Package ‘fromo’

January 30, 2019

Type Package
Maintainer Steven E. Pav <shabbychef@gmail.com>
Version 0.2.1
Date 2019-01-29
License LGPL-3
Title Fast Robust Moments

BugReports https://github.com/shabbychef/fromo/issues

Description Fast, numerically robust computation of weighted moments via 'Rcpp'.
Supports computation on vectors and matrices, and Monoidal append of moments.
Moments and cumulants over running fixed length windows can be computed,
as well as over time-based windows.
Moment computations are via a generalization of Welford's method, as described

Imports Rcpp (>= 0.12.3), methods
LinkingTo Rcpp
Suggests knitr, testthat, moments, PDQutils, e1071, microbenchmark

RoxygenNote 6.0.1

URL https://github.com/shabbychef/fromo
VignetteBuilder knitr
Collate 'fromo.r' 'RcppExports.R' 'zzz_centsums.R'

NeedsCompilation yes
Author Steven E. Pav [aut, cre] (<https://orcid.org/0000-0002-4197-6195>)
Repository CRAN

Date/Publication 2019-01-30 07:10:03 UTC

R topics documented:

fromo-package .............................................. 2
accessor ..................................................... 3
Description

Fast, numerically robust moments computations, along with computation of cumulants, running means, etc.

Robust Moments

Welford described a method for ‘robust’ one-pass computation of the standard deviation. By ‘robust’, we mean robust to round-off caused by a large shift in the mean. This method was generalized by Terriberry, and Bennett et. al. to the case of higher-order moments. This package provides those algorithms for computing moments.

Generally we should find that the stock implementations of sd, skewness and so on are already robust and likely using these algorithms under the hood. This package was written for a few reasons:

1. As an exercise to learn Rcpp.
2. Often I found I needed the first $k$ moments. For example, when computing the Z-score, the standard deviation and mean must be computed separately, which is inefficient. Similarly Merten’s correction for the standard error of the Sharpe ratio uses the first four moments. These are all computed as a side effect of computation of the kurtosis, but discarded by the standard methods.
Legal Mumbo Jumbo

fromo is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

Note

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

This package was developed as an exercise in learning Rcpp.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

References


Description

Access slot data from a centsums object.

Usage

```r
sums(x)
```

## S4 method for signature 'centsums'
```
sums(x)
```

```r
moments(x, type = c("central", "raw", "standardized"))
```

## S4 method for signature 'centsums'
```
moments(x, type = c("central", "raw", "standardized"))
```
Arguments

- `x`: a centcosums object.
- `type`: the type of moment to compute.

Note

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Description

Coerce to a centcosums object.

Usage

```r
as.centcosums(x, order=2, na.omit=TRUE)
```

## Default S3 method:
```r
as.centcosums(x, order = 2, na.omit = TRUE)
```

Arguments

- `x`: a matrix.
- `order`: the order, defaulting to 2.
- `na.omit`: whether to remove rows with NA.

Details

Computes the raw cosums on data, and stuffs the results into a centcosums object.

Value

A centcosums object.

Note

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.
as.centsums

Author(s)
Steven E. Pav <shabbychef@gmail.com>

Examples

```r
set.seed(123)
x <- matrix(rnorm(100*3), ncol=3)
cs <- as.centcosums(x, order=2)
```

Description

Coerce to a centsums object.

Usage

```r
as.centsums(x, order=3, na.rm=TRUE)
```

## Default S3 method:
```
as.centsums(x, order = 3, na.rm = TRUE)
```

Arguments

- `x` a numeric, array, or matrix.
- `order` the order, defaulting to `length(sums)+1`.
- `na.rm` whether to remove NA.

Details

Computes the raw sums on data, and stuffs the results into a centsums object.

Value

A centsums object.

Note

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Author(s)

Steven E. Pav <shabbychef@gmail.com>
Examples

```r
set.seed(123)
x <- rnorm(1000)
cs <- as.centsums(x, order=5)
```

Description

Concatenate centcosums objects.

Usage

```r
\method{c}{centcosums}(\ldots)
```

Arguments

... centcosums objects

See Also

join_cent_cosums

Description

Concatenate centcosums objects.

Usage

```r
\method{c}{centsums}(\ldots)
```

Arguments

... centsums objects

See Also

join_cent_sums
**cent2raw**

Convert between different types of moments, raw, central, standardized.

**Description**

Given raw or central or standardized moments, convert to another type.

**Usage**

```r
cent2raw(input)
```

**Arguments**

- `input`: a vector of the count, then the mean, then the 2 through k raw or central moments.

**Note**

The moment computations provided by fromo are numerically robust, but will often *not* provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

**Author(s)**

Steven E. Pav <shabbychef@gmail.com>

---

**centcosums-accessor**

Accessor methods.

**Description**

Access slot data from a centcosums object.

**Usage**

```r
cosums(x)
```

```r
# S4 method for signature 'centcosums'
cosums(x)
```

```r
comoments(x, type = c("central", "raw"))
```

```r
# S4 method for signature 'centcosums'
comoments(x, type = c("central", "raw"))
```
centcosums-class

Arguments

x a centcosums object.
type the type of moment to compute.

Note

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the ‘standard’ implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

centcosums-class

centcosums Class.

Description

An S4 class to store (centered) cosums of data, and to support operations on the same.

Usage

## S4 method for signature 'centcosums'
initialize(.Object, cosums, order = NA_real_)

centcosums(cosums, order = NULL)

Arguments

.Object a centcosums object, or proto-object.
cosums the output of cent_cosums, say.
order the order, defaulting to 2.

Details

A centcosums object contains a multidimensional array (now only 2-dimensional), as output by cent_cosums.

Value

An object of class centcosums.

Slots

cosums a multidimensional array of the cosums.
order the maximum order, ignored for now.
Note

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

References


See Also

cent_cosums

Examples

obj <- new("centcosums", cosums=cent_cosums(matrix(rnorm(100*3), ncol=3), max_order=2), order=2)

centsums-class  centsums Class.

Description

An S4 class to store (centered) sums of data, and to support operations on the same.

Usage

## S4 method for signature 'centsums'
initialize(.Object, sums, order = NA_real_)

centsums(sums, order = NULL)
Arguments

.Object  a centsums object, or proto-object.
sums    a numeric vector.
order   the order, defaulting to length(sums)+1.

Details

A centsums object contains a vector value of the data count, the mean, and the \( k \)th centered sum, for \( k \) up to some maximum order.

Value

An object of class centsums.

Slots

sums  a numeric vector of the sums.
order the maximum order.

Note

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the ‘standard’ implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

References


Examples

obj <- new("centsums", sums=c(1000,1.234,0.235), order=2)
**cent_cosums**

Multivariate centered sums: join and unjoined.

**Description**
Compute, join, or unjoin multivariate centered (co-) sums.

**Usage**

```r
cent_cosums(v, max_order = 2L, na_omit = FALSE)
cent_comoments(v, max_order = 2L, used_df = 0L, na_omit = FALSE)
join_cent_cosums(ret1, ret2)
unjoin_cent_cosums(ret3, ret2)
```

**Arguments**

- `v`: an `m` by `n` matrix, each row an independent observation of some `n` variate variable.
- `max_order`: the maximum order of cosum to compute. For now this can only be 2; in the future higher order cosums should be possible.
- `na_omit`: a boolean; if TRUE, then only rows of `v` with complete observations will be used.
- `used_df`: the number of degrees of freedom consumed, used in the denominator of the centered moments computation. These are subtracted from the number of observations.
- `ret1`, `ret2`, `ret3`: a multidimensional array as output by `cent_cosums`.

**Value**

A multidimensional array of dimension `max_order`, each side of length `1 + n`. For the case currently implemented where `max_order` must be 2, the output is a symmetric matrix, where the element in the `1,1` position is the count of complete rows of `v`, the `2: (n+1), 1` column is the mean, and the `2: (n+1), 2: (n+1)` is the co-sums matrix, which is the covariance up to scaling by the count. `cent_comoments` performs this normalization for you.

**Note**

The moment computations provided by fromo are numerically robust, but will often *not* provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.
cent_sums

Author(s)
Steven E. Pav <shabbychef@gmail.com>

References


See Also

cent_sums

Examples

```r
set.seed(1234)
x1 <- matrix(rnorm(1e3*5,mean=1),ncol=5)
x2 <- matrix(rnorm(1e3*5,mean=1),ncol=5)
max_ord <- 2L
rs1 <- cent_cosums(x1,max_ord)
rs2 <- cent_cosums(x2,max_ord)
rs3 <- cent_cosums(rbind(x1,x2),max_ord)
rs3alt <- join_cent_cosums(rs1,rs2)
stopifnot(max(abs(rs3 - rs3alt)) < 1e-7)
rs1alt <- unjoin_cent_cosums(rs3,rs2)
rs2alt <- unjoin_cent_cosums(rs3,rs1)
stopifnot(max(abs(rs1 - rs1alt)) < 1e-7)
stopifnot(max(abs(rs2 - rs2alt)) < 1e-7)
```

cent_sums

Centered sums; join and unjoined.

