

Package: rjd3filters (via r-universe)

October 10, 2026

Type Package

Title Trend-Cycle Extraction with Linear Filters with 'JDemetra+ 3.x'

Version 2.4.0

Description Build and apply symmetric and asymmetric moving averages (= linear filters) for trend-cycle extraction. In particular, it implements several modern approaches for real-time estimates from the viewpoint of revisions and time delay in detecting turning points. It includes the local polynomial approach of Proietti and Luati (2008), the Reproducing Kernel Hilbert Space (RKHS) of Dagum and Bianconcini (2008) and the Fidelity-Smoothness-Timeliness approach of Grun-Rehomme, Guggemos, and Ladiray (2018). It is based on 'Java' libraries developed in 'JDemetra+' (<<https://github.com/jdemetra>>), time series analysis software.

Depends R (>= 4.1.0)

Imports rJava (>= 1.0-6), methods, MASS, graphics, stats, utils, rjd3jars (>= 0.0.5), rjd3toolkit (>= 3.7.1)

SystemRequirements Java (>= 21)

License EUPL

LazyData TRUE

URL <https://github.com/rjdverse/rjd3filters>,
<https://rjdverse.github.io/rjd3filters/>

BugReports <https://github.com/rjdverse/rjd3filters/issues>

Suggests knitr, rmarkdown, spelling

Encoding UTF-8

Config/roxygen2/version 8.1.0

Language en-GB

NeedsCompilation no

Author Alain Quartier-la-Tente [aut, cre, cph] (ORCID:
<<https://orcid.org/0000-0001-7890-3857>>), Jean Palate [aut],
Tanguy Barthelemy [ctb], Anna Smyk [ctb]

Maintainer Alain Quartier-la-Tente <alain.quartier@yahoo.fr>

Config/pak/sysreqs default-jdk

Repository <https://cran.r-universe.dev>

Date/Publication 2026-10-10 13:53:14 UTC

RemoteUrl <https://github.com/cran/rjd3filters>

RemoteRef HEAD

RemoteSha d1f0dae69968721e8aae8d5c4e30794994a083c5

Contents

.jd2r_finitefilters	3
confint_filter	3
df_var	5
dfa_filter	6
diagnostic_matrix	7
diagnostics-fit	8
filter	10
filters_operations	12
finite_filters	15
fst	16
fst_filter	17
get_kernel	19
get_moving_average	20
get_properties_function	20
implicit_forecasts	21
impute_last_obs	22
localpolynomials	24
lp_filter	25
mmsre_filter	26
moving_average	28
mse	31
plot_filters	32
polynomial_matrix	35
print,finite_filters-method	36
retailsa	37
rkhs_filter	38
rkhs_kernel	39
rkhs_optimal_bw	40
rkhs_optimization_fun	41
simple_ma	43
underlying_forecasts	44
var_estimator	45

Index	47
--------------	-----------

<code>.jd2r_finitefilters</code>	<i>Java Utility Function</i>
----------------------------------	------------------------------

Description

These functions are used in all JDemetra+ 3.0 packages to easily interact between R and Java objects.

Usage

```
.jd2r_finitefilters(jf)
```

Arguments

<code>jf</code>	Java linear filter object.
-----------------	----------------------------

Value

`.jd2r_finitefilters()` returns a `finite_filters` object.

<code>confint_filter</code>	<i>Confidence intervals</i>
-----------------------------	-----------------------------

Description

Confidence intervals

Usage

```
confint_filter(  
  x,  
  coef,  
  coef_var = coef,  
  level = 0.95,  
  asymmetric_var = TRUE,  
  gaussian_distribution = FALSE,  
  exact_df = TRUE,  
  ...  
)
```

Arguments

x	input time series.
coef	moving-average (<code>moving_average()</code>) or finite filter (<code>finite_filters()</code>) used to filter the series.
coef_var	moving-average (<code>moving_average()</code>) or finite filter (<code>finite_filters()</code>) used to compute the variance (throw <code>var_estimator()</code>). By default equal to coef.
level	confidence level.
asymmetric_var	if <code>asymmetric_var = TRUE</code> then the variance is estimated for each asymmetric filters instead of using the variance associated the symmetric estimates.
gaussian_distribution	if TRUE use the normal distribution to compute the confidence interval, otherwise use the t-distribution.
exact_df	if TRUE compute the exact degrees of freedom for the t-distribution (when <code>gaussian_distribution = FALSE</code>), otherwise uses an approximation.
...	other arguments passed to the function <code>moving_average()</code> to convert coef to a "moving_average" object.

Details

Let $(\theta_i)_{-p \leq i \leq q}$ be a moving average of length $p + q + 1$ used to filter a time series $(y_i)_{1 \leq i \leq n}$. Let denote $\hat{\mu}_t$ the filtered series computed at time t as:

$$\hat{\mu}_t = \sum_{i=-p}^q \theta_i y_{t+i}.$$

If $\hat{\mu}_t$ is unbiased, a approximate confidence for the true mean is:

$$\left[\hat{\mu}_t - z_{1-\alpha/2} \hat{\sigma} \sqrt{\sum_{i=-p}^q \theta_i^2}; \hat{\mu}_t + z_{1-\alpha/2} \hat{\sigma} \sqrt{\sum_{i=-p}^q \theta_i^2} \right],$$

where $z_{1-\alpha/2}$ is the quantile $1 - \alpha/2$ of the standard normal distribution.

The estimate of the variance $\hat{\sigma}$ is obtained using `var_estimator()` with the parameter `coef_var`. The assumption that $\hat{\mu}_t$ is unbiased is rarely exactly true, so variance estimates and confidence intervals are usually computed at small bandwidths where bias is small.

When `coef` (or `coef_var`) is a finite filter, the last points of the confidence interval are computed using the corresponding asymmetric filters

Value

A matrix or mts object with the filtered series and the lower and upper bounds of the confidence interval.

References

Loader, Clive. 1999. Local regression and likelihood. New York: Springer-Verlag.

Examples

```
x <- retailsa$DrinkingPlaces
coef <- lp_filter(6)
confint <- confint_filter(x, coef)
graphics::plot(confint, plot.type = "single",
  col = c("red", "black", "black"),
  lty = c(1, 2, 2))
```

df_var

*Compute the degrees of freedom for the variance estimator***Description**

Compute the degrees of freedom for the variance estimator

Usage

```
df_var(n, coef, exact_df = FALSE)
```

Arguments

n	number of observations
coef	moving average (moving_average()) used to filter the series.
exact_df	if TRUE compute the exact degrees of freedom for the t-distribution (when <code>gaussian_distribution = FALSE</code>), otherwise uses an approximation.

Value

Numeric value with the statistic (the degree of freedom is not necessary an integer).

See Also

[var_estimator\(\)](#).

Examples

```
df_var(n = 100, coef = lp_filter(6)$filter)
```

dfa_filter

*Direct Filter Approach***Description**

Direct Filter Approach

Usage

```
dfa_filter(
  horizon = 6,
  degree = 0,
  density = c("uniform", "rw"),
  targetfilter = lp_filter(horizon = horizon)@sfilter,
  passband = 2 * pi/12,
  accuracy.weight = 1/3,
  smoothness.weight = 1/3,
  timeliness.weight = 1/3
)
```

Arguments

horizon	horizon (bandwidth) of the symmetric filter.
degree	degree of polynomial.
density	hypothesis on the spectral density: "uniform" (= white noise, the default) or "rw" (= random walk).
targetfilter	the weights of the symmetric target filters (by default the Henderson filter).
passband	passband threshold.
accuracy.weight, smoothness.weight, timeliness.weight	the weight used for the optimisation. The weight associated to the residual is derived so that the sum of the four weights are equal to 1.

Details

Moving average computed by a minimisation of a weighted mean of three criteria under polynomials constraints. The criteria come from the decomposition of the mean squared error between the trend-cycle

Let $\theta = (\theta_{-p}, \dots, \theta_f)'$ be a moving average where p and f are two integers defined by the parameter lags and leads. The three criteria are:

Value

A `finite_filters()` object.

Examples

```
dfa_filter(horizon = 6, degree = 0)
dfa_filter(horizon = 6, degree = 2)
```

diagnostic_matrix	<i>Compute quality criteria for asymmetric filters</i>
-------------------	--

Description

Function to compute a diagnostic matrix of quality criteria for asymmetric filters.

Usage

```
diagnostic_matrix(x, lags, passband = pi/6, sweights = NULL, ...)
```

Arguments

x	moving_average object or weights of the asymmetric filter (from -lags to m).
lags	Lags of the filter (should be positive), only used if x is not a moving_average.
passband	passband threshold.
sweights	Weights of the symmetric filter (from 0 to lags or -lags to lags). If missing, the criteria from the functions mse are not computed.
...	optional arguments to mse .

