Package 'h5mread'

July 29, 2025

Title A fast HDF5 reader

Description The main function in the h5mread package is h5mread(), which allows reading arbitrary data from an HDF5 dataset into R, similarly to what the h5read() function from the rhdf5 package does.
In the case of h5mread(), the implementation has been optimized to make it as fast and memory-efficient as possible.

biocViews Infrastructure, DataRepresentation, DataImport

URL https://bioconductor.org/packages/h5mread

BugReports https://github.com/Bioconductor/h5mread/issues

Version 1.1.1

License Artistic-2.0

Encoding UTF-8

Depends R (>= 4.5), methods, rhdf5, BiocGenerics, SparseArray

Imports stats, tools, rhdf5filters, S4Vectors, IRanges, S4Arrays

LinkingTo Rhdf5lib, S4Vectors

SystemRequirements GNU make

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

VignetteBuilder knitr

Collate utils.R h5dim.R H5File-class.R h5ls.R H5DSetDescriptor-class.R uaselection.R h5mread.R h5summarize.R h5mread_from_reshaped.R h5dimscales.R h5writeDimnames.R zzz.R

git_url https://git.bioconductor.org/packages/h5mread

git_branch devel

git_last_commit 52c036a

git_last_commit_date 2025-05-15

Repository Bioconductor 3.22

Date/Publication 2025-07-29

Author Hervé Pagès [aut, cre] (ORCID: <https://orcid.org/0009-0002-8272-4522>)

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

Contents

`h5dim` *Get the dimensions of an HDF5 dataset*

Description

Two convenience functions to obtain the dimensions of an HDF5 dataset as well as the dimensions of its chunks.

Usage

```
h5dim(filepath, name, as.integer=TRUE)

h5chunkdim(filepath, name, adjust=FALSE)
```

Arguments

`filepath`	The path (as a single string) to an HDF5 file.
`name`	The name (as a single string) of a dataset in the HDF5 file.
`as.integer`	By default `h5dim()` returns the dimensions of the dataset in an integer vector and will raise an error if any dimension is greater than `.Machine$integer.max` (= 2^31 - 1). Use `as.integer=FALSE` to support datasets with dimensions greater than `.Machine$integer.max`. In this case the dimensions are returned in a numeric vector.
`adjust`	By default `h5chunkdim()` returns the dimensions of the chunks as reported by `H5Pget_chunk` from the C HDF5 API. Note that the HDF5 specs allow some or all the dimensions of the chunks to be greater than the dimensions of the dataset. You can use `adjust=TRUE` to request `h5chunkdim()` to return *adjusted chunk dimensions*, that is, dimensions that do not exceed the dimensions of the dataset. The *adjusted chunk dimensions* are simply obtained by replacing those dimensions in the vector of chunk dimensions that are greater than the corresponding dataset dimension with the latter.

Value

An integer (or numeric) vector of length the number of dimensions of the HDF5 dataset.

Examples

```
test_h5 <- system.file("extdata", "test.h5", package="h5mread")
h5ls(test_h5)

h5dim(test_h5, "m2")
h5chunkdim(test_h5, "m2")

h5dim(test_h5, "a3")
h5chunkdim(test_h5, "a3")
```

`H5File-class` *H5File objects*

Description

The H5File class provides a formal representation of an HDF5 file (local or remote).

Usage

```
## Constructor function:
H5File(filepath, s3=FALSE, s3credentials=NULL, .no_rhdf5_h5id=FALSE)
```

Arguments

`filepath`	A single string specifying the path or URL to an HDF5 file.
`s3`	`TRUE` or `FALSE`. Should the `filepath` argument be treated as the URL to a file stored in an Amazon S3 bucket, rather than the path to a local file?
`s3credentials`	A list of length 3, providing the credentials for accessing files stored in a private Amazon S3 bucket. See ?`H5Pset_fapl_ros3` in the **rhdf5** package for more information.
`.no_rhdf5_h5id`	For internal use only. Don't use.

Details

IMPORTANT NOTE ABOUT H5File OBJECTS AND PARALLEL EVALUATION

The short story is that H5File objects cannot be used in the context of parallel evaluation at the moment.

