# 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
by Bennett et. (2009) <doi:10.1109/CLUSTR.2009.5289161>.

**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` *Fast Robust Moments.*

---

## 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



### 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

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

---

## accessor — *Accessor methods.*

---

### Description

Access slot data from a `centsums` object.

### Usage

```
sums(x)

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

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

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

### Arguments

| | |
|---|---|
| `x` | a `centsums` 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>

---

## `as.centcosums` *Coerce to a centcosums object.*

---

### Description

Convert data to a `centcosums` object.

### Usage

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

## Default S3 method:
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.

### Author(s)

Steven E. Pav <shabbychef@gmail.com>

### Examples

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

---

## as.centsums — *Coerce to a centsums object.*

---

### Description

Convert data to a `centsums` object.

### Usage

```
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

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

---

## c.centcosums *concatenate centcosums objects.*

---

### Description

Concatenate centcosums objects.

### Usage

```
\method{c}{centcosums}(...)
```

### Arguments

| | |
|---|---|
| `...` | `centcosums` objects |

### See Also

join_cent_cosums

---

## c.centsums *concatenate centsums objects.*

---

### Description

Concatenate centsums objects.

### Usage

```
\method{c}{centsums}(...)
```

### 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

```
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

```
cosums(x)

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

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

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

## 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-diemnsional), 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

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

## 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

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

### 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

```
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` | a multdimensional array as output by `cent_cosums`. |
| `ret2` | a multdimensional array as output by `cent_cosums`. |
| `ret3` | a multdimensional array as output by `cent_cosums`. |

### Value

a multidimensional arry 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.

## Author(s)

Steven E. Pav <shabbychef@gmail.com>

## References

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

## See Also

cent_sums

## Examples

```
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

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

## Examples

```
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.

### 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. |

| | |
|---|---|
| `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`. |
| `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

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

## See Also

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

## Examples

```
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 = 1, 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 = 1, 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 = 1, 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 = 1, restart_period = 100L,
  check_wts = FALSE, normalize_wts = TRUE)

running_tstat(v, window = NULL, wts = NULL, na_rm = FALSE, min_df = 0L,
  used_df = 1, 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 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. |
| 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. |
| 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 vector of $x_{i-window+1}, x_{i-window+2}, ..., x_i$, 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 $m_i = \sqrt{n_i}\mu_i/\sigma_i$.

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 $j = i + lookahead$: 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


Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation


## See Also

t_running_centered, scale

## Examples

```
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)
    rzsco <- 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(rzsco - ((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 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 $M_{i,j}$ 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

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

## Examples

```
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

```
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 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. |
| `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 $x_{i-window+1}, x_{i-window+2}, ..., x_i$. 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

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

Kahan, W. "Further remarks on reducing truncation errors," Communications of the ACM, 8 (1), 1965. https://doi.org/10.1145/363707.363723

Wikipedia contributors "Kahan summation algorithm," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Kahan_summation_algorithm&oldid=777164752 (accessed May 31, 2017).

## Examples

```
x <- rnorm(1e5)
xs <- 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

```
sd3(v, na_rm = FALSE, wts = NULL, sg_df = 1, check_wts = FALSE,
  normalize_wts = TRUE)

skew4(v, na_rm = FALSE, wts = NULL, sg_df = 1, check_wts = FALSE,
  normalize_wts = TRUE)

kurt5(v, na_rm = FALSE, wts = NULL, sg_df = 1, check_wts = FALSE,
  normalize_wts = TRUE)

cent_moments(v, max_order = 5L, used_df = 0L, na_rm = FALSE, wts = NULL,
  check_wts = FALSE, normalize_wts = TRUE)

std_moments(v, max_order = 5L, used_df = 0L, na_rm = FALSE, wts = NULL,
  check_wts = FALSE, normalize_wts = TRUE)

cent_cumulants(v, max_order = 5L, used_df = 0L, na_rm = FALSE,
  wts = NULL, check_wts = FALSE, normalize_wts = TRUE)

std_cumulants(v, max_order = 5L, used_df = 0L, na_rm = FALSE,
  wts = NULL, check_wts = FALSE, normalize_wts = TRUE)
```

**Arguments**

- `v` — a vector
- `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`.
- `sg_df` — 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'.
- `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.
- `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.

**Details**

Computes the number of elements, the mean, and the 2nd through kth centered standardized moment, for $k = 2, 3, 4$. These are computed via the numerically robust one-pass method of Bennett *et. al.* In general they will *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 kth central sum is computed as

$$\mu = \sum_i \left(x_i - \mu\right)^k w_i.$$

Let $n = \sum_i w_i$ be the sum of weights (or number of observations in the unweighted case). Then the weighted kth 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 kth standardized moment is the central moment divided by the second central moment to the $k/2$ power:

$$\tilde{\mu}_k = \frac{\mu_k}{\mu_2^{k/2}}.$$

The (centered) rth 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 rth cumulant is the rth centered cumulant divided by $\mu_2^{r/2}$.

## Value

a vector, filled out as follows:

**sd3** A vector of the (sample) standard devation, mean, and number of elements (or the total weight when `wts` are given).

**skew4** A vector of the (sample) skewness, standard devation, mean, and number of elements (or the total weight when `wts` are given).