Description

Compute, join, or unjoin centered sums.
Usage

cent_sums(v, max_order = 5L, na_rm = FALSE, wts = NULL,
     check_wts = FALSE, normalize_wts = TRUE)

join_cent_sums(ret1, ret2)

unjoin_cent_sums(ret3, ret2)

Arguments

v a vector
max_order the maximum order of the centered moment to be computed.
na_rm whether to remove NA, false by default.
wts an optional vector of weights. Weights are ‘replication’ weights, meaning a
   value of 2 is shorthand for having two observations with the corresponding v
   value. If NULL, corresponds to equal unit weights, the default. Note that weights
   are typically only meaningfully defined up to a multiplicative constant, meaning
   the units of weights are immaterial, with the exception that methods which check
   for minimum df will, in the weighted case, check against the sum of weights. For
   this reason, weights less than 1 could cause NA to be returned unexpectedly due
   to the minimum condition. When weights are NA, the same rules for checking
   v are applied. That is, the observation will not contribute to the moment if the
   weight is NA when na_rm is true. When there is no checking, an NA value will
   cause the output to be NA.
check_wts a boolean for whether the code shall check for negative weights, and throw an
   error when they are found. Default false for speed.
normalize_wts a boolean for whether the weights should be renormalized to have a mean value
   of 1. This mean is computed over elements which contribute to the moments, so
   if na_rm is set, that means non-NA elements of wts that correspond to non-NA
   elements of the data vector.
ret1 an ord + 1 vector as output by cent_sums consisting of the count, the mean,
   then the k through ordth centered sum of some observations.
ret2 an ord + 1 vector as output by cent_sums consisting of the count, the mean,
   then the k through ordth centered sum of some observations.
ret3 an ord + 1 vector as output by cent_sums consisting of the count, the mean,
   then the k through ordth centered sum of some observations.

Value

a vector the same size as the input consisting of the adjusted version of the input. When there are
not sufficient (non-nan) elements for the computation, NaN are returned.

Note

The moment computations provided by fromo are numerically robust, but will often not provide
the same results as the 'standard' implementations, due to differences in roundoff. We make every
attempt to balance speed and robustness. User assumes all risk from using the fromo package.
Author(s)
Steven E. Pav <shabbychef@gmail.com>

References


Examples

```r
set.seed(1234)
x1 <- rnorm(1e3,mean=1)
x2 <- rnorm(1e3,mean=1)
max_ord <- 6L
rs1 <- cent_sums(x1,max_ord)
rs2 <- cent_sums(x2,max_ord)
rs3 <- cent_sums(c(x1,x2),max_ord)
rs3alt <- join_cent_sums(rs1,rs2)
stopifnot(max(abs(rs3 - rs3alt)) < 1e-7)
rs1alt <- unjoin_cent_sums(rs3,rs2)
rs2alt <- unjoin_cent_sums(rs3,rs1)
stopifnot(max(abs(rs1 - rs1alt)) < 1e-7)
stopifnot(max(abs(rs2 - rs2alt)) < 1e-7)
```

---

fromo-NEWS

News for package 'fromo':

Description

News for package 'fromo'

Version 0.2.1 (2019-01-29)

- fix memory leak for case where the mean only need be computed via a Welford object.
running_apx_quantiles

Version 0.2.0 (2019-01-12)
- add std_cumulants
- add running_sum, running_mean.
- Kahan compensated summation for these.
- Welford object under the hood.
- add weighted moments computation.
- add time-based running window computations.
- some speedups for obviously fast cases: no checking of NA, etc.
- move github figures to location CRAN understands.

Version 0.1.3 (2016-04-04)
- submit to CRAN

Initial Version 0.1.0 (2016-03-25)
- start work

running_apx_quantiles  Compute approximate quantiles over a sliding window

Description
Computes cumulants up to some given order, then employs the Cornish-Fisher approximation to compute approximate quantiles using a Gaussian basis.

Usage
running_apx_quantiles(v, p, window = NULL, wts = NULL, max_order = 5L, na_rm = FALSE, min_df = 0L, used_df = 0, restart_period = 100L, check_wts = FALSE, normalize_wts = TRUE)

running_apx_median(v, window = NULL, wts = NULL, max_order = 5L, na_rm = FALSE, min_df = 0L, used_df = 0, restart_period = 100L, check_wts = FALSE, normalize_wts = TRUE)

Arguments
- v a vector
- p the probability points at which to compute the quantiles. Should be in the range (0,1).
- window the window size. if given as finite integer or double, passed through. If NULL, NA_integer_, NA_real_ or Inf are given, equivalent to an infinite window size. If negative, an error will be thrown.
**running_apx_quantiles**

- **wts**: an optional vector of weights. Weights are ‘replication’ weights, meaning a value of 2 is shorthand for having two observations with the corresponding value. If NULL, corresponds to equal unit weights, the default. Note that weights are typically only meaningfully defined up to a multiplicative constant, meaning the units of weights are immaterial, with the exception that methods which check for minimum df will, in the weighted case, check against the sum of weights. For this reason, weights less than 1 could cause NA to be returned unexpectedly due to the minimum condition. When weights are NA, the same rules for checking are applied. That is, the observation will not contribute to the moment if the weight is NA when na.rm is true. When there is no checking, an NA value will cause the output to be NA.

- **max_order**: the maximum order of the centered moment to be computed.

- **na.rm**: whether to remove NA, false by default.

- **min_df**: the minimum df to return a value, otherwise NaN is returned. This can be used to prevent moments from being computed on too few observations. Defaults to zero, meaning no restriction.

- **used_df**: the number of degrees of freedom consumed, used in the denominator of the centered moments computation. These are subtracted from the number of observations.

- **restart_period**: the recompute period. Because subtraction of elements can cause loss of precision, the computation of moments is restarted periodically based on this parameter. Larger values mean fewer restarts and faster, though less accurate results.

- **check_wts**: a boolean for whether the code shall check for negative weights, and throw an error when they are found. Default false for speed.

- **normalize_wts**: a boolean for whether the weights should be renormalized to have a mean value of 1. This mean is computed over elements which contribute to the moments, so if na.rm is set, that means non-NA elements of wts that correspond to non-NA elements of the data vector.

**Details**

Computes the cumulants, then approximates quantiles using AS269 of Lee & Lin.

**Value**

A matrix, with one row for each element of x, and one column for each element of q.

**Note**

The current implementation is not as space-efficient as it could be, as it first computes the cumulants for each row, then performs the Cornish-Fisher approximation on a row-by-row basis. In the future, this computation may be moved earlier into the pipeline to be more space efficient. File an issue if the memory footprint is an issue for you.

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the ‘standard’ implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.
Note that when weights are given, they are treated as replication weights. This can have subtle effects on computations which require minimum degrees of freedom, since the sum of weights will be compared to that minimum, not the number of data points. Weight values (much) less than 1 can cause computations to return NA somewhat unexpectedly due to this condition, while values greater than one might cause the computation to spuriously return a value with little precision.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

References


See Also

t_running_apx_quantiles, running_cumulants, PDQutils::qapx_cf, PDQutils::AS269.

Examples

```r
x <- rnorm(1e5)
xq <- running_apx_quantiles(x,c(0.1,0.25,0.5,0.75,0.9))
xm <- running_apx_median(x)
```

running_centered

Compare data to moments computed over a sliding window.

Description

Computes moments over a sliding window, then adjusts the data accordingly, centering, or scaling, or z-scoring, and so on.
Usage

running_centered(v, window = NULL, wts = NULL, na_rm = FALSE,
                   min_df = 0L, used_df = 1L, lookahead = 0L, restart_period = 100L,
                   check_wts = FALSE, normalize_wts = FALSE)

running_scaled(v, window = NULL, wts = NULL, na_rm = FALSE, min_df = 0L,
                used_df = 1L, lookahead = 0L, restart_period = 100L, check_wts = FALSE,
                normalize_wts = TRUE)

running_zscored(v, window = NULL, wts = NULL, na_rm = FALSE,
                min_df = 0L, used_df = 1L, lookahead = 0L, restart_period = 100L,
                check_wts = FALSE, normalize_wts = TRUE)

running_sharpe(v, window = NULL, wts = NULL, na_rm = FALSE,
                compute_se = FALSE, min_df = 0L, used_df = 1L, restart_period = 100L,
                check_wts = FALSE, normalize_wts = TRUE)

running_tstat(v, window = NULL, wts = NULL, na_rm = FALSE, min_df = 0L,
              used_df = 1L, restart_period = 100L, check_wts = FALSE,
              normalize_wts = TRUE)

Arguments

v a vector

window the window size. If given as finite integer or double, passed through. If NULL, 
         NA_integer_, NA_real_ or Inf are given, equivalent to an infinite window size. 
         If negative, an error will be thrown.

wts an optional vector of weights. Weights are ‘replication’ weights, meaning a 
     value of 2 is shorthand for having two observations with the corresponding v 
     value. If NULL, corresponds to equal unit weights, the default. Note that weights 
     are typically only meaningfully defined up to a multiplicative constant, meaning 
     the units of weights are immaterial, with the exception that methods which check 
     for minimum df will, in the weighted case, check against the sum of weights. For 
     this reason, weights less than 1 could cause NA to be returned unexpectedly due 
     to the minimum condition. When weights are NA, the same rules for checking 
     v are applied. That is, the observation will not contribute to the moment if the 
     weight is NA when na_rm is true. When there is no checking, an NA value will 
     cause the output to be NA.

na_rm whether to remove NA, false by default.

min_df the minimum df to return a value, otherwise NaN is returned. This can be used 
         to prevent e.g. Z-scores from being computed on only 3 observations. Defaults 
         to zero, meaning no restriction, which can result in infinite Z-scores during the 
         burn-in period.

used_df the number of degrees of freedom consumed, used in the denominator of the 
         centered moments computation. These are subtracted from the number of ob-
For some of the operations, the value is compared to mean and standard deviation possibly using 'future' or 'past' information by means of a non-zero lookahead. Positive values mean data are taken from the future.