Here is why:

H5File objects contain an identifier to an open connection to the HDF5 file. This identifier becomes invalid in the 2 following situations:

- After serialization/deserialization, that is, after loading a serialized H5File object with `readRDS()` or `load()`.
- In the context of parallel evaluation, when using the SnowParam parallelization backend. This is because, unlike the MulticoreParam backend which used a system `fork`, the SnowParam backend uses serialization/deserialization to transmit the object to the workers.

In both cases, the connection to the file is lost and any attempt to read data from the H5File object will fail. Note that the above also happens to any H5File object that got serialized indirectly i.e. as part of a bigger object. For example, if an HDF5Array object was constructed from an H5File object, then it contains the H5File object and therefore `blockApply(..., BPPARAM=SnowParam(4))` cannot be used on it.

Furthermore, even if sometimes an H5File object *seems* to work fine with the MulticoreParam parallelization backend, this is highly unreliable and must be avoided.

Value

An H5File object.

See Also

- H5Pset_fapl_ros3 in the **rhdf5** package for detailed information about how to pass your S3 credentials to the `s3credentials` argument.
- The HDF5Array class defined in the **HDF5Array** package for representing and operating on a conventional (a.k.a. dense) HDF5 dataset.
- The H5SparseMatrix class defined in the **HDF5Array** package for representing and operating on an HDF5 sparse matrix.
- The H5ADMatrix class defined in the **HDF5Array** package for representing and operating on the central matrix of an `h5ad` file, or any matrix in its `/layers` group.
- The TENxMatrix class defined in the **HDF5Array** package for representing and operating on a 10x Genomics dataset.
- The `h5mread` function in this package (**h5mread**) that is used internally by HDF5Array, TENxMatrix, and H5ADMatrix objects, for (almost) all their data reading needs.
- `h5ls` to list the content of an HDF5 file.
- `bplapply`, `MulticoreParam`, and `SnowParam`, in the **BiocParallel** package.

Examples

```
## ---------------------------------------------------------------------
## A. BASIC USAGE
## ---------------------------------------------------------------------

## With a local file:
test_h5 <- system.file("extdata", "test.h5", package="h5mread")
h5file1 <- H5File(test_h5)
h5ls(h5file1)
path(h5file1)

h5mread(h5file1, "m2", list(1:10, 1:6))
get_h5mread_returned_type(h5file1, "m2")

## With a file stored in an Amazon S3 bucket:
if (Sys.info()[["sysname"]] != "Darwin") {
  public_S3_url <-
   "https://rhdf5-public.s3.eu-central-1.amazonaws.com/rhdf5ex_t_float_3d.h5"
  h5file2 <- H5File(public_S3_url, s3=TRUE)
  h5ls(h5file2)

  h5mread(h5file2, "a1")
  get_h5mread_returned_type(h5file2, "a1")
```

```
}

## ------------------------------------------------------------------------
## B. H5File OBJECTS AND PARALLEL EVALUATION
## ------------------------------------------------------------------------
## H5File objects cannot be used in the context of parallel evaluation
## at the moment!

library(BiocParallel)

FUN1 <- function(i, h5file, name)
    sum(h5mread::h5mread(h5file, name, list(i, NULL)))

FUN2 <- function(i, h5file, name)
    sum(h5mread::h5mread(h5file, name, list(i, NULL, NULL)))

## With the SnowParam parallelization backend, the H5File object
## does NOT work on the workers:
## Not run:
## ERROR!
res1 <- bplapply(1:150, FUN1, h5file1, "m2", BPPARAM=SnowParam(3))
## ERROR!
res2 <- bplapply(1:5, FUN2, h5file2, "a1", BPPARAM=SnowParam(3))

## End(Not run)

## With the MulticoreParam parallelization backend, the H5File object
## might seem to work on the workers. However this is highly unreliable
## and must be avoided:
## Not run:
if (.Platform$OS.type != "windows") {
  ## UNRELIABLE!
  res1 <- bplapply(1:150, FUN1, h5file1, "m2", BPPARAM=MulticoreParam(3))
  ## UNRELIABLE!
  res2 <- bplapply(1:5, FUN2, h5file2, "a1", BPPARAM=MulticoreParam(3))
}

## End(Not run)
```

---

h5ls *A wrapper to rhdf5::h5ls() that works on H5File objects*

Description

Like `rhdf5::h5ls()`, but works on an H5File object.

