# Package 'pbapply'

July 20, 2025

**Type** Package

**Title** Adding Progress Bar to '*apply' Functions

**Version** 1.7-4

**Date** 2025-07-19

**Maintainer** Peter Solymos <psolymos@gmail.com>

**Description** A lightweight package that adds
progress bar to vectorized R functions
('*apply'). The implementation can easily be added
to functions where showing the progress is
useful (e.g. bootstrap). The type and style of the
progress bar (with percentages or remaining time)
can be set through options.
Supports several parallel processing backends including mirai and future.

**Depends** R (>= 3.2.0)

**Imports** parallel

**Suggests** shiny, future, future.apply

**License** GPL (>= 2)

**URL** https://github.com/psolymos/pbapply,
https://peter.solymos.org/pbapply/

**BugReports** https://github.com/psolymos/pbapply/issues

**NeedsCompilation** no

**Author** Peter Solymos [aut, cre] (ORCID:
<https://orcid.org/0000-0001-7337-1740>),
Zygmunt Zawadzki [aut],
Henrik Bengtsson [ctb],
R Core Team [cph, ctb]

**Repository** CRAN

**Date/Publication** 2025-07-20 18:50:02 UTC

# Contents



---

pbapply *Adding Progress Bar to '*apply' Functions*

---

## Description

Adding progress bar to `*apply` functions, possibly leveraging parallel processing.

## Usage

```
pblapply(X, FUN, ..., cl = NULL)
pbeapply(env, FUN, ..., all.names = FALSE, USE.NAMES = TRUE, cl = NULL)
pbwalk(X, FUN, ..., cl = NULL)

pbapply(X, MARGIN, FUN, ..., simplify = TRUE, cl = NULL)

pbsapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE, cl = NULL)
pbvapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE, cl = NULL)
pbreplicate(n, expr, simplify = "array", ..., cl = NULL)

.pb_env
pbmapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
pb.mapply(FUN, dots, MoreArgs)
pbMap(f, ...)

pbtapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE, cl = NULL)

pbby(data, INDICES, FUN, ..., simplify = TRUE, cl = NULL)
```

## Arguments

| | |
|---|---|
| `X` | For `pbsapply`, `pblapply`, and `pbwalk` a vector (atomic or list) or an expressions vector (other objects including classed objects will be coerced by `as.list`.) For `pbapply` an array, including a matrix. For `pbtapply` an R object for which a `split` method exists. Typically vector-like, allowing subsetting with `[`. |
| `MARGIN` | A vector giving the subscripts which the function will be applied over. `1` indicates rows, `2` indicates columns, `c(1,2)` indicates rows and columns. |

| | |
|---|---|
| FUN, f | The function to be applied to each element of X: see apply, sapply, and lapply. In the case of functions like +, %*%, etc., the function name must be backquoted or quoted. If FUN is NULL, pbtapply returns a vector which can be used to subscript the multi-way array pbtapply normally produces. |
| ... | Optional arguments to FUN and also to underlying functions (e.g. parLapply and mclapply when cl is not NULL). |
| dots | List of arguments to vectorize over (vectors or lists of strictly positive length, or all of zero length); see .mapply. |
| env | Environment to be used. |
| FUN.VALUE | A (generalized) vector; a template for the return value from FUN. See 'Details' for vapply. |
| simplify, SIMPLIFY | Logical; should the result be simplified to a vector or matrix if possible? pbtapply returns an array of mode "list" (in other words, a list with a dim attribute) when FALSE; if TRUE (the default), then if FUN always returns a scalar, pbtapply returns an array with the mode of the scalar. |
| USE.NAMES | Logical; if TRUE and if X is character, use X as names for the result unless it had names already. |
| all.names | Logical, indicating whether to apply the function to all values. |
| n | Number of replications. |
| expr | Expression (language object, usually a call) to evaluate repeatedly. |
| cl | A cluster object created by makeCluster, or an integer to indicate number of child-processes (integer values are ignored on Windows) for parallel evaluations (see Details on performance). It can also be "future" to use a future backend (see Details), NULL (default) refers to sequential evaluation. |
| MoreArgs | A list of other arguments to FUN. |
| INDEX | A list of one or more factors, each of same length as X. The elements are coerced to factors by as.factor. |
| INDICES | A factor or a list of factors, each of length nrow(data). |
| data | An R object, normally a data frame, possibly a matrix. |
| default | Only in the case of simplification to an array, the value with which the array is initialized as array(default, dim = ..). Before R 3.4.0, this was hard coded to array()'s default NA. If it is NA (the default), the missing value of the answer type, e.g. NA_real_, is chosen (as.raw(0) for "raw"). In a numerical case, it may be set, e.g., to FUN(integer(0)), e.g., in the case of FUN = sum to 0 or 0L. |