The recompute period. Because subtraction of elements can cause loss of precision, the computation of moments is restarted periodically based on this parameter. Larger values mean fewer restarts and faster, though less accurate results.

A boolean for whether the code shall check for negative weights, and throw an error when they are found. Default false for speed.

A boolean for whether the weights should be renormalized to have a mean value of 1. This mean is computed over elements which contribute to the moments, so if na.rm is set, that means non-NA elements of wts that correspond to non-NA elements of the data vector.

For running_sharpe, return an extra column of the standard error, as computed by Mertens' correction.

Details

Given the length n vector x, for a given index i, define \( x^{(i)} \) as the vector of \( x_{i-window+1}, x_{i-window+2}, \ldots, x_i \), where we do not run over the 'edge' of the vector. In code, this is essentially \( x(\max(1, i-window+1):i) \).

Then define \( \mu_i, \sigma_i \) and \( n_i \) as, respectively, the sample mean, standard deviation and number of non-NA elements in \( x^{(i)} \).

We compute output vector m the same size as x. For the 'centered' version of x, we have \( m_i = x_i - \mu_i \). For the 'scaled' version of x, we have \( m_i = x_i / \sigma_i \). For the 'z-scored' version of x, we have \( m_i = (x_i - \mu_i) / \sigma_i \). For the 't-scored' version of x, we have \( m_i = x_i \sqrt{n_i / \mu_i / \sigma_i} \).

We also allow a 'lookahead' for some of these operations. If positive, the moments are computed using data from larger indices; if negative, from smaller indices. Letting \( j = i + \text{lookahead} \): For the 'centered' version of x, we have \( m_i = x_i - \mu_j \). For the 'scaled' version of x, we have \( m_i = x_i / \sigma_j \). For the 'z-scored' version of x, we have \( m_i = (x_i - \mu_j) / \sigma_j \).

Value

A vector the same size as the input consisting of the adjusted version of the input. When there are not sufficient (non-nan) elements for the computation, NaN are returned.

Note

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Note that when weights are given, they are treated as replication weights. This can have subtle effects on computations which require minimum degrees of freedom, since the sum of weights will be compared to that minimum, not the number of data points. Weight values (much) less than 1 can cause computations to return NA somewhat unexpectedly due to this condition, while values greater than one might cause the computation to spuriously return a value with little precision.

Author(s)

Steven E. Pav <shabbychef@gmail.com>
References


See Also

t_running_centered, scale

Examples

```r
if (require(moments)) { 
  set.seed(123)
  x <- rnorm(5e1)
  window <- 10L
  rm1 <- t(sapply(seq_len(length(x)),function(iii) {
    xrang <- x[max(1,iii-window+1):iii]
    c(sd(xrang),mean(xrang),length(xrang)),
    simplify=TRUE))
  rcent <- running_centered(x,window=window)
  rscal <- running_scaled(x,window=window)
  rzesco <- running_zscored(x,window=window)
  rshrp <- running_sharpe(x,window=window)
  rtsco <- running_tstat(x,window=window)
  rsrse <- running_sharpe(x,window=window,compute_se=TRUE)
  stopifnot(max(abs(rcent - (x - rm1[,2])),na.rm=TRUE) < 1e-12)
  stopifnot(max(abs(rscal - (x / rm1[,1])),na.rm=TRUE) < 1e-12)
  stopifnot(max(abs(rzesco - ((x - rm1[,2]) / rm1[,1])),na.rm=TRUE) < 1e-12)
  stopifnot(max(abs(rshrp - (rm1[,2] / rm1[,1])),na.rm=TRUE) < 1e-12)
  stopifnot(max(abs(rtsco - ((sqrt(rm1[,3]) * rm1[,2]) / rm1[,1])),na.rm=TRUE) < 1e-12)
  stopifnot(max(abs(rsrse[,1] - rshrp),na.rm=TRUE) < 1e-12)

  rm2 <- t(sapply(seq_len(length(x)),function(iii) {
    xrang <- x[max(1,iii-window+1):iii]
    c(kurtosis(xrang)-3.0,skewness(xrang)),
    simplify=TRUE))
  mertens_se <- sqrt((1 + ((2 + rm2[,1])/4) * rshrp^2 - rm2[,2]*rshrp) / rm1[,3])
  stopifnot(max(abs(rsrse[,2] - mertens_se),na.rm=TRUE) < 1e-12)
}
```
running_sd3
Compute first K moments over a sliding window

Description
Compute the (standardized) 2nd through kth moments, the mean, and the number of elements over an infinite or finite sliding window, returning a matrix.

Usage
running_sd3(v, window = NULL, wts = NULL, na_rm = FALSE, min_df = 0L,
used_df = 1, restart_period = 100L, check_wts = FALSE,
normalize_wts = TRUE)
running_skew4(v, window = NULL, wts = NULL, na_rm = FALSE, min_df = 0L,
used_df = 1, restart_period = 100L, check_wts = FALSE,
normalize_wts = TRUE)
running_kurt5(v, window = NULL, wts = NULL, na_rm = FALSE, min_df = 0L,
used_df = 1, restart_period = 100L, check_wts = FALSE,
normalize_wts = TRUE)
running_sd(v, window = NULL, wts = NULL, na_rm = FALSE, min_df = 0L,
used_df = 1, restart_period = 100L, check_wts = FALSE,
normalize_wts = TRUE)
running_skew(v, window = NULL, wts = NULL, na_rm = FALSE, min_df = 0L,
used_df = 1, restart_period = 100L, check_wts = FALSE,
normalize_wts = TRUE)
running_kurt(v, window = NULL, wts = NULL, na_rm = FALSE, min_df = 0L,
used_df = 1, restart_period = 100L, check_wts = FALSE,
normalize_wts = TRUE)
running_cent_moments(v, window = NULL, wts = NULL, max_order = 5L,
na_rm = FALSE, max_order_only = FALSE, min_df = 0L, used_df = 0,
restart_period = 100L, check_wts = FALSE, normalize_wts = TRUE)
running_std_moments(v, window = NULL, wts = NULL, max_order = 5L,
na_rm = FALSE, min_df = 0L, used_df = 0, restart_period = 100L,
check_wts = FALSE, normalize_wts = TRUE)
running_cumulants(v, window = NULL, wts = NULL, max_order = 5L,
na_rm = FALSE, min_df = 0L, used_df = 0, restart_period = 100L,
check_wts = FALSE, normalize_wts = TRUE)
Arguments

v  

a vector

window  

the window size. If given as a finite integer or double, passed through. If NULL, NA_integer_, NA_real_ or Inf are given, equivalent to an infinite window size. If negative, an error will be thrown.

wts  

an optional vector of weights. Weights are ‘replication’ weights, meaning a value of 2 is shorthand for having two observations with the corresponding v value. If NULL, corresponds to equal unit weights, the default. Note that weights are typically only meaningfully defined up to a multiplicative constant, meaning the units of weights are immaterial, with the exception that methods which check for minimum df will, in the weighted case, check against the sum of weights. For this reason, weights less than 1 could cause NA to be returned unexpectedly due to the minimum condition. When weights are NA, the same rules for checking v are applied. That is, the observation will not contribute to the moment if the weight is NA when na.rm is true. When there is no checking, an NA value will cause the output to be NA.

na_rm  

whether to remove NA, false by default.

min_df  

the minimum df to return a value, otherwise NaN is returned. This can be used to prevent moments from being computed on too few observations. Defaults to zero, meaning no restriction.

used_df  

the number of degrees of freedom consumed, used in the denominator of the centered moments computation. These are subtracted from the number of observations.

restart_period  

the recompute period. Because subtraction of elements can cause loss of precision, the computation of moments is restarted periodically based on this parameter. Larger values mean fewer restarts and faster, though less accurate results.

check_wts  

a boolean for whether the code shall check for negative weights, and throw an error when they are found. Default false for speed.

normalize_wts  

a boolean for whether the weights should be renormalized to have a mean value of 1. This mean is computed over elements which contribute to the moments, so if na_rm is set, that means non-NA elements of wts that correspond to non-NA elements of the data vector.

max_order  

the maximum order of the centered moment to be computed.

max_order_only  

for running_cent_moments, if this flag is set, only compute the maximum order centered moment, and return in a vector.

Details

Computes the number of elements, the mean, and the 2nd through kth centered (and typically standardized) moments, for k = 2, 3, 4. These are computed via the numerically robust one-pass method of Bennett et al.

Given the length n vector x, we output matrix M where Mt,j is the order − j + 1 moment (i.e. excess kurtosis, skewness, standard deviation, mean or number of elements) of x_{i−window+1}, x_{i−window+2}, ..., x_{i}.

Barring NA or NaN, this is over a window of size window. During the ‘burn-in’ phase, we take fewer elements.
Value

Typically a matrix, where the first columns are the kth, k-1th through 2nd standardized, centered moments, then a column of the mean, then a column of the number of (non-nan) elements in the input, with the following exceptions:

**running_cent_moments** Computes arbitrary order centered moments. When max_order_only is set, only a column of the maximum order centered moment is returned.

**running_std_moments** Computes arbitrary order standardized moments, then the standard deviation, the mean, and the count. There is not yet an option for max_order_only, but probably should be.

**running_cumulants** Computes arbitrary order cumulants, and returns the kth, k-1th, through the second (which is the variance) cumulant, then the mean, and the count.

