Package 'hutilscpp'

March 20, 2025

Title Miscellaneous Functions in C++

Version 0.10.10

Description Provides utility functions that are simply, frequently used, but may require higher performance that what can be obtained from base R. Incidentally provides support for 'reverse geocoding', such as matching a point with its nearest neighbour in another array. Used as a complement to package 'hutils' by sacrificing compilation or installation time for higher running speeds. The name is a portmanteau of the author and 'Rcpp'.

URL https://github.com/hughparsonage/hutilscpp

BugReports https://github.com/hughparsonage/hutilscpp/issues

License GPL-2

Encoding UTF-8

Depends R (>= 3.3.0)

Imports data.table, hutils, magrittr, utils

RoxygenNote 7.3.2

Suggests bench, parallel, TeXCheckR, withr, tinytest, covr

NeedsCompilation yes

Author Hugh Parsonage [aut, cre], Simon Urbanek [ctb] (fastmatch components)

Maintainer Hugh Parsonage <hugh.parsonage@gmail.com>

Repository CRAN

Date/Publication 2025-03-20 08:10:02 UTC

Contents

`abs_diff` *Absolute difference*

Description

Equivalent to `abs(x - y)` but aims to be faster by avoiding allocations.

Usage

```
abs_diff(x, y, nThread = getOption("hutilscpp.nThread", 1L), option = 1L)

max_abs_diff(x, y, nThread = getOption("hutilscpp.nThread", 1L))
```

Arguments

x, y	Atomic, numeric, equilength vectors.
nThread	Number of threads to use.
option	An integer, provides backwards-compatible method to change results. **0** Return `max(abs(x - y))` (without allocation). **1** Return `abs(x - y)` with the expectation that every element will be `integer`, returning a `double` only if required. **2** Return `abs(x - y)` but always a `double` vector, regardless of necessity. **3** Return `which.max(abs(x - y))`

Examples

```
x <- sample(10)
y <- sample(10)
abs_diff(x, y)
max_abs_diff(x, y)
```

---

allNA *Is a vector empty?*

Description

A vector is empty if `all(is.na(x))` with a special case for `length(x) == 0`.

Usage

```
allNA(
  x,
  expected = FALSE,
  len0 = FALSE,
  nThread = getOption("hutilscpp.nThread", 1L)
)
```

Arguments

x	A vector. Only atomic vectors are supported.
expected	`TRUE | FALSE` Whether it is expected that `x` is empty. If `TRUE` the function will be marginally faster if `x` is empty but likely slower if not.
len0	The result if `length(x) == 0`.
nThread	Number of threads to use (only applicable if `expected` is `TRUE`)

Examples

```
allNA(c(NA, NA))
allNA(c(NA, NA, 1))
```

anyOutside *Are any values outside the interval specified?*

Description

Are any values outside the interval specified?

Usage

```
anyOutside(x, a, b, nas_absent = NA, na_is_outside = NA)
```

Arguments

x	A numeric vector.
a, b	Single numeric values designating the interval.
nas_absent	Are NAs *known* to be absent from x? If nas_absent = NA, the default, x will be searched for NAs; if nas_absent = TRUE, x will not be checked; if nas_absent = FALSE, the answer is NA_integer_ if na.rm = FALSE otherwise only non-NA values outside [a, b]. If nas_absent = TRUE but x has missing values then the result is unreliable.
na_is_outside	(logical, default: NA) How should NAs in x be treated? **If** NA the default, then the first value in x that is either outside [a, b] or NA is detected: if it is NA, then NA_integer_ is returned; otherwise the position of that value is returned.#' **If** FALSE then NA values are effectively skipped; the position of the first *known* value outside [a, b] is returned. **If** TRUE the position of the first value that is either outside [a, b] or NA is returned.

Value

0L if no values in x are outside [a, b]. Otherwise, the position of the first value of x outside [a, b].

Examples

```
anyOutside(1:10, 1L, 10L)
anyOutside(1:10, 1L, 7L)

# na_is_outside = NA
anyOutside(c(1:10, NA), 1L, 7L)     # Already outside before the NA
anyOutside(c(NA, 1:10, NA), 1L, 7L) # NA since it occurred first

anyOutside(c(1:7, NA), 1L, 7L, na_is_outside = FALSE)
anyOutside(c(1:7, NA), 1L, 7L, na_is_outside = TRUE)
```

```
##
# N <- 500e6
N <- 500e3
x <- rep_len(hutils::samp(-5:6, size = 23), N)
bench_system_time(anyOutside(x, -5L, 6L))
#    process      real
#  453.125ms 459.758ms
```

`are_even` *Are elements of a vector even?*

Description

Are elements of a vector even?