## Details

The behavior of the progress bar is controlled by the option type in pboptions, it can take values c("txt", "win", "tk", "none",) on Windows, and c("txt", "tk", "none",) on Unix systems.

Other options have elements that are arguments used in the functions timerProgressBar, txtProgressBar, and tkProgressBar. See pboptions for how to conveniently set these.

Parallel processing can be enabled through the cl argument. parLapply is called when cl is a 'cluster' object, mclapply is called when cl is an integer. Showing the progress bar increases

the communication overhead between the main process and nodes / child processes compared to the parallel equivalents of the functions without the progress bar. The functions fall back to their original equivalents when the progress bar is disabled (i.e. `getOption("pboptions")$type == "none"` or `dopb()` is `FALSE`). This is the default when `interactive()` if `FALSE` (i.e. called from command line R script).

When doing parallel processing, other objects might need to pushed to the workers, and random numbers must be handled with care (see Examples).

Updating the progress bar with `mclapply` can be slightly slower compared to using a Fork cluster (i.e. calling `makeForkCluster`). Care must be taken to set appropriate random numbers in this case.

Note the `use_lb` option (see `pboptions`) for using load balancing when running in parallel clusters. If using `mclapply`, the `...` passes arguments to the underlying function for further control.

`pbwalk` is similar to `pblapply` but it calls `FUN` only for its side-effect and returns the input `X` invisibly (this behavior is modeled after 'purrr::walk').

Note that when `cl = "future"`, you might have to specify the `future.seed` argument (passed as part of `...`) when using random numbers in parallel.

Note also that if your code prints messages or you encounter warnings during execution, the condition messages might cause the progress bar to break up and continue on a new line.

## Value

Similar to the value returned by the standard `*apply` functions.

A progress bar is showed as a side effect.

## Note

Progress bar can add an overhead to the computation.

## Author(s)

Peter Solymos <solymos@ualberta.ca>

## See Also

Progress bars used in the functions: `txtProgressBar`, `tkProgressBar`, `timerProgressBar`

Sequential `*apply` functions: `apply`, `sapply`, `lapply`, `replicate`, `mapply`, `.mapply`, `tapply`

Parallel `*apply` functions from package 'parallel': `parLapply`, `mclapply`.

Setting the options: `pboptions`

Conveniently add progress bar to `for`-like loops: `startpb`, `setpb`, `getpb`, `closepb`

