArray-matrixStats), 20
rowSums2,SparseArray-method
(SparseArray-matrixStats), 45
rowVars (SparseArray-matrixStats), 45
rowVars,SparseArray-method
(SparseArray-matrixStats), 45
rpois, *27*, *28*
rsparsematrix, *28*, *57*

S4groupGeneric, *15–20*, *39*, *41–44*
sd,NaArray-method
(NaArray-summarization), 26
sd,SparseArray-method
(SparseArray-summarization), 52
set_SparseArray_nthread, *22*, *46*, *54*
set_SparseArray_nthread
(thread-control), 59
show,NaArray-method (NaArray), 12
show,SparseArray-method (SparseArray), 33
signif,NaArray-method
(NaArray-Math-methods), 19
signif,SVT_SparseArray-method
(SparseArray-Math-methods), 44
SparseArray, *4*, *6*, *7*, *10*, *11*, *22*, *23*, *27*, *28*, *30*, *32*, 33, *37–39*, *41–46*, *48–51*, *53*, *56*, *57*, *59*
SparseArray-abind, 37
SparseArray-aperm, 38
SparseArray-Arith
(SparseArray-Arith-methods), 39
SparseArray-arith
(SparseArray-Arith-methods), 39
SparseArray-Arith-methods, 39
SparseArray-arith-methods
(SparseArray-Arith-methods), 39
SparseArray-class (SparseArray), 33

SparseArray-combine (SparseArray-abind), 37
SparseArray-Compare (SparseArray-Compare-methods), 41
SparseArray-compare (SparseArray-Compare-methods), 41
SparseArray-Compare-methods, 41
SparseArray-compare-methods (SparseArray-Compare-methods), 41
SparseArray-Complex (SparseArray-Complex-methods), 42
SparseArray-complex (SparseArray-Complex-methods), 42
SparseArray-Complex-methods, 42
SparseArray-complex-methods (SparseArray-Complex-methods), 42
SparseArray-dim-tuning, 42
SparseArray-Logic (SparseArray-Logic-methods), 43
SparseArray-logic (SparseArray-Logic-methods), 43
SparseArray-Logic-methods, 43
SparseArray-logic-methods (SparseArray-Logic-methods), 43
SparseArray-Math (SparseArray-Math-methods), 44
SparseArray-math (SparseArray-Math-methods), 44
SparseArray-Math-methods, 44
SparseArray-math-methods (SparseArray-Math-methods), 44
SparseArray-Math2 (SparseArray-Math-methods), 44
SparseArray-math2 (SparseArray-Math-methods), 44
SparseArray-Math2-methods (SparseArray-Math-methods), 44
SparseArray-math2-methods (SparseArray-Math-methods), 44
SparseArray-matrixStats, 45
SparseArray-misc (SparseArray-misc-methods), 48
SparseArray-misc-methods, 48
SparseArray-subassignment, 50
SparseArray-subsetting, 51
SparseArray-summarization, 52
SparseArray-transposition (SparseArray-aperm), 38
SparseArray_abind, *35*
SparseArray_abind (SparseArray-abind), 37
SparseArray_aperm, *35*
SparseArray_aperm (SparseArray-aperm), 38
SparseArray_Arith, *35*
SparseArray_Arith (SparseArray-Arith-methods), 39
SparseArray_arith (SparseArray-Arith-methods), 39
SparseArray_Arith-methods (SparseArray-Arith-methods), 39
SparseArray_arith-methods (SparseArray-Arith-methods), 39
SparseArray_combine (SparseArray-abind), 37
SparseArray_Compare, *35*
SparseArray_Compare (SparseArray-Compare-methods), 41
SparseArray_compare (SparseArray-Compare-methods), 41
SparseArray_Compare-methods (SparseArray-Compare-methods), 41
SparseArray_compare-methods (SparseArray-Compare-methods), 41
SparseArray_Complex, *35*
SparseArray_Complex (SparseArray-Complex-methods), 42
SparseArray_complex (SparseArray-Complex-methods), 42
SparseArray_Complex_methods (SparseArray-Complex-methods), 42
SparseArray_complex_methods (SparseArray-Complex-methods), 42
SparseArray_dim_tuning (SparseArray-dim-tuning), 42
SparseArray_Logic, *35*
SparseArray_Logic (SparseArray-Logic-methods), 43
SparseArray_logic (SparseArray-Logic-methods), 43