Note

the kurtosis is *excess kurtosis*, with a 3 subtracted, and should be nearly zero for Gaussian input.

The moment computations provided by fromo are numerically robust, but will often *not* provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Note that when weights are given, they are treated as replication weights. This can have subtle effects on computations which require minimum degrees of freedom, since the sum of weights will be compared to that minimum, not the number of data points. Weight values (much) less than 1 can cause computations to return NA somewhat unexpectedly due to this condition, while values greater than one might cause the computation to spuriously return a value with little precision.

As this code may add and remove observations, numerical imprecision may result in negative estimates of squared quantities, like the second or fourth moments. We do not currently correct for this issue, although it may be somewhat mitigated by setting a smaller restart_period. In the future we will add a check for this case. Post an issue if you experience this bug.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

References


running_sum

Examples

```r
x <- rnorm(1e5)
xs3 <- running_sd3(x, 10)
xs4 <- running_skew4(x, 10)

if (require(moments)) {
  set.seed(123)
  x <- rnorm(5e1)
  window <- 10L
  kt5 <- running_kurt5(x, window = window)
  rm1 <- t(sapply(seq_len(length(x)), function(iii) {
    xrang <- x[max(1, iii - window + 1):iii]
    c(moments::kurtosis(xrang) - 3, 0, moments::skewness(xrang),
    sd(xrang), mean(xrang), length(xrang)) },
    simplify = TRUE))
  stopifnot(max(abs(kt5 - rm1), na.rm = TRUE) < 1e-12)
}

xc6 <- running_cent_moments(x, window = 100L, max_order = 6L)
```

running_sum

Compute sums or means over a sliding window.

Description

Compute the mean or sum over an infinite or finite sliding window, returning a vector the same size as the input.

Usage

```r
running_sum(v, window = NULL, wts = NULL, na.rm = FALSE,
             restart_period = 10000L, check_wts = FALSE)

running_mean(v, window = NULL, wts = NULL, na.rm = FALSE, min_df = 0L,
              restart_period = 10000L, check_wts = FALSE)
```

Arguments

- `v` a vector.
- `window` the window size. If given as integer or double, passed through. If `NULL`, `NA_integer_`, `NA_real_` or `Inf` are given, equivalent to an infinite window size. If negative, an error will be thrown.
- `wts` an optional vector of weights. Weights are ‘replication’ weights, meaning a value of 2 is shorthand for having two observations with the corresponding `v` value. If `NULL`, corresponds to equal unit weights, the default. Note that weights are typically only meaningfully defined up to a multiplicative constant, meaning
the units of weights are immaterial, with the exception that methods which check for minimum df will, in the weighted case, check against the sum of weights. For this reason, weights less than 1 could cause NA to be returned unexpectedly due to the minimum condition. When weights are NA, the same rules for checking v are applied. That is, the observation will not contribute to the moment if the weight is NA when na_rm is true. When there is no checking, an NA value will cause the output to be NA.

na_rm whether to remove NA, false by default.

restart_period the recompute period. Because subtraction of elements can cause loss of precision, the computation of moments is restarted periodically based on this parameter. Larger values mean fewer restarts and faster, though potentially less accurate results. Unlike in the computation of even order moments, loss of precision is unlikely to be disastrous, so the default value is rather large.

check_wts a boolean for whether the code shall check for negative weights, and throw an error when they are found. Default false for speed.

min_df the minimum df to return a value, otherwise NaN is returned, only for the means computation. This can be used to prevent moments from being computed on too few observations. Defaults to zero, meaning no restriction.

Details

Computes the mean or sum of the elements, using a Kahan’s Compensated Summation Algorithm, a numerically robust one-pass method.

Given the length n vector x, we output matrix M where M_{i,1} is the sum or mean of \( x_{i-window+1}, x_{i-window+2}, \ldots, x_{i} \). Barring NA or NaN, this is over a window of size window. During the ‘burn-in’ phase, we take fewer elements. If fewer than min_df for running_mean, returns NA.

Value

A vector the same size as the input.

Note

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the ‘standard’ implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

References


Examples

```r
x <- rnorm(1e5)
x <- running_sum(x,10)
xm <- running_mean(x,100)
```

---

sd3

**Compute first K moments**

**Description**

Compute the (standardized) 2nd through kth moments, the mean, and the number of elements.

**Usage**

```r
sd3(v, na.rm = FALSE, wts = NULL, sg_df = 1, check_wts = FALSE,
    normalize_wts = TRUE)
```

```r
skew4(v, na.rm = FALSE, wts = NULL, sg_df = 1, check_wts = FALSE,
    normalize_wts = TRUE)
```

```r
kurt5(v, na.rm = FALSE, wts = NULL, sg_df = 1, check_wts = FALSE,
    normalize_wts = TRUE)
```

```r
cent_moments(v, max_order = 5L, used_df = 0L, na.rm = FALSE, wts = NULL,
    check_wts = FALSE, normalize_wts = TRUE)
```

```r
std_moments(v, max_order = 5L, used_df = 0L, na.rm = FALSE, wts = NULL,
    check_wts = FALSE, normalize_wts = TRUE)
```

```r
cent_cumulants(v, max_order = 5L, used_df = 0L, na.rm = FALSE,
    wts = NULL, check_wts = FALSE, normalize_wts = TRUE)
```

```r
std_cumulants(v, max_order = 5L, used_df = 0L, na.rm = FALSE,
    wts = NULL, check_wts = FALSE, normalize_wts = TRUE)
```
Arguments

\begin{itemize}
  \item \texttt{v} \quad \text{a vector}
  \item \texttt{na.rm} \quad \text{whether to remove NA, false by default.}
  \item \texttt{wts} \quad \text{an optional vector of weights. Weights are 'replication' weights, meaning a value of 2 is shorthand for having two observations with the corresponding \( v \) value. If NULL, corresponds to equal unit weights, the default. Note that weights are typically only meaningfully defined up to a multiplicative constant, meaning the units of weights are immaterial, with the exception that methods which check for minimum df will, in the weighted case, check against the sum of weights. For this reason, weights less than 1 could cause NA to be returned unexpectedly due to the minimum condition. When weights are NA, the same rules for checking \( v \) are applied. That is, the observation will not contribute to the moment if the weight is NA when \texttt{na.rm} is true. When there is no checking, an NA value will cause the output to be NA.}
  \item \texttt{sg_df} \quad \text{the number of degrees of freedom consumed in the computation of the variance or standard deviation. This defaults to 1 to match the 'Bessel correction'.}
  \item \texttt{check_wts} \quad \text{a boolean for whether the code shall check for negative weights, and throw an error when they are found. Default false for speed.}
  \item \texttt{normalize_wts} \quad \text{a boolean for whether the weights should be renormalized to have a mean value of 1. This mean is computed over elements which contribute to the moments, so if \texttt{na.rm} is set, that means non-NA elements of \texttt{wts} that correspond to non-NA elements of the data vector.}
  \item \texttt{max_order} \quad \text{the maximum order of the centered moment to be computed.}
  \item \texttt{used_df} \quad \text{the number of degrees of freedom consumed, used in the denominator of the centered moments computation. These are subtracted from the number of observations.}
\end{itemize}

Details

Computes the number of elements, the mean, and the 2nd through \( k \)th centered standardized moment, for \( k = 2, 3, 4 \). These are computed via the numerically robust one-pass method of Bennett \textit{et al}. In general they will \textit{not} match exactly with the 'standard' implementations, due to differences in roundoff.

These methods are reasonably fast, on par with the 'standard' implementations. However, they will usually be faster than calling the various standard implementations more than once.

Moments are computed as follows, given some values \( x_i \) and optional weights \( w_i \), defaulting to 1, the weighted mean is computed as
\[
\mu = \frac{\sum_i x_i w_i}{\sum w_i}.
\]

The weighted \( k \)th central sum is computed as
\[
\mu = \sum_i (x_i - \mu)^k w_i.
\]

Let \( n = \sum_i w_i \) be the sum of weights (or number of observations in the unweighted case). Then the weighted \( k \)th central moment is computed as that weighted sum divided by the adjusted sum
weights:

\[
\mu_k = \frac{\sum_i (x_i - \mu)^k w_i}{n - \nu},
\]

where \( \nu \) is the ‘used df’, provided by the user to adjust the denominator. (Typical values are 0 or 1.) The weighted \( k \)th standardized moment is the central moment divided by the second central moment to the \( k/2 \) power:

\[
\tilde{\mu}_k = \frac{\mu_k}{\mu_k^{k/2}}.
\]

The (centered) \( r \)th cumulant, for \( r \geq 2 \) is then computed using the formula of Willink, namely

\[
\kappa_r = \mu_r - \sum_{j=0}^{r-2} \binom{r-1}{j} \mu_j \kappa_r - j.
\]

The standardized \( r \)th cumulant is the \( r \)th centered cumulant divided by \( \mu_r^{r/2} \).

