

# Package ‘NPE’

August 16, 2020

**Type** Package

**Title** Non-Parametric Estimators

**Version** 1.0

**Date** 2020-08-03

**Author** Sumit Sethi

**Maintainer** Sumit Sethi <sumit.sethi@nyu.edu>

## Description

Functions for calculating a variety of nonparametric estimators, including estimators of location (median, Hodges-Lehmann), of dispersion (Median Absolute Deviation), and of dependency-covariance (Theil-Sen).

It also implements the nonparametric statistics of the Wilcoxon Signed Rank test, the Mann-Whitney-Wilcoxon

Rank Sum test, and the Kruskal-Wallis test. It also implements PCA.

**License** MPL-2.0

**Imports** Rcpp (>= 1.0.4.6), RcppParallel

**LinkingTo** Rcpp, RcppArmadillo, RcppParallel, BH

**SystemRequirements** GNU make

**RoxygenNote** 7.1.0

**Encoding** UTF-8

## R topics documented:

calc_pca . . . . .	2
calc_ranksWithTies . . . . .	2
hle . . . . .	3
kruskalWalliceTest . . . . .	4
medianAbsoluteDeviation . . . . .	5
med_ian . . . . .	6
rolling_mad . . . . .	7
rolling_median . . . . .	7
theilSenEstimator . . . . .	8
wilcoxonMannWhitneyTest . . . . .	9
wilcoxonSignedRankTest . . . . .	10

<b>Index</b>	<b>12</b>
--------------	-----------

---

calc_pca	<i>Performs a principle component analysis on given matrix or time series using RcppArmadillo.</i>
----------	----------------------------------------------------------------------------------------------------

---

### Description

Performs a principle component analysis on given *matrix* or *time series* using RcppArmadillo.

### Usage

```
calc_pca(mat_rix)
```

### Arguments

mat\_rix            *A matrix or a time series.*

### Details

The function calc\_pca() performs a principle component analysis on a *matrix* using RcppArmadillo.

### Value

*A matrix* of variable loadings (i.e. a matrix whose columns contain the eigenvectors).

### Examples

```
## Not run:
# Create a matrix of random returns
re_returns <- matrix(rnorm(5e6), nc=5)
# Compare calc_pca() with standard prcomp()
all.equal(drop(NPE::calc_pca(re_returns),
  prcomp(re_returns))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=NPE::calc_pca(re_returns),
  rcode=prcomp(re_returns),
  times=10))[, c(1, 4, 5)] # end microbenchmark summary

## End(Not run)
```

---

calc_ranksWithTies	<i>Calculate the ranks of the elements of a vector or a single-column time series using RcppArmadillo and boost.</i>
--------------------	----------------------------------------------------------------------------------------------------------------------

---

### Description

Calculate the ranks of the elements of a *vector* or a single-column *time series* using RcppArmadillo and boost.

**Usage**

```
calc_ranksWithTies(vec_tor)
```

**Arguments**

`vec_tor`            *A vector or a single-column time series.*

**Details**

The function `calc_ranks()` calculates the ranks of the elements of a *vector* or a single-column *time series*. It *averages* the ranks in case of ties. It uses the boost function `boost::sort::parallel_stable_sort` for sorting array in parallel fashion.

**Value**

*A double vector with the ranks of the elements of the vector.*

**Examples**

```
## Not run:
# Create a vector of random data
da_ta <- round(runif(7), 2)
# Calculate the ranks of the elements in two ways
all.equal(rank(da_ta), drop(NPE::calc_ranksWithTies(da_ta)))
# Create a time series of random data
da_ta <- xts::xts(runif(7), seq.Date(Sys.Date(), by=1, length.out=7))
# Calculate the ranks of the elements in two ways
all.equal(rank(coredata(da_ta)), drop(NPE::calc_ranksWithTies(da_ta)))
# Compare the speed of this function with RcppArmadillo and R code
da_ta <- runif(7)
library(microbenchmark)
summary(microbenchmark(
  rcpp=calc_ranks(da_ta),
  rcode=rank(da_ta),
  boost=calc_ranksWithTies(da_ta)
  times=10))[, c(1, 4, 5)] # end microbenchmark summary

## End(Not run)
```

---

hle	<i>Calculate the non parametric Hodges-Lehmann estimator of location for a vector or a single-column time series using RcppArmadillo and RcppParallel.</i>
-----	------------------------------------------------------------------------------------------------------------------------------------------------------------

---

**Description**

Calculate the non parametric Hodges-Lehmann estimator of location for a *vector* or a single-column *time series* using RcppArmadillo and RcppParallel.