## Examples

```
## --- simple linear model simulation ---
set.seed(1234)
n <- 200
x <- rnorm(n)
```

```
y <- rnorm(n, crossprod(t(model.matrix(~ x)), c(0, 1)), sd = 0.5)
d <- data.frame(y, x)
## model fitting and bootstrap
mod <- lm(y ~ x, d)
ndat <- model.frame(mod)
B <- 100
bid <- sapply(1:B, function(i) sample(nrow(ndat), nrow(ndat), TRUE))
fun <- function(z) {
    if (missing(z))
        z <- sample(nrow(ndat), nrow(ndat), TRUE)
    coef(lm(mod$call$formula, data=ndat[z,]))
}

## standard '*apply' functions
system.time(res1 <- lapply(1:B, function(i) fun(bid[,i])))
system.time(res2 <- sapply(1:B, function(i) fun(bid[,i])))
system.time(res3 <- apply(bid, 2, fun))
system.time(res4 <- replicate(B, fun()))

## 'pb*apply' functions
## try different settings:
## "none", "txt", "tk", "win", "timer"
op <- pboptions(type = "timer") # default
system.time(res1pb <- pblapply(1:B, function(i) fun(bid[,i])))
pboptions(op)

pboptions(type = "txt")
system.time(res2pb <- pbsapply(1:B, function(i) fun(bid[,i])))
pboptions(op)

pboptions(type = "txt", style = 1, char = "=")
system.time(res3pb <- pbapply(bid, 2, fun))
pboptions(op)

pboptions(type = "txt", char = ":")
system.time(res4pb <- pbreplicate(B, fun()))
pboptions(op)

## Not run:
## parallel evaluation using the parallel package
## (n = 2000 and B = 1000 will give visible timing differences)

library(parallel)
cl <- makeCluster(2L)
clusterExport(cl, c("fun", "mod", "ndat", "bid"))

## parallel with no progress bar: snow type cluster
## (RNG is set in the main process to define the object bid)
system.time(res1cl <- parLapply(cl = cl, 1:B, function(i) fun(bid[,i])))
system.time(res2cl <- parSapply(cl = cl, 1:B, function(i) fun(bid[,i])))
system.time(res3cl <- parApply(cl, bid, 2, fun))

## parallel with  progress bar: snow type cluster
```

```
## (RNG is set in the main process to define the object bid)
system.time(res1pbcl <- pblapply(1:B, function(i) fun(bid[,i]), cl = cl))
system.time(res2pbcl <- pbsapply(1:B, function(i) fun(bid[,i]), cl = cl))
## (RNG needs to be set when not using bid)
parallel::clusterSetRNGStream(cl, iseed = 0L)
system.time(res4pbcl <- pbreplicate(B, fun(), cl = cl))
system.time(res3pbcl <- pbapply(bid, 2, fun, cl = cl))

stopCluster(cl)

if (.Platform$OS.type != "windows") {
    ## parallel with no progress bar: multicore type forking
    ## (mc.set.seed = TRUE in parallel::mclapply by default)
    system.time(res2mc <- mclapply(1:B, function(i) fun(bid[,i]), mc.cores = 2L))
    ## parallel with  progress bar: multicore type forking
    ## (mc.set.seed = TRUE in parallel::mclapply by default)
    system.time(res1pbmc <- pblapply(1:B, function(i) fun(bid[,i]), cl = 2L))
    system.time(res2pbmc <- pbsapply(1:B, function(i) fun(bid[,i]), cl = 2L))
    system.time(res4pbmc <- pbreplicate(B, fun(), cl = 2L))
}

## End(Not run)

## --- Examples taken from standard '*apply' functions ---

## --- sapply, lapply, and replicate ---

require(stats); require(graphics)

x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
# compute the list mean for each list element
pblapply(x, mean)
pbwalk(x, mean)
# median and quartiles for each list element
pblapply(x, quantile, probs = 1:3/4)
pbsapply(x, quantile)
i39 <- sapply(3:9, seq) # list of vectors
pbsapply(i39, fivenum)
pbvapply(i39, fivenum,
       c(Min. = 0, "1st Qu." = 0, Median = 0, "3rd Qu." = 0, Max. = 0))

## sapply(*, "array") -- artificial example
(v <- structure(10*(5:8), names = LETTERS[1:4]))
f2 <- function(x, y) outer(rep(x, length.out = 3), y)
(a2 <- pbsapply(v, f2, y = 2*(1:5), simplify = "array"))
a.2 <- pbvapply(v, f2, outer(1:3, 1:5), y = 2*(1:5))
stopifnot(dim(a2) == c(3,5,4), all.equal(a2, a.2),
          identical(dimnames(a2), list(NULL,NULL,LETTERS[1:4])))

summary(pbreplicate(100, mean(rexp(10))))

