# Package 'hutilscpp'

October 16, 2019

**Title** Miscellaneous Functions in C++

**Version** 0.3.0

**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

**LazyData** true

**LinkingTo** Rcpp

**Imports** Rcpp, data.table, hutils, utils

**RoxygenNote** 6.1.1

**Suggests** bench, testthat (>= 2.1.0), TeXCheckR, covr

**NeedsCompilation** yes

**Author** Hugh Parsonage [aut, cre]

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

**Repository** CRAN

**Date/Publication** 2019-10-16 11:20:02 UTC

## R topics documented:

---

`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 `NA`s *known* to be absent from `x`? If `nas_absent = NA`, the default, `x` will be searched for `NA`s; 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]`.<br>If `nas_absent = TRUE` but `x` has missing values then the result is unreliable. |
| `na_is_outside` | (logical, default: `NA`) How should `NA`s in `x` be treated?<br>**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.#'<br>**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)
```

---

# are_even *Are even*

---

## Description

Are even

## Usage

```
are_even(x, check_integerish = TRUE)

which_are_even(x, check_integerish = TRUE)
```

## Arguments

| | |
|---|---|
| x | An integer vector. Double vectors may also be used. |
| 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. |

## 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. |

---

`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*. |
| y | Optional: a numeric vector the same length as x to cumulatively sum. |

## Value

If y is a double vector, a double vector of cumulative sums, resetting whenever x is FALSE; otherwise an integer vector.

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))
```

## do_pmaxC *Internal pmaxC helpers*

### Description

Internal functions used when the overheads of assertions would be too expensive. The _IP_ flavours modify in place.

### Usage

```
do_pmaxC_dbl(x, a, in_place = FALSE)

do_pmaxC_int(x, a, in_place = FALSE)

do_pmax0(x, in_place = FALSE)

do_pmaxIPnum0(x)

do_pmaxIPint0(x)
```

### Arguments

| | |
|---|---|
| x | A numeric/integer vector. |
| a | A single numeric/integer. |
| in_place | Modify x in place? |

## do_pmaxV *Parallel maximum in C++*

### Description

A faster pmax().

### Usage

```
do_pmaxNumNum(x, y, in_place = FALSE)

do_pmaxIntInt(x, y, in_place = FALSE)
```

### Arguments

| | |
|---|---|
| x | A numeric vector. |
| y | A numeric vector, the same length as x. |
| in_place | (bool, default: false) Should the function operate on x in-place? |

### Value

The parallel maximum of the input values.

---

`do_pminC` *Parallel maximum*

---

### Description

A faster `pmin()`.

### Arguments

| | |
|---|---|
| `x` | A numeric vector. |
| `a` | A single numeric value. |
| `in_place` | (bool, default: `false`) Should the function operate on `x` in-place? |

### Value

The parallel minimum of the input values. The `0` versions are shortcuts for `a = 0`.

### Note

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

---

`do_pminV` *Parallel maximum*

---

### Description

A faster `pmin()`.

### Usage

```
do_pminV_dbl(x, y, in_place = FALSE)

do_pminV_int(x, y, in_place = FALSE)
```

### Arguments

| | |
|---|---|
| `x` | A numeric vector. |
| `y` | A numeric vector, the same length as x. |
| `in_place` | (bool, default: `false`) Modify `x` in-place? |

### Value

The parallel maximum of the input values.

---

## helper *Helper*

---

### Description

Helper

### Usage

```
helper(expr)
```

### Arguments

| | |
|---|---|
| expr | An expression |

### Value

The expression evaluated.

### Examples

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

---

## 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)

isntConstant(x)
```

## Arguments

| | |
|---|---|
| x | An atomic vector. Only logical, integer, double, and character vectors are supported. Others may work but have not been tested. |

## Value

Whether or not the vector x is constant:

`is_constant` `TRUE` or `FALSE`. Missing values are considered to be the same.

`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)`.

## Examples

```
library(hutilscpp)
library(data.table)
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
```

---

`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.

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.)<br>For example, close_enough = 10 means the first location within ten metres will be matched, even if a closer match occurs later.<br>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 $\ell^\infty$. |

## 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, 0L)
match_nrst_haversine(lat1, lon1, lat1, lon1, 11:12, excl_self = TRUE)
```

---

# pmaxC *Parallel maximum/minimum*

---

## Description

Faster `pmax()` and `pmin()`.

## Usage

```
pmaxC(x, a, in_place = FALSE)

pmax0(x, in_place = FALSE, sorted = FALSE)

pmaxV(x, y, in_place = FALSE)

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

## Arguments

| | |
|---|---|
| `x` | A numeric vector. |
| `a` | A single numeric value. |
| `in_place` | (logical, default: `FALSE`) Should `x` be modified in-place. |
| `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` | Other numeric vectors the same length as `x` |

## Value

The parallel maximum/minimum of the input values. `pmax0(x)` is shorthand for `pmaxC(x,0)`, i.e. convert negative values in `x` to 0.

## 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.

If `x` is nonnegative so `pmax0(x) = identity(x)` the function will be much faster still, as the `C++` code only starts allocating once a negative value is found.

## Examples

```
pmaxC(-5:5, 2)
```

---

# pminC *Parallel minimum*

---

## Description

Parallel minimum

## Usage

```
pmin0(x, in_place = FALSE)

pminV(x, y, in_place = FALSE)

pminC(x, a = 0L, in_place = FALSE)

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

## Arguments

| | |
|---|---|
| `x` | A numeric vector. |
| `in_place` | (logical, default: `FALSE`) Should `x` be modified in-place. |
| `y, z` | Other numeric vectors. |
| `a` | A single number. |

## Details

The type of `x` is preserved as far as possible.

## Value

Same as `pmin(x,0)`.

`pmin0(x)` = `pmin(x,0)`

`pminV(x,y)` = `pmin(x,y)`

`pminC(x,a)` = `pmin(x,a)` for length-one `a`.

`pmin3(x,y,z)` = `pmin(x,pmin(y,z))`.

## Examples

```
pminV(10:1, 1:10)
pmin0(-5:5)
seq_out <- function(x, y) seq(x, y, length.out = 10)
pmin3(seq_out(0, 10), seq_out(-5, 50), seq_out(20, -10))
```

---

# 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_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)
```

### 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`. |

### Examples

```
sum_isna(c(1:5, NA))
```

## 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`. |

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)
```

## Arguments

| | |
|---|---|
| expr | An expression, such as x == 2. |
| verbose | (logical, default: FALSE) If TRUE a message is emitted if expr could not be handled in the advertised way. |

## Details

If the expr is of the form LHS <operator> RHS and LHS is a single symbol, operator is one of ==, !=, >, >=, <, <=, or %in%. and RHS is a single numeric value, 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.

## 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_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