*Journal of Statistical Software*

*August 2013, Volume 54, Issue 6.* *http://www.jstatsoft.org/*

# Sustainable, Extensible Documentation Generation Using inlinedocs

**Toby Dylan Hocking**
INRIA Paris

**Thomas Wutzler**
MPI-BGC Jena

**Keith Ponting**
Aurix Ltd.

**Philippe Grosjean**
University of Mons

**Abstract**

This article presents **inlinedocs**, an R package for generating documentation from comments. The concept of structured, interwoven code and documentation has existed for many years, but existing systems that implement this for the R programming language do not tightly integrate with R code, leading to several drawbacks. This article attempts to address these issues and presents 2 contributions for documentation generation for the R community. First, we propose a new syntax for inline documentation of R code within comments adjacent to the relevant code, which allows for highly readable and maintainable code and documentation. Second, we propose an extensible system for parsing these comments, which allows the syntax to be easily augmented.

*Keywords*: R, Rd, documentation, documentation generation, literate programming.

## 1. Introduction

In this article, we present **inlinedocs**, an R package which allows R documentation to be written in comments. The standard way to distribute R code is in a package along with Rd files that document the code (R Core Team 2013). There are several existing methods for documenting a package by writing R comments, which are later processed and converted into standard Rd files. We first review these efforts, emphasizing the key issues that justify the introduction of a new package like **inlinedocs**.

### 1.1. Existing documentation generation systems for R

For report generation and literate programming, the mature `Sweave` (Leisch 2003) format allows integration of R code and results within LaTeX documents (Lamport 1986). However, the goal of **inlinedocs** is different. It aims for integration of documentation inside of R code files, to generate Rd files using R code and markup in R comments. Thus for **inlinedocs** we need to extract the documentation specified in R code, and the `Sweave` system can not be easily applied to this parsing task.

The `package.skeleton` function that ships with base R is intended to ease the generation of Rd files from R code. After specifying some input R code files or objects to use for the package, it produces some minimal documentation that must be completed using a text editor. Although `package.skeleton` is sufficient for creating small packages that are published once and forgotten, it offers little help for continued maintenance of packages for which Rd files are frequently updated.

The other existing approaches, **Rdoc** (Bengtsson 2010) and **roxygen** (Danenberg 2011), attempt to address this sustainability problem using Rd generation from comments in R code. The documentation is thus written closer to the code it documents, which is easier to maintain. These packages are a step toward seamless integration of code and documentation, but they have three major drawbacks:

1. They only use comments to generate documentation, ignoring the information already defined in the code. This is particularly problematic for documenting function arguments, which requires the repetition of the argument names in the function definition and the documentation. This repetition is a possible source of disagreement between code and documentation if both are not simultaneously updated.

2. The documentation for an object appears in comments above its definition. These comment blocks can grow to be quite large, and thus they tend to be far away from the relevant code.

3. Examples are defined either in comments or in supplementary R code files. Examples in comments are not easy to test and debug with the R interpreter, and supplementary R code files reintroduce the separation of code and documentation that these tools are supposed to eliminate.

There are many tools that accomplish documentation alongside code in other programming languages. Notable examples include docstrings in Lisp and Python, **Javadoc** for Java, and **Doxygen**, which supports several languages (Wikipedia 2013). These systems use large comments in headers, and do not support R. In contrast, **inlinedocs** is designed for R packages, uses smaller comments alongside the code, and exploits the code structure to reduce the need to repeat information in the documentation.

### 1.2. Documentation using inline comments

The **inlinedocs** package addresses the aforementioned issues by proposing a new syntax for inline documentation of R packages. Using **inlinedocs**, one writes documentation in comments right next to the relevant code, and examples in the `ex` attribute of the relevant object. By

design, **inlinedocs** exploits the structure of the R code so that only minimal documentation comments are required, reducing duplication and simplifying code maintenance.

The remainder of the article is organized as follows. In Section 2, we discuss the details of the **inlinedocs** syntax for writing documentation in R comments. In Section 3, we discuss the design and implementation of **inlinedocs**, and explain how the syntax can be extended. In Section 4, we conclude and offer some ideas for future improvements. Finally, in Appendix A, we show a concrete application by porting the base `apply` function to **inlinedocs**.

## 2. inlinedocs syntax for inline documentation of R packages

The main idea of **inlinedocs** is to document an R object using `###` and `##<<` comments directly adjacent to its source code. Furthermore, **inlinedocs** allows documentation wherever it is most relevant in the code using `##section<<` comments. These special comment strings are designed to work well with the default behavior of common editing environments, such as **Emacs** with the **Emacs Speaks Statistics** (Rossini, Heiberger, Sparapani, Maechler, and Hornik 2004) add-on package:

- `###` is aligned to the left margin, providing maximum space for comment text.
- `##<<` is aligned with the start of adjacent code lines, so that comments using this form in the middle of a function do not obscure the code structure.