**Usage**

```
hle(vec_tor)
```

**Arguments**

`vec_tor`                      A *vector* or a single-column *time series*.

**Details**

The function `hle()` calculates the Hodges-Lehmann estimator of the *vector*, using `RcppArmadillo` and `RcppParallel`. The function `hle()` is very much faster than function `wilcox.test()` in R.

**Value**

A single *double* value representing Hodges-Lehmann estimator of the vector.

**Examples**

```
## Not run:
# Create a vector of random returns
re_turns <- rnorm(1e6)
# Compare hle() with wilcox.test()
all.equal(drop(NPE:hle(re_turns)),
  wilcox.test(re_turns, conf.int = TRUE))
# Compare the speed of RcppParallel with R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=NPE:hle(re_turns),
  rcode=wilcox.test(re_turns, conf.int = TRUE),
  times=10))[, c(1, 4, 5)] # end microbenchmark summary

## End(Not run)
```

---

<code>kruskalWalliceTest</code>	<i>Performs a Kruskal-Wallis rank sum test. using Rcpp and boost.</i>
---------------------------------	-----------------------------------------------------------------------

---

**Description**

Performs a Kruskal-Wallis rank sum test. using Rcpp and boost.

**Usage**

```
kruskalWalliceTest(x)
```

**Arguments**

`x`                              A *List* of numeric data vectors

**Details**

The function `kruskalWalliceTest()` performs a Kruskal-Wallis rank sum test of the null hypothesis that the location parameters of the distribution of `x` are the same in each group. The alternative is that they differ in at least in one.

**Value**

A *double* indicating p-value of the test.

**Examples**

```
## Not run:
x <- c(2.9, 3.0, 2.5, 2.6, 3.2) # normal subjects
y <- c(3.8, 2.7, 4.0, 2.4)      # with obstructive airway disease
z <- c(2.8, 3.4, 3.7, 2.2, 2.0) # with asbestosis

# Carry out Kruskal wallice rank sum test on the elements in two ways
all.equal(kruskal.test(list(x, y, z))$p.value, drop(NPE::kruskalWalliceTest(list(x, y, z))))
# Compare the speed of Rcpp and R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=kruskalWalliceTest(list(x, y, z)),
  rcode=kruskal.test(list(x, y, z))$p.value,
  times=10))[, c(1, 4, 5)] # end microbenchmark summary

## End(Not run)
```

---

medianAbsoluteDeviation

*Calculate the Median absolute deviation of a vector or a single-column time series using RcppArmadillo.*

---

**Description**

Calculate the Median absolute deviation of a *vector* or a single-column *time series* using RcppArmadillo.

**Usage**

```
medianAbsoluteDeviation(vec_tor)
```

**Arguments**

`vec_tor`            A *vector* or a single-column *time series*.

**Details**

The function `medianAbsoluteDeviation()` calculates the median of the *vector*, using RcppArmadillo. The function `medianAbsoluteDeviation()` is several times faster than `mad()` in R.

**Value**

A single *double* value representing median absolute deviation of the vector.

**Examples**

```
## Not run:
# Create a vector of random returns
re_turns <- rnorm(1e6)
# Compare medianAbsoluteDeviation() with mad()
all.equal(drop(NPE::medianAbsoluteDeviation(re_turns)),
  mad(re_turns))
# Compare the speed of RcppArmadillo with R code
```

```
library(microbenchmark)
summary(microbenchmark(
  rcpp=NPE::medianAbsoluteDeviation(re_returns),
  rcode=mad(re_returns),
  times=10))[, c(1, 4, 5)] # end microbenchmark summary

## End(Not run)
```

---

med_ian	<i>Calculate the median of a vector or a single-column time series using RcppArmadillo.</i>
---------	---------------------------------------------------------------------------------------------

---

## Description

Calculate the median of a *vector* or a single-column *time series* using RcppArmadillo.

## Usage

```
med_ian(vec_tor)
```

## Arguments

vec\_tor      *A vector or a single-column time series.*

## Details

The function med\_ian() calculates the median of the *vector*, using RcppArmadillo. The function med\_ian() is several times faster than median() in R.

## Value

A single *double* value representing median of the vector.