Usage

```
are_even(
  x,
  check_integerish = TRUE,
  keep_nas = TRUE,
  nThread = getOption("hutilscpp.nThread", 1L)
)

which_are_even(x, check_integerish = TRUE)
```

Arguments

`x`	An integer vector. Double vectors may also be used, but will be truncated, with a warning if any element are not integers. Long vectors are not supported unless `x` is integer and `keep_nas = FALSE`.
`check_integerish`	(logical, default: `TRUE`) Should the values in `x` be checked for non-integer values if `x` is a double vector. If `TRUE` and values are found to be non-integer a warning is emitted.
`keep_nas`	(logical, default: `TRUE`) Should `NA`s in `x` return `NA` in the result? If `FALSE`, will return `TRUE` since the internal representation of `x` is even. Only applies if `is.integer(x)`.
`nThread`	Number of threads to use.

Value

For `are_even`, a logical vector the same length as `x`, `TRUE` whenever `x` is even.

For `which_are_even` the integer positions of even values in `x`.

as_integer_if_safe *Coerce from double to integer if safe*

Description

The same as `as.integer(x)` but only if `x` consists only of whole numbers and is within the range of integers.

Usage

```
as_integer_if_safe(x)
```

Arguments

`x`	A double vector. If not a double vector, it is simply returned without any coercion.

Examples

```
N <- 1e6  # run with 1e9
x <- rep_len(as.double(sample.int(100)), N)
alt_as_integer <- function(x) {
  xi <- as.integer(x)
  if (isTRUE(all.equal(x, xi))) {
    xi
  } else {
    x
  }
}
bench_system_time(as_integer_if_safe(x))
#> process    real
#>  6.453s  6.452s
bench_system_time(alt_as_integer(x))
#> process    real
#> 15.516s 15.545s
bench_system_time(as.integer(x))
#> process    real
#>  2.469s  2.455s
```

bench_system_time *Evaluate time of computation*

Description

(Used for examples and tests)

Usage

```
bench_system_time(expr)
```

Arguments

expr	Passed to system_time.

---

character2integer *Character to numeric*

Description

Character to numeric

Usage

```
character2integer(x, na.strings = NULL, allow.double = FALSE, option = 0L)
```

Arguments

x	A character vector.
na.strings	A set of strings that shall be coerced to NA_integer_
allow.double	logical(1) If TRUE, a double vector may be returned. If FALSE, an error will be emitted. If NA, numeric values outside integer range are coerced to NA_integer_, silently.
option	Control behaviour: **0** Strip commas.

---

coalesce0 *Convenience function for coalescing to zero*

Description

Convenience function for coalescing to zero

Usage

```
coalesce0(x, nThread = getOption("hutilscpp.nThread", 1L))

COALESCE0(x, nThread = getOption("hutilscpp.nThread", 1L))
```

Arguments

x	An atomic vector. Or a list for COALESCE0.
nThread	Number of threads to use.

Value

Equivalent to `hutils::coalesce(x, 0)` for an appropriate type of zero. `COALESCE0(x)`

For complex numbers, each component is coalesced. For unsupported types, the vector is returned, silently.

Examples

```
coalesce0(c(NA, 2:3))
coalesce0(NaN + 1i)
```

Comma *Faster version of* `scales::comma`

Description

Faster version of `scales::comma`

Usage

```
Comma(x, digits = 0L, big.mark = c(",", " ", "'", "_", "~", "\"", "/"))
```

Arguments

`x`	A numeric vector.
`digits`	An integer, similar to `round`.
`big.mark`	A single character, the thousands separator.

Value

Similar to `prettyNum(round(x, digits), big.mark = ',')` but rounds down and `-1 < x < 0` will output `"-0"`.

---

count_logical *Count logicals*

Description

Count the number of `FALSE`, `TRUE`, and `NA`s.

Usage

```
count_logical(x, nThread = getOption("hutilscpp.nThread", 1L))
```

Arguments

x	A logical vector.
nThread	Number of threads to use.

Value

A vector of 3 elements: the number of FALSE, TRUE, and NA values in x.

---

cumsum_reset *Cumulative sum unless reset*

Description

Cumulative sum unless reset

Usage

```
cumsum_reset(x, y = as.integer(x))
```

Arguments

x	A logical vector indicating when the sum should *continue*. Missing values in x is an error.
y	Optional: a numeric vector the same length as x to cumulatively sum.

Value

A vector of cumulative sums, resetting whenever x is FALSE. The return type is double if y is double; otherwise an integer vector. Integer overflow wraps around, rather than being promoted to double type, as this function is intended for 'shortish' runs of cumulative sums.

If `length(x) == 0`, y is returned (i.e. `integer(0)` or `double(0)`.