**Value**

a vector, filled out as follows:

- **sd3** A vector of the (sample) standard deviation, mean, and number of elements (or the total weight when \( wts \) are given).
- **skew4** A vector of the (sample) skewness, standard deviation, mean, and number of elements (or the total weight when \( wts \) are given).
- **kurt5** A vector of the (sample) excess kurtosis, skewness, standard deviation, mean, and number of elements (or the total weight when \( wts \) are given).
- **cent_moments** A vector of the (sample) \( k \)th centered moment, then \( k-1 \)th centered moment, ..., then the variance, the mean, and number of elements (total weight when \( wts \) are given).
- **std_moments** A vector of the (sample) \( k \)th standardized (and centered) moment, then \( k-1 \)th, ..., then standard deviation, mean, and number of elements (total weight).
- **cent_cumulants** A vector of the (sample) \( k \)th (centered, but this is redundant) cumulant, then the \( k-1 \)th, ..., then the variance (which is the second cumulant), then the mean, then the number of elements (total weight).
- **std_cumulants** A vector of the (sample) \( k \)th standardized (and centered, but this is redundant) cumulant, then the \( k-1 \)th, ..., down to the third, then the variance, the mean, then the number of elements (total weight).

**Note**

The first centered (and standardized) moment is often defined to be identically 0. Instead cent_moments and std_moments returns the mean. Similarly, the second standardized moments defined to be identically 1; std_moments instead returns the standard deviation. The reason is that a user can always decide to ignore the results and fill in a 0 or 1 as they need, but could not efficiently compute the mean and standard deviation from scratch if we discard it. The antepenultimate element of the output of std_cumulants is not a one, even though that ‘should’ be the standardized second cumulant.

The antepenultimate element of the output of cent_moments, cent_cumulants and std_cumulants is the variance, not the standard deviation. All other code return the standard deviation in that place.
The kurtosis is excess kurtosis, with a 3 subtracted, and should be nearly zero for Gaussian input.
The term 'centered cumulants' is redundant. The intent was to avoid possible collision with existing code named 'cumulants'.
The moment computations provided by fromo are numerically robust, but will often not provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.
Note that when weights are given, they are treated as replication weights. This can have subtle effects on computations which require minimum degrees of freedom, since the sum of weights will be compared to that minimum, not the number of data points. Weight values (much) less than 1 can cause computations to return NA somewhat unexpectedly due to this condition, while values greater than one might cause the computation to spuriously return a value with little precision.

Author(s)
Steven E. Pav <shabbychef@gmail.com>

References

Examples
```r
x <- rnorm(1e5)
sd3(x)[1] - sd(x)
skew4(x)[4] - length(x)
skew4(x)[3] - mean(x)
skew4(x)[2] - sd(x)
if (require(moments)) {
  skew4(x)[1] - skewness(x)
}
```

```
# check 'robustness'; only the mean should change:
kurt5(x + 1e12) - kurt5(x)
# check speed
if (require(microbenchmark) & require(moments)) {
  dumbk <- function(x) { c(kurtosis(x) - 3.0,skewness(x),sd(x),mean(x),length(x)) }
```
set.seed(1234)
x <- rnorm(1e6)
print(kurt(x) - dumbk(x))
microbenchmark(dumbk(x),kurt(x),times=10L)
}
y <- std_moments(x,6)
cml <- cent_cumulants(x,6)
std <- std_cumulants(x,6)

# check that skew matches moments::skewness
if (require(moments)) {
  set.seed(1234)
x <- rnorm(1000)
resu <- fromo::skew(x)

msku <- moments::skewness(x)
stopifnot(abs(msku - resu[1]) < 1e-14)
}

# check skew vs e1071 skewness, which has a different denominator
if (require(e1071)) {
  set.seed(1234)
x <- rnorm(1000)
resu <- fromo::skew(x)

esku <- e1071::skewness(x,type=3)
nobs <- resu[4]
stopifnot(abs(esku - resu[1] * ((nobs-1)/nobs)^((3/2)) < 1e-14)

# similarly:
resu <- fromo::std_moments(x,max_order=3,used_df=0)
stopifnot(abs(esku - resu[1] * ((nobs-1)/nobs)^((3/2)) < 1e-14)
}

---

**show**  
*Show a centsums object.*

**Description**

Displays the centsums object.

**Usage**

```
show(object)
```

## S4 method for signature 'centsums'

```
show(object)
```
t_running_apx_quantiles

Argument

object a centsums object.

Note

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

Examples

set.seed(123)
x <- rnorm(1000)
obj <- as.centsums(x, order=5)
obj

t_running_apx_quantiles

Compute approximate quantiles over a sliding time window

Description

Computes cumulants up to some given order, then employs the Cornish-Fisher approximation to compute approximate quantiles using a Gaussian basis.

Usage

```r
T_running_apx_quantiles(v, p, time = NULL, time_deltas = NULL,
window = NULL, wts = NULL, lb_time = NULL, max_order = 5L,
na_rm = FALSE, min_df = 0L, used_df = 0, restart_period = 100L,
variable_win = FALSE, wts_as_delta = TRUE, check_wts = FALSE,
normalize_wts = TRUE)
```

```r
T_running_apx_median(v, time = NULL, time_deltas = NULL, window = NULL,
wts = NULL, lb_time = NULL, max_order = 5L, na_rm = FALSE,
min_df = 0L, used_df = 0, restart_period = 100L, variable_win = FALSE,
wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)
```
Arguments

v

a vector of data.

p

the probability points at which to compute the quantiles. Should be in the range (0,1).

time

an optional vector of the timestamps of v. If given, must be the same length as v. If not given, we try to infer it by summing the time_deltas.

time_deltas

an optional vector of the deltas of timestamps. If given, must be the same length as v. If not given, and wts are given and wts_as_delta is true, we take the wts as the time deltas. The deltas must be positive. We sum them to arrive at the times.

window

the window size, in time units. If given as finite integer or double, passed through. If NULL, NA_integer_, NA_real_ or Inf are given, and variable_win is true, then we infer the window from the lookback times: the first window is infinite, but the remaining is the deltas between lookback times. If variable_win is false, then these undefined values are equivalent to an infinite window. If negative, an error will be thrown.

wts

an optional vector of weights. Weights are ‘replication’ weights, meaning a value of 2 is shorthand for having two observations with the corresponding v value. If NULL, corresponds to equal unit weights, the default. Note that weights are typically only meaningfully defined up to a multiplicative constant, meaning the units of weights are immaterial, with the exception that methods which check for minimum df will, in the weighted case, check against the sum of weights. For this reason, weights less than 1 could cause NA to be returned unexpectedly due to the minimum condition. When weights are NA, the same rules for checking v are applied. That is, the observation will not contribute to the moment if the weight is NA when na_rm is true. When there is no checking, an NA value will cause the output to be NA.

lb_time

a vector of the times from which lookback will be performed. The output should be the same size as this vector. If not given, defaults to time.

max_order

the maximum order of the centered moment to be computed.

na_rm

whether to remove NA, false by default.

min_df

the minimum df to return a value, otherwise NaN is returned. This can be used to prevent moments from being computed on too few observations. Defaults to zero, meaning no restriction.

used_df

the number of degrees of freedom consumed, used in the denominator of the centered moments computation. These are subtracted from the number of observations.

restart_period

the recompute period. because subtraction of elements can cause loss of precision, the computation of moments is restarted periodically based on this parameter. Larger values mean fewer restarts and faster, though less accurate results.

variable_win

if true, and the window is not a concrete number, the computation window becomes the time between lookback times.

wts_as_delta

if true and the time and time_deltas are not given, but wts are given, we take wts as the time_deltas.
check_wts  

A boolean for whether the code shall check for negative weights, and throw an error when they are found. Default false for speed.

normalize_wts  

A boolean for whether the weights should be renormalized to have a mean value of 1. This mean is computed over elements which contribute to the moments, so if na_rm is set, that means non-NA elements of wts that correspond to non-NA elements of the data vector.

Details  

Computes the cumulants, then approximates quantiles using AS269 of Lee & Lin.

Value  

A matrix, with one row for each element of x, and one column for each element of q.

Time Windowing  

This function supports time (or other counter) based running computation. Here the input are the data x_i, and optional weights vectors, w_i, defaulting to 1, and a vector of time indices, t_i of the same length as x. The times must be non-decreasing:

\[
t_1 \leq t_2 \leq \ldots
\]

It is assumed that t_0 = -\infty. The window, W is now a time-based window. An optional set of lookback times are also given, b_j, which may have different length than the x and w. The output will correspond to the lookback times, and should be the same length. The jth output is computed over indices i such that

\[
b_j - W < t_i \leq b_j.
\]

For comparison functions (like Z-score, rescaling, centering), which compare values of x_i to local moments, the lookbacks may not be given, but a lookahead L is admitted. In this case, the jth output is computed over indices i such that

\[
t_j - W + L < t_i \leq t_j + L.
\]

If the times are not given, ‘deltas’ may be given instead. If δ_i are the deltas, then we compute the times as

\[
t_i = \sum_{1 \leq j \leq i} \delta_j.
\]

The deltas must be the same length as x. If times and deltas are not given, but weights are given and the ‘weights as deltas’ flag is set true, then the weights are used as the deltas.

Some times it makes sense to have the computational window be the space between lookback times. That is, the jth output is to be computed over indices i such that

\[
b_{j-1} - W < t_i \leq b_j.
\]

This can be achieved by setting the ‘variable window’ flag true and setting the window to null. This will not make much sense if the lookback times are equal to the times, since each moment computation is over a set of a single index, and most moments are underdefined.
Note

The current implementation is not as space-efficient as it could be, as it first computes the cumulants for each row, then performs the Cornish-Fisher approximation on a row-by-row basis. In the future, this computation may be moved earlier into the pipeline to be more space efficient. File an issue if the memory footprint is an issue for you.