The following sections illustrate common usage of **inlinedocs** comments through **fermat**, an example package inspired by the **roxygen** vignette (Danenberg 2011). The examples were processed and checked for validity using **inlinedocs** version 2013.9.3. For brevity, only the most frequently used **inlinedocs** features will be discussed, and the reader is directed to the **inlinedocs** web site for complete documentation: http://inlinedocs.R-Forge.R-project.org/.

### 2.1. Documenting function arguments and return values

The following example demonstrates the minimal documentation a package author should provide for every function. Note that the location of white space, brackets, default arguments and commas is quite flexible.

```
fermat.test <- function
### Test an integer for primality using Fermat's Little Theorem.
(n ##<< The integer to test.
 ){
  a <- floor(runif(1, min = 1, max = n))
  a^n %% n == a
### Whether the integer passes the Fermat test for a randomized
### \eqn{0 < a < n}.
}
```

The comments correspond to the following sections of the `fermat.test.Rd` file:

- `###` comments following the line of `function` form the `description` section.

- For each argument, an item is created in the `arguments` section using a `##<<` comment on the same line.
- `###` comments at the end of the function form the `value` section.

By default, `name`, `alias` and `title` Rd sections are set to the function name, so this minimal level of documentation is enough to make a working package that passes `R CMD check` with no errors or warnings.

## 2.2. Inline titles, arguments, and other sections

The following example shows some optional **inlinedocs** comments that allow detailed and flexible specification of Rd files.

```
is.pseudoprime <- function # Check an integer for pseudo-primality.
### A number is pseudo-prime if it is probably prime, the basis of
### which is the probabalistic Fermat test; if it passes two such
### tests, the chances are better than 3 out of 4 that \eqn{n} is
### prime.
##references<< Abelson, Hal; Jerry Sussman, and Julie
##Sussman. Structure and Interpretation of Computer
##Programs. Cambridge: MIT Press, 1984.
(n,    ##<< Integer to test for pseudoprimality.
 times
### Number of Fermat tests to perform. More tests are more likely to
### give accurate results.
 ){
  if(times == 0) TRUE
  ##seealso<< \code{\link{fermat.test}}
  else if(fermat.test(n)) is.pseudoprime(n, times - 1)
  else FALSE
### logical TRUE if n is probably prime.
}
```

On the first line, the `#` comment specifies the title. On the lines after an argument, `###` comments specify its documentation. This is a useful alternative to inline `##<<` comments for longer, multi-line documentation of function arguments.

A `##section<<` comment can be used anywhere within a function, for any documentation section except `examples`, which is handled in a special manner as shown below in section 2.3. In each comment, arbitrary Rd may be written, as shown in the `##seealso<<` section above. Each `##section<<` may occur several times in the documentation for a single object. Such multiple occurrences are normally concatenated as separate paragraphs, but special processing is applied to match the intended use of the following documentation sections:

- `title` sections are concatenated into a single line.
- `description` sections should be brief, so are concatenated into a single paragraph.
- `alias` contents are split to give one alias per line of text.

- `keyword` contents are split at white space, each generating a separate `\keyword` entry.

The `###` and `##<<` documentation styles may be freely mixed. In general, `###` or `#` lines are processed first, followed by any corresponding `##<<` or `##section<<` comments. Section 3 will explain in more detail how comments are processed.

### 2.3. Examples and named lists

The following code demonstrates inline documentation of named lists, and the preferred method of writing examples:

```
try.several.times <- structure(function
### Test an integer for primality using different numbers of tests.
(n,    ##<< integer to test for primality.
 times ##<< vector of number of tests to try.
 ){
  is.prime <- sapply(times, function(t) is.pseudoprime(n, t))
  ##value<< data.frame with columns:
  data.frame(times, ##<< number of Fermat tests.
             is.prime, ##<< TRUE if probably prime
             n) ##<< Integer tested.
  ##end<<
},ex=function(){
  try.several.times(6, 1:5)
  try.several.times(5, 1:5)
})
```

On the final lines of the function definition, a `##value<<` comment allows documentation of lists or data frames using the names defined in the code. The entries are documented using `##<<` in the same way as function arguments, and this even works for nested lists. The `##end<<` comment closes the return value documentation block.

The examples are written using `structure` to put them in the `ex` attribute as the body of a function without arguments. This method for documenting examples was motivated by the desire to express examples in R code rather than in R comments, to keep the examples close to the object definition, and to avoid repetition of the object name. When examples are in R code, they are easily transferred to the R interpreter, and thus are easy to debug. Furthermore, when examples are written close to the object definition, it is easy to keep examples up to date and informative.

An alternative is to use `attr(try.several.times, "ex") <- function(){`*`code`*`}` later in the code. However, we prefer using `structure` since it keeps the examples near the object definition, and avoids repetition of the object name.

The simplicity of adding examples and generating a package using **inlinedocs** also allows for routine regression testing of functions with very little extra work. Even for small collections of functions, one can use `R CMD check` to run the examples and optionally check the output with reference output.