Examples

```
cumsum_reset(c(TRUE, TRUE, FALSE, TRUE, TRUE, TRUE, FALSE))
cumsum_reset(c(TRUE, TRUE, FALSE, TRUE, TRUE, TRUE, FALSE),
             c(1000, 1000, 10000,   10,   20,   33,     0))
```

diam *What is the diameter of set of points?*

Description

Equivalent to `diff(minmax(x))`

Usage

```
diam(x, nThread = getOption("hutilscpp.nThread", 1L))

thinner(x, width, nThread = getOption("hutilscpp.nThread", 1L))
```

Arguments

x	A numeric vector.
nThread	Number of threads to use.
width	`numeric(1)` (For `thinner`, the maximum width)

Value

A single value:

`diam` The difference of `minmax(x)`

`thinner` Equivalent to `diam(x) <= width`

divisible *Divisibility*

Description

Divisibility

Usage

```
divisible(x, d, nThread = getOption("hutilscpp.nThread", 1L))

divisible2(x, nThread = getOption("hutilscpp.nThread", 1L))

divisible16(x, nThread = getOption("hutilscpp.nThread", 1L))
```

Arguments

x	An integer vector
d	`integer(1)`. The divisor.
nThread	The number of threads to use.

Value

Logical vector: TRUE where x is divisible by d.

divisible2,divisible16 are short for (and quicker than) divisible(x, 2) and divisble(x, 16).

---

every_int *Every integer*

Description

Every integer

Usage

```
every_int(nThread = getOption("hutilsc.nThread", 1L), na = NA_integer_)
```

Arguments

nThread	Number of threads.
na	Value for NA_INTEGER.

---

fmatchp *Parallel fastmatching*

Description

fastmatch::fmatch and logical versions, with parallelization.

Usage

```
fmatchp(
  x,
  table,
  nomatch = NA_integer_,
  nThread = getOption("hutilscpp.nThread", 1L),
  fin = FALSE,
  whichFirst = 0L,
  .raw = 0L
)

finp(x, table, nThread = getOption("hutilscpp.nThread", 1L), .raw = 0L)

fnotinp(x, table, nThread = getOption("hutilscpp.nThread", 1L), .raw = 0L)
```

Arguments

x, table, nomatch	As in match.
nThread	Number of threads to use.
fin	TRUE | FALSE Behaviour of return value when value found in table. If FALSE, return the index of table; if TRUE, return TRUE.
whichFirst	integer(1) If 0L, not used. If positive, returns the index of the first element in x found in table; if negative, returns the last element in x found in table.
.raw	integer(1) **0** Return integer or logical as required. **1** Return raw if possible.

Examples

```
x <- c(1L, 4:5)
y <- c(2L, 4:5)
fmatchp(x, y)
fmatchp(x, y, nomatch = 0L)
finp(x, y)
```

---

helper *Helper*

Description

Helper

Usage

```
helper(expr)
```

Arguments

expr	An expression

Value

The expression evaluated.

Examples

```
x6 <- 1:6
helper(x6 + 1)
```

Implies *Implies*

Description

Implies

Usage

```
Implies(x, y, anyNAx = TRUE, anyNAy = TRUE)
```

Arguments

x, y	Logical vectors of equal length.
anyNAx, anyNAy	Whether x,y may contain NA. If FALSE, the function runs faster, but under that assumption.

Value

Logical implies: TRUE unless x is TRUE and y is FALSE.

NA in either x or y results in NA if and only if the result is unknown. In particular NA %implies% TRUE is TRUE and FALSE %implies% NA is TRUE.

If x or y are length-one, the function proceeds as if the length-one vector were recycled to the length of the other.

Examples

```
library(data.table)
CJ(x = c(TRUE,
         FALSE),
   y = c(TRUE,
         FALSE))[, ` x => y` := Implies(x, y)][]

#>        x     y  x => y
#> 1: FALSE FALSE    TRUE
#> 2: FALSE  TRUE    TRUE
#> 3:  TRUE FALSE   FALSE
#> 4:  TRUE  TRUE    TRUE

# NA results:
#> 5:    NA    NA      NA
#> 6:    NA FALSE      NA
#> 7:    NA  TRUE    TRUE
#> 8: FALSE    NA    TRUE
#> 9:  TRUE    NA      NA
```

`is_constant` *Is a vector constant?*

Description

Efficiently decide whether an atomic vector is constant; that is, contains only one value.

Equivalent to

`data.table::uniqueN(x) == 1L`

or

`forecast::is.constant(x)`

Usage

```
is_constant(x, nThread = getOption("hutilscpp.nThread", 1L))

isntConstant(x)
```

Arguments

`x`	An atomic vector. Only logical, integer, double, and character vectors are supported. Others may work but have not been tested.
`nThread`	`integer(1)` Number of threads to use in `is_constant`.