Details

For a moving average of coefficients $\theta = (\theta_i)_{-p \leq i \leq q}$ `diagnostic_matrix` returns a vector with the following ten criteria:

- **b_c** Constant bias (if $b_c = 0$, θ preserve constant trends)

$$\sum_{i=-p}^q \theta_i - 1$$

- **b_l** Linear bias (if $b_c = b_l = 0$, θ preserve constant trends)

$$\sum_{i=-p}^q i\theta_i$$

- **b_q** Quadratic bias (if $b_c = b_l = b_q = 0$, θ preserve quadratic trends)

$$\sum_{i=-p}^q i^2\theta_i$$

- F_g Fidelity criterium of Grun-Rehomme et al (2018)
- S_g Smoothness criterium of Grun-Rehomme et al (2018)
- T_g Timeliness criterium of Grun-Rehomme et al (2018)
- A_w Accuracy criterium of Wildi and McElroy (2019)
- S_w Smoothness criterium of Wildi and McElroy (2019)
- T_w Timeliness criterium of Wildi and McElroy (2019)
- R_w Residual criterium of Wildi and McElroy (2019)

A_w, S_w, T_w and R_w are computed only if the weights of the symmetric filter are provided (sweights).

Value

A vector of length 10 with the quality criteria.

References

Grun-Rehomme, Michel, Fabien Guggemos, and Dominique Ladiray (2018). “Asymmetric Moving Averages Minimizing Phase Shift”. In: Handbook on Seasonal Adjustment.

Wildi, Marc and McElroy, Tucker (2019). “The trilemma between accuracy, timeliness and smoothness in real-time signal extraction”. In: International Journal of Forecasting 35.3, pp. 1072–1084.

Examples

```
filter <- lp_filter(horizon = 6, kernel = "Henderson", endpoints = "LC")
sweights <- filter[, "q=6"]
aweight <- filter[, "q=0"]
diagnostic_matrix(aweight, sweights = sweights)
```

diagnostics-fit

Diagnostics and goodness of fit of filtered series

Description

Set of functions to compute diagnostics and goodness of fit of filtered series: cross validation (cv()) and cross validate estimate (cve()), leave-one-out cross validation estimate (loocve), CP statistic (cp()) and Rice's T statistics (rt()).

Usage

```
cve(x, coef, ...)

cv(x, coef, ...)

loocve(x, coef, ...)

rt(x, coef, ...)

cp(x, coef, var, ...)
```

Arguments

x	input time series.
coef	vector of coefficients or a moving-average (moving_average()).
...	other arguments passed to the function moving_average() to convert coef to a "moving_average" object.
var	variance used to compute the CP statistic (cp()).

Details

Let $(\theta_i)_{-p \leq i \leq q}$ be a moving average of length $p + q + 1$ used to filter a time series $(y_i)_{1 \leq i \leq n}$. Let denote $\hat{\mu}_t$ the filtered series computed at time t as:

$$\hat{\mu}_t = \sum_{i=-p}^q \theta_i y_{t+i}.$$

The cross validation estimate ([cve\(\)](#)) is defined as the time series $Y_t - \hat{\mu}_{-t}$ where $\hat{\mu}_{-t}$ is the leave-one-out cross validation estimate ([loocve\(\)](#)) defined as the filtered series computed deleting the observation t and remaining all the other points. The cross validation statistics ([cv\(\)](#)) is defined as:

$$CV = \frac{1}{n - (p + q)} \sum_{t=p+1}^{n-q} (y_t - \hat{\mu}_{-t})^2.$$

In the case of filtering with a moving average, we can show that:

$$\hat{\mu}_{-t} = \frac{\hat{\mu}_t - \theta_0 y_t}{1 - \theta_0}$$

and

$$CV = \frac{1}{n - (p + q)} \sum_{t=p+1}^{n-q} \left(\frac{y_t - \hat{\mu}_t}{1 - \theta_0} \right)^2.$$

In the case of filtering with a moving average, the CP estimate of risk (introduced by Mallows (1973); [cp\(\)](#)) can be defined as:

$$CP = \frac{1}{\sigma^2} \sum_{t=p+1}^{n-q} (y_t - \hat{\mu}_t)^2 - (n - (p + q))(1 - 2\theta_0).$$

The CP method requires an estimate of σ^2 (var parameter). The usual use of CP is to compare several different fits (for example different bandwidths): one should use the same estimate of $\hat{\sigma}^2$ for all fits (using for example `var_estimator()`). The recommendation of Cleveland and Devlin (1988) is to compute $\hat{\sigma}^2$ from a fit at the smallest bandwidth under consideration, at which one should be willing to assume that bias is negligible.

The Rice's T statistic (`rt()`) is defined as:

$$\frac{1}{n - (p + q)} \sum_{t=p+1}^{n-q} \frac{(y_t - \hat{\mu}_t)^2}{1 - 2\theta_0}$$

Value

Numeric value for the statistic and `ts()` or `vector()` for the estimates.

References

Loader, Clive. 1999. Local regression and likelihood. New York: Springer-Verlag.

Mallows, C. L. (1973). Some comments on Cp. *Technometrics* 15, 661–675.

Cleveland, W. S. and S. J. Devlin (1988). Locally weighted regression: An approach to regression analysis by local fitting. *Journal of the American Statistical Association* 83, 596–610.

Examples

```
x <- retailsa$DrinkingPlaces
h13 <- lp_filter(6)sfilter
cv(x, h13)
cve(x, h13)
loocve(x, h13)
rt(x, h13)
cp(x, h13, var = var_estimator(x, h13))
```

filter

Linear Filtering on a Time Series

Description

Applies linear filtering to a univariate time series or to each series separately of a multivariate time series using either a moving average (symmetric or asymmetric) or a combination of symmetric moving average at the center and asymmetric moving averages at the bounds.

Usage

```
filter(x, coefs, remove_missing = TRUE)
```

Arguments

- `x` a univariate or multivariate time series.
- `coefs` a matrix or a list that contains all the coefficients of the asymmetric and symmetric filters. (from the symmetric filter to the shortest). See details.
- `remove_missing` if TRUE (default) leading and trailing NA are removed before filtering.

Details

The functions `filter` extends `filter` allowing to apply every kind of moving averages (symmetric and asymmetric filters) or to apply a set of multiple moving averages to deal with the boundaries.

Let x_t be the input time series to filter.

- If `coef` is an object `moving_average()`, of length q , the result y is equal at time t to:

$$y[t] = x[t - \text{lags}] * \text{coef}[1] + x[t - \text{lags} + 1] * \text{coef}[1] + \dots + x[t - \text{lags} + q] * \text{coef}[q]$$

. It extends the function `filter` that would add NA at the end of the time series.

- If `coef` is a matrix, list or `finite_filters()` object, at the center, the symmetric moving average is used (first column/element of `coefs`). At the boundaries, the last moving average of `coefs` is used to compute the filtered time series $y[n]$ (no future point known), the second to last to compute the filtered time series $y[n - 1]$ (one future point known)...

Value

An object of the same class as `x` (ts, mts, vector or matrix) with the filtered time series.

Examples

```
x <- retailsa$DrinkingPlaces

lags <- 6
leads <- 2
fst_coef <- fst_filter(lags = lags, leads = leads, smoothness.weight = 0.3, timeliness.weight = 0.3)
lpp_coef <- lp_filter(horizon = lags, kernel = "Henderson", endpoints = "LC")

fst_ma <- filter(x, fst_coef)
lpp_ma <- filter(x, lpp_coef[, "q=2"])

graphics::plot(stats::ts.union(x, fst_ma, lpp_ma),
               plot.type = "single",
               col = c("black", "red", "blue"))

trend <- filter(x, lpp_coef)
# This is equivalent to:
trend <- localpolynomials(x, horizon = 6)
```

filters_operations	<i>Operations on Filters</i>
--------------------	------------------------------

Description

Manipulation of `moving_average()` or `finite_filters()` objects.