## 2.4. Documenting classes and methods

S3 methods may be defined using plain R, or using `setConstructorS3` and `setMethodS3` from the **R.oo** package (Bengtsson 2003). The **inlinedocs** package detects S3 methods using `utils::getKnownS3generics` and `utils::findGeneric`, and updates the generated documentation automatically. S4 class declarations using the `setClass` function are also supported. The following example is from the source of **inlinedocs**:

```
setClass("DocLink", # Link documentation among related functions
### The \code{DocLink} class provides the basis for hooking together
### documentation of related classes/functions/objects. The aim is that
### documentation sections missing from the child are inherited from
### the parent class.
    representation(name = "character", ##<< name of object
                   created = "character", ##<< how created
                   parent = "character", ##<< parent class or NA
                   code = "character", ##<< actual source lines
                   description = "character") ##<< preceding description
         )
```

The inheritance referred to in this example is designed to avoid the need for repetitive documentation when defining a class hierarchy. The argument descriptions and other documentation sections default to those defined in the parent class. At present it only functions when all the definitions are within a single source file and this "documentation inheritance" is strictly linear within the file.

## 2.5. `package.skeleton.dx` for generating Rd files

The main function that the **inlinedocs** package provides is `package.skeleton.dx`, which generates Rd files for a package, and should be run before `R CMD build`. For example, `package.skeleton.dx("fermat")` processes R code found in `fermat/R`, and generates Rd files in `fermat/man` for each object in the package. Documentation is generated even for objects that are not exported. The generated Rd files should be treated as object files, since any edits will be overwritten the next time the Rd files are generated.

Package authors with existing Rd files will have to convert them to **inlinedocs** comments manually. However, for new adopters of inlinedocs, it is possible to mix static Rd files and **inlinedocs** in the same package. For example, the following code specifies that `file1.Rd` and `file2.Rd` are static Rd files and so should not be generated by **inlinedocs**:

```
my.parsers <- c(default.parsers, list(do.not.generate("file1", "file2")))
package.skeleton.dx(parsers = my.parsers)
```

By design, **inlinedocs** is incapable of generating Rd files that document multiple objects, but package authors may write these Rd files manually using this mechanism.

More generally, the `parsers` argument to `package.skeleton.dx` should be a list of parser functions. Next, in Section 3, we explain how to write parser functions.

# 3. inlinedocs system of extensible documentation generators

The previous section explains how to write inline documentation in R code using the standard **inlinedocs** syntax, then process it to generate Rd files using `package.skeleton.dx`. For most users of **inlinedocs** this should be sufficient for everyday use.

For users who wish to extend the syntax of **inlinedocs**, here we explain the internal organization of the **inlinedocs** package. The two central concepts are parser functions and documentation lists. Parser functions are used to extract documentation from R code, which is then stored in a documentation list before writing Rd files.

## 3.1. Documentation lists store the structured content of Rd files

A documentation list is a list of lists that describes all of the documentation to write to the Rd files. The elements of the outer list correspond to Rd files in the package, and the elements of the inner list correspond to tags in an Rd file. For example, consider the code in Figure 1 and its corresponding documentation list.

Parser functions examine the lines of code on the left that define the functions, and return the documentation list of tags shown on the right. This list describes the tags in the Rd files that will be written for these functions. The names of the outer list specify the Rd file, and the names of the inner list specify the Rd tag.

To store parsed documentation, another intermediate representation that we considered instead of the documentation list was the `"Rd"` object, as described by Murdoch and Urbanek (2009). It is a recursive structure of lists and character strings, which is similar to the documentation list format of **inlinedocs**. However, we chose the documentation list format since it allows rapid development of parser functions which are straightforward to read, write, and modify.

R code

```
give.me.a.break <- function
### Create some line breaks.
(times=1,
### The number of line breaks.
 collapse=""
### String to paste in between.
 ){
  paste(rep("\n",times),
        collapse=collapse)
### Character vector of length 1.
}

give.me.five <- function
(times=1 ##<< the number of fives
 ){
  rep(5,times)
### a vector of fives
}
```

Documentation list

```
List of 2
 $ give.me.a.break:List of 5
  ..$ description   : chr "Create some line breaks."
  ..$ item{times}   : chr "The number of line breaks."
  ..$ item{collapse}: chr "String to paste in between."
  ..$ value         : chr "Character vector of length 1."
  ..$ title         : chr "give me a break"
 $ give.me.five   :List of 3
  ..$ value      : chr "a vector of fives"
  ..$ item{times}: chr "the number of fives"
  ..$ title      : chr "give me five"
```

Figure 1: Example for R code and its corresponding documentation list.

| Argument | Description |
|---|---|
| `code` | Character vector of all lines of R code in the package. |
| `env` | Environment in which the lines of code are evaluated. |
| `objs` | List of all R objects defined in the package. |
| `docs` | Documentation list from previous parser functions. |
| `desc` | 1-row matrix of `DESCRIPTION` metadata, as read by `read.dcf`. |

Table 1: Arguments that are passed to every parser function.