Value

Whether or not the vector `x` is constant:

`is_constant` `TRUE` or `FALSE`. Missing values are considered to be the same as each other, so a vector entirely composed of missing values is considered constant. Note that `is_constant(c(NA_real_, NaN))` is `TRUE`.

`isntConstant` If constant, `0L`; otherwise, the first integer position at which `x` has a different value to the first.

This has the virtue of `!isntConstant(x) == is_constant(x)`.

Multithreaded `is_constant(x, nThread)` should only be used if `x` is expected to be true. It will be faster when `x` is constant but much slower otherwise.

Empty vectors are constant, as are length-one vectors.

Examples

```
library(hutilscpp)
library(data.table)
setDTthreads(1L)
N <- 1e9L
N <- 1e6  # to avoid long-running examples on CRAN
```

```
## Good-cases
nonconst <- c(integer(1e5), 13L, integer(N))
bench_system_time(uniqueN(nonconst) == 1L)
#> process    real
#> 15.734s  2.893s
bench_system_time(is_constant(nonconst))
#> process    real
#>   0.000   0.000
bench_system_time(isntConstant(nonconst))
#> process    real
#>   0.000   0.000

## Worst-cases
consti <- rep(13L, N)
bench_system_time(uniqueN(consti) == 1L)
#> process    real
#>  5.734s  1.202s
bench_system_time(is_constant(consti))
#>   process      real
#> 437.500ms 437.398ms
bench_system_time(isntConstant(consti))
#>   process      real
#> 437.500ms 434.109ms

nonconsti <- c(consti, -1L)
bench_system_time(uniqueN(nonconsti) == 1L)
#> process    real
#> 17.812s  3.348s
bench_system_time(is_constant(nonconsti))
#>   process      real
#> 437.500ms 431.104ms
bench_system_time(isntConstant(consti))
#>   process      real
#> 484.375ms 487.588ms

constc <- rep("a", N)
bench_system_time(uniqueN(constc) == 1L)
#> process    real
#> 11.141s  3.580s
bench_system_time(is_constant(constc))
#> process    real
#>  4.109s  4.098s

nonconstc <- c(constc, "x")
bench_system_time(uniqueN(nonconstc) == 1L)
#> process    real
#> 22.656s  5.629s
bench_system_time(is_constant(nonconstc))
#> process    real
#>  5.906s  5.907s
```

is_sorted *Is a vector sorted?*

Description

Is a vector sorted?

Usage

```
is_sorted(x, asc = NA)

isntSorted(x, asc = NA)
```

Arguments

x	An atomic vector.
asc	Single logical. If NA, the default, a vector is considered sorted if it is either sorted ascending or sorted descending; if FALSE, a vector is sorted only if sorted descending; if TRUE, a vector is sorted only if sorted ascending.

Value

is_sorted returns TRUE or FALSE

isntSorted returns 0 if sorted or the first position that proves the vector is not sorted

logical3 *Vectorized logical with support for short-circuits*

Description

Vectorized logical with support for short-circuits

Usage

```
and3(x, y, z = NULL, nas_absent = FALSE)

or3(x, y, z = NULL)
```

Arguments

x, y, z	Logical vectors. If z is NULL the function is equivalent to the binary versions; only x and y are used.
nas_absent	(logical, default: FALSE) Can it be assumed that x,y,z have no missing values? Set to TRUE when you are sure that that is the case; setting to TRUE falsely has no defined behaviour.

Value

For `and3`, the same as `x & y & z`; for `or3`, the same as `x | y | z`, designed to be efficient when component-wise short-circuiting is available.

`logical3s` *Complex logical expressions*

Description

Performant implementations of `&` et `or`. Performance is high when the expressions are long (i.e. over 10M elements) and in particular when they are of the form `lhs <op> rhs` for binary `<op>`.

Usage

```
and3s(
  exprA,
  exprB = NULL,
  exprC = NULL,
  ...,
  nThread = getOption("hutilscpp.nThread", 1L),
  .parent_nframes = 1L,
  type = c("logical", "raw", "which")
)

or3s(
  exprA,
  exprB = NULL,
  exprC = NULL,
  ...,
  nThread = getOption("hutilscpp.nThread", 1L),
  .parent_nframes = 1L,
  type = c("logical", "raw", "which")
)
```

Arguments

`exprA`, `exprB`, `exprC`, `...`
: Expressions of the form `x <op> y`. with `<op>` one of the standard binary operators.

  Only `exprA` is required, all following expressions are optional.

`nThread`
: `integer(1)` Number of threads to use.

`.parent_nframes`
: `integer(1)` For internal use. Passed to `eval.parent`.