Usage

```
h5ls(file, recursive=TRUE, all=FALSE, datasetinfo=TRUE,
           index_type=h5default("H5_INDEX"), order=h5default("H5_ITER"),
           s3=FALSE, s3credentials=NULL, native=FALSE)
```

Arguments

`file`, `recursive`, `all`, `datasetinfo`, `index_type`, `order`, `s3`, `s3credentials`, `native`

See `?rhdf5::h5ls` in the **rhdf5** package for a description of these arguments.

Note that the only difference with `rhdf5::h5ls()` is that, with `h5mread::h5ls()`, `file` can be an H5File object.

Value

See `?rhdf5::h5ls` in the **rhdf5** package.

See Also

- `h5ls` in the **rhdf5** package.
- H5File objects.

Examples

```
test_h5 <- system.file("extdata", "test.h5", package="h5mread")
h5ls(test_h5)

h5file <- H5File(test_h5)
h5ls(h5file)

## See '?H5File' for more examples.
```

---

h5mread *An alternative to* `rhdf5::h5read`

Description

An efficient and flexible alternative to `rhdf5::h5read()`.

Usage

```
h5mread(filepath, name, starts=NULL, counts=NULL, noreduce=FALSE,
        as.vector=NA, as.integer=FALSE, as.sparse=FALSE,
        method=0L, use.H5Dread_chunk=FALSE)

get_h5mread_returned_type(filepath, name, as.integer=FALSE)
```

Arguments

`filepath`	The path (as a single string) to the HDF5 file where the dataset to read from is located, or an H5File object. Note that you must create and use an H5File object if the HDF5 file to access is stored in an Amazon S3 bucket. See `?H5File` for how to do this. Also please note that H5File objects must NOT be used in the context of parallel evaluation at the moment.
`name`	The name of the dataset in the HDF5 file.

`starts`, `counts` `starts` and `counts` are used to specify the *array selection*. Each argument can be either `NULL` or a list with one list element per dimension in the dataset.

If `starts` and `counts` are both `NULL`, then the entire dataset is read.

If `starts` is a list, each list element in it must be a vector of valid positive indices along the corresponding dimension in the dataset. An empty vector (`integer(0)`) is accepted and indicates an empty selection along that dimension. A `NULL` is accepted and indicates a *full* selection along the dimension so has the same meaning as a missing subscript when subsetting an array-like object with `[`. (Note that for `[` a `NULL` subscript indicates an empty selection.)

Each list element in `counts` must be `NULL` or a vector of non-negative integers of the same length as the corresponding list element in `starts`. Each value in the vector indicates how many positions to select starting from the associated start value. A `NULL` indicates that a single position is selected for each value along the corresponding dimension.

If `counts` is `NULL`, then each index in each `starts` list element indicates a single position selection along the corresponding dimension. Note that in this case the `starts` argument is equivalent to the `index` argument of `h5read` and `extract_array` (with the caveat that `h5read` doesn't accept empty selections).

Finally note that when `counts` is not `NULL` then the selection described by `starts` and `counts` must be *strictly ascending* along each dimension.

`as.vector` Should the data be returned in a vector instead of an array? By default (i.e. when set to `NA`), the data is returned in an ordinary array when reading from a multidimensional dataset, and in an ordinary vector when reading from a 1D dataset. You can override this by setting `as.vector` to `TRUE` or `FALSE`.

`as.integer` If set to `TRUE` then the data is loaded in an ordinary array (or vector) of `type()` `"integer"`. This will typically reduce the memory footprint of the returned array or vector by half if the values in the HDF5 dataset are floating point values.

Note that, when `as.integer=TRUE`, the values loaded from the HDF5 dataset get coerced to integers at the C level as early as possible so this transformation is very efficient.

`as.sparse` By default `h5mread()` returns the data in an ordinary array or vector. Use `as.sparse=TRUE` to return it in a SparseArray derivative from the **SparseArray** package. This will significantly reduce the memory footprint of the returned object if the HDF5 dataset contains mostly zeros.

`noreduce`, `method`, `use.H5Dread_chunk`
For testing and advanced usage only. Do not use.

Value

`h5mread()` returns an ordinary array or vector if `as.sparse` is `FALSE` (the default), and a COO_SparseArray object if `as.sparse` is `TRUE`.