### 3.2. Structure of a parser function and `forall`/`forfun`

The job of a parser function is to return a documentation list for a package. To do this, a parser function requires knowledge of what is defined in the package, so the arguments in Table 1 are supplied by **inlinedocs**.

The R code files in the package are concatenated into `code` and then parsed into `objs`, and the `DESCRIPTION` metadata is available as `desc`. These arguments allow complete flexibility in the construction of parser functions that take apart the package and extract meaningful documentation lists. In addition, the `docs` argument allows for checking of what previous parser functions have already extracted.

In principle, one could write a single monolithic parser function that extracts all tags for all Rd files for the package, then returns the entire documentation list. However, in practice, this results in one unwieldly parser function that does many things and is hard to maintain. A simpler strategy is to write several smaller parser functions, each of which produces an inner documentation list for a specific Rd file, such as the following:

```
title.from.firstline <- function (src, ...) {
  first <- src[1]
  if (grepl("#", first)) {
    list(title = gsub("[^#]*#\\s*(.*)", "\\1", first, perl = TRUE))
  } else list()
}
```

This function takes `src`, a character vector of R code lines that define a function, and looks for a comment on the first line. If there is a comment, `title.from.firstline` returns the comment as the title in an inner documentation list. This a very simple and readable way to define a parser function.

But how does this parser function get access to the `src` argument, the source code of an individual function? We introduce the `forall` and `forfun` functions, which transform an object-specific parser function such as `title.from.firstline` to a parser function that can work on an entire package. These functions examine the `objs` and `docs` arguments, and call the object-specific parser function on each object in turn. The `forfun` function applies to every function in the package, whereas the `forall` function applies to every documentation object in the package.

Thus, when using a parser function such as `forfun(title.from.firstline)`, the additional arguments in Table 2 can be used in the definition of `title.from.firstline`, in addition to the arguments in Table 1 that are passed to every parser function.

| Argument | Description |
|---|---|
| o | The R object. |
| name | The name of the object. |
| src | The source code lines that define the object. |
| doc | The inner documentation list already constructed for this object. |

Table 2: Arguments passed to each parser function, when used with `forall` or `forfun`.

This design choice of **inlinedocs** allows the development of modular parser functions. For example, there is one parser function for `###` comments, another for `##<<` comments, another for adding the `author` tag using the Author line of the `DESCRIPTION` file, etc. Each of these parser functions is relatively small and thus easy to maintain.

### 3.3. Extending the syntax with custom parser functions

The `parsers` argument to `package.skeleton.dx` specifies the list of parser functions used to create the Documentation List. The parser functions will be called in sequence, and their results will be combined to form the final documentation list that will be used to write Rd files. Thus, the **inlinedocs** syntax can be extended by simply writing new parser functions. To illustrate how **inlinedocs** may be extended using this mechanism, consider this parser function, which extracts documentation from single-`#` comments:

```
simple <- function (src, ...) {# a simple parser function
  #item{src} character vector of R source code.
  noquotes <- gsub("([\"'`]).*\\1", "", src)
  comments <- grep("#", noquotes, value = TRUE)
  doc.pattern <- "[^#]*#([^ ]*) (.*)"
  tags <- gsub(doc.pattern, "\\1", comments)
  docs <- as.list( gsub(doc.pattern, "\\2", comments) )
  names(docs) <- tags
  #value all the tags with a single pound sign.
  docs[ tags != "" ]
}
```

We can then define a list of custom parser functions as follows:

```
simple.parsers <- list(forfun(title.from.firstline), forfun(simple))
```

These custom parser functions can be used to extract the following documentation list from the definition above of `simple`:

```
List of 1
 $ simple:List of 3
  ..$ title    : chr "a simple parser function"
  ..$ item{src}: chr "character vector of R source code."
  ..$ value    : chr "all the tags with a single pound sign."
```

In conclusion, a new syntax for inline documentation can be quickly specified using parser functions, and then **inlinedocs** takes care of the details of converting the documentation list to Rd files.

## 4. Conclusions and future work

We have presented **inlinedocs**, which is both a new syntax for inline documentation of R packages, and an extensible system for parsing this syntax and generating Rd files. It has been in development since 2009 on R-Forge (Theußl and Zeileis 2009) at `http://inlinedocs.R-Forge.R-project.org/`, has seen several releases on CRAN at `http://CRAN.R-project.org/package=inlinedocs`, and has been used to generate documentation for itself and several other R packages. In practice, we have found that **inlinedocs** significantly reduces the amount of time it takes to create a package that passes `R CMD check`. In addition, **inlinedocs** facilitates rapid package updates since the documentation is written in comments right next to the relevant code.

For quality assurance, we currently have implemented unit tests for documentation lists, which assure that parser functions work as described. We also have unit tests which ensure that the generated Rd passes `R CMD check` without errors or warnings.

A potential criticism of **inlinedocs** is that excessive inline comments may obscure the meaning of code. Indeed, this is a design choice, and can be seen as a bug, but we prefer to see it as a feature: the documentation is always near the object definition, for quick reference.

