Package 'onlineforecast'

May 10, 2022

Type Package

Title Forecast Modelling for Online Applications

Version 1.0.1

Description
A framework for fitting adaptive forecasting models. Provides a way to use forecasts as input to models, e.g. weather forecasts for energy related forecasting. The models can be fitted recursively and can easily be setup for updating parameters when new data arrives. See the included vignettes, the website <https://onlineforecasting.org> and the pre-print paper ``onlineforecast: An R package for adaptive and recursive forecasting" <arXiv:2109.12915>.

License GPL-3

Encoding UTF-8

LazyData true

Depends R (>= 3.0.0)

Imports Rcpp (>= 0.12.18), R6 (>= 2.2.2), splines (>= 3.1.1), pbs, digest

LinkingTo Rcpp, RcppArmadillo

Suggests knitr, rmarkdown, R.rsp, testthat (>= 3.0.0), data.table, plotly

VignetteBuilder knitr

RoxygenNote 7.1.2

URL https://onlineforecasting.org

BugReports https://lab.compute.dtu.dk/packages/onlineforecast/-/issues

Config/testthat/edition 3

NeedsCompilation yes

Author Peder Bacher [cre], Hjorleifur G Bergsteinsson [aut]

Maintainer Peder Bacher <pbac@dtu.dk>

Repository CRAN

Date/Publication 2022-05-10 09:30:05 UTC

R topics documented:

`==.data.list` *Determine if two data.lists are identical*

Description

Compare two data.lists

Usage

```
## S3 method for class 'data.list'
x == y
```

Arguments

x	first data.list
y	second data.list

Details

Returns TRUE if the two data.lists are fully identical, so all data, order of variables etc. must be fully identical

Value

logical

Examples

```
Dbuilding == Dbuilding

D <- Dbuilding
D$Ta$k2[1] <- NA
Dbuilding == D

D <- Dbuilding
names(D)[5] <- "I"
names(D)[6] <- "Ta"
Dbuilding == D
```

AR *Auto-Regressive (AR) input*

Description

Generate auto-regressive (AR) inputs in a model

Usage

```
AR(lags)
```

Arguments

lags	integer vector: The lags of the AR to include.

Details

The AR function can be used in an onlineforecast model formulation. It creates the input matrices for including AR inputs in a model during the transformation stage. It takes the values from the model output in the provided data does the needed lagging.

The lags must be given according to the one-step ahead model, e.g.:

`AR(lags=c(0,1))` will give: Y_t+1|t = ϕ_1 y_t-0 + ϕ_2 y_t-1 + ϵ_t+1

and:

`AR(lags=c(0,3,12))` will give: Y_t+1|t = ϕ_1 y_t-0 + ϕ_2 y_t-3 + ϕ_3 y_t-12 + ϵ_t+1

Note, that

For k>1 the coefficients will be fitted individually for each horizon, e.g.:

`AR(lags=c(0,1))` will be the multi-step AR: Y_t+k|t = ϕ_1,k y_t-0 + ϕ_2,k y_t-1 + ϵ_t+k|t

See the details in ??(ref til vignette).

Value

A list of matrices, one for each lag in lags, each with columns according to model$kseq.

Examples

```
# Setup data and a model for the example
D <- Dbuilding
model <- forecastmodel$new()
model$output = "heatload"
# Use the AR in the transformation stage
model$add_