Usage

```
## S3 method for class 'moving_average'
sum(..., na.rm = FALSE)

## S4 method for signature 'moving_average,numeric'
x[i]

## S4 method for signature 'moving_average,logical'
x[i]

## S4 replacement method for signature 'moving_average,ANY,missing,numeric'
x[i] <- value

## S3 method for class 'moving_average'
cbind(..., zero_as_na = FALSE)

## S3 method for class 'moving_average'
rbind(...)

## S4 method for signature 'moving_average,moving_average'
e1 + e2

## S4 method for signature 'moving_average,numeric'
e1 + e2

## S4 method for signature 'numeric,moving_average'
e1 + e2

## S4 method for signature 'moving_average,missing'
e1 + e2

## S4 method for signature 'moving_average,missing'
e1 - e2

## S4 method for signature 'moving_average,moving_average'
e1 - e2

## S4 method for signature 'moving_average,numeric'
e1 - e2
```

```
## S4 method for signature 'numeric,moving_average'
e1 - e2

## S4 method for signature 'moving_average,moving_average'
e1 * e2

## S4 method for signature 'moving_average,numeric'
e1 * e2

## S4 method for signature 'numeric,moving_average'
e1 * e2

## S4 method for signature 'ANY,moving_average'
e1 * e2

## S4 method for signature 'moving_average,ANY'
e1 * e2

## S4 method for signature 'moving_average,numeric'
e1 / e2

## S4 method for signature 'moving_average,numeric'
e1 ^ e2

## S4 method for signature 'finite_filters,moving_average'
e1 * e2

## S4 method for signature 'moving_average,finite_filters'
e1 * e2

## S4 method for signature 'finite_filters,numeric'
e1 * e2

## S4 method for signature 'ANY,finite_filters'
e1 * e2

## S4 method for signature 'finite_filters,ANY'
e1 * e2

## S4 method for signature 'numeric,finite_filters'
e1 + e2

## S4 method for signature 'finite_filters,moving_average'
e1 + e2

## S4 method for signature 'moving_average,finite_filters'
e1 + e2
```

```

## S4 method for signature 'finite_filters,missing'
e1 + e2

## S4 method for signature 'finite_filters,missing'
e1 - e2

## S4 method for signature 'finite_filters,moving_average'
e1 - e2

## S4 method for signature 'moving_average,finite_filters'
e1 - e2

## S4 method for signature 'finite_filters,numeric'
e1 - e2

## S4 method for signature 'numeric,finite_filters'
e1 - e2

## S4 method for signature 'finite_filters,numeric'
e1 / e2

## S4 method for signature 'finite_filters,numeric'
e1 ^ e2

## S4 method for signature 'finite_filters,finite_filters'
e1 * e2

## S4 method for signature 'finite_filters,finite_filters'
e1 + e2

## S4 method for signature 'finite_filters,finite_filters'
e1 - e2

## S4 method for signature 'finite_filters,missing'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'finite_filters,ANY'
x[i, j, ..., drop = TRUE]

```

Arguments

<code>..., drop, na.rm</code>	other parameters.
<code>x, e1, e2</code>	object.
<code>i, j, value</code>	indices specifying elements to extract or replace and the new value
<code>zero_as_na</code>	boolean indicating if, when merging several moving averages (cbind) if trailing and leading zeros added to have a matrix form should be replaced by NA.

Value

A `moving_average()` or `finite_filters()` object resulting from the operation.

Examples

```
e1 <- moving_average(rep(1,12), lags = -6)
e1 <- e1/sum(e1)
e2 <- moving_average(rep(1/12, 12), lags = -5)
M2X12 <- (e1 + e2)/2
M2X12
h13 <- lp_filter(horizon = 6)
h13
h13 * M2X12
```

finite_filters

Manipulating Finite Filters

Description

Finite filters are a set of moving averages used to compute the filtered values of a time series. The symmetric filter is used on the central points (i.e. to obtain final estimates), while the right and left filters are used on the boundaries (at the end and beginning of the series) when the symmetric filter cannot be applied due to lack of data.

Usage

```
finite_filters(
  sfilter,
  rfilters = NULL,
  lfilters = NULL,
  first_to_last = FALSE
)
```

```
is.finite_filters(x)
```

```
## S3 method for class 'finite_filters'
to_seasonal(x, s)
```

Arguments

<code>sfilter</code>	the symmetric filter (<code>moving_average()</code> object) or a matrix or list with all the coefficients.
<code>rfilters</code>	the right filters (used on the last points).
<code>lfilters</code>	the left filters (used on the first points).
<code>first_to_last</code>	boolean indicating if the first element of <code>rfilters</code> is the first asymmetric filter (when only one observation is missing) or the last one (real-time estimates).

x object to test the class.
s seasonal period for the `to_seasonal()` function.

Value

`finite_filters()` returns a "finite_filters" object, which is an S4 class with three slots:

- `sfilter`: the symmetric filter (`moving_average()` object) used on the central points.
- `rfilters`: a list of `moving_average()` objects used on the boundaries at the end of the series (when the symmetric filter cannot be applied due to missing future data).
- `lfilters`: a list of `moving_average()` objects used on the boundaries at the beginning of the series (when the symmetric filter cannot be applied due to missing past data).

`is.finite_filters()` returns TRUE if the object is a "finite_filters" object, FALSE otherwise.

See Also

[print,finite_filters-method.](#)

Examples

```
ff_lp <- lp_filter()
ff_simple_ma <- finite_filters(moving_average(c(1, 1, 1), lags = -1)/3,
                             rfilters = list(moving_average(c(1, 1), lags = -1)/2))

ff_lp
ff_simple_ma
ff_lp * ff_simple_ma
# To get the symmetric filter and the right filters:
ff_lp@sfilter
ff_lp@rfilters
```

fst	<i>FST criteria</i>
-----	---------------------

Description

Compute the Fidelity, Smoothness and Timeliness (FST) criteria

Usage

```
fst(weights, lags, passband = pi/6, ...)
```

Arguments

weights either a "moving_average" or a numeric vector containing weights.
lags Lags of the moving average (when weights is not a "moving_average").
passband Passband threshold for timeliness criterion.
... other unused arguments.

Value

A named vector with the values of the 3 criteria.

References

Grun-Rehorme, Michel, Fabien Guggemos, and Dominique Ladiray (2018). “Asymmetric Moving Averages Minimizing Phase Shift”. In: Handbook on Seasonal Adjustment, <https://ec.europa.eu/eurostat/web/products-manuals-and-guidelines/-/ks-gq-18-001>.

Examples

```
filter <- lp_filter(horizon = 6, kernel = "Henderson", endpoints = "LC")
fst(filter[, "q=0"])
# To compute the statistics on all filters:
fst(filter)
```

fst_filter	<i>Estimation of a filter using the Fidelity-Smoothness-Timeliness criteria</i>
------------	---

Description

Estimation of a filter using the Fidelity-Smoothness-Timeliness criteria

Usage

```
fst_filter(
  lags = 6,
  leads = 0,
  pdegree = 2,
  smoothness.weight = 1,
  smoothness.degree = 3,
  timeliness.weight = 0,
  timeliness.passband = pi/6,
  timeliness.antiphase = TRUE
)
```

Arguments

lags	Lags of the filter (should be positive).
leads	Leads of the filter (should be positive or 0).
pdegree	Local polynomials preservation: max degree.
smoothness.weight	Weight for the smoothness criterion (in $[0, 1]$).
smoothness.degree	Degree of the smoothness criterion (3 for Henderson).

`timeliness.weight`
 Weight for the Timeliness criterion (in $[0, 1]$). `sweight+twweight` should be in $[0, 1]$.

`timeliness.passband`
 Passband for the timeliness criterion (in radians). The phase effect is computed in $[0, \text{passband}]$.

`timeliness.antiphase`
 boolean indicating if the timeliness should be computed analytically (TRUE) or numerically (FALSE).

Details

Moving average computed by a minimisation of a weighted mean of three criteria under polynomial constraints. Let $\theta = (\theta_{-p}, \dots, \theta_f)'$ be a moving average where p and f are two integers defined by the parameter lags and leads. The three criteria are:

- *Fidelity*, F_g : it's the variance reduction ratio.

$$F_g(\theta) = \sum_{k=-p}^{+f} \theta_k^2$$

- *Smoothness*, S_g : it measures the flexibility of the coefficient curve of a filter and the smoothness of the trend.

$$S_g(\theta) = \sum_j (\nabla^q \theta_j)^2$$

The integer q is defined by parameter `smoothness.degree`. By default, the Henderson criteria is used (`smoothness.degree = 3`).

- *Timeliness*, T_g :

$$T_g(\theta) = \int_0^{\omega_2} f(\rho_\theta(\omega), \varphi_\theta(\omega)) d\omega$$

with ρ_θ and φ_θ the gain and phase shift functions of θ , and f a penalty function defined as $f: (\rho, \varphi) \mapsto \rho^2 \sin(\varphi)^2$ to have an analytically solvable criterium. ω_2 is defined by the parameter `timeliness.passband` and is by default equal to $2\pi/12$: for monthly time series, we focus on the timeliness associated to cycles of 12 months or more.