`get_h5mread_returned_type()` returns the type of the array or vector that will be returned by `h5mread()`. Equivalent to (but more efficient than):

```
typeof(h5mread(filepath, name, rep(list(integer(0)), ndim)))
```

where `ndim` is the number of dimensions (a.k.a. *rank* in HDF5 jargon) of the dataset.

See Also

- H5File objects.
- h5read in the **rhdf5** package.
- extract_array in the **S4Arrays** package.
- COO_SparseArray objects in the **SparseArray** package.
- h5mread_from_reshaped to read data from a virtually reshaped HDF5 dataset.

Examples

```
## ---------------------------------------------------------------------
## BASIC EXAMPLES
## ---------------------------------------------------------------------
test_h5 <- system.file("extdata", "test.h5", package="h5mread")
h5ls(test_h5)

m1 <- h5mread(test_h5, "m1")  # 12 x 5 integer matrix

m <- h5mread(test_h5, "m1", starts=list(c(8, 12:7), NULL))
m

## Sanity check:
stopifnot(identical(m1[c(8, 12:7), ], m))

m <- h5mread(test_h5, "m1", starts=list(c(8, 12:7), integer(0)))
m

## Sanity check:
stopifnot(identical(m1[c(8, 12:7), NULL], m))

m2 <- h5mread(test_h5, "m2")  # 4000 x 90 double matrix

m2a <- h5mread(test_h5, "m2", starts=list(31, 1), counts=list(10, 8))
m2a

## Sanity check:
stopifnot(identical(m2[31:40, 1:8], m2a))

m2b <- h5mread(test_h5, "m2", starts=list(31, 1), counts=list(10, 8),
               as.integer=TRUE)
m2b

## Sanity check:
storage.mode(m2a) <- "integer"
stopifnot(identical(m2a, m2b))

a3 <- h5mread(test_h5, "a3")  # 180 x 75 x 4 integer array

starts <- list(c(21, 101), NULL, 3:4)
counts <- list(c( 5,  22), NULL, NULL)
a <- h5mread(test_h5, "a3", starts=starts, counts=counts)
dim(a)
a[1:10, 1:12, ]

## Sanity check:
stopifnot(identical(a3[c(21:25, 101:122), , 3:4, drop=FALSE], a))
```

```
## ---------------------------------------------------------------------
## RETURNING THE DATA AS A SPARSE ARRAY
## ---------------------------------------------------------------------

starts <- list(c(21:25, 101:122), NULL, 3:4)
coo <- h5mread(test_h5, "a3", starts=starts, as.sparse=TRUE)
coo

class(coo)  # COO_SparseArray object (see ?COO_SparseArray)
dim(coo)

## Sanity check:
stopifnot(is(coo, "COO_SparseArray"), identical(a, as.array(coo)))

## ---------------------------------------------------------------------
## PERFORMANCE
## ---------------------------------------------------------------------
library(ExperimentHub)
hub <- ExperimentHub()

## With the "sparse" TENxBrainData dataset
## ---------------------------------------
fname0 <- hub[["EH1039"]]
h5ls(fname0)  # all datasets are 1D datasets

index <- list(77 * sample(34088679, 5000, replace=TRUE))
## h5mread() is about 4x faster than h5read():
system.time(a <- h5mread::h5mread(fname0, "mm10/data", index))
system.time(b <- h5read(fname0, "mm10/data", index=index))
stopifnot(identical(a, as.vector(b)))

index <- list(sample(1306127, 7500, replace=TRUE))
## h5mread() is about 14x faster than h5read():
system.time(a <- h5mread::h5mread(fname0, "mm10/barcodes", index))
system.time(b <- h5read(fname0, "mm10/barcodes", index=index))
stopifnot(identical(a, as.vector(b)))

## With the "dense" TENxBrainData dataset
## --------------------------------------
fname1 <- hub[["EH1040"]]
h5ls(fname1)  # "counts" is a 2D dataset

set.seed(33)
index <- list(sample(27998, 300), sample(1306127, 450))
## h5mread() is about 2x faster than h5read():
system.time(a <- h5mread::h5mread(fname1, "counts", index))
system.time(b <- h5read(fname1, "counts", index=index))
stopifnot(identical(a, b))

## Alternatively 'as.sparse=TRUE' can be used to reduce memory usage:
system.time(coo <- h5mread::h5mread(fname1, "counts", index, as.sparse=TRUE))
stopifnot(identical(a, as.array(coo)))

## The bigger the selection, the greater the speedup between
## h5read() and h5mread():
```

```
index <- list(sample(27998, 1000), sample(1306127, 1000))
## h5mread() about 4x faster than h5read() (12s vs 48s):
system.time(a <- h5mread::h5mread(fname1, "counts", index))
system.time(b <- h5read(fname1, "counts", index=index))
stopifnot(identical(a, b))

## With 'as.sparse=TRUE' (about the same speed as with 'as.sparse=FALSE'):
system.time(coo <- h5mread::h5mread(fname1, "counts", index, as.sparse=TRUE))
stopifnot(identical(a, as.array(coo)))
```