## use of replicate() with parameters:
foo <- function(x = 1, y = 2) c(x, y)
```

```
# does not work: bar <- function(n, ...) replicate(n, foo(...))
bar <- function(n, x) pbreplicate(n, foo(x = x))
bar(5, x = 3)

## --- apply ---

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

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

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

## keeping named dimnames
names(dimnames(x)) <- c("row", "col")
x3 <- array(x, dim = c(dim(x),3),
    dimnames = c(dimnames(x), list(C = paste0("cop.",1:3))))
identical(x,  pbapply( x,  2,  identity))
identical(x3, pbapply(x3, 2:3, identity))

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

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

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

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

## --- mapply and .mapply ---

pbmapply(rep, 1:4, 4:1)
pbmapply(rep, times = 1:4, x = 4:1)
pbmapply(rep, times = 1:4, MoreArgs = list(x = 42))
pbmapply(function(x, y) seq_len(x) + y,
```

```
       c(a =  1, b = 2, c = 3),  # names from first
       c(A = 10, B = 0, C = -10))
word <- function(C, k) paste(rep.int(C, k), collapse = "")
utils::str(pbmapply(word, LETTERS[1:6], 6:1, SIMPLIFY = FALSE))

pb.mapply(rep,
          dots = list(1:4, 4:1),
          MoreArgs = list())
pb.mapply(rep,
          dots = list(times = 1:4, x = 4:1),
          MoreArgs = list())
pb.mapply(rep,
          dots = list(times = 1:4),
          MoreArgs = list(x = 42))
pb.mapply(function(x, y) seq_len(x) + y,
          dots = list(c(a =  1, b = 2, c = 3),  # names from first
                      c(A = 10, B = 0, C = -10)),
          MoreArgs = list())

## --- Map ---

pbMap(`+`, 1,         1 : 3) ;         1 + 1:3

## --- eapply ---

env <- new.env(hash = FALSE)
env$a <- 1:10
env$beta <- exp(-3:3)
env$logic <- c(TRUE, FALSE, FALSE, TRUE)
pbeapply(env, mean)
unlist(pbeapply(env, mean, USE.NAMES = FALSE))
pbeapply(env, quantile, probs = 1:3/4)
pbeapply(env, quantile)

## --- tapply ---

require(stats)
groups <- as.factor(rbinom(32, n = 5, prob = 0.4))
pbtapply(groups, groups, length) #- is almost the same as
table(groups)

## contingency table from data.frame : array with named dimnames
pbtapply(warpbreaks$breaks, warpbreaks[,-1], sum)
pbtapply(warpbreaks$breaks, warpbreaks[, 3, drop = FALSE], sum)

n <- 17; fac <- factor(rep_len(1:3, n), levels = 1:5)
table(fac)
pbtapply(1:n, fac, sum)
pbtapply(1:n, fac, sum, default = 0) # maybe more desirable
pbtapply(1:n, fac, sum, simplify = FALSE)
pbtapply(1:n, fac, range)
pbtapply(1:n, fac, quantile)
pbtapply(1:n, fac, length) ## NA's
```

```
pbtapply(1:n, fac, length, default = 0) # == table(fac)

## example of ... argument: find quarterly means
pbtapply(presidents, cycle(presidents), mean, na.rm = TRUE)

ind <- list(c(1, 2, 2), c("A", "A", "B"))
table(ind)
pbtapply(1:3, ind) #-> the split vector
pbtapply(1:3, ind, sum)

## Some assertions (not held by all patch propsals):
nq <- names(quantile(1:5))
stopifnot(
  identical(pbtapply(1:3, ind), c(1L, 2L, 4L)),
  identical(pbtapply(1:3, ind, sum),
            matrix(c(1L, 2L, NA, 3L), 2, dimnames = list(c("1", "2"), c("A", "B")))),
  identical(pbtapply(1:n, fac, quantile)[-1],
            array(list(`2` = structure(c(2, 5.75, 9.5, 13.25, 17), .Names = nq),
                 `3` = structure(c(3, 6, 9, 12, 15), .Names = nq),
                 `4` = NULL, `5` = NULL), dim=4, dimnames=list(as.character(2:5)))))

## --- by ---

pbby(warpbreaks[, 1:2], warpbreaks[,"tension"], summary)
pbby(warpbreaks[, 1],   warpbreaks[, -1],       summary)
pbby(warpbreaks, warpbreaks[,"tension"],
   function(x) lm(breaks ~ wool, data = x))
tmp <- with(warpbreaks,
            pbby(warpbreaks, tension,
               function(x) lm(breaks ~ wool, data = x)))
sapply(tmp, coef)
```