`type`
: The type of the result. `which` corresponds to the indices of `TRUE` in the result. Type `raw` is available for a memory-constrained result, though the result will not be interpreted as logical.

Value

and3s and or3s return exprA & exprB & exprC and exprA | exprB | exprC respectively. If any expression is missing it is considered TRUE for and3s and FALSE for or3s; in other words only the results of the other expressions count towards the result.

`match_nrst_haversine` *Match coordinates to nearest coordinates*

Description

When geocoding coordinates to known addresses, an efficient way to match the given coordinates with the known is necessary. This function provides this efficiency by using C++ and allowing approximate matching.

Usage

```
match_nrst_haversine(
  lat,
  lon,
  addresses_lat,
  addresses_lon,
  Index = seq_along(addresses_lat),
  cartesian_R = NULL,
  close_enough = 10,
  excl_self = FALSE,
  as.data.table = TRUE,
  .verify_box = TRUE
)
```

Arguments

lat, lon	Coordinates to be geocoded. Numeric vectors of equal length.
addresses_lat, addresses_lon	Coordinates of known locations. Numeric vectors of equal length (likely to be a different length than the length of lat, except when excl_self = TRUE).
Index	A vector the same length as lat to encode the match between lat,lon and addresses_lat,addresses_lon. The default is to use the integer position of the nearest match to addresses_lat,addresses_lon.
cartesian_R	The maximum radius of any address from the points to be geocoded. Used to accelerate the detection of minimum distances. Note, as the argument name suggests, the distance is in cartesian coordinates, so a small number is likely.
close_enough	The distance, in metres, below which a match will be considered to have occurred. (The distance that is considered "close enough" to be a match.) For example, close_enough = 10 means the first location within ten metres will be matched, even if a closer match occurs later. May be provided as a string to emphasize the units, e.g. close_enough = "0.25km". Only km and m are permitted.

excl_self	(bool, default: FALSE) For each x_i of the first coordinates, exclude the y_i-th point when determining closest match. Useful to determine the nearest neighbour within a set of coordinates, *viz.* match_nrst_haversine(x, y, x, y, excl_self = TRUE).
as.data.table	Return result as a data.table? If FALSE, a list is returned. TRUE by default to avoid dumping a huge list to the console.
.verify_box	Check the initial guess against other points within the box of radius ℓ^∞.

Value

A list (or data.table if as.data.table = TRUE) with two elements, both the same length as lat, giving for point lat,lon:

pos the position (or corresponding value in Table) in addresses_lat,addresses_lon nearest to lat, lon.

dist the distance, in kilometres, between the two points.

Examples

```
lat2 <- runif(5, -38, -37.8)
lon2 <- rep(145, 5)

lat1 <- c(-37.875, -37.91)
lon1 <- c(144.96, 144.978)

match_nrst_haversine(lat1, lon1, lat2, lon2)
match_nrst_haversine(lat1, lon1, lat1, lon1, 11:12, excl_self = TRUE)
```

---

minmax *Minimum and maximum*

Description

Minimum and maximum

Usage

```
minmax(x, empty_result = NULL, nThread = getOption("hutilscpp.nThread", 1L))
```

Arguments

x	An atomic vector.
empty_result	What should be returned when length(x) == 0?
nThread	Number of threads to be used.

Value

Vector of two elements, the minimum and maximum of x, or NULL.

ModeC *Most common element*

Description

Most common element

Usage

```
ModeC(
  x,
  nThread = getOption("hutilscpp.nThread", 1L),
  .range_fmatch = 1000000000,
  option = 1L
)
```

Arguments

x	An atomic vector.
nThread	Number of threads to use.
.range_fmatch	If the range of x differs by more than this amount, the mode will be calculated via fmatchp.
option	integer(1) Handle exceptional cases: **0** Returns NULL quietly. **1** Returns an error if the mode cannot be calculated. **2** Emits a warning if the mode cannot be calculate, falls back to hutils::Mode

Examples

```
ModeC(c(1L, 1L, 2L))
```

pmaxC *Parallel maximum/minimum*

Description

Faster pmax() and pmin().