Currently, the **inlinedocs** package relies on the `srcref` attribute of a function to access its definition. For S4 classes, we use `parse` on the source files. In the future, we would like to develop parser functions that use this approach to extract documentation for S4 methods and reference classes, which are currently unsupported in **inlinedocs**.

For the future, we would like to make use of Rd manipulation tools such as `parse_Rd`, as described by Murdoch (2010). For package authors who want to convert Rd files to inlinedocs comments, we may be able to use `parse_Rd` to develop a converter that takes R source code and Rd, then outputs R code with documentation in comments.

Also, it would be advantageous to have functions for converting documentation lists to and from `Rd` objects. For example, after converting an inner documentation list to an `Rd` object, we could use its `print` method to write the Rd file. This could be simpler than the current system of starting from the Rd files from `package.skeleton` and then doing find and replace. Furthermore, a converter from `Rd` objects to documentation lists would permit unit tests for the content of the Rd generated by **inlinedocs**.

Finally, we thank a reviewer for an idea for integrating **inlinedocs** into the `R CMD build` process. Currently, the `package.skeleton.dx` function must be run by the package author before each `R CMD build`. Documentation generation could be integrated into the package building process if package authors could write a `.onBuild` function that would be run prior to each package build. Packages that use **inlinedocs** could include a call to `package.skeleton.dx` in the `.onBuild` function for automatic documentation generation prior to each package build.

## References

Bengtsson H (2003). "The **R.oo** package – Object-Oriented Programming with References Using Standard R Code." In K Hornik, F Leisch, A Zeileis (eds.), *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*. Vienna, Austria. ISSN 1609-395X.

Bengtsson H (2010). "**Aroma** Project Developers' Corner." URL http://www.aroma-project.org/developers.

Danenberg P (2011). "**roxygen** Vignette." Version 2011-12-23, URL http://CRAN.R-project.org/package=roxygen.

Lamport L (1986). *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts.

Leisch F (2003). "**Sweave**, Part II: Package Vignettes." *R News*, **3**(2), 21–24. URL http://CRAN.R-project.org/doc/Rnews/.

Murdoch D (2010). *Parsing Rd Files*. URL http://developer.R-project.org/parseRd.pdf.

Murdoch D, Urbanek S (2009). "The New R Help System." *The R Journal*, **1**(2), 60–65. URL http://journal.R-project.org/archive/2009-2/RJournal_2009-2_Murdoch+Urbanek.pdf.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Rossini AJ, Heiberger RM, Sparapani RA, Maechler M, Hornik K (2004). "**Emacs Speaks Statistics**: A Multiplatform, Multipackage Development Environment for Statistical Analysis." *Journal of Computational and Graphical Statistics*, **13**(1), 247–261.

Theußl S, Zeileis A (2009). "Collaborative Software Development Using R-Forge." *The R Journal*, **1**(1), 9–14. URL http://journal.R-project.org/2009-1/RJournal_2009-1_Theussl+Zeileis.pdf.

Wikipedia (2013). "Comparison of Documentation Generators — Wikipedia, The Free Encyclopedia." URL http://en.wikipedia.org/wiki/Comparison_of_documentation_generators.

# A. The base function `apply` converted to inlinedocs

In this appendix, we show a concrete application by converting the source and documentation of the base function `apply` to **inlinedocs**.

## A.1. Source and inline documentation

We use the following source code and comments to define the `apply` function and its documentation.