## Examples

```
## Not run:
# Create a vector of random returns
re_returns <- rnorm(1e6)
# Compare med_ian() with median()
all.equal(drop(NPE::med_ian(re_returns)),
  median(re_returns))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=NPE::med_ian(re_returns),
  rcode=median(re_returns),
  times=10))[, c(1, 4, 5)] # end microbenchmark summary

## End(Not run)
```

---

rolling_mad	<i>Calculate the rolling median absolute deviation over a vector or a single-column time series using RcppArmadillo and RcppParallel.</i>
-------------	-------------------------------------------------------------------------------------------------------------------------------------------

---

**Description**

Calculate the rolling median absolute deviation over a *vector* or a single-column *time series* using RcppArmadillo and RcppParallel.

**Usage**

```
rolling_mad(vec_tor, look_back)
```

**Arguments**

vec_tor	<i>A vector or a single-column time series.</i>
look_back	The length of look back interval, equal to the number of elements of data used for calculating the median.

**Details**

The function rolling\_mad() calculates a vector of rolling medians, over a *vector* of data, using RcppArmadillo and RcppParallel.

**Value**

A column *vector* of the same length as the argument vect\_tor.

**Examples**

```
## Not run:
# Create a vector of random returns
re_returns <- rnorm(1e6)
rolling_mad(re_returns)

## End(Not run)
```

---

rolling_median	<i>Calculate the rolling median over a vector or a single-column time series using RcppArmadillo and RcppParallel.</i>
----------------	------------------------------------------------------------------------------------------------------------------------

---

**Description**

Calculate the rolling median over a *vector* or a single-column *time series* using RcppArmadillo and RcppParallel.

**Usage**

```
rolling_median(vec_tor, look_back)
```

**Arguments**

<code>vec_tor</code>	A <i>vector</i> or a single-column <i>time series</i> .
<code>look_back</code>	The length of look back interval, equal to the number of elements of data used for calculating the median.

**Details**

The function `rolling_median()` calculates a vector of rolling medians, over a *vector* of data, using *RcppArmadillo* and *RcppParallel*. The function `rolling_median()` is faster than `roll::roll_median()` which uses *Rcpp*.

**Value**

A column *vector* of the same length as the argument `vec_tor`.

**Examples**

```
## Not run:
# Create a vector of random returns
re_turns <- rnorm(1e6)
# Compare rolling_median() with roll::roll_median()
all.equal(drop(NPE::rolling_median(re_turns, look_back=11)),
  roll::roll_median(re_turns, width=11))
# Compare the speed of RcppArmadillo with R code
library(microbenchmark)
summary(microbenchmark(
  parallel_rcpp=NPE::rolling_median(re_turns),
  rcpp=roll::roll_median(re_turns),
  times=10))[, c(1, 4, 5)] # end microbenchmark summary

## End(Not run)
```

---

<code>theilSenEstimator</code>	<i>Calculate the non parametric Theil-Sen estimator of dependency-covariance for two vectors using RcppArmadillo</i>
--------------------------------	----------------------------------------------------------------------------------------------------------------------

---

**Description**

Calculate the non parametric Theil-Sen estimator of dependency-covariance for two *vectors* using *RcppArmadillo*

**Usage**

```
theilSenEstimator(x, y)
```

**Arguments**

<code>vector_x</code>	A <i>vector</i> independent (explanatory) data.
<code>vector_y</code>	A <i>vector</i> dependent data.



**Details**

The function `theilSenEstimator()` calculates the Theil-Sen estimator of the *vector*, using `RcppArmadillo`. The function `theilSenEstimator()` is significantly faster than function `WRS::tsreg()` in R.

**Value**

A column *vector* containing two values i.e intercept and slope

**Examples**

```
## Not run:
# Create a vector of random returns
vector_x <- rnorm(10)
vector_y <- rnorm(10)
# Compare theilSenEstimator() with tsreg()
# Compare the speed of RcppParallel with R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=NPE::theilSenEstimator(vector_x, vector_y),
  rcode=WRS(vector_x, vector_y),
  times=10))[, c(1, 4, 5)] # end microbenchmark summary

## End(Not run)
```

---

wilcoxonMannWhitneyTest

*Performs two sample Wilcoxon-Mann-Whitney rank sum test also known as Mann-Whitney U Test on vector or a single-column time series using RcppArmadillo and boost.*

---

**Description**

Performs two sample Wilcoxon-Mann-Whitney rank sum test also known as Mann-Whitney U Test on *vector* or a single-column *time series* using `RcppArmadillo` and `boost`.