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Note that when weights are given, they are treated as replication weights. This can have subtle effects on computations which require minimum degrees of freedom, since the sum of weights will be compared to that minimum, not the number of data points. Weight values (much) less than 1 can cause computations to return NA somewhat unexpectedly due to this condition, while values greater than one might cause the computation to spuriously return a value with little precision.

Author(s)
Steven E. Pav <shabbychef@gmail.com>

References


See Also

running_apx_quantiles, t_running_cumulants, PDQutils::qapx Cf, PDQutils::AS269.

Examples

```r
x <- rnorm(1e5)
xq <- t_running_apx_quantiles(x, c(0.1, 0.25, 0.5, 0.75, 0.9),
   time=seq_along(x), window=200, lb_time=c(100, 200, 400))

xq <- t_running_apx_median(x, time=seq_along(x), window=200, lb_time=c(100, 200, 400))
xq <- t_running_apx_median(x, time=seq_along(x), window=200, lb_time=c(100, 200, 400))

# weighted median?
wts <- runif(length(x), min=1, max=5)
xq <- t_running_apx_median(x, wts=wts, wts_as_delta=TRUE, window=1000, lb_time=seq(1000, 10000, by=1000))
```
t_running_centered

# these should give the same answer:
xr <- running_apx_median(x, window=200);  
xt <- t_running_apx_median(x, time=seq_along(x), window=199.99)

---

**t_running_centered**  
*Compare data to moments computed over a time sliding window.*

**Description**

Computes moments over a sliding window, then adjusts the data accordingly, centering, or scaling, or z-scoring, and so on.

**Usage**

```r

t_running_centered(v, time = NULL, time_deltas = NULL, window = NULL,  
wts = NULL, na_rm = FALSE, min_df = 0L, used_df = 1, lookahead = 0,  
restart_period = 100L, variable_win = FALSE, wts_as_delta = TRUE,  
check_wts = FALSE, normalize_wts = TRUE)

t_running_scaled(v, time = NULL, time_deltas = NULL, window = NULL,  
wts = NULL, na_rm = FALSE, min_df = 0L, used_df = 1, lookahead = 0,  
restart_period = 100L, variable_win = FALSE, wts_as_delta = TRUE,  
check_wts = FALSE, normalize_wts = TRUE)

t_running_zscored(v, time = NULL, time_deltas = NULL, window = NULL,  
wts = NULL, na_rm = FALSE, min_df = 0L, used_df = 1, lookahead = 0,  
restart_period = 100L, variable_win = FALSE, wts_as_delta = TRUE,  
check_wts = FALSE, normalize_wts = TRUE)

t_running_sharpe(v, time = NULL, time_deltas = NULL, window = NULL,  
wts = NULL, lb_time = NULL, na_rm = FALSE, compute_se = FALSE,  
min_df = 0L, used_df = 1, restart_period = 100L, variable_win = FALSE,  
wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)

t_running_tstat(v, time = NULL, time_deltas = NULL, window = NULL,  
wts = NULL, lb_time = NULL, na_rm = FALSE, compute_se = FALSE,  
min_df = 0L, used_df = 1, restart_period = 100L, variable_win = FALSE,  
wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)
```

**Arguments**

- `v`  
a vector of data.

- `time`  
an optional vector of the timestamps of `v`. If given, must be the same length as `v`. If not given, we try to infer it by summing the `time_deltas`. 
time_deltas

an optional vector of the deltas of timestamps. If given, must be the same length as v. If not given, and wts are given and wts_as_delta is true, we take the wts as the time deltas. The deltas must be positive. We sum them to arrive at the times.

window

the window size, in time units. If given as finite integer or double, passed through. If NULL, NA_integer_, NA_real_ or Inf are given, and variable_win is true, then we infer the window from the lookback times: the first window is infinite, but the remaining is the deltas between lookback times. If variable_win is false, then these undefined values are equivalent to an infinite window. If negative, an error will be thrown.

wts

an optional vector of weights. Weights are 'replication' weights, meaning a value of 2 is shorthand for having two observations with the corresponding v value. If NULL, corresponds to equal unit weights, the default. Note that weights are typically only meaningfully defined up to a multiplicative constant, meaning the units of weights are immaterial, with the exception that methods which check for minimum df will, in the weighted case, check against the sum of weights. For this reason, weights less than 1 could cause NA to be returned unexpectedly due to the minimum condition. When weights are NA, the same rules for checking v are applied. That is, the observation will not contribute to the moment if the weight is NA when na_rm is true. When there is no checking, an NA value will cause the output to be NA.

na_rm

whether to remove NA, false by default.

min_df

the minimum df to return a value, otherwise NaN is returned. This can be used to prevent e.g. Z-scores from being computed on only 3 observations. Defaults to zero, meaning no restriction, which can result in infinite Z-scores during the burn-in period.

used_df

the number of degrees of freedom consumed, used in the denominator of the centered moments computation. These are subtracted from the number of observations.

lookahead

for some of the operations, the value is compared to mean and standard deviation possibly using 'future' or 'past' information by means of a non-zero lookahead. Positive values mean data are taken from the future. This is in time units, and so should be a real.

restart_period

the recompute period. Because subtraction of elements can cause loss of precision, the computation of moments is restarted periodically based on this parameter. Larger values mean fewer restarts and faster, though less accurate results.

variable_win

if true, and the window is not a concrete number, the computation window becomes the time between lookback times.

wts_as_delta

if true and the time and time_deltas are not given, but wts are given, we take wts as the time_deltas.

check_wts

a boolean for whether the code shall check for negative weights, and throw an error when they are found. Default false for speed.

normalize_wts

a boolean for whether the weights should be renormalized to have a mean value of 1. This mean is computed over elements which contribute to the moments, so if na_rm is set, that means non-NA elements of wts that correspond to non-NA elements of the data vector.
`t_running_centered`

**lb_time**

A vector of the times from which lookback will be performed. The output should be the same size as this vector. If not given, defaults to time.

**compute_se**

For `running_sharpe`, return an extra column of the standard error, as computed by Mertens’ correction.

**Details**

Given the length `n` vector `x`, for a given index `i`, define `x^{(i)}` as the elements of `x` defined by the sliding time window (see the section on time windowing). Then define `\mu_i`, `\sigma_i` and `n_i` as, respectively, the sample mean, standard deviation and number of non-NA elements in `x^{(i)}`.

We compute output vector `m` the same size as `x`. For the ‘centered’ version of `x`, we have `m_i = x_i - \mu_i`. For the ‘scaled’ version of `x`, we have `m_i = x_i / \sigma_i`. For the ‘z-scored’ version of `x`, we have `m_i = (x_i - \mu_i) / \sigma_i`. For the ‘t-scored’ version of `x`, we have `m_i = \sqrt{n_i} \mu_i / \sigma_i`.

We also allow a ‘lookahead’ for some of these operations. If positive, the moments are computed using data from larger indices; if negative, from smaller indices.

**Value**

A vector the same size as the input consisting of the adjusted version of the input. When there are not sufficient (non-nan) elements for the computation, `NaN` are returned.

**Time Windowing**

This function supports time (or other counter) based running computation. Here the input are the data `x_i`, and optional weights vectors, `w_i`, defaulting to 1, and a vector of time indices, `t_i` of the same length as `x`. The times must be non-decreasing:

\[ t_1 \leq t_2 \leq \ldots \]

It is assumed that `t_0 = -\infty`. The window, `W` is now a time-based window. An optional set of `lookback times` are also given, `b_j`, which may have different length than the `x` and `w`. The output will correspond to the lookback times, and should be the same length. The `j`th output is computed over indices `i` such that

\[ b_j - W < t_i \leq b_j. \]

For comparison functions (like Z-score, rescaling, centering), which compare values of `x_i` to local moments, the lookbacks may not be given, but a lookahead `L` is admitted. In this case, the `j`th output is computed over indices `i` such that

\[ t_j - W + L < t_i \leq t_j + L. \]

If the times are not given, ‘deltas’ may be given instead. If `\delta_i` are the deltas, then we compute the times as

\[ t_i = \sum_{1 \leq j \leq i} \delta_j. \]

The deltas must be the same length as `x`. If times and deltas are not given, but weights are given and the ‘weights as deltas’ flag is set true, then the weights are used as the deltas.
Some times it makes sense to have the computational window be the space between lookback times. That is, the $j$th output is to be computed over indices $i$ such that

$$b_{j-1} - W < t_i \leq b_j.$$  

This can be achieved by setting the ‘variable window’ flag true and setting the window to null. This will not make much sense if the lookback times are equal to the times, since each moment computation is over a set of a single index, and most moments are underdefined.

**Note**

The moment computations provided by fromo are numerically robust, but will often *not* provide the same results as the ‘standard’ implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Note that when weights are given, they are treated as replication weights. This can have subtle effects on computations which require minimum degrees of freedom, since the sum of weights will be compared to that minimum, not the number of data points. Weight values (much) less than 1 can cause computations to return NA somewhat unexpectedly due to this condition, while values greater than one might cause the computation to spuriously return a value with little precision.

**Author(s)**

Steven E. Pav <shabbychef@gmail.com>

**References**


**See Also**

`running_centered`, `scale`
t_running_sd3

**Compute first K moments over a sliding time-based window**

**Description**

Compute the (standardized) 2nd through kth moments, the mean, and the number of elements over an infinite or finite sliding time based window, returning a matrix.