The moving average is then computed solving the problem:

$$\begin{cases} \min_{\theta} & J(\theta) = (1 - \beta - \gamma)F_g(\theta) + \beta S_g(\theta) + \gamma T_g(\theta) \\ \text{s.t.} & C\theta = a \end{cases}$$

Where $C\theta = a$ represents linear constraints to have a moving average that preserve polynomials of degree q (`pdegree`):

$$C = \begin{pmatrix} 1 & \cdots & 1 \\ -h & \cdots & h \\ \vdots & \cdots & \vdots \\ (-h)^d & \cdots & h^d \end{pmatrix}, \quad a = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Value

A `finite_filters()` object.

References

Grun-Rehomme, Michel, Fabien Guggemos, and Dominique Ladiray (2018). “Asymmetric Moving Averages Minimizing Phase Shift”. In: Handbook on Seasonal Adjustment, <https://ec.europa.eu/eurostat/web/products-manuals-and-guidelines/-/ks-gq-18-001>.

Examples

```
filter <- fst_filter(lags = 6, leads = 0)
filter
```

get_kernel	<i>Get the coefficients of a kernel</i>
------------	---

Description

Function to get the coefficient associated to a kernel. Those coefficients are then used to compute the different filters.

Usage

```
get_kernel(
  kernel = c("Henderson", "Uniform", "Triangular", "Epanechnikov", "Parabolic",
    "BiWeight", "TriWeight", "Tricube", "Trapezoidal", "Gaussian"),
  horizon,
  sd_gauss = 0.25
)
```

Arguments

kernel	kernel uses.
horizon	horizon (bandwidth) of the symmetric filter.
sd_gauss	standard deviation for gaussian kernel. By default 0.25.

Value

tskernel object (see [kernel](#)).

Examples

```
get_kernel("Henderson", horizon = 3)
```

get_moving_average	<i>Get Moving Averages from ARIMA model</i>
--------------------	---

Description

Get Moving Averages from ARIMA model

Usage

```
get_moving_average(x, ...)
```

Arguments

x	the object.
...	unused parameters

Value

A `moving_average()` object.

Examples

```
fit <- stats::arima(log10(AirPassengers), c(0, 1, 1),  
seasonal = list(order = c(0, 1, 1), period = 12))  
get_moving_average(fit)
```

get_properties_function	<i>Get properties of filters</i>
-------------------------	----------------------------------

Description

Get properties of filters

Usage

```
get_properties_function(  
  x,  
  component = c("Symmetric Gain", "Symmetric Phase", "Symmetric transfer",  
    "Asymmetric Gain", "Asymmetric Phase", "Asymmetric transfer"),  
  ...  
)
```

Arguments

x a "moving_average" or "finite_filters" object.
 component the component to extract.
 ... unused other arguments.

Value

A function that takes a frequency as input and returns the value of the selected component.

Examples

```
filter <- lp_filter(3, kernel = "Henderson")
sgain <- get_properties_function(filter, "Symmetric Gain")
graphics::plot(sgain, xlim= c(0, pi/12))
```

implicit_forecasts	<i>Retrieve implicit forecasts corresponding to the asymmetric filters</i>
--------------------	--

Description

Function to retrieve the implicit forecasts corresponding to the asymmetric filters

Usage

```
implicit_forecasts(x, coefs)
```

Arguments

x a univariate or multivariate time series.
 coefs a matrix or a list that contains all the coefficients of the asymmetric and symmetric filters. (from the symmetric filter to the shortest). See details.

Details

Let h be the bandwidth of the symmetric filter, $v_{-h}, \dots, v_h$ the coefficients of the symmetric filter and $w_{-h}^q, \dots, w_h^q$ the coefficients of the asymmetric filter used to estimate the trend when q future values are known (with the convention $w_{q+1}^q = \dots = w_h^q = 0$). Let denote $y_{-h}, \dots, y_0$ the last h available values of the input times series. The implicit forecast $y_1^*, \dots, y_h^*$ induced by $w^0, \dots, w^{h-1}$ are defined by:

$$\forall q \in \{0, \dots, h-1\}, \quad \sum_{i=-h}^0 v_i y_i + \sum_{i=1}^h v_i y_i^* = \sum_{i=-h}^0 w_i^q y_i + \sum_{i=1}^h w_i^q y_i^*.$$

which is equivalent to

$$\forall q, \quad \sum_{i=1}^h (v_i - w_i^q) y_i^* = \sum_{i=-h}^0 (w_i^q - v_i) y_i.$$

Note that this is solved numerically: the solution isn't exact.

Value

An object of the same class as `x` (ts, mts, vector or matrix) with the implicit forecasts.

Examples

```
x <- retailsa$AllOtherGenMerchandiseStores
ql <- lp_filter(horizon = 6, kernel = "Henderson", endpoints = "QL")
lc <- lp_filter(horizon = 6, kernel = "Henderson", endpoints = "LC")
f_ql <- implicit_forecasts(x, ql)
f_lc <- implicit_forecasts(x, lc)

graphics::plot(window(x, start = 2007),
               xlim = c(2007,2012))
graphics::lines(
  x = stats::ts(
    c(utils::tail(x,1), f_ql),
    frequency = stats::frequency(x),
    start = stats::end(x)
  ),
  col = "red",
  lty = 2
)
graphics::lines(
  x = stats::ts(c(utils::tail(x,1), f_lc),
    frequency = stats::frequency(x),
    start = stats::end(x)
  ),
  col = "blue",
  lty = 2
)
```

impute_last_obs

Impute Incomplete Finite Filters

Description

Impute Incomplete Finite Filters

Usage

```
impute_last_obs(x, n = NULL, nperiod = 1, backward = TRUE, forward = TRUE)
```

Arguments

`x` a `finite_filters()` object.

`n` integer specifying the number of imputed periods. By default all the missing moving averages are imputed.

`nperiod` integer specifying how to impute missing date.

- `nperiod = 1` means imputation using last filtered data (1 period backward),
- `nperiod = 12` with monthly data means imputation using last year filtered data, etc.

`backward, forward` boolean indicating if the imputation should be done backward (on left filters), forward (on right filters).

Details

When combining finite filters and a moving average, the first and/or the last points cannot be computed.

For example, using the M2X12 moving average, that is to say the symmetric moving average with coefficients

$$\theta = \frac{1}{24}B^6 + \frac{1}{12}B^5 + \dots + \frac{1}{12}B^{-5} + \frac{1}{24}B^{-6},$$

the first and last 6 points cannot be computed.

`impute_last_obs()` allows to impute the first/last points using the `nperiod` previous filtered data. With `nperiod = 1`, the last filtered data is used for the imputation, with `nperiod = 12` and monthly data, the last year filtered data is used for the imputation, etc.

Value

A `finite_filters()` object.

Examples

```
y <- window(retailsa$AllOtherGenMerchandiseStores, start = 2008)
M3 <- moving_average(rep(1/3, 3), lags = -1)
M3X3 <- M3 * M3
M2X12 <- (simple_ma(12, -6) + simple_ma(12, -5)) / 2
composite_ma <- M3X3 * M2X12
# The last 6 points cannot be computed
composite_ma
composite_ma * y
# they can be computed using the last filtered data
# e.g. to impute the first 3 missing months with last period:
impute_last_obs(composite_ma, n = 3, nperiod = 1) * y
# or using the filtered data of the same month in previous year
impute_last_obs(composite_ma, n = 6, nperiod = 12) * y
```

localpolynomials	<i>Apply Local Polynomials Filters</i>
------------------	--

Description

Apply Local Polynomials Filters

Usage

```
localpolynomials(
  x,
  horizon = 6,
  degree = 3,
  kernel = c("Henderson", "Uniform", "Biweight", "Trapezoidal", "Triweight", "Tricube",
    "Gaussian", "Triangular", "Parabolic"),
  endpoints = c("LC", "QL", "CQ", "CC", "DAF"),
  ic = 3.5,
  tweight = 0,
  passband = pi/12
)
```

Arguments

x	input time-series.
horizon	horizon (bandwidth) of the symmetric filter.
degree	degree of polynomial.
kernel	kernel uses.
endpoints	method for endpoints.
ic	ic ratio.
tweight	timeliness weight.
passband	passband threshold.

Value

A vector or ts object with the filtered time series.

References

Proietti, Tommaso and Alessandra Luati (2008). “Real time estimation in local polynomial regression, with application to trend-cycle analysis”.

See Also

[lp_filter\(\)](#).