**kurt5** A vector of the (sample) excess kurtosis, skewness, standard devation, 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 devation, 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

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

Willink, R. "Relationships Between Central Moments and Cumulants, with Formulae for the Central Moments of Gamma Distributions." Communications in Statistics - Theory and Methods, 32 no 4 (2003): 701-704. https://doi.org/10.1081/STA-120018823

### Examples

```
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(kurt5(x) - dumbk(x))
    microbenchmark(dumbk(x),kurt5(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::skew4(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::skew4(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)
```

### Arguments

| | |
|---|---|
| `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

```
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)

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:

$$t_1 \le t_2 \le \ldots$$

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

$$b_j - W < t_i \le b_j.$$

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

$$t_j - W + L < t_i \le t_j + L.$$

If the times are not given, 'deltas' may be given instead. If $\delta_i$ are the deltas, then we compute the times as

$$t_i = \sum_{1 \le j \le i} \delta_j.$$

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

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

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

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

## See Also

`running_apx_quantiles`, `t_running_cumulants`, `PDQutils::qapx_cf`, `PDQutils::AS269`.

## Examples

```
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=cumsum(runif(length(x),min=0.5,max=1.5)),
      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))
```

```
# 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

```
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.

`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:

$$t_1 \le t_2 \le \ldots$$

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

$$b_j - W < t_i \le b_j.$$

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

$$t_j - W + L < t_i \le t_j + L.$$

If the times are not given, 'deltas' may be given instead. If $\delta_i$ are the deltas, then we compute the times as

$$t_i = \sum_{1 \le j \le i} \delta_j.$$

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

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

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

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

## 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

```
t_running_sd3(v, time = NULL, time_deltas = NULL, window = NULL,
  wts = NULL, lb_time = NULL, na_rm = 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_skew4(v, time = NULL, time_deltas = NULL, window = NULL,
  wts = NULL, lb_time = NULL, na_rm = 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_kurt5(v, time = NULL, time_deltas = NULL, window = NULL,
  wts = NULL, lb_time = NULL, na_rm = 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_sd(v, time = NULL, time_deltas = NULL, window = NULL,
  wts = NULL, lb_time = NULL, na_rm = 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_skew(v, time = NULL, time_deltas = NULL, window = NULL,
  wts = NULL, lb_time = NULL, na_rm = 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_kurt(v, time = NULL, time_deltas = NULL, window = NULL,
  wts = NULL, lb_time = NULL, na_rm = 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_cent_moments(v, time = NULL, time_deltas = NULL, window = NULL,
  wts = NULL, lb_time = NULL, max_order = 5L, na_rm = FALSE,
  max_order_only = 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_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.
- `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. 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.

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 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:

$$t_1 \le t_2 \le \ldots$$

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

$$b_j - W < t_i \le b_j.$$

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

$$t_j - W + L < t_i \le t_j + L.$$

If the times are not given, 'deltas' may be given instead. If $\delta_i$ are the deltas, then we compute the times as

$$t_i = \sum_{1 \le j \le i} \delta_j.$$

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

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

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>

## References


Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation


## See Also

running_sd3.

## Examples

```
x <- rnorm(1e5)
xs3 <- t_running_sd3(x,time=seq_along(x),window=10)
xs4 <- t_running_skew4(x,time=seq_along(x),window=10)
# but what if you only cared about some middle values?
xs4 <- t_running_skew4(x,time=seq_along(x),lb_time=(length(x) / 2) + 0:10,window=20)
```

---

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

```
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)

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 `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:

$$t_1 \le t_2 \le \ldots$$

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

$$b_j - W < t_i \le b_j.$$

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

$$t_j - W + L < t_i \le t_j + L.$$

If the times are not given, 'deltas' may be given instead. If $\delta_i$ are the deltas, then we compute the times as

$$t_i = \sum_{1 \le j \le i} \delta_j.$$

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

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

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

Terriberry, T. "Computing Higher-Order Moments Online." http://people.xiph.org/~tterribe/notes/homs.html

J. Bennett, et. al., "Numerically Stable, Single-Pass, Parallel Statistics Algorithms," Proceedings of IEEE International Conference on Cluster Computing, 2009. https://www.semanticscholar.org/paper/Numerically-stable-single-pass-parallel-statistics-Bennett-Grout/a83ed72a5ba86622d5eb6395

Cook, J. D. "Accurately computing running variance." http://www.johndcook.com/standard_deviation.html

Cook, J. D. "Comparing three methods of computing standard deviation." http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation

Kahan, W. "Further remarks on reducing truncation errors," Communications of the ACM, 8 (1), 1965. https://doi.org/10.1145/363707.363723

Wikipedia contributors "Kahan summation algorithm," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Kahan_summation_algorithm&oldid=777164752 (accessed May 31, 2017).

## Examples

```
x <- rnorm(1e5)
xs <- t_running_sum(x,time=seq_along(x),window=10)
xm <- t_running_mean(x,time=cumsum(runif(length(x))),window=7.3)
```

---

# %-%,centcosums,centcosums-method

*unconcatenate centcosums objects.*

---

## Description

Unconcatenate centcosums objects.

## Usage

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

## Arguments

| | |
|---|---|
| x | a `centcosums` objects |
| y | a `centcosums` objects |

## See Also

unjoin_cent_cosums

---

`%-%` *unconcatenate centsums objects.*

---

## Description

Unconcatenate centsums objects.

## Usage

```
x %-% y

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

## Arguments

| | |
|---|---|
| x | a `centsums` objects |
| y | a `centsums` objects |

## See Also

unjoin_cent_sums

# Index