Usage

```
pmaxC(
  x,
  a,
  in_place = FALSE,
  keep_nas = FALSE,
  dbl_ok = NA,
  nThread = getOption("hutilscpp.nThread", 1L)
)

pminC(
  x,
  a,
  in_place = FALSE,
  keep_nas = FALSE,
  dbl_ok = NA,
  nThread = getOption("hutilscpp.nThread", 1L)
)

pmax0(
  x,
  in_place = FALSE,
  sorted = FALSE,
  keep_nas = FALSE,
  nThread = getOption("hutilscpp.nThread", 1L)
)

pmin0(
  x,
  in_place = FALSE,
  sorted = FALSE,
  keep_nas = FALSE,
  nThread = getOption("hutilscpp.nThread", 1L)
)

pmaxV(
  x,
  y,
  in_place = FALSE,
  dbl_ok = TRUE,
  nThread = getOption("hutilscpp.nThread", 1L)
)

pminV(
  x,
  y,
  in_place = FALSE,
  dbl_ok = TRUE,
```

```
  nThread = getOption("hutilscpp.nThread", 1L)
)

pmax3(x, y, z, in_place = FALSE)

pmin3(x, y, z, in_place = FALSE)
```

Arguments

x	`numeric(n)` A numeric vector.
a	`numeric(1)` A single numeric value.
in_place	`TRUE | FALSE`**, default:** `FALSE` Should `x` be modified in-place? For advanced use only.
keep_nas	`TRUE | FALSE`**, default:** `FALSE` Should `NA`s values be preserved? By default, `FALSE`, so the behaviour of the function is dependent on the representation of `NA`s at the C++ level.
dbl_ok	`logical(1)`**, default:** `NA` Is it acceptable to return a non-integer vector if `x` is integer? This argument will have effect `a` is both double and cannot be coerced to `integer`: If `NA`, the default, a message is emitted whenever a double vector needs to be returned. If `FALSE`, an error is returned. If `TRUE`, neither an error nor a message is returned.
nThread	`integer(1)` The number of threads to use. Combining `nThread > 1` and `in_place = TRUE` is not supported.
sorted	`TRUE | FALSE`**, default:** `FALSE` Is `x` known to be sorted? If `TRUE`, `x` is assumed to be sorted. Thus the first zero determines whether the position at which zeroes start or end.
y, z	`numeric(n)` Other numeric vectors the same length as `x`

Value

Versions of `pmax` and `pmin`, designed for performance.

When `in_place = TRUE`, the values of `x` are modified in-place. For advanced users only.

The differences are:

`pmaxC(x, a)` **and** `pminC(x, a)` Both `x` and `a` must be numeric and `a` must be length-one.

Note

This function will always be faster than `pmax(x, a)` when `a` is a single value, but can be slower than `pmax.int(x, a)` when `x` is short. Use this function when comparing a numeric vector with a single value.

Use `in_place = TRUE` only within functions when you are sure it is safe, i.e. not a reference to something outside the environment.

By design, the functions first check whether `x` will be modified before allocating memory to a new vector. For example, if all values in `x` are nonnegative, the vector is returned.

Examples

```
pmaxC(-5:5, 2)
pmaxC(1:4, 5.5)
pmaxC(1:4, 5.5, dbl_ok = TRUE)
# pmaxC(1:4, 5.5, dbl_ok = FALSE)  # error
```

`poleInaccessibility` *Find a binary pole of inaccessibility*

Description

Find a binary pole of inaccessibility

Usage

```
poleInaccessibility2(
  x = NULL,
  y = NULL,
  DT = NULL,
  x_range = NULL,
  y_range = NULL,
  copy_DT = TRUE
)

poleInaccessibility3(
  x = NULL,
  y = NULL,
  DT = NULL,
  x_range = NULL,
  y_range = NULL,
  copy_DT = TRUE,
  test_both = TRUE
)
```

Arguments

x, y	Coordinates.
DT	A `data.table` containing `LONGITUDE` and `LATITUDE` to define the `x` and `y` coordinates.
x_range, y_range	Numeric vectors of length-2; the range of `x` and `y`. Use this rather than the default when the 'vicinity' of `x,y` is different from the minimum closed rectangle covering the points.
copy_DT	(logical, default: `TRUE`) Run `copy` on `DT` before proceeding. If `FALSE`, `DT` have additional columns updated by reference.
test_both	(logical, default: `TRUE`) For `3`, test both stretching vertically then horizontally and horizontally then vertically.

Value

`poleInaccessibility2` A named vector containing the `xmin`, `xmax` and `ymin`, `ymax` coordinates of the largest rectangle of width an integer power of two that is empty.

`poleInaccessibility3` Starting with the rectangle formed by `poleInaccessibility2`, the rectangle formed by stretching it out vertically and horizontally until the edges intersect the points `x,y`

Examples

```
library(data.table)
library(hutils)
# A square with a 10 by 10 square of the northeast corner removed
x <- runif(1e4, 0, 100)
y <- runif(1e4, 0, 100)
DT <- data.table(x, y)
# remove the NE corner
DT_NE <- DT[implies(x > 90, y < 89)]
DT_NE[, poleInaccessibility2(x, y)]
DT_NE[, poleInaccessibility3(x, y)]
```

---

range_rcpp *Range C++*

Description

Range of a vector using Rcpp.