Examples

```
x <- retailsa$AllOtherGenMerchandiseStores
trend <- localpolynomials(x, horizon = 6)
graphics::plot(x)
graphics::lines(trend, col = "red")
```

lp_filter

*Local Polynomials Filters***Description**

Local Polynomials Filters

Usage

```
lp_filter(
  horizon = 6,
  degree = 3,
  kernel = c("Henderson", "Uniform", "Biweight", "Trapezoidal", "Triweight", "Tricube",
    "Gaussian", "Triangular", "Parabolic"),
  endpoints = c("LC", "QL", "CQ", "CC", "DAF", "CN"),
  ic = 3.5,
  tweight = 0,
  passband = pi/12
)
```

Arguments

horizon	horizon (bandwidth) of the symmetric filter.
degree	degree of polynomial.
kernel	kernel uses.
endpoints	method for endpoints.
ic	ic ratio.
tweight	timeliness weight.
passband	passband threshold.

Details

- "LC": Linear-Constant filter
- "QL": Quadratic-Linear filter
- "CQ": Cubic-Quadratic filter
- "CC": Constant-Constant filter
- "DAF": Direct Asymmetric filter
- "CN": Cut and Normalized Filter

Value

a `finite_filters()` object.

A `finite_filters()` object.

References

Proietti, Tommaso and Alessandra Luati (2008). “Real time estimation in local polynomial regression, with application to trend-cycle analysis”.

See Also

`mmsre_filter()` `localpolynomials()`.

Examples

```
henderson_f <- lp_filter(horizon = 6, kernel = "Henderson")
plot_coef(henderson_f)
```

mmsre_filter

Mean Square Revision Error (mmsre) filter

Description

Provides an asymmetric filter based on the given reference filter (usually symmetric) minimizing the mean square revision error.

Usage

```
mmsre_filter(
  ref_filter,
  q,
  U,
  Z = NULL,
  delta = NULL,
  kernel = NULL,
  tweight = 0,
  passband = pi/12
)
```

Arguments

<code>ref_filter</code>	The reference filter (a <code>moving_average()</code> object).
<code>q</code>	The horizon of the asymmetric filter.
<code>U</code>	Matrix of the constraints.
<code>Z</code>	Matrix of the bias (can be NULL).

delta	Coefficients of the linear model.
kernel	The kernel used for weighting factors, by default, no weight is used. See lp_filter() for the available kernels.
tweight	timeliness weight.
passband	passband threshold.

Details

The asymmetric filter $\mathbf{v} = (v_{-h}, \dots, v_q)'$ minimizes the mean square revision error (mmsre) relative to the reference filter $\boldsymbol{\theta} = (\theta_{-h}, \dots, \theta_h)'$. The series follows the model

$$\mathbf{y} = \mathbf{U}\boldsymbol{\gamma} + \mathbf{Z}\boldsymbol{\delta} + \boldsymbol{\varepsilon}, \quad \boldsymbol{\varepsilon} \sim \mathcal{N}(0, \sigma^2 \mathbf{K}^{-1}).$$

With $\mathbf{K}$ a set of weights (kernel), by default (kernel = NULL) no weight is used. The matrix $\mathbf{U}$ represents the constraints of the symmetric filter (usually polynomials preservations), $\boldsymbol{\theta}$, imposed to the asymmetric filter, $\mathbf{v}$. Partitionning the matrix $\mathbf{U} = (\mathbf{U}_p' \quad \mathbf{U}_f')'$ with $\mathbf{U}_p$ the first $h + q + 1$ rows and $\mathbf{U}_f$ the remaining rows, the constraints are $\mathbf{U}_p' \mathbf{v} = \mathbf{U}' \boldsymbol{\theta}$.

The matrix $\mathbf{Z}$ represents the bias of the asymmetric filter: usually constraints imposed to the symmetric filter but not to the asymmetric filter.

Value

A [moving_average\(\)](#) object.

References

Proietti, Tommaso and Alessandra Luati (2008). “Real time estimation in local polynomial regression, with application to trend-cycle analysis”.

See Also

[lp_filter\(\)](#).

Examples

```
QL <- lp_filter(endpoints = "QL", ic = 3.5)
LC <- lp_filter(endpoints = "LC", ic = 3.5)
DAF <- lp_filter(endpoints = "DAF")
h6 <- QL[, "q=6"]
# To reproduce DAF filter
mmsre_filter(
  ref_filter = h6, q = 0,
  U = polynomial_matrix(l = - 6, d0 = 0, d1 = 3),
  kernel = "Henderson"
)
DAF[, "q=0"]
# To reproduce QL filter
mmsre_filter(
  ref_filter = h6, q = 1,
  delta = 2 / (sqrt(pi) * 3.5),
```

```

    U = polynomial_matrix(l = -6, d0 = 0, d1 = 1),
    Z = polynomial_matrix(l = -6, d0 = 2, d1 = 2)
)
QL[, "q=1"]

# Or using the Uniform kernel
mmsre_filter(
  ref_filter = h6, q = 2,
  # we multiply by the square root of the inverse of weights (1/13)
  # to get the same result as the QL filter
  delta = 2 / (sqrt(pi) * 3.5) * (sqrt(13)),
  U = polynomial_matrix(l = -6, d0 = 0, d1 = 0),
  Z = polynomial_matrix(l = -6, d0 = 1, d1 = 1),
  kernel = "Uniform"
)
LC[, "q=2"]

```

moving_average

Manipulation of moving averages

Description

Manipulation of moving averages

Usage

```

moving_average(
  x,
  lags = -length(x),
  trailing_zero = FALSE,
  leading_zero = FALSE
)

is.moving_average(x)

## S3 method for class 'moving_average'
coef(object, ...)

is_symmetric(x)

upper_bound(x)

lower_bound(x)

mirror(x)

## S3 method for class 'moving_average'

```

```

rev(x)

## S3 method for class 'moving_average'
length(x)

to_seasonal(x, s)

## S4 method for signature 'moving_average'
show(object)

```

Arguments

x vector of coefficients.
lags integer indicating the number of lags of the moving average.
trailing_zero, leading_zero boolean indicating whether to remove leading/trailing zero and NA.
object moving_average object.
... other unused parameters.
s seasonal period for the to_seasonal() function.

Details

A moving average is defined by a set of coefficient $\theta = (\theta_{-p}, \dots, \theta_f)'$ such all time series X_t are transformed as:

$$M_{\theta}(X_t) = \sum_{k=-p}^{+f} \theta_k X_{t+k} = \left(\sum_{k=-p}^{+f} \theta_k B^{-k} \right) X_t.$$

The integer p is defined by the parameter `lags` and can be retrieved by the function `lower_bound()`. The integer f is defined by the length of the coefficients and can be retrieved by the function `upper_bound()`. The length of the moving average is defined as $f + p + 1$ and can be retrieved by the function `length()`. A moving average is symmetric (`is_symmetric()`) if $p = f$ and $\theta_{-k} = \theta_k$ for all $k = 0, \dots, p$.

The function `to_seasonal()` transforms the moving average θ to:

$$M_{\theta'}(X_t) = \sum_{k=-p}^{+f} \theta_k X_{t+k} = \left(\sum_{k=-p}^{+f} \theta_k B^{-k} \right) X_t.$$

The functions `mirror()` and `rev()` transforms the moving average θ to:

$$M_{\theta'}(X_t) = \sum_{k=-f}^{+p} \theta_k X_{t+k} = \left(\sum_{k=-f}^{+p} \theta_k B^{-k} \right) X_t.$$

Value

`moving_average()` returns a "moving_average" object which is an S4 class with three slots:

- `coefficients`: a numeric vector of coefficients.
- `lower_bound`: an integer indicating the number of lags.
- `upper_bound`: an integer indicating the number of leads (automatically computed).

`is.moving_average()` and `is_symmetric()` return a boolean. `length()`, `numeric()`, `lower_bound()` and `upper_bound()` return an integer. `mirror()`, `rev()` and `to_seasonal()` return a "moving_average" object.