**Usage**

```
wilcoxonMannWhitneyTest(x, y, mu = 0, alternative = "two.sided",
  exact = FALSE, correct = TRUE)
```

**Arguments**

- |             |                                                                                                                                                                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| x           | A <i>vector</i> or a single-column <i>time series</i> .                                                                                                                                                                                                             |
| y           | A <i>vector</i> or a single-column <i>time series</i> .                                                                                                                                                                                                             |
| mu          | A <i>double</i> specifying an optional parameter used to form null hypothesis. Default value is <i>zero</i> .                                                                                                                                                       |
| alternative | a <i>character</i> string specifying the alternative hypothesis. It must be one of : <ul style="list-style-type: none"> <li>• "two.sided" two tailed test.</li> <li>• "greater" greater(right) tailed test.</li> <li>• "less" smaller(left) tailed test.</li> </ul> |

(The default is *two.sided* test.)

exact	A boolean indicating whether an exact p-value should be computed.
correct	A boolean indicating whether to apply continuity correction in normal approximation for the p-value.

### Details

The function `wilcoxonMannWhitneyTest()` carries out the wilcoxon-Mann-Whitney signed rank test on  $x$  &  $y$  and returns the *p-value* of the test. By default (if `exact` is not specified), an exact p-value is computed if sample contains less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

### Value

A *double* indicating p-value of the test.

### Examples

```
## Not run:
# Create a vector of random data
x <- round(runif(10), 2)
y <- round(runif(10), 2)
# Carry out WMW signed rank test on the elements in two ways
all.equal(wilcox.test(x, y)$p.value, drop(NPE::wilcoxonMannWhitneyTest(x, y)))
# Compare the speed of Rcpp and R code
library(microbenchmark)
summary(microbenchmark(
  rcpp=wilcoxonMannWhitneyTest(x, y),
  rcode=wilcox.test(x, y),
  times=10))[, c(1, 4, 5)] # end microbenchmark summary

## End(Not run)
```

---

`wilcoxonSignedRankTest`

*Performs one sample Wilcoxon ranked sum test on vector or a single-column time series using RcppArmadillo and boost.*

---

### Description

Performs one sample Wilcoxon ranked sum test on *vector* or a single-column *time series* using RcppArmadillo and boost.

### Usage

```
wilcoxonSignedRankTest(x, mu = 0, alternative = "two.sided",
  exact = FALSE, correct = TRUE)
```

**Arguments**

<code>x</code>	A <i>vector</i> or a single-column <i>time series</i> .
<code>mu</code>	A <i>double</i> specifying an optional parameter used to form null hypothesis. Default value is <i>zero</i> .
<code>alternative</code>	a <i>character</i> string specifying the alternative hypothesis. It must be one of : <ul style="list-style-type: none"> <li>• "two.sided" two tailed test.</li> <li>• "greater" greater(right) tailed test.</li> <li>• "less" smaller(left) tailed test.</li> </ul> (The default is <i>two.sided</i> test.)
<code>exact</code>	A boolean indicating whether an exact p-value should be computed.
<code>correct</code>	A boolean indicating whether to apply continuity correction in normal approximation for the p-value.

**Details**

The function `wilcoxonSignedRankTest()` carries out the wilcoxon signed rank test on *vec\_tor* and returns the *p-value* of the test. By default (if `exact` is not specified), an exact p-value is computed if sample contains less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

**Value**

A *double* indicating p-value of the test.

**Examples**

```
## Not run:
# Create a vector of random data
da_ta <- round(runif(7), 2)
# Carry out wilcoxon signed rank test on the elements in two ways
all.equal(wilcox.test(da_ta)$p.value, drop(HighFreq::WilcoxonSignedRankTest(da_ta)))
# Create a time series of random data
da_ta <- xts::xts(runif(7), seq.Date(Sys.Date(), by=1, length.out=7))
# Calculate the ranks of the elements in two ways
all.equal(wilcox.test(coredata(da_ta))$p.value, drop(NPE::wilcoxonSignedRankTest(da_ta)))
# Compare the speed of Rcpp and R code
da_ta <- runif(10)
library(microbenchmark)
summary(microbenchmark(
  rcpp=wilcoxonSignedRankTest(da_ta),
  rcode=wilcox.test(da_ta),
  times=10))[, c(1, 4, 5)] # end microbenchmark summary

## End(Not run)
```

# Index

`calc_pca`, [2](#)  
`calc_ranksWithTies`, [2](#)  
  
`hle`, [3](#)  
  
`kruskalWalliceTest`, [4](#)  
  
`med_ian`, [6](#)  
`medianAbsoluteDeviation`, [5](#)  
  
`rolling_mad`, [7](#)  
`rolling_median`, [7](#)  
  
`theilSenEstimator`, [8](#)  
  
`wilcoxonMannWhitneyTest`, [9](#)  
`wilcoxonSignedRankTest`, [10](#)