Usage

```
range_rcpp(
  x,
  anyNAx = anyNA(x),
  warn_empty = TRUE,
  integer0_range_is_integer = FALSE
)
```

Arguments

`x`	A vector for which the range is desired. Vectors with missing values are not supported and have no definite behaviour.
`anyNAx`	(logical, default: `anyNA(x)` lazily). Set to `TRUE` only if `x` is known to contain no missing values (including `NaN`).
`warn_empty`	(logical, default: `TRUE`) If `x` is empty (i.e. has no length), should a warning be emitted (like `range`)?

integer0_range_is_integer
: (logical, default: FALSE) If x is a length-zero integer, should the result also be an integer? Set to FALSE by default in order to be compatible with range, but can be set to TRUE if an integer result is desired, in which case range_rcpp(integer()) is (INT_MAX, -INT_MAX).

Value

A length-4 vector, the first two positions give the range and the next two give the positions in x where the max and min occurred.

This is almost equivalent to c(range(x), which.min(x), which.max(x)). Note that the type is not strictly preserved, but no loss should occur. In particular, logical x results in an integer result, and a double x will have double values for which.min(x) and which.max(x).

A completely empty, logical x returns c(NA, NA, NA, NA) as an integer vector.

Examples

```
x <- rnorm(1e3) # Not noticeable at this scale
bench_system_time(range_rcpp(x))
bench_system_time(range(x))
```

---

squish *Squish into a range*

Description

Squish into a range

Usage

```
squish(x, a, b, in_place = FALSE)
```

Arguments

x	A numeric vector.
a, b	Upper and lower bounds
in_place	(logical, default: FALSE) Should the function operate on x in place?

Value

A numeric/integer vector with the values of x "squished" between a and b; values above b replaced with b and values below a replaced with a.

Examples

```
squish(-5:5,-1L, 1L)
```

`sum_and3s` *Sum of logical expressions*

Description

Sum of logical expressions

Usage

```
sum_and3s(
  exprA,
  exprB,
  exprC,
  ...,
  nThread = getOption("hutilscpp.nThread", 1L),
  .env = parent.frame()
)

sum_or3s(
  exprA,
  exprB,
  exprC,
  ...,
  .env = parent.frame(),
  nThread = getOption("hutilscpp.nThread", 1L)
)
```

Arguments

`exprA`, `exprB`, `exprC`, `...`	Expressions of the form `x <op> y`. with `<op>` one of the standard binary operators.
`nThread`	`integer(1)` Number of threads to use.
`.env`	The environment in which the expressions are to be evaluated.

Value

Equivalent to `sum(exprA & exprB & exprC)` or `sum(exprA | exprB | exprC)` as desired.

sum_isna *Number of missing values*

Description

The count of missing values in an atomic vector, equivalent to to `sum(is.na(x))`.

Usage

```
sum_isna(x, do_anyNA = TRUE, nThread = getOption("hutilscpp.nThread", 1L))
```

Arguments

x	An atomic vector.
do_anyNA	Should `anyNA(x)` be executed before an attempt to count the `NA`'s in `x` one-by-one? By default, set to `TRUE`, since it is generally quicker. It will only be slower when `NA` is rare and occurs late in `x`. Ignored silently if `nThread != 1`.
nThread	`nThread` Number of threads to use.

Examples

```
sum_isna(c(1:5, NA))
sum_isna(c(NaN, NA))  # 2 from v0.4.0 (Sep 2020)
```

unique_fmatch *Distinct elements*

Description

Using the `fastmatch` hash functions, determine the unique elements of a vector, and the number of distinct elements.

Usage

```
unique_fmatch(x, nThread = getOption("hutilscpp.nThread", 1L))

uniqueN_fmatch(x, nThread = getOption("hutilscpp.nThread", 1L))
```

Arguments

x	An atomic vector.
nThread	Number of threads to use.

Value

Equivalent to `unique(x)` or `data.table::uniqueN(x)` respectively.

which3 *which of three vectors are the elements (all, any) true?*

Description

which of three vectors are the elements (all, any) true?

Usage

```
which3(
  x,
  y,
  z,
  And = TRUE,
  anyNAx = anyNA(x),
  anyNAy = anyNA(y),
  anyNAz = anyNA(z)
)
```

Arguments

x, y, z	Logical vectors. Either the same length or length-1
And	Boolean. If TRUE, only indices where all of x, y, z are TRUE are returned; if FALSE, any index where x, y, z are TRUE are returned.
anyNAx, anyNAy, anyNAz	Whether or not the inputs have NA.

whichs *Separated which*

Description

Same as which(exprA) where exprA is a binary expression.