**Usage**

```r
t_running_sd3(v, time = NULL, time_deltas = NULL, window = NULL, wts = NULL, lb_time = NULL, na_rm = FALSE, min_df = 0L, used_df = 1L, restart_period = 100L, variable_win = FALSE, wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)
```

```r
t_running_skew4(v, time = NULL, time_deltas = NULL, window = NULL, wts = NULL, lb_time = NULL, na_rm = FALSE, min_df = 0L, used_df = 1L, restart_period = 100L, variable_win = FALSE, wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)
```

```r
t_running_kurt5(v, time = NULL, time_deltas = NULL, window = NULL, wts = NULL, lb_time = NULL, na_rm = FALSE, min_df = 0L, used_df = 1L, restart_period = 100L, variable_win = FALSE, wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)
```

```r
t_running_sd(v, time = NULL, time_deltas = NULL, window = NULL, wts = NULL, lb_time = NULL, na_rm = FALSE, min_df = 0L, used_df = 1L, restart_period = 100L, variable_win = FALSE, wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)
```

```r
t_running_skew(v, time = NULL, time_deltas = NULL, window = NULL, wts = NULL, lb_time = NULL, na_rm = FALSE, min_df = 0L, used_df = 1L, restart_period = 100L, variable_win = FALSE, wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)
```

```r
t_running_kurt(v, time = NULL, time_deltas = NULL, window = NULL, wts = NULL, lb_time = NULL, na_rm = FALSE, min_df = 0L, used_df = 1L, restart_period = 100L, variable_win = FALSE, wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)
```

```r
t_running_cent_moments(v, time = NULL, time_deltas = NULL, window = NULL, max_order = NULL, na_rm = FALSE, max_order_only = NULL, min_df = 0L, used_df = 0L, restart_period = 100L, variable_win = FALSE, wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)
```

```r
t_running_std_moments(v, time = NULL, time_deltas = NULL, window = NULL,
```
wts = NULL, lb_time = NULL, max_order = 5L, na_rm = FALSE,
min_df = 0L, used_df = 0, restart_period = 100L, variable_win = FALSE,
wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)

t_running_cumulants(v, time = NULL, time_deltas = NULL, window = NULL,
wts = NULL, lb_time = NULL, max_order = 5L, na_rm = FALSE,
min_df = 0L, used_df = 0, restart_period = 100L, variable_win = FALSE,
wts_as_delta = TRUE, check_wts = FALSE, normalize_wts = TRUE)

Arguments

v  
  a vector of data.

时间为
  an optional vector of the timestamps of v. If given, must be the same length as
v. If not given, we try to infer it by summing the time_deltas.

time_deltas
  an optional vector of the deltas of timestamps. If given, must be the same length
as v. If not given, and wts are given and wts_as_delta is true, we take the wts
as the time deltas. The deltas must be positive. We sum them to arrive at the
times.

window
  the window size, in time units. if given as finite integer or double, passed
through. If NULL, NA_integer_, NA_real_ or Inf are given, and variable_win
is true, then we infer the window from the lookback times: the first window is in-
finite, but the remaining is the deltas between lookback times. If variable_win
is false, then these undefined values are equivalent to an infinite window. If
negative, an error will be thrown.

wts
  an optional vector of weights. Weights are 'replication' weights, meaning a
value of 2 is shorthand for having two observations with the corresponding v
value. If NULL, corresponds to equal unit weights, the default. Note that weights
are typically only meaningfully defined up to a multiplicative constant, meaning
the units of weights are immaterial, with the exception that methods which check
for minimum df will, in the weighted case, check against the sum of weights. For
this reason, weights less than 1 could cause NA to be returned unexpectedly due
to the minimum condition. When weights are NA, the same rules for checking
v are applied. That is, the observation will not contribute to the moment if the
weight is NA when na_rm is true. When there is no checking, an NA value will
cause the output to be NA.

lb_time
  a vector of the times from which lookback will be performed. The output should
be the same size as this vector. If not given, defaults to time.

na_rm
  whether to remove NA, false by default.

min_df
  the minimum df to return a value, otherwise NaN is returned. This can be used
to prevent moments from being computed on too few observations. Defaults to
zero, meaning no restriction.

used_df
  the number of degrees of freedom consumed, used in the denominator of the
centered moments computation. These are subtracted from the number of ob-
servations.

restart_period
  the recompute period. because subtraction of elements can cause loss of preci-
sion, the computation of moments is restarted periodically based on this param-
eter. Larger values mean fewer restarts and faster, though less accurate results.
variable_win
if true, and the window is not a concrete number, the computation window becomes the time between lookback times.

wts_as_delta
if true and the time and time_deltas are not given, but wts are given, we take wts as the time_deltas.

check_wts
a boolean for whether the code shall check for negative weights, and throw an error when they are found. Default false for speed.

normalize_wts
a boolean for whether the weights should be renormalized to have a mean value of 1. This mean is computed over elements which contribute to the moments, so if na.rm is set, that means non-NA elements of wts that correspond to non-NA elements of the data vector.

max_order
the maximum order of the centered moment to be computed.

max_order_only
for running_cent_moments, if this flag is set, only compute the maximum order centered moment, and return in a vector.

Details

Computes the number of elements, the mean, and the 2nd through kth centered (and typically standardized) moments, for \( k = 2, 3, 4 \). These are computed via the numerically robust one-pass method of Bennett et. al.

Given the length \( n \) vector \( x \), we output matrix \( M \) where \( M_{i,j} \) is the order \( j + 1 \) moment (i.e. excess kurtosis, skewness, standard deviation, mean or number of elements) of some elements \( x_i \) defined by the sliding time window. Barring NA or NaN, this is over a window of time width \( \text{window} \).

Value

Typically a matrix, where the first columns are the kth, k-1th through 2nd standardized, centered moments, then a column of the mean, then a column of the number of (non-nan) elements in the input, with the following exceptions:

- **t_running_cent_moments** Computes arbitrary order centered moments. When max_order_only is set, only a column of the maximum order centered moment is returned.

- **t_running_std_moments** Computes arbitrary order standardized moments, then the standard deviation, the mean, and the count. There is not yet an option for max_order_only, but probably should be.

- **t_running_cumulants** Computes arbitrary order cumulants, and returns the kth, k-1th, through the second (which is the variance) cumulant, then the mean, and the count.

Time Windowing

This function supports time (or other counter) based running computation. Here the input are the data \( x_i \), and optional weights vectors, \( w_i \), defaulting to 1, and a vector of time indices, \( t_i \) of the same length as \( x \). The times must be non-decreasing:

\[
t_1 \leq t_2 \leq \ldots
\]

It is assumed that \( t_0 = -\infty \). The window, \( W \) is now a time-based window. An optional set of lookback times are also given, \( b_j \), which may have different length than the \( x \) and \( w \). The output
will correspond to the lookback times, and should be the same length. The jth output is computed over indices i such that

\[ b_j - W < t_i \leq b_j. \]

For comparison functions (like Z-score, rescaling, centering), which compare values of \( x_i \) to local moments, the lookbacks may not be given, but a lookahead \( L \) is admitted. In this case, the jth output is computed over indices i such that

\[ t_j - W + L < t_i \leq t_j + L. \]

If the times are not given, ‘deltas’ may be given instead. If \( \delta_i \) are the deltas, then we compute the times as

\[ t_i = \sum_{1 \leq j \leq i} \delta_j. \]

The deltas must be the same length as \( x \). If times and deltas are not given, but weights are given and the ‘weights as deltas’ flag is set true, then the weights are used as the deltas.

Some times it makes sense to have the computational window be the space between lookback times. That is, the jth output is to be computed over indices i such that

\[ b_{j-1} - W < t_i \leq b_j. \]

This can be achieved by setting the ‘variable window’ flag true and setting the window to null. This will not make much sense if the lookback times are equal to the times, since each moment computation is over a set of a single index, and most moments are underdefined.

**Note**

the kurtosis is *excess kurtosis*, with a 3 subtracted, and should be nearly zero for Gaussian input.

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the ‘standard’ implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Note that when weights are given, they are treated as replication weights. This can have subtle effects on computations which require minimum degrees of freedom, since the sum of weights will be compared to that minimum, not the number of data points. Weight values (much) less than 1 can cause computations to return NA somewhat unexpectedly due to this condition, while values greater than one might cause the computation to spuriously return a value with little precision.

As this code may add and remove observations, numerical imprecision may result in negative estimates of squared quantities, like the second or fourth moments. We do not currently correct for this issue, although it may be somewhat mitigated by setting a smaller restart_period. In the future we will add a check for this case. Post an issue if you experience this bug.

**Author(s)**

Steven E. Pav <shabbychef@gmail.com>
t_running_sum

Compute sums or means over a sliding time window.

Description

Compute the mean or sum over an infinite or finite sliding time window, returning a vector the same size as the lookback times.

Usage

```r
t_running_sum(v, time = NULL, time_deltas = NULL, window = NULL, wts = NULL, lb_time = NULL, na_rm = FALSE, min_df = 0L, restart_period = 10000L, variable_win = FALSE, wts_as_delta = TRUE, check_wts = FALSE)
```

```r
t_running_mean(v, time = NULL, time_deltas = NULL, window = NULL, wts = NULL, lb_time = NULL, na_rm = FALSE, min_df = 0L, restart_period = 10000L, variable_win = FALSE, wts_as_delta = TRUE, check_wts = FALSE)
```
Arguments

**v** a vector.

**time** an optional vector of the timestamps of v. If given, must be the same length as v. If not given, we try to infer it by summing the time_deltas.

**time_deltas** an optional vector of the deltas of timestamps. If given, must be the same length as v. If not given, and wts are given and wts_as_delta is true, we take the wts as the time deltas. The deltas must be positive. We sum them to arrive at the times.