`h5mread_from_reshaped` *Read data from a virtually reshaped HDF5 dataset*

Description

An `h5mread` wrapper that reads data from a virtually reshaped HDF5 dataset.

Usage

```
h5mread_from_reshaped(filepath, name, dim, starts, noreduce=FALSE,
                      as.integer=FALSE, method=0L)
```

Arguments

`filepath`	The path (as a single string) to the HDF5 file where the dataset to read from is located, or an H5File object. Note that you must create and use an H5File object if the HDF5 file to access is stored in an Amazon S3 bucket. See ?`H5File` for how to do this. Also please note that H5File objects must NOT be used in the context of parallel evaluation at the moment.
`name`	The name of the dataset in the HDF5 file.
`dim`	A vector of dimensions that describes the virtual reshaping i.e. the reshaping that is virtually applied upfront to the HDF5 dataset to read from. Note that the HDF5 dataset is treated as read-only so never gets *effectively* reshaped, that is, the dataset dimensions encoded in the HDF5 file are not mmodified. Also please note that arbitrary reshapings are not supported. Only reshapings that reduce the number of dimensions by collapsing a group of consecutive dimensions into a single dimension are supported. For example, reshaping a 10 x 3 x 5 x 1000 array as a 10 x 15 x 1000 array or as a 150 x 1000 matrix is supported.
`starts`	A multidimensional subsetting index *with respect to the reshaped dataset*, that is, a list with one list element per dimension in the reshaped dataset. Each list element in `starts` must be a vector of valid positive indices along the corresponding dimension in the reshaped dataset. An empty vector (`integer(0)`) is accepted and indicates an empty selection along that dimension. A `NULL` is accepted and indicates a *full* selection along the dimension so has the same meaning as a missing subscript when subsetting an array-like object with `[`. (Note that for `[` a `NULL` subscript indicates an empty selection.)
`noreduce`, `as.integer`, `method`	See ?`h5mread` for a description of these arguments.

Value

An array.

See Also

- H5File objects.
- h5mread.

Examples

```
temp_h5 <- tempfile(fileext=".h5")

## ---------------------------------------------------------------------
## BASIC USAGE
## ---------------------------------------------------------------------
a1 <- array(1:350, c(10, 5, 7))
h5write(a1, temp_h5, "A1")

## Collapse the first 2 dimensions:
h5mread_from_reshaped(temp_h5, "A1", dim=c(50, 7),
                      starts=list(8:11, NULL))
h5mread_from_reshaped(temp_h5, "A1", dim=c(50, 7),
                      starts=list(8:11, NULL))

## Collapse the last 2 dimensions:
h5mread_from_reshaped(temp_h5, "A1", dim=c(10, 35),
                      starts=list(NULL, 3:11))

a2 <- array(1:150000 + 0.1*runif(150000), c(10, 3, 5, 1000))
h5write(a2, temp_h5, name="A2")

## Collapse the 2nd and 3rd dimensions:
h5mread_from_reshaped(temp_h5, "A2", dim=c(10, 15, 1000),
                      starts=list(NULL, 8:11, 999:1000))

## Collapse the first 3 dimensions:
h5mread_from_reshaped(temp_h5, "A2", dim=c(150, 1000),
                      starts=list(71:110, 999:1000))
```

---

h5writeDimnames *Write/read the dimnames of an HDF5 dataset*

Description

h5writeDimnames and h5readDimnames can be used to write/read the dimnames of an HDF5 dataset to/from the HDF5 file.