---

pboptions *Creating Progress Bar and Setting Options*

---

## Description

Creating progress bar and setting options.

## Usage

```
pboptions(...)
startpb(min = 0, max = 1)
setpb(pb, value)
getpb(pb)
closepb(pb)
dopb()
doshiny()
pbtypes()
```

## Arguments

| | |
|---|---|
| ... | Arguments in tag = value form, or a list of tagged values. The tags must come from the parameters described below. |
| pb | A progress bar object created by startpb. |
| min, max | Finite numeric values for the extremes of the progress bar. Must have min < max. |
| value | New value for the progress bar. |

## Details

pboptions is a convenient way of handling options related to progress bar.

Other functions can be used for conveniently adding progress bar to for-like loops (see Examples).

## Value

When parameters are set by pboptions, their former values are returned in an invisible named list. Such a list can be passed as an argument to pboptions to restore the parameter values. Tags are the following:

| | |
|---|---|
| type | Type of the progress bar: timer ("timer"), text ("txt"), Windows ("win"), TclTk ("tk"), none ("none"), or Shiny ("shiny"). Default value is "timer" progress bar with estimated remaining time when in interactive mode, and "none" otherwise. See pbtypes() for available progress bar types depending on operating system. |
| char | The character (or character string) to form the progress bar. Default value is "+". |
| txt.width | The width of the text based progress bar, as a multiple of the width of char. If NA, the number of characters is that which fits into getOption("width"). Default value is 50. |
| gui.width | The width of the GUI based progress bar in pixels: the dialogue box will be 40 pixels wider (plus frame). Default value is 300. |
| style | The style of the bar, see txtProgressBar and timerProgressBar. Default value is 3. |
| initial | Initial value for the progress bar. Default value is 0. |
| title | Character string giving the window title on the GUI dialogue box. Default value is "R progress bar". |
| label | Character string giving the window label on the GUI dialogue box. Default value is "". |
| nout | Integer, the maximum number of times the progress bar is updated. The default value is 100. Smaller value minimizes the running time overhead related to updating the progress bar. This can be especially important for forking type parallel runs. |
| min_time | Minimum time in seconds. timerProgressBar output is printed only if estimated completion time is higher than this value. The default value is 0. |
| use_lb | Switch for using load balancing when running in parallel clusters. The default value is FALSE. |

For `startpb` a progress bar object.

For `getpb` and `setpb`, a length-one numeric vector giving the previous value (invisibly for `setpb`). The return value is `NULL` if the progress bar is turned off by `getOption("pboptions")$type` (`"none"` or `NULL` value).

`dopb` returns a logical value if progress bar is to be shown based on the option `getOption("pboptions")$type`. It is `FALSE` if the type of progress bar is `"none"` or `NULL`.

`doshiny` returns a logical value, `TRUE` when the shiny package namespace is available (i.e. the suggested package is installed), the `type` option is set to `"shiny"`, and a shiny application is running.

For `closepb` closes the connection for the progress bar.