Examples

```
y <- retailsa$AllOtherGenMerchandiseStores
e1 <- moving_average(rep(1,12), lags = -6)
e1 <- e1/sum(e1)
e2 <- moving_average(rep(1/12, 12), lags = -5)
M2X12 <- (e1 + e2)/2
stats::coef(M2X12)
M3 <- moving_average(rep(1/3, 3), lags = -1)
M3X3 <- M3 * M3
# M3X3 moving average applied to each month
M3X3
M3X3_seasonal <- to_seasonal(M3X3, 12)
# M3X3_seasonal moving average applied to the global series
M3X3_seasonal

def.par <- par(no.readonly = TRUE)
par(mai = c(0.5, 0.8, 0.3, 0))
layout(matrix(c(1,2), nrow = 1))
plot_gain(M3X3, main = "M3X3 applied to each month")
plot_gain(M3X3_seasonal, main = "M3X3 applied to the global series")
par(def.par)

# To apply the moving average
t <- y * M2X12
# Or use the filter() function:
t <- filter(y, M2X12)
si <- y - t
s <- si * M3X3_seasonal
# or equivalently:
s_mm <- M3X3_seasonal * (1 - M2X12)
s <- y * s_mm
graphics::plot(s)
```

mse	<i>Accuracy/smoothness/timeliness criteria through spectral decomposition</i>
-----	---

Description

Accuracy/smoothness/timeliness criteria through spectral decomposition

Usage

```
mse(aweights, sweights, density = c("uniform", "rw"), passband = pi/6, ...)

## S3 method for class 'finite_filters'
mse(
  aweights,
  sweights = aweights@sfilter,
  density = c("uniform", "rw"),
  passband = pi/6,
  sfilter = TRUE,
  rfilters = TRUE,
  lfilters = FALSE,
  ...
)
```

Arguments

<code>aweights</code>	moving_average or finite_filters object or weights of the asymmetric filter (from -n to m).
<code>sweights</code>	moving_average object or weights of the symmetric filter (from 0 to n or -n to n). If aweights is a finite_filters() object, sweights is by default the symmetric filter.
<code>density</code>	hypothesis on the spectral density: "uniform" (= white noise, the default) or "rw" (= random walk).
<code>passband</code>	passband threshold.
<code>...</code>	other unused arguments.
<code>sfilter, rfilters, lfilters</code>	boolean indicating if the criteria should be computed for the symmetric filter (<code>sfilter = TRUE</code> by default), the right filters (<code>rfilters = TRUE</code> by default) and the left filters (<code>lfilters = FALSE</code> by default).

Value

If sweights is a [finite_filters\(\)](#) object, a matrix with the value of the three criteria for asymmetric and symmetric filters (depending on the value of `sfilter`, `rfilters` and `lfilters`).

Otherwise, a named vector with the value of the three criteria is returned.

References

Wildi, Marc and McElroy, Tucker (2019). “The trilemma between accuracy, timeliness and smoothness in real-time signal extraction”. In: International Journal of Forecasting 35.3, pp. 1072–1084.

Examples

```
filter <- lp_filter(horizon = 6, kernel = "Henderson", endpoints = "LC")
sweights <- filter[, "q=6"]
aweights <- filter[, "q=0"]
mse(aweights, sweights)
# Or to compute directly the criteria on all asymmetric filters:
mse(filter)
```

plot_filters

Plots filters properties

Description

Functions to plot the coefficients, the gain and the phase functions.

Usage

```
plot_coef(x, nxlab = 7, add = FALSE, ..., xlab = "", ylab = "coefficient")

## Default S3 method:
plot_coef(
  x,
  nxlab = 7,
  add = FALSE,
  zero_as_na = TRUE,
  q = 0,
  legend = FALSE,
  legend.pos = "topright",
  ...,
  xlab = "",
  ylab = "coefficient"
)

## S3 method for class 'moving_average'
plot_coef(x, nxlab = 7, add = FALSE, ..., xlab = "", ylab = "coefficient")

## S3 method for class 'finite_filters'
plot_coef(
  x,
  nxlab = 7,
  add = FALSE,
```

```
    zero_as_na = TRUE,
    q = 0,
    legend = length(q) > 1,
    legend.pos = "topright",
    ...,
    xlab = "",
    ylab = "coefficient"
)

plot_gain(
  x,
  nxlab = 7,
  add = FALSE,
  xlim = c(0, pi),
  ...,
  xlab = "",
  ylab = "gain"
)

## S3 method for class 'moving_average'
plot_gain(
  x,
  nxlab = 7,
  add = FALSE,
  xlim = c(0, pi),
  ...,
  xlab = "",
  ylab = "gain"
)

## S3 method for class 'finite_filters'
plot_gain(
  x,
  nxlab = 7,
  add = FALSE,
  xlim = c(0, pi),
  q = 0,
  legend = length(q) > 1,
  legend.pos = "topright",
  n = 101,
  ...,
  xlab = "",
  ylab = "gain"
)

plot_phase(
  x,
  nxlab = 7,
```

```

    add = FALSE,
    xlim = c(0, pi),
    normalized = FALSE,
    ...,
    xlab = "",
    ylab = "phase"
)

## S3 method for class 'moving_average'
plot_phase(
  x,
  nxlab = 7,
  add = FALSE,
  xlim = c(0, pi),
  normalized = FALSE,
  ...,
  xlab = "",
  ylab = "phase"
)

## S3 method for class 'finite_filters'
plot_phase(
  x,
  nxlab = 7,
  add = FALSE,
  xlim = c(0, pi),
  normalized = FALSE,
  q = 0,
  legend = length(q) > 1,
  legend.pos = "topright",
  n = 101,
  ...,
  xlab = "",
  ylab = "phase"
)

```

Arguments

x	coefficients, gain or phase.
nxlab	number of xlab.
add	boolean indicating if the new plot is added to the previous one.
...	other arguments to <code>matplot</code> .
xlab, ylab	labels of axis.
zero_as_na	boolean indicating if the trailing zero of the coefficients should be plotted (FALSE) or removed (TRUE).
q	q.
legend	boolean indicating if the legend is printed.

legend.pos	position of the legend.
xlim	vector containing x limits.
n	number of points used to plot the functions.
normalized	boolean indicatif if the phase function is normalized by the frequency.

Value

No return value, called for side effects.

Examples

```
filter <- lp_filter(6, endpoints = "DAF", kernel = "Henderson")
plot_coef(filter, q = c(0,3), legend = TRUE)
plot_gain(filter, q = c(0,3), legend = TRUE)
plot_phase(filter, q = c(0,3), legend = TRUE)
```

polynomial_matrix	<i>Create polynomial matrix</i>
-------------------	---------------------------------

Description

Create polynomial matrix used in local polynomial regression (see details).

Usage

```
polynomial_matrix(l, u = abs(l), d0 = 0, d1 = 3)
```

Arguments

l, u	lower bound (usually negative) and upper bound (usually positive) of the polynomial matrix.
d0, d1	lower and upper polynomial degree of the polynomial matrix.

Details

polynomial_matrix() computes the following matrix:

$$\begin{pmatrix} (l)^{d_0} & (l)^{d_0+1} & \dots & (l)^{d_1} \\ (l+1)^{d_0} & (l+1)^{d_0+1} & \dots & (l+1)^{d_1} \\ \vdots & \vdots & \dots & \vdots \\ (u)^{d_0} & (u)^{d_0+1} & \dots & (u)^{d_1} \end{pmatrix}.$$

Value

A "matrix" of size $(u - l + 1) \times (d_1 - d_0 + 1)$.

Examples

```
# For example to reproduce DAF filters
daf <- lp_filter(horizon = 6, endpoints = "DAF")
q <- 0
X <- polynomial_matrix(l = -6, u = q, d0 = 0, d1 = 3)
K <- diag(sapply(-6:q, function(i) get_kernel(horizon = 6)[i]))
e_1 <- c(1, 0, 0, 0)
q0 <- K %*% X %*% solve(t(X) %*% K %*% X, e_1)
q0
daf[, "q=0"]
```

```
print,finite_filters-method
```

Print method for Finite Filters Objects

Description

Printing objects of class "finite_filters".

Usage

```
## S4 method for signature 'finite_filters'
print(
  x,
  sfilter = TRUE,
  rfilters = TRUE,
  lfilters = FALSE,
  zero_as_na = FALSE,
  ...
)

## S4 method for signature 'finite_filters'
show(object)

## S3 method for class 'finite_filters'
as.matrix(
  x,
  sfilter = TRUE,
  rfilters = TRUE,
  lfilters = FALSE,
  zero_as_na = FALSE,
  ...
)
```

Arguments

<code>x, object</code>	An object of class "finite_filters".
<code>sfilter, rfilters, lfilters</code>	Logical indicating whether to add the symmetric filter (<code>sfilter = TRUE</code> by default), the right filters (<code>rfilters = TRUE</code> by default) or the left filters (<code>lfilters = FALSE</code> by default).
<code>zero_as_na</code>	Logical. Asymmetric filters have fewer terms than the symmetric filter. If <code>TRUE</code> , the missing past/future terms in the matrix representation are filled with <code>NA</code> instead of <code>0</code> . Defaults to <code>FALSE</code> .
<code>...</code>	Further arguments to be passed to or from methods.