Note that h5writeDimnames is used internally by HDF5Array::writeHDF5Array(x, ..., with.dimnames=TRUE) to write the dimnames of x to the HDF5 file together with the array data.

set_h5dimnames and get_h5dimnames are low-level utilities that can be used to attach existing HDF5 datasets along the dimensions of a given HDF5 dataset, or to retrieve the names of the HDF5 datasets that are attached along the dimensions of a given HDF5 dataset.

Usage

```
h5writeDimnames(dimnames, filepath, name, group=NA, h5dimnames=NULL)
h5readDimnames(filepath, name, as.character=FALSE)

set_h5dimnames(filepath, name, h5dimnames, dry.run=FALSE)
get_h5dimnames(filepath, name)
```

Arguments

dimnames	The dimnames to write to the HDF5 file. Must be supplied as a list (possibly named) with one list element per dimension in the HDF5 dataset specified via the name argument. Each list element in dimnames must be an atomic vector or a NULL. When not a NULL, its length must equal the extent of the corresponding dimension in the HDF5 dataset.
filepath	For h5writeDimnames and h5readDimnames: The path (as a single string) to the HDF5 file where the dimnames should be written to or read from. For set_h5dimnames and get_h5dimnames: The path (as a single string) to the HDF5 file where to set or get the *h5dimnames*.
name	For h5writeDimnames and h5readDimnames: The name of the dataset in the HDF5 file for which the dimnames should be written or read. For set_h5dimnames and get_h5dimnames: The name of the dataset in the HDF5 file for which to set or get the *h5dimnames*.
group	NA (the default) or the name of the HDF5 group where to write the dimnames. If set to NA then the group name is automatically generated from name. If set to the empty string ("") then no group will be used. Except when group is set to the empty string, the names in h5dimnames (see below) must be relative to the group.
h5dimnames	For h5writeDimnames: NULL (the default) or a character vector containing the names of the HDF5 datasets (one per list element in dimnames) where to write the dimnames. Names associated with NULL list elements in dimnames are ignored and should typically be NAs. If set to NULL then the names are automatically set to numbers indicating the associated dimensions ("1" for the first dimension, "2" for the second, etc...) For set_h5dimnames: A character vector containing the names of the HDF5 datasets to attach as dimnames of the dataset specified in name. The vector must have one element per dimension in dataset name. NAs are allowed and indicate dimensions along which nothing should be attached.
as.character	Even though the dimnames of an HDF5 dataset are usually stored as datasets of type "character" (H5 datatype "H5T_STRING") in the HDF5 file, this is not a requirement. By default h5readDimnames will return them *as-is*. Set as.character to TRUE to make sure that they are returned as character vectors. See example below.
dry.run	When set to TRUE, set_h5dimnames doesn't make any change to the HDF5 file but will still raise errors if the operation cannot be done.

Value

h5writeDimnames and set_h5dimnames return nothing.

h5readDimnames returns a list (possibly named) with one list element per dimension in HDF5 dataset name and containing its dimnames retrieved from the file.

get_h5dimnames returns a character vector containing the names of the HDF5 datasets that are currently set as the dimnames of the dataset specified in name. The vector has one element per dimension in dataset name. NAs in the vector indicate dimensions along which nothing is set.

See Also

- writeHDF5Array in the **HDF5Array** package for a high-level function to write an array-like object and its dimnames to an HDF5 file.
- h5write in the **rhdf5** package that h5writeDimnames uses internally to write the dimnames to the HDF5 file.
- h5mread in this package (**h5mread**) that h5readDimnames uses internally to read the dimnames from the HDF5 file.
- h5ls to list the content of an HDF5 file.