**window** the window size, in time units. if given as finite integer or double, passed through. If NULL, NA_integer_, NA_real_ or Inf are given, and variable_win is true, then we infer the window from the lookback times: the first window is infinite, but the remaining is the deltas between lookback times. If variable_win is false, then these undefined values are equivalent to an infinite window. If negative, an error will be thrown.

**wts** an optional vector of weights. Weights are ‘replication’ weights, meaning a value of 2 is shorthand for having two observations with the corresponding v value. If NULL, corresponds to equal unit weights, the default. Note that weights are typically only meaningfully defined up to a multiplicative constant, meaning the units of weights are immaterial, with the exception that methods which check for minimum df will, in the weighted case, check against the sum of weights. For this reason, weights less than 1 could cause NA to be returned unexpectedly due to the minimum condition. When weights are NA, the same rules for checking v are applied. That is, the observation will not contribute to the moment if the weight is NA when na_rm is true. When there is no checking, an NA value will cause the output to be NA.

**lb_time** a vector of the times from which lookback will be performed. The output should be the same size as this vector. If not given, defaults to time.

**na_rm** whether to remove NA, false by default.

**min_df** the minimum df to return a value, otherwise NaN is returned, only for the means computation. This can be used to prevent moments from being computed on too few observations. Defaults to zero, meaning no restriction.

**restart_period** the recompute period. because subtraction of elements can cause loss of precision, the computation of moments is restarted periodically based on this parameter. Larger values mean fewer restarts and faster, though potentially less accurate results. Unlike in the computation of even order moments, loss of precision is unlikely to be disastrous, so the default value is rather large.

**variable_win** if true, and the window is not a concrete number, the computation window becomes the time between lookback times.

**wts_as_delta** if true and the time and time_deltas are not given, but wts are given, we take wts as the time_deltas.

**check_wts** a boolean for whether the code shall check for negative weights, and throw an error when they are found. Default false for speed.
Details

Computes the mean or sum of the elements, using a Kahan’s Compensated Summation Algorithm, a numerically robust one-pass method.

Given the length n vector \( x \), we output matrix \( M \) where \( M_{i,1} \) is the sum or mean of some elements \( x_i \) defined by the sliding time window. Barring NA or NaN, this is over a window of time width \( \text{window} \).

Value

A vector the same size as the lookback times.

Time Windowing

This function supports time (or other counter) based running computation. Here the input are the data \( x_i \), and optional weights vectors, \( w_i \), defaulting to 1, and a vector of time indices, \( t_i \) of the same length as \( x \). The times must be non-decreasing:

\[
t_1 \leq t_2 \leq \ldots
\]

It is assumed that \( t_0 = -\infty \). The window, \( W \) is now a time-based window. An optional set of lookback times are also given, \( b_j \), which may have different length than the \( x \) and \( w \). The output will correspond to the lookback times, and should be the same length. The \( j \)th output is computed over indices \( i \) such that

\[
b_j - W < t_i \leq b_j.
\]

For comparison functions (like Z-score, rescaling, centering), which compare values of \( x_i \) to local moments, the lookbacks may not be given, but a lookahead \( L \) is admitted. In this case, the \( j \)th output is computed over indices \( i \) such that

\[
t_j - W + L < t_i \leq t_j + L.
\]

If the times are not given, ‘deltas’ may be given instead. If \( \delta_i \) are the deltas, then we compute the times as

\[
t_i = \sum_{1 \leq j \leq i} \delta_j.
\]

The deltas must be the same length as \( x \). If times and deltas are not given, but weights are given and the ‘weights as deltas’ flag is set true, then the weights are used as the deltas.

Some times it makes sense to have the computational window be the space between lookback times. That is, the \( j \)th output is to be computed over indices \( i \) such that

\[
b_{j-1} - W < t_i \leq b_j.
\]

This can be achieved by setting the ‘variable window’ flag true and setting the window to null. This will not make much sense if the lookback times are equal to the times, since each moment computation is over a set of a single index, and most moments are underdefined.
Note

The moment computations provided by fromo are numerically robust, but will often not provide the same results as the 'standard' implementations, due to differences in roundoff. We make every attempt to balance speed and robustness. User assumes all risk from using the fromo package.

Note that when weights are given, they are treated as replication weights. This can have subtle effects on computations which require minimum degrees of freedom, since the sum of weights will be compared to that minimum, not the number of data points. Weight values (much) less than 1 can cause computations to return NA somewhat unexpectedly due to this condition, while values greater than one might cause the computation to spuriously return a value with little precision.

Author(s)

Steven E. Pav <shabbychef@gmail.com>

References


Examples

```r
x <- rnorm(1e5)
x <- c_running_sum(x, time=seq_along(x), window=10)
xm <- c_running_mean(x, time=cumsum(runif(length(x))), window=7.3)
```

Description

Unconcatenate centcosums objects.
### Usage

```r
## S4 method for signature 'centcosums,centcosums'
x %-% y
```

### Arguments

- `x`: A `centcosums` object
- `y`: A `centcosums` object

### See Also

- `unjoin_cent_cosums`

---

### Description

Unconcatenate `centcosums` objects.

### Usage

```r
x %-% y
```

```r
## S4 method for signature 'centsums,centsums'
x %-% y
```

### Arguments

- `x`: A `centsums` object
- `y`: A `centsums` object

### See Also

- `unjoin_cent_sums`
Index

*Topic moments
  centcosums-class, 8
  centsums-class, 9

*Topic package
  fromo-package, 2
  kurt5 (sd3), 26
  running_apx_median (running_apx_quantiles), 15
  running_apx_quantiles, 15, 34
  running_cent_moments (running_sdS), 21
  running_centered, 17, 38
  running_cumulants, 17
  running_cumulants (running_sdS), 21
  running_kurt (running_sdS), 21
  running_kurt5 (running_sdS), 21
  running_mean, 15
  running_mean (running_sum), 24
  running_scaled (running_centered), 17
  running_sd (running_sd3), 21
  running_sd3, 21, 43
  running_sharpe (running_centered), 17
  running_skew (running_sdS), 21
  running_skew4 (running_sd3), 21
  running_std_moments (running_sdS), 21
  running_sum, 15, 24
  running_tstat (running_centered), 17
  running_zscored (running_centered), 17
  scale, 20, 38
  sd3, 26
  show, 30
  show, centsums-method (show), 30
  skew4 (sd3), 26
  std_cumulants (sdS), 26

accessor, 3
as.centcosums, 4
as.centsums, 5
c.centcosums, 6
c.centsums, 6
cent2raw, 7
cent_comoments (cent_cosums), 11
cent_cosums, 8, 11, 11
cent_cumulants (sdS), 26
cent_moments (sdS), 26
cent_sums, 12, 13
centcosums (centcosums-class), 8
centcosums-accessor, 7
centcosums-class, 8
centsums (centsums-class), 9
centsums-class, 9
comoments (centcosums-accessor), 7
comoments, centcosums-method (centcosums-accessor), 7
cosums (centcosums-accessor), 7
cosums, centcosums-method (centcosums-accessor), 7

fromo-NEWS, 14
fromo-package, 2
initialize, centcosums-class (centcosums-class), 8
initialize, centcosums-method (centcosums-class), 8
initialize, centsums-class (centsums-class), 9
initialize, centsums-method (centsums-class), 9
join_cent_cosums (cent_cosums), 11
join_cent_sums (cent_sums), 12
kurt5 (sd3), 26
moments (accessor), 3
moments, centsums-method (accessor), 3
running_apx_median (running_apx_quantiles), 15
running_apx_quantiles, 15, 34
running_cent_moments (running_sdS), 21
running_centered, 17, 38
running_cumulants, 17
running_cumulants (running_sdS), 21
running_kurt (running_sdS), 21
running_kurt5 (running_sdS), 21
running_mean, 15
running_mean (running_sum), 24
running_scaled (running_centered), 17
running_sd (running_sd3), 21
running_sd3, 21, 43
running_sharpe (running_centered), 17
running_skew (running_sdS), 21
running_skew4 (running_sd3), 21
running_std_moments (running_sdS), 21
running_sum, 15, 24
running_tstat (running_centered), 17
running_zscored (running_centered), 17
scale, 20, 38
sd3, 26
show, 30
show, centsums-method (show), 30
skew4 (sd3), 26
std_cumulants (sdS), 26
std_moments (sd3), 26
sums (accessor), 3
sums, centcosums-method
  (centcosums-accessor), 7
sums, centsums-method (accessor), 3

t_running_apx_median
  (t_running_apx_quantiles), 31
t_running_apx_quantiles, 17, 31
t_running_cent_moments (t_running_sd3), 39
t_running_centered, 20, 35
t_running_cumulants, 34
t_running_cumulants (t_running_sd3), 39
t_running_kurt (t_running_sd3), 39
t_running_kurt5 (t_running_sd3), 39
t_running_mean (t_running_sum), 43
t_running_scaled (t_running_centered), 35
t_running_sd (t_running_sd3), 39
t_running_sd3, 39
t_running_sharpe (t_running_centered), 35
t_running_skew (t_running_sd3), 39
t_running_skew4 (t_running_sd3), 39
t_running_std_moments (t_running_sd3), 39
t_running_sum, 43
t_running_tstat (t_running_centered), 35
t_running_zscored (t_running_centered), 35

unjoin_cent_cosums (cent_cosums), 11
unjoin_cent_sums (cent_sums), 12