Details

The `print` method just calls the `as.matrix()` method and prints the resulting matrix.

Value

The `print` method returns the argument `x`, invisibly, as for all `base::print()` methods.

`as.matrix.finite_filters()` returns a matrix containing the coefficients of the symmetric and asymmetric filters. The columns are named according to the number of future points used (for right filters). For example, column `q=0` corresponds to the asymmetric filter using no future points, while `q=1` corresponds to the filter using one future point, and so on. A similar convention with negative values is used for left filters (when printed).

See Also

`finite_filters()`.

Examples

```
ff_lp <- lp_filter()
as.matrix(ff_lp)
```

retailsa

Seasonally Adjusted Retail Sales

Description

A dataset containing monthly seasonally adjusted retail sales

Usage

```
retailsa
```

Format

A list of `ts` objects from january 1992 to december 2010.

rkhs_filter

*Reproducing Kernel Hilbert Space (RKHS) Filters***Description**

Estimation of a filter using Reproducing Kernel Hilbert Space (RKHS)

Usage

```
rkhs_filter(
  horizon = 6,
  degree = 2,
  kernel = c("BiWeight", "Henderson", "Epanechnikov", "Triangular", "Uniform",
    "TriWeight"),
  asymmetricCriterion = c("Timeliness", "FrequencyResponse", "Accuracy", "Smoothness",
    "Undefined"),
  density = c("uniform", "rw"),
  passband = 2 * pi/12,
  optimalbw = TRUE,
  optimal.minBandwidth = horizon,
  optimal.maxBandwidth = 3 * horizon,
  bandwidth = horizon + 1
)
```

Arguments

horizon	horizon (bandwidth) of the symmetric filter.
degree	degree of polynomial.
kernel	kernel uses.
asymmetricCriterion	the criteria used to compute the optimal bandwidth. If "Undefined", $m + 1$ is used.
density	hypothesis on the spectral density: "uniform" (= white noise, the default) or "rw" (= random walk).
passband	passband threshold.
optimalbw	boolean indicating if the bandwidth should be choosen by optimisation (between optimal.minBandwidth and optimal.minBandwidth using the criteria asymmetricCriterion). If optimalbw = FALSE then the bandwidth specified in bandwidth will be used.
optimal.minBandwidth, optimal.maxBandwidth	the range used for the optimal bandwidth selection.
bandwidth	the bandwidth to use if optimalbw = FALSE.

Value

A `finite_filters()` object.

References

Dagum, Estela Bee and Silvia Bianconcini (2008). “The Henderson Smoother in Reproducing Kernel Hilbert Space”. In: Journal of Business & Economic Statistics 26, pp. 536–545. URL: <https://ideas.repec.org/a/bes/jnlbes/v26y2008p536-545.html>.

Examples

```
rkhs <- rkhs_filter(horizon = 6, asymmetricCriterion = "Timeliness")
plot_coef(rkhs)
```

rkhs_kernel	<i>Get RKHS kernel function</i>
-------------	---------------------------------

Description

Get RKHS kernel function

Usage

```
rkhs_kernel(
  kernel = c("Biweight", "Henderson", "Epanechnikov", "Triangular", "Uniform",
    "Triweight"),
  degree = 2,
  horizon = 6
)
```

Arguments

kernel	kernel uses.
degree	degree of polynomial.
horizon	horizon (bandwidth) of the symmetric filter.

Value

A function that takes a numeric input and returns the value of the RKHS kernel.

Examples

```
biweight <- rkhs_kernel(kernel = "Biweight")
triangular <- rkhs_kernel(kernel = "Triangular")
graphics::plot(biweight, -1, 1)
graphics::plot(triangular, -1, 1, add = TRUE, col = "orange")
```

rkhs_optimal_bw	<i>Optimal Bandwidth of Reproducing Kernel Hilbert Space (RKHS) Filters</i>
-----------------	---

Description

Function to compute the optimal bandwidths used in Reproducing Kernel Hilbert Space (RKHS) filters

Usage

```
rkhs_optimal_bw(
  horizon = 6,
  degree = 2,
  kernel = c("Biweight", "Henderson", "Epanechnikov", "Triangular", "Uniform",
    "Triweight"),
  asymmetricCriterion = c("Timeliness", "FrequencyResponse", "Accuracy", "Smoothness"),
  density = c("uniform", "rw"),
  passband = 2 * pi/12,
  optimal.minBandwidth = horizon,
  optimal.maxBandwidth = 3 * horizon
)
```

Arguments

horizon	horizon (bandwidth) of the symmetric filter.
degree	degree of polynomial.
kernel	kernel uses.
asymmetricCriterion	the criteria used to compute the optimal bandwidth. If "Undefined", $m + 1$ is used.
density	hypothesis on the spectral density: "uniform" (= white noise, the default) or "rw" (= random walk).
passband	passband threshold.
optimal.minBandwidth, optimal.maxBandwidth	the range used for the optimal bandwidth selection.

Value

A vector of optimal bandwidths for each lead time.

Examples

```
rkhs_optimal_bw(asymmetricCriterion = "Timeliness")
rkhs_optimal_bw(asymmetricCriterion = "Timeliness", optimal.minBandwidth = 6.2)
```

rkhs_optimization_fun *Optimization Function of Reproducing Kernel Hilbert Space (RKHS) Filters*

Description

Export function used to compute the optimal bandwidth of Reproducing Kernel Hilbert Space (RKHS) filters.

Usage

```
rkhs_optimization_fun(
  horizon = 6,
  leads = 0,
  degree = 2,
  kernel = c("Biweight", "Henderson", "Epanechnikov", "Triangular", "Uniform",
    "Triweight"),
  asymmetricCriterion = c("Timeliness", "FrequencyResponse", "Accuracy", "Smoothness"),
  density = c("uniform", "rw"),
  passband = 2 * pi/12
)
```

Arguments

horizon	horizon (bandwidth) of the symmetric filter.
leads	Leads of the filter (should be positive or 0).
degree	degree of polynomial.
kernel	kernel uses.
asymmetricCriterion	the criteria used to compute the optimal bandwidth. If "Undefined", $m + 1$ is used.
density	hypothesis on the spectral density: "uniform" (= white noise, the default) or "rw" (= random walk).
passband	passband threshold.

Value

A function that takes a bandwidth as input and returns the value of the optimization criterion.

Examples

```
graphics::plot(
  rkhs_optimization_fun(
    horizon = 6,
    leads = 0,
    degree = 3,
```

```

        asymmetricCriterion = "Timeliness"
    ),
    5.5,
    6 * 3,
    ylab = "Timeliness",
    main = "6X0 filter"
)
graphics::plot(
    rkhs_optimization_fun(
        horizon = 6,
        leads = 1,
        degree = 3,
        asymmetricCriterion = "Timeliness"
    ),
    5.5,
    6 * 3,
    ylab = "Timeliness",
    main = "6X1 filter"
)
graphics::plot(
    rkhs_optimization_fun(
        horizon = 6,
        leads = 2,
        degree = 3,
        asymmetricCriterion = "Timeliness"
    ),
    5.5,
    6 * 3,
    ylab = "Timeliness",
    main = "6X2 filter"
)
graphics::plot(
    rkhs_optimization_fun(
        horizon = 6,
        leads = 3,
        degree = 3,
        asymmetricCriterion = "Timeliness"
    ),
    5.5,
    6 * 3,
    ylab = "Timeliness",
    main = "6X3 filter"
)
graphics::plot(
    rkhs_optimization_fun(
        horizon = 6,
        leads = 4,
        degree = 3,
        asymmetricCriterion = "Timeliness"
    ),
    5.5,
    6 * 3,
    ylab = "Timeliness",

```

```

    main = "6X4 filter"
  )
  graphics::plot(
    rkhs_optimization_fun(
      horizon = 6,
      leads = 5,
      degree = 3,
      asymmetricCriterion = "Timeliness"
    ),
    5.5,
    6 * 3,
    ylab = "Timeliness",
    main = "6X5 filter"
  )

```

simple_ma

*Simple Moving Average***Description**

A simple moving average is a moving average whose coefficients are all equal and whose sum is 1

Usage

```
simple_ma(order, lags = -trunc((order - 1)/2))
```

Arguments

order	number of terms of the moving_average
lags	integer indicating the number of lags of the moving average.

Value

A `moving_average()` object.