Usage

```
whichs(
  exprA,
  .env = parent.frame(),
  nThread = getOption("hutilscpp.nThread", 1L)
)
```

Arguments

exprA	An expression. Useful when of the form a <op> b for a an atomic vector. Long expressions are not supported.
.env	The environment in which exprA is to be evaluated.
nThread	Number of threads to use.

Value

Integer vector, the indices of exprA that return TRUE.

---

which_first *Where does a logical expression first return* TRUE*?*

Description

A faster and safer version of which.max applied to simple-to-parse logical expressions.

Usage

```
which_first(
  expr,
  verbose = FALSE,
  reverse = FALSE,
  sexpr,
  eval_parent_n = 1L,
  suppressWarning = getOption("hutilscpp_suppressWarning", FALSE),
  use.which.max = FALSE
)

which_last(
  expr,
  verbose = FALSE,
  reverse = FALSE,
  suppressWarning = getOption("hutilscpp_suppressWarning", FALSE)
)
```

Arguments

expr	An expression, such as x == 2.
verbose	logical(1), **default:** FALSE If TRUE a message is emitted if expr could not be handled in the advertised way.
reverse	logical(1), **default:** FALSE Scan expr in reverse.
sexpr	Equivalent to substitute(expr). For internal use.
eval_parent_n	Passed to eval.parent, the environment in which expr is evaluated.

suppressWarning
: Either a FALSE or TRUE, whether or not warnings should be suppressed. Also supports a string input which suppresses a warning if it matches as a regular expression.

use.which.max
: If TRUE, which.max is dispatched immediately, even if expr would be amenable to separation. Useful when evaluating many small expr's when these are known in advance.

Details

If the expr is of the form LHS <operator> RHS and LHS is a single symbol, operator is one of ==, !=, >, >=, <, <=, %in%, or %between%, and RHS is numeric, then expr is not evaluated directly; instead, each element of LHS is compared individually.

If expr is not of the above form, then expr is evaluated and passed to which.max.

Using this function can be significantly faster than the alternatives when the computation of expr would be expensive, though the difference is only likely to be clear when length(x) is much larger than 10 million. But even for smaller vectors, it has the benefit of returning 0L if none of the values in expr are TRUE, unlike which.max.

Compared to Position for an appropriate choice of f the speed of which_first is not much faster when the expression is TRUE for some position. However, which_first is faster when all elements of expr are FALSE. Thus which_first has a smaller worst-case time than the alternatives for most x.

Missing values on the RHS are handled specially. which_first(x %between% c(NA, 1)) for example is equivalent to which_first(x <= 1), as in data.table::between.

Value

The same as which.max(expr) or which(expr)[1] but returns 0L when expr has no TRUE values.

Examples

```
N <- 1e5
# N <- 1e8  ## too slow for CRAN

# Two examples, from slowest to fastest,
# run with N = 1e8 elements

                                      # seconds
x <- rep_len(runif(1e4, 0, 6), N)
bench_system_time(x > 5)
bench_system_time(which(x > 5))       # 0.8
bench_system_time(which.max(x > 5))   # 0.3
bench_system_time(which_first(x > 5)) # 0.000

## Worst case: have to check all N elements
x <- double(N)
bench_system_time(x > 0)
bench_system_time(which(x > 0))       # 1.0
bench_system_time(which.max(x > 0))   # 0.4  but returns 1, not 0
```

```
bench_system_time(which_first(x > 0))  # 0.1

x <- as.character(x)
# bench_system_time(which(x == 5))     # 2.2
bench_system_time(which.max(x == 5))   # 1.6
bench_system_time(which_first(x == 5)) # 1.3
```

`which_firstNA` *First/last position of missing values*

Description

Introduced in v `1.6.0`

Usage

```
which_firstNA(x)

which_lastNA(x)
```

Arguments

`x`	An atomic vector.

Value

The position of the first/last missing value in `x`.

Examples

```
N <- 1e8
N <- 1e6  # for CRAN etc
x <- c(1:1e5, NA, integer(N))
bench_system_time(which.max(is.na(x))) # 123ms
bench_system_time(Position(is.na, x))  #  22ms
bench_system_time(which_firstNA(x))    #  <1ms
```

which_true_onwards *At which point are all values true onwards*

Description

At which point are all values true onwards

Usage

```
which_true_onwards(x)
```

Arguments

x	A logical vector. NA values are not permitted.

Value

The position of the first TRUE value in x at which all the following values are TRUE.

Examples

```
which_true_onwards(c(TRUE, FALSE, TRUE, TRUE, TRUE))
```

xor2 *Exclusive or*

Description

Exclusive or

Usage

```
xor2(x, y, anyNAx = TRUE, anyNAy = TRUE)
```

Arguments

x, y	Logical vectors.
anyNAx, anyNAy	Could x and y possibly contain NA values? Only set to FALSE if known to be free of NA.

Index