Characters that are special in Rd do not need to be escaped in **inlinedocs**, such as `%` in the documentation of the `FUN` argument.

```
apply <- structure(function # Apply Functions Over Array Margins
### Returns a vector or array or list of values obtained by applying a
### function to margins of an array or matrix.
(X, ##<< an array, including a matrix.
 MARGIN,
### a vector giving the subscripts which the function will be applied
### over. E.g., for a matrix \code{1} indicates rows, \code{2}
### indicates columns, \code{c(1, 2)} indicates rows and
### columns. Where \code{X} has named dimnames, it can be a character
### vector selecting dimension names.
 FUN,
### the function to be applied: see \sQuote{Details}. In the case of
### functions like \code{+}, \code{%*%}, etc., the function name
### must be backquoted or quoted.
 ... ##<< optional arguments to \code{FUN}.
 ){
  ##keyword<< iteration array

  ##details<< \code{FUN} is found by a call to \code{\link{match.fun}}
  ## and typically is either a function or a symbol (e.g. a backquoted
  ## name) or a character string specifying a function to be searched
  ## for from the environment of the call to \code{apply}.
  FUN <- match.fun(FUN)

  ##details<< If \code{X} is not an array but an object of a class
  ## with a non-null \code{\link{dim}} value (such as a data frame),
  ## \code{apply} attempts to coerce it to an array via
  ## \code{as.matrix} if it is two-dimensional (e.g., a data frame) or
  ## via \code{as.array}.
  dl <- length(dim(X))
  if(!dl) stop("dim(X) must have a positive length")
  if(is.object(X))
    X <- if(dl == 2L) as.matrix(X) else as.array(X)

### Body of function contains no inline docs and is omitted for brevity.

  return(ans)
### If each call to \code{FUN} returns a vector of length \code{n},
### then \code{apply} returns an array of dimension \code{c(n,
### dim(X)[MARGIN])} if \code{n > 1}.  If \code{n} equals \code{1},
```

```
### \code{apply} returns a vector if \code{MARGIN} has length 1 and an
### array of dimension \code{dim(X)[MARGIN]} otherwise.  If \code{n}
### is \code{0}, the result has length 0 but not necessarily the
### \sQuote{correct} dimension.
###
### If the calls to \code{FUN} return vectors of different lengths,
### \code{apply} returns a list of length \code{prod(dim(X)[MARGIN])}
### with \code{dim} set to \code{MARGIN} if this has length greater
### than one.
###
### In all cases the result is coerced by \code{\link{as.vector}} to
### one of the basic vector types before the dimensions are set, so
### that (for example) factor results will be coerced to a character
### array.
},ex=function(){
  ## Compute row and column sums for a matrix:
  x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
  dimnames(x)[[1]] <- letters[1:8]
  apply(x, 2, mean, trim = .2)
  col.sums <- apply(x, 2, sum)
  row.sums <- apply(x, 1, sum)
  rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))

  stopifnot( apply(x, 2, is.vector))

  ## Sort the columns of a matrix
  apply(x, 2, sort)

  ##- function with extra args:
  cave <- function(x, c1, c2) c(mean(x[c1]), mean(x[c2]))
  apply(x,1, cave,  c1="x1", c2=c("x1","x2"))

  ma <- matrix(c(1:4, 1, 6:8), nrow = 2)
  ma
  apply(ma, 1, table)  #--> a list of length 2
  apply(ma, 1, stats::quantile)# 5 x n matrix with rownames

  stopifnot(dim(ma) == dim(apply(ma, 1:2, sum)))

  ## Example with different lengths for each call
  z <- array(1:24, dim=2:4)
  zseq <- apply(z, 1:2, function(x) seq_len(max(x)))
  zseq         ## a 2 x 3 matrix
  typeof(zseq) ## list
  dim(zseq) ## 2 3
  zseq[1,]
  apply(z, 3, function(x) seq_len(max(x)))
  ## a list without a dim attribute
})
```

## A.2. Documentation list

Running the **inlinedocs** default parser functions on the source code results in the following documentation list, which summarizes the extracted documentation.

Again, note in `item{FUN}` that `%` is not escaped in the documentation list. It is the job of **inlinedocs** to convert this documentation list to valid Rd, so the parser function programmer does not need to worry about escaping special characters.

```
$apply
$apply$description
[1] "Returns a vector or array or list of values obtained by applying a"
[2] "function to margins of an array or matrix."

$apply$`item{MARGIN}`
[1] "a vector giving the subscripts which the function will be applied"
[2] "over. E.g., for a matrix \\code{1} indicates rows, \\code{2}"
[3] "indicates columns, \\code{c(1, 2)} indicates rows and"
[4] "columns. Where \\code{X} has named dimnames, it can be a character"
[5] "vector selecting dimension names."

$apply$`item{FUN}`
[1] "the function to be applied: see \\sQuote{Details}. In the case of"
[2] "functions like \\code{+}, \\code{%*%}, etc., the function name"
[3] "must be backquoted or quoted."

$apply$value
 [1] "If each call to \\code{FUN} returns a vector of length \\code{n},"
 [2] "then \\code{apply} returns an array of dimension \\code{c(n,"
 [3] "dim(X)[MARGIN])} if \\code{n > 1}.  If \\code{n} equals \\code{1},"
 [4] "\\code{apply} returns a vector if \\code{MARGIN} has length 1 and an"
 [5] "array of dimension \\code{dim(X)[MARGIN]} otherwise.  If \\code{n}"
 [6] "is \\code{0}, the result has length 0 but not necessarily the"
 [7] "\\sQuote{correct} dimension."
 [8] ""
 [9] "If the calls to \\code{FUN} return vectors of different lengths,"
[10] "\\code{apply} returns a list of length \\code{prod(dim(X)[MARGIN])}"
[11] "with \\code{dim} set to \\code{MARGIN} if this has length greater"
[12] "than one."
[13] ""
[14] "In all cases the result is coerced by \\code{\\link{as.vector}} to"
[15] "one of the basic vector types before the dimensions are set, so"
[16] "that (for example) factor results will be coerced to a character"
[17] "array."

$apply$`item{X}`
[1] "an array, including a matrix."

$apply$`item{\dots}`
[1] "optional arguments to \\code{FUN}."

$apply$keyword
[1] "iteration}"      "\\keyword{array"
```

```
$apply$details
 [1] "\\code{FUN} is found by a call to \\code{\\link{match.fun}}"
 [2] "and typically is either a function or a symbol (e.g. a backquoted"
 [3] "name) or a character string specifying a function to be searched"
 [4] "for from the environment of the call to \\code{apply}."
 [5] ""
 [6] "If \\code{X} is not an array but an object of a class"
 [7] "with a non-null \\code{\\link{dim}} value (such as a data frame),"
 [8] "\\code{apply} attempts to coerce it to an array via"
 [9] "\\code{as.matrix} if it is two-dimensional (e.g., a data frame) or"
[10] "via \\code{as.array}."

$apply$title
[1] "Apply Functions Over Array Margins"

$apply$examples
 [1] ""
 [2] "## Compute row and column sums for a matrix:"
 [3] "x <- cbind(x1 = 3, x2 = c(4:1, 2:5))"
 [4] "dimnames(x)[[1]] <- letters[1:8]"
 [5] "apply(x, 2, mean, trim = .2)"
 [6] "col.sums <- apply(x, 2, sum)"
 [7] "row.sums <- apply(x, 1, sum)"
 [8] "rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))"
 [9] ""
[10] "stopifnot( apply(x, 2, is.vector))"
[11] ""
[12] "## Sort the columns of a matrix"
[13] "apply(x, 2, sort)"
[14] ""
[15] "##- function with extra args:"
[16] "cave <- function(x, c1, c2) c(mean(x[c1]), mean(x[c2]))"
[17] "apply(x,1, cave,  c1=\"x1\", c2=c(\"x1\",\"x2\"))"
[18] ""
[19] "ma <- matrix(c(1:4, 1, 6:8), nrow = 2)"
[20] "ma"
[21] "apply(ma, 1, table)  #--> a list of length 2"
[22] "apply(ma, 1, stats::quantile)# 5 x n matrix with rownames"
[23] ""
[24] "stopifnot(dim(ma) == dim(apply(ma, 1:2, sum)))"
[25] ""
[26] "## Example with different lengths for each call"
[27] "z <- array(1:24, dim=2:4)"
[28] "zseq <- apply(z, 1:2, function(x) seq_len(max(x)))"
[29] "zseq         ## a 2 x 3 matrix"
[30] "typeof(zseq) ## list"
[31] "dim(zseq) ## 2 3"
[32] "zseq[1,]"
[33] "apply(z, 3, function(x) seq_len(max(x)))"
[34] "## a list without a dim attribute"
```