Examples

```
## ---------------------------------------------------------------------
## BASIC EXAMPLE
## ---------------------------------------------------------------------
library(rhdf5)  # for h5write()

m0 <- matrix(1:60, ncol=5)
colnames(m0) <- LETTERS[1:5]

h5file <- tempfile(fileext=".h5")
h5write(m0, h5file, "M0")  # h5write() ignores the dimnames
h5ls(h5file)

h5writeDimnames(dimnames(m0), h5file, "M0")
h5ls(h5file)

get_h5dimnames(h5file, "M0")
h5readDimnames(h5file, "M0")

## Reconstruct 'm0' from HDF5 file:
m1 <- h5mread(h5file, "M0")
dimnames(m1) <- h5readDimnames(h5file, "M0")
stopifnot(identical(m0, m1))

## Create an HDF5Array object that points to HDF5 dataset M0:
HDF5Array::HDF5Array(h5file, "M0")

## Sanity checks:
stopifnot(
  identical(dimnames(m0), h5readDimnames(h5file, "M0")),
  identical(dimnames(m0), dimnames(HDF5Array::HDF5Array(h5file, "M0")))
)

## ---------------------------------------------------------------------
## SHARED DIMNAMES
## ---------------------------------------------------------------------
## If a collection of HDF5 datasets share the same dimnames, the
## dimnames only need to be written once in the HDF5 file. Then they
## can be attached to the individual datasets with set_h5dimnames():
```

```
h5write(array(runif(240), c(12, 5:4)), h5file, "A1")
set_h5dimnames(h5file, "A1", get_h5dimnames(h5file, "M0"))
get_h5dimnames(h5file, "A1")
h5readDimnames(h5file, "A1")
HDF5Array::HDF5Array(h5file, "A1")

h5write(matrix(sample(letters, 60, replace=TRUE), ncol=5), h5file, "A2")
set_h5dimnames(h5file, "A2", get_h5dimnames(h5file, "M0"))
get_h5dimnames(h5file, "A2")
h5readDimnames(h5file, "A2")
HDF5Array::HDF5Array(h5file, "A2")

## Sanity checks:
stopifnot(identical(dimnames(m0), h5readDimnames(h5file, "A1")[1:2]))
stopifnot(identical(dimnames(m0), h5readDimnames(h5file, "A2")))

## ---------------------------------------------------------------------
## USE h5writeDimnames() AFTER A CALL TO writeHDF5Array()
## ---------------------------------------------------------------------
## After calling writeHDF5Array(x, ..., with.dimnames=FALSE) the
## dimnames on 'x' can still be written to the HDF5 file by doing the
## following:

## 1. Write 'm0' to the HDF5 file and ignore the dimnames (for now):
HDF5Array::writeHDF5Array(m0, h5file, "M2", with.dimnames=FALSE)

## 2. Use h5writeDimnames() to write 'dimnames(m0)' to the file and
##    associate them with the "M2" dataset:
h5writeDimnames(dimnames(m0), h5file, "M2")

## 3. Use the HDF5Array() constructor to make an HDF5Array object that
##    points to the "M2" dataset:
HDF5Array::HDF5Array(h5file, "M2")

## Note that at step 2. you can use the extra arguments of
## h5writeDimnames() to take full control of where the dimnames
## should be stored in the file:
HDF5Array::writeHDF5Array(m0, h5file, "M3", with.dimnames=FALSE)
h5writeDimnames(dimnames(m0), h5file, "M3",
                group="a_secret_place", h5dimnames=c("NA", "M3_dim2"))
h5ls(h5file)
## h5readDimnames() and HDF5Array() still "find" the dimnames:
h5readDimnames(h5file, "M3")
HDF5Array::HDF5Array(h5file, "M3")

## Sanity checks:
stopifnot(
  identical(dimnames(m0), h5readDimnames(h5file, "M3")),
  identical(dimnames(m0), dimnames(HDF5Array::HDF5Array(h5file, "M3")))
)

## ---------------------------------------------------------------------
## STORE THE DIMNAMES AS NON-CHARACTER TYPES
## ---------------------------------------------------------------------
HDF5Array::writeHDF5Array(m0, h5file, "M4", with.dimnames=FALSE)
dimnames <- list(1001:1012, as.raw(11:15))
h5writeDimnames(dimnames, h5file, "M4")
```

```
h5ls(h5file)

h5readDimnames(h5file, "M4")
h5readDimnames(h5file, "M4", as.character=TRUE)

## Sanity checks:
stopifnot(identical(dimnames, h5readDimnames(h5file, "M4")))
dimnames(m0) <- dimnames
stopifnot(identical(
    dimnames(m0),
    h5readDimnames(h5file, "M4", as.character=TRUE)
))
```

Index