SparseArray_Logic-methods
(SparseArray-Logic-methods), 43
SparseArray_logic-methods
(SparseArray-Logic-methods), 43
SparseArray_Math, *35*
SparseArray_Math
(SparseArray-Math-methods), 44
SparseArray_math
(SparseArray-Math-methods), 44
SparseArray_Math2
(SparseArray-Math-methods), 44
SparseArray_math2
(SparseArray-Math-methods), 44
SparseArray_Math2_methods
(SparseArray-Math-methods), 44
SparseArray_math2_methods
(SparseArray-Math-methods), 44
SparseArray_Math_methods
(SparseArray-Math-methods), 44
SparseArray_math_methods
(SparseArray-Math-methods), 44
SparseArray_matrixStats, *35*, *59*
SparseArray_matrixStats
(SparseArray-matrixStats), 45
SparseArray_misc, *35*
SparseArray_misc
(SparseArray-misc-methods), 48
SparseArray_misc_methods
(SparseArray-misc-methods), 48
SparseArray_subassignment, *35*
SparseArray_subassignment
(SparseArray-subassignment), 50
SparseArray_subsetting, *35*
SparseArray_subsetting
(SparseArray-subsetting), 51
SparseArray_summarization, *35*
SparseArray_summarization
(SparseArray-summarization), 52
SparseArray_transposition
(SparseArray-aperm), 38
SparseMatrix, *28*, *30*, *32*, *33*, *37*, *46*, *54*
SparseMatrix (SparseArray), 33
SparseMatrix-class (SparseArray), 33
SparseMatrix-mult, 54
sparseMatrix-utils, 55
SparseMatrix_mult, *35*, *59*
SparseMatrix_mult (SparseMatrix-mult),
54
sparseMatrix_utils
(sparseMatrix-utils), 55
sparsity (is_nonzero), 9
SVT_SparseArray, *3*, *4*, *6*, *12*, *13*, *15–19*, *28*,
32–35, *39*, *41*, *43*, *44*, *46*, *53*
SVT_SparseArray
(SVT_SparseArray-class), 55
SVT_SparseArray-class, 55
SVT_SparseMatrix, *28*, *30*, *33–35*
SVT_SparseMatrix
(SVT_SparseArray-class), 55
SVT_SparseMatrix-class
(SVT_SparseArray-class), 55

t,NaMatrix-method (NaArray-aperm), 15
t,SVT_SparseMatrix-method
(SparseArray-aperm), 38
t.NaMatrix (NaArray-aperm), 15
t.SVT_SparseMatrix (SparseArray-aperm),
38
tcrossprod, *54*
tcrossprod (SparseMatrix-mult), 54
tcrossprod,ANY,SparseMatrix-method
(SparseMatrix-mult), 54
tcrossprod,matrix,SparseMatrix-method
(SparseMatrix-mult), 54
tcrossprod,SparseMatrix,ANY-method
(SparseMatrix-mult), 54
tcrossprod,SparseMatrix,matrix-method
(SparseMatrix-mult), 54
tcrossprod,SparseMatrix,missing-method
(SparseMatrix-mult), 54
tcrossprod,SparseMatrix,SparseMatrix-method
(SparseMatrix-mult), 54
thread-control, 59
thread_control (thread-control), 59
tolower, *49*
tolower (SparseArray-misc-methods), 48
tolower,COO_SparseArray-method
(SparseArray-misc-methods), 48
toupper (SparseArray-misc-methods), 48
toupper,COO_SparseArray-method
(SparseArray-misc-methods), 48
tune_Array_dims,NaArray-method
(NaArray-subsetting), 24
tune_Array_dims,SVT_SparseArray-method
(SparseArray-dim-tuning), 42
type, *7*, *32*, *54*
type,COO_SparseArray-method
(COO_SparseArray-class), 3
type,NaArray-method (NaArray), 12
type,SVT_SparseArray-method
(SVT_SparseArray-class), 55
type<-,COO_SparseArray-method
(COO_SparseArray-class), 3
type<-,NaArray-method (NaArray), 12

type<-,SVT_SparseArray-method
(SVT_SparseArray-class), 55

var,NaArray,ANY-method
(NaArray-summarization), 26
var,SparseArray,ANY-method
(SparseArray-summarization), 52

which, *8*, *11*, *35*
writeSparseCSV (readSparseCSV), 29