### A.3. Generated Rd

The Rd produced by **inlinedocs** is shown below. In particular, note that the % characters have been correctly escaped.

```
\name{apply}
\alias{apply}
\title{Apply Functions Over Array Margins}
\description{Returns a vector or array or list of values obtained by applying a
function to margins of an array or matrix.}
\usage{apply(X, MARGIN, FUN, ...)}
\arguments{
  \item{X}{an array, including a matrix.}
  \item{MARGIN}{a vector giving the subscripts which the function will be applied
over. E.g., for a matrix \code{1} indicates rows, \code{2}
indicates columns, \code{c(1, 2)} indicates rows and
columns. Where \code{X} has named dimnames, it can be a character
vector selecting dimension names.}
  \item{FUN}{the function to be applied: see \sQuote{Details}. In the case of
functions like \code{+}, \code{\%*\%}, etc., the function name
must be backquoted or quoted.}
  \item{\dots}{optional arguments to \code{FUN}.}
}
\details{\code{FUN} is found by a call to \code{\link{match.fun}}
and typically is either a function or a symbol (e.g. a backquoted
name) or a character string specifying a function to be searched
for from the environment of the call to \code{apply}.

If \code{X} is not an array but an object of a class
with a non-null \code{\link{dim}} value (such as a data frame),
\code{apply} attempts to coerce it to an array via
\code{as.matrix} if it is two-dimensional (e.g., a data frame) or
via \code{as.array}.}
\value{If each call to \code{FUN} returns a vector of length \code{n},
then \code{apply} returns an array of dimension \code{c(n,
dim(X)[MARGIN])} if \code{n > 1}.  If \code{n} equals \code{1},
\code{apply} returns a vector if \code{MARGIN} has length 1 and an
array of dimension \code{dim(X)[MARGIN]} otherwise.  If \code{n}
is \code{0}, the result has length 0 but not necessarily the
\sQuote{correct} dimension.

If the calls to \code{FUN} return vectors of different lengths,
\code{apply} returns a list of length \code{prod(dim(X)[MARGIN])}
with \code{dim} set to \code{MARGIN} if this has length greater
than one.

In all cases the result is coerced by \code{\link{as.vector}} to
one of the basic vector types before the dimensions are set, so
that (for example) factor results will be coerced to a character
array.}

\author{Toby Dylan Hocking}
```

```
\examples{
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
apply(x, 2, mean, trim = .2)
col.sums <- apply(x, 2, sum)
row.sums <- apply(x, 1, sum)
rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))

stopifnot( apply(x, 2, is.vector))

## Sort the columns of a matrix
apply(x, 2, sort)

##- function with extra args:
cave <- function(x, c1, c2) c(mean(x[c1]), mean(x[c2]))
apply(x,1, cave,  c1="x1", c2=c("x1","x2"))

ma <- matrix(c(1:4, 1, 6:8), nrow = 2)
ma
apply(ma, 1, table)  #--> a list of length 2
apply(ma, 1, stats::quantile)# 5 x n matrix with rownames

stopifnot(dim(ma) == dim(apply(ma, 1:2, sum)))

## Example with different lengths for each call
z <- array(1:24, dim=2:4)
zseq <- apply(z, 1:2, function(x) seq_len(max(x)))
zseq         ## a 2 x 3 matrix
typeof(zseq) ## list
dim(zseq) ## 2 3
zseq[1,]
apply(z, 3, function(x) seq_len(max(x)))
## a list without a dim attribute
}

\keyword{iteration}
\keyword{array}
```