Examples

```

# The M2X12 moving average is computed as
(simple_ma(12, -6) + simple_ma(12, -5)) / 2
# The M3X3 moving average is computed as
simple_ma(3, -1) ^ 2
# The M3X5 moving average is computed as
simple_ma(3, -1) * simple_ma(5, -2)

```

underlying_forecasts *Retrieve underlying forecasts corresponding to the asymmetric filters*

Description

Function to retrieve the underlying forecasts corresponding to the asymmetric filters: the asymmetric filtering produce the same result than the symmetric filters applied to the extended series.

Usage

```
underlying_forecasts(x, coefs)
```

Arguments

x a univariate or multivariate time series.

coefs a matrix or a list that contains all the coefficients of the asymmetric and symmetric filters. (from the symmetric filter to the shortest). See details.

Details

Let h be the bandwidth of the symmetric filter, $v_{-h}, \dots, v_h$ the coefficients of the symmetric filter and $w_{-h}^q, \dots, w_h^q$ the coefficients of the asymmetric filter used to estimate the trend when q future values are known (with the convention $w_{q+1}^q = \dots = w_h^q = 0$). Let denote $y_{-2h+1}, \dots, y_0$ the last $2h$ available values of the input times series. The underlying forecast $y_1^*, \dots, y_h^*$ induced by $w^0, \dots, w^{h-1}$ are defined by:

$$\forall q \in \{0, \dots, h-1\}, \quad \sum_{i=-h}^q v_i y_{i-q} + \sum_{i=q+1}^h v_i y_{i-q}^* = \sum_{i=-h}^q w_i^q y_{i-q}.$$

Note that this is solved numerically: the solution isn't exact.

Value

An object of the same class as x (ts, mts, vector or matrix) with the underlying forecasts.

Examples

```
x <- retailsa$AllOtherGenMerchandiseStores

ql <- lp_filter(horizon = 6, kernel = "Henderson", endpoints = "QL")
lc <- lp_filter(horizon = 6, kernel = "Henderson", endpoints = "LC")
f_ql <- underlying_forecasts(x, ql)
f_lc <- underlying_forecasts(x, lc)

graphics::plot(window(x, start = 2007), xlim = c(2007, 2012))
graphics::lines(
  stats::ts(
    c(utils::tail(x, 1), f_ql),
```

```

        frequency = stats::frequency(x),
        start = stats::end(x)
    ),
    col = "red",
    lty = 2
)
graphics::lines(
    stats::ts(
        c(utils::tail(x, 1), f_lc),
        frequency = stats::frequency(x),
        start = stats::end(x)
    ),
    col = "blue",
    lty = 2
)

```

var_estimator

*Variance Estimator***Description**

Variance Estimator

Usage

var_estimator(x, coef, ...)

Arguments

x input time series.

coef vector of coefficients or a moving-average ([moving_average\(\)](#)).

... other arguments passed to the function [moving_average\(\)](#) to convert coef to a "moving_average" object.

Details

Let $(\theta_i)_{-p \leq i \leq q}$ be a moving average of length $p + q + 1$ used to filter a time series $(y_i)_{1 \leq i \leq n}$. It is equivalent to a local regression and the associated error variance σ^2 can be estimated using the normalized residual sum of squares, which can be simplified as:

$$\hat{\sigma}^2 = \frac{1}{n - (p + q)} \sum_{t=p+1}^{n-q} \frac{(y_t - \hat{\mu}_t)^2}{1 - 2w_0^2 + \sum_{i=-p}^q w_i^2}$$

Value

Numeric value with the statistic.

References

Loader, Clive. 1999. Local regression and likelihood. New York: Springer-Verlag.

See Also

[df_var\(\)](#).

Examples

```
x <- retailsa$DrinkingPlaces
h13 <- lp_filter(6)$sfilter
var_estimator(x, h13)
```

Index

```
* datasets  
    retailsa, 37  
*, ANY, finite_filters-method  
    (filters_operations), 12  
*, ANY, moving_average-method  
    (filters_operations), 12  
*, finite_filters, ANY-method  
    (filters_operations), 12  
*, finite_filters, finite_filters-method  
    (filters_operations), 12  
*, finite_filters, moving_average-method  
    (filters_operations), 12  
*, finite_filters, numeric-method  
    (filters_operations), 12  
*, moving_average, ANY-method  
    (filters_operations), 12  
*, moving_average, finite_filters-method  
    (filters_operations), 12  
*, moving_average, moving_average-method  
    (filters_operations), 12  
*, moving_average, numeric-method  
    (filters_operations), 12  
*, numeric, moving_average-method  
    (filters_operations), 12  
+, finite_filters, finite_filters-method  
    (filters_operations), 12  
+, finite_filters, missing-method  
    (filters_operations), 12  
+, finite_filters, moving_average-method  
    (filters_operations), 12  
+, moving_average, finite_filters-method  
    (filters_operations), 12  
+, moving_average, missing-method  
    (filters_operations), 12  
+, moving_average, moving_average-method  
    (filters_operations), 12  
+, moving_average, numeric-method  
    (filters_operations), 12  
+, numeric, finite_filters-method  
    (filters_operations), 12  
+, numeric, moving_average-method  
    (filters_operations), 12  
-, finite_filters, finite_filters-method  
    (filters_operations), 12  
-, finite_filters, missing-method  
    (filters_operations), 12  
-, finite_filters, moving_average-method  
    (filters_operations), 12  
-, finite_filters, numeric-method  
    (filters_operations), 12  
-, moving_average, finite_filters-method  
    (filters_operations), 12  
-, moving_average, missing-method  
    (filters_operations), 12  
-, moving_average, moving_average-method  
    (filters_operations), 12  
-, moving_average, numeric-method  
    (filters_operations), 12  
-, numeric, finite_filters-method  
    (filters_operations), 12  
-, numeric, moving_average-method  
    (filters_operations), 12  
.jd2r_finitefilters, 3  
/, finite_filters, numeric-method  
    (filters_operations), 12  
/, moving_average, numeric-method  
    (filters_operations), 12  
[, finite_filters, ANY-method  
    (filters_operations), 12  
[, finite_filters, missing-method  
    (filters_operations), 12  
[, moving_average, logical-method  
    (filters_operations), 12  
[, moving_average, numeric-method  
    (filters_operations), 12  
[<-, moving_average, ANY, missing, numeric-method  
    (filters_operations), 12  
^, finite_filters, numeric-method
```

- (filters_operations), 12
- ^, moving_average, numeric-method
(filters_operations), 12
- as.matrix.finite_filters
(print, finite_filters-method),
36
- base::print(), 37
- cbind.moving_average
(filters_operations), 12
- coef.moving_average (moving_average), 28
- confint_filter, 3
- cp (diagnostics-fit), 8
- cv (diagnostics-fit), 8
- cve (diagnostics-fit), 8
- df_var, 5
- df_var(), 46
- dfa_filter, 6
- diagnostic_matrix, 7
- diagnostics-fit, 8
- filter, 10, 11
- filters_operations, 12
- finite_filters, 15
- finite_filters(), 4, 6, 11, 12, 15, 19, 22,
23, 26, 31, 37, 38
- fst, 16
- fst_filter, 17
- get_kernel, 19
- get_moving_average, 20
- get_properties_function, 20
- implicit_forecasts, 21
- impute_last_obs, 22
- is.finite_filters (finite_filters), 15
- is.moving_average (moving_average), 28
- is_symmetric (moving_average), 28
- kernel, 19
- length.moving_average (moving_average),
28
- localpolynomials, 24
- localpolynomials(), 26
- loocve (diagnostics-fit), 8
- lower_bound (moving_average), 28
- lp_filter, 25
- lp_filter(), 24, 27
- mirror (moving_average), 28
- mmsre_filter, 26
- mmsre_filter(), 26
- moving_average, 28
- moving_average(), 4, 5, 9, 11, 12, 15, 16, 20,
26, 27, 43, 45
- mse, 7, 31
- plot_coef (plot_filters), 32
- plot_filters, 32
- plot_gain (plot_filters), 32
- plot_phase (plot_filters), 32
- polynomial_matrix, 35
- print, finite_filters-method, 16, 36
- rbind.moving_average
(filters_operations), 12
- retailsa, 37
- rev.moving_average (moving_average), 28
- rkhs_filter, 38
- rkhs_kernel, 39
- rkhs_optimal_bw, 40
- rkhs_optimization_fun, 41
- rt (diagnostics-fit), 8
- show, finite_filters-method
(print, finite_filters-method),
36
- show, moving_average-method
(moving_average), 28
- simple_ma, 43
- sum.moving_average
(filters_operations), 12
- to_seasonal (moving_average), 28
- to_seasonal.finite_filters
(finite_filters), 15
- underlying_forecasts, 44
- upper_bound (moving_average), 28
- var_estimator, 45
- var_estimator(), 4, 5, 10