`pbtypes` prints the available progress bar types depending on the operating system (i.e. `"win"` available on Windows only).

## Author(s)

Peter Solymos <solymos@ualberta.ca>

## See Also

Progress bars used in the functions: `timerProgressBar`, `txtProgressBar`, `tkProgressBar`

## Examples

```
## increase sluggishness to admire the progress bar longer
sluggishness <- 0.01

## for loop
fun1 <- function() {
    pb <- startpb(0, 10)
    on.exit(closepb(pb))
    for (i in 1:10) {
        Sys.sleep(sluggishness)
        setpb(pb, i)
    }
    invisible(NULL)
}
## while loop
fun2 <- function() {
    pb <- startpb(0, 10-1)
    on.exit(closepb(pb))
    i <- 1
    while (i < 10) {
        Sys.sleep(sluggishness)
        setpb(pb, i)
        i <- i + 1
    }
    invisible(NULL)
}
## using original settings
fun1()
## resetting pboptions
```

```
opb <- pboptions(style = 1, char = ">")
## check new settings
getOption("pboptions")
## running again with new settings
fun2()
## resetting original
pboptions(opb)
## check reset
getOption("pboptions")
fun1()

## dealing with nested progress bars
## when only one the 1st one is needed
f <- function(x) Sys.sleep(sluggishness)
g <- function(x) pblapply(1:10, f)
tmp <- lapply(1:10, g) # undesirable
## here is the desirable solution
h <- function(x) {
    opb <- pboptions(type="none")
    on.exit(pboptions(opb))
    pblapply(1:10, f)
}
tmp <- pblapply(1:10, h)

## list available pb types
pbtypes()
```

---

`splitpb` *Divide Tasks for Progress-bar Friendly Distribution in a Cluster*

---

## Description

Divides up `1:nx` into approximately equal sizes (`ncl`) as a way to allocate tasks to nodes in a cluster repeatedly while updating a progress bar.

## Usage

```
splitpb(nx, ncl, nout = NULL)
```

## Arguments

| | |
|---|---|
| `nx` | Number of tasks. |
| `ncl` | Number of cluster nodes. |
| `nout` | Integer, maximum number of partitions in the output (must be > 0). |

## Value

A list of length `min(nout, ceiling(nx / ncl))`, each element being an integer vector of length `ncl * k` or less, where `k` is a tuning parameter constrained by the other arguments (`k = max(1L, ceiling(ceiling(nx / ncl) / nout))` and `k = 1` if `nout = NULL`).

### Author(s)

Peter Solymos <solymos@ualberta.ca>

### See Also

Parallel usage of `pbapply` and related functions.

### Examples

```
## define 1 job / worker at a time and repeat
splitpb(10, 4)
## compare this to the no-progress-bar split
## that defines all the jubs / worker up front
parallel::splitIndices(10, 4)

## cap the length of the output
splitpb(20, 2, nout = NULL)
splitpb(20, 2, nout = 5)
```

---

## timerProgressBar *Timer Progress Bar*

---

### Description

Text progress bar with timer in the R console.

### Usage

```
timerProgressBar(min = 0, max = 1, initial = 0, char = "=",
    width = NA, title, label, style = 1, file = "", min_time = 0)
getTimerProgressBar(pb)
setTimerProgressBar(pb, value, title = NULL, label = NULL)
getTimeAsString(time)
```