## A.4. Generated PDF

In Figures 2 and 3, we show the generated documentation converted to PDF via `R CMD Rd2pdf`.

# R documentation

of 'man/apply.Rd'

August 15, 2013

---

apply *Apply Functions Over Array Margins*

---

**Description**

Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

**Usage**

```
apply(X, MARGIN, FUN, ...)
```

**Arguments**

| | |
|---|---|
| X | an array, including a matrix. |
| MARGIN | a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where X has named dimnames, it can be a character vector selecting dimension names. |
| FUN | the function to be applied: see 'Details'. In the case of functions like +, %*%, etc., the function name must be backquoted or quoted. |
| ... | optional arguments to FUN. |

**Details**

FUN is found by a call to match.fun and typically is either a function or a symbol (e.g. a backquoted name) or a character string specifying a function to be searched for from the environment of the call to apply.

If X is not an array but an object of a class with a non-null dim value (such as a data frame), apply attempts to coerce it to an array via as.matrix if it is two-dimensional (e.g., a data frame) or via as.array.



Figure 2: Documentation of `apply.R` as converted to PDF via `R CMD Rd2pdf` (page 1).

**Value**

If each call to FUN returns a vector of length n, then apply returns an array of dimension c(n, dim(X)[MARGIN]) if n > 1. If n equals 1, apply returns a vector if MARGIN has length 1 and an array of dimension dim(X)[MARGIN] otherwise. If n is 0, the result has length 0 but not necessarily the 'correct' dimension.

If the calls to FUN return vectors of different lengths, apply returns a list of length prod(dim(X)[MARGIN]) with dim set to MARGIN if this has length greater than one.

In all cases the result is coerced by as.vector to one of the basic vector types before the dimensions are set, so that (for example) factor results will be coerced to a character array.

**Author(s)**

Toby Dylan Hocking

**Examples**

```
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
apply(x, 2, mean, trim = .2)
col.sums <- apply(x, 2, sum)
row.sums <- apply(x, 1, sum)
rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))

stopifnot( apply(x, 2, is.vector))

## Sort the columns of a matrix
apply(x, 2, sort)

##- function with extra args:
cave <- function(x, c1, c2) c(mean(x[c1]), mean(x[c2]))
apply(x,1, cave,  c1="x1", c2=c("x1","x2"))

ma <- matrix(c(1:4, 1, 6:8), nrow = 2)
ma
apply(ma, 1, table)  #--> a list of length 2
apply(ma, 1, stats::quantile)# 5 x n matrix with rownames

stopifnot(dim(ma) == dim(apply(ma, 1:2, sum)))

## Example with different lengths for each call
z <- array(1:24, dim=2:4)
zseq <- apply(z, 1:2, function(x) seq_len(max(x)))
zseq         ## a 2 x 3 matrix
typeof(zseq) ## list
dim(zseq) ## 2 3
zseq[1,]
apply(z, 3, function(x) seq_len(max(x)))
## a list without a dim attribute
```

Figure 3: Documentation of `apply.R` as converted to PDF via `R CMD Rd2pdf` (page 2).

**Affiliation:**

Toby Dylan Hocking
INRIA – Sierra Project for Machine Learning Research
23, avenue d'Italie
CS 81321
75214 Paris Cedex 13, France
Telephone: +33/1 39 63 54 99
E-mail: Toby.Hocking@inria.fr
URL: http://cbio.ensmp.fr/~thocking/

Thomas Wutzler
Max Planck Institute for Biogeochemistry
Hans-Knöll-Strasse 10
07745 Jena, Germany
Telephone: +49/3641 576271
E-mail: twutz@bgc-jena.mpg.de

Keith Ponting
Aurix Ltd.
Malvern Hills Science Park
Geraldine Road
Great Malvern
Worcestershire, WR14 3SZ, United Kingdom
E-mail: k.ponting@aurix.com

Philippe Grosjean
Numerical Ecology of Aquatic Systems
University of Mons
20 Place du Parc, 7000 Mons, Belgium
E-mail: Philippe.Grosjean@umons.ac.be

*Journal of Statistical Software* http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 54, Issue 6 *Submitted:* 2011-05-16
August 2013 *Accepted:* 2013-01-22