### Arguments

| | |
|---|---|
| `min`, `max` | (finite) numeric values for the extremes of the progress bar. Must have `min` < `max`. |
| `initial`, `value` | initial or new value for the progress bar. See Details for what happens with invalid values. |
| `char` | he character (or character string) to form the progress bar. If number of characters is >1, it is silently stripped to length 1 unless `style` is 5 or 6 (see Details). |
| `width` | the width of the progress bar, as a multiple of the width of char. If `NA`, the default, the number of characters is that which fits into `getOption("width")`. |

| | |
|---|---|
| style | the style taking values between 1 and 6. 1: progress bar with elapsed and remaining time, remaining percentage is indicated by spaces between pipes (default for this function), 2: throbber with elapsed and remaining time, 3: progress bar with remaining time printing elapsed time at the end, remaining percentage is indicated by spaces between pipes (default for style option in pboptions), 4: throbber with remaining time printing elapsed time at the end, 5: progress bar with elapsed and remaining time with more flexible styling (see Details and Examples), 6: progress bar with remaining time printing elapsed time at the end with more flexible styling (see Details and Examples). |
| file | an open connection object or "" which indicates the console. |
| min_time | numeric, minimum processing time (in seconds) required to show a progress bar. |
| pb | an object of class "timerProgressBar". |
| title, label | ignored, for compatibility with other progress bars. |
| time | numeric of length 1, time in seconds. |

## Details

timerProgressBar will display a progress bar on the R console (or a connection) via a text representation.

setTimerProgessBar will update the value. Missing (NA) and out-of-range values of value will be (silently) ignored. (Such values of initial cause the progress bar not to be displayed until a valid value is set.)

The progress bar should be closed when finished with: this outputs the final newline character (see closepb).

If style is 5 or 6, it is possible to define up to 4 characters for the char argument (as a single string) for the left end, elapsed portion, remaining portion, and right end of the progress bar (|= | by default). Remaining portion cannot be the same as the elapsed portion (space is used for remaining in such cases). If 1 character is defined, it is taken for the elapsed portion. If 2-4 characters are defined, those are interpreted in sequence (left and right end being the same when 2-3 characters defined), see Examples.

getTimeAsString converts time in seconds into ~HHh MMm SSs format to be printed by timerProgressBar.

## Value

For timerProgressBar an object of class "timerProgressBar" inheriting from "txtProgressBar".

For getTimerProgressBar and setTimerProgressBar, a length-one numeric vector giving the previous value (invisibly for setTimerProgressBar).

getTimeAsString returns time in ~HHh MMm SSs format as character. Returns "calculating" when time=NULL.

## Author(s)

Zygmunt Zawadzki <zawadzkizygmunt@gmail.com>

Peter Solymos <solymos@ualberta.ca>

## See Also

The `timerProgressBar` implementation follows closely the code of `txtProgressBar`.

## Examples

```
## increase sluggishness to admire the progress bar longer
sluggishness <- 0.02

test_fun <- function(...)
{
    pb <- timerProgressBar(...)
    on.exit(close(pb))
    for (i in seq(0, 1, 0.05)) {
        Sys.sleep(sluggishness)
        setTimerProgressBar(pb, i)
    }
    invisible(NULL)
}

## check the different styles
test_fun(width = 35, char = "+", style = 1)
test_fun(style = 2)
test_fun(width = 50, char = ".", style = 3)
test_fun(style = 4)
test_fun(width = 35, char = "[=-]", style = 5)
test_fun(width = 50, char = "{*.}", style = 6)

## no bar only percent and elapsed
test_fun(width = 0, char = "    ", style = 6)

## this should produce a progress bar based on min_time
(elapsed <- system.time(test_fun(width = 35, min_time = 0))["elapsed"])
## this should not produce a progress bar based on min_time
system.time(test_fun(min_time = 2 * elapsed))["elapsed"]

## time formatting
getTimeAsString(NULL)
getTimeAsString(15)
getTimeAsString(65)
getTimeAsString(6005)

## example usage of getTimeAsString, use sluggishness <- 1
n <- 10
t0 <- proc.time()[3]
ETA <- NULL
for (i in seq_len(n)) {
    cat(i, "/", n, "- ETA:", getTimeAsString(ETA))
    flush.console()
    Sys.sleep(sluggishness)
    dt <- proc.time()[3] - t0
    cat(" - elapsed:", getTimeAsString(dt), "\n")
    ETA <- (n - i) * dt / i
```

```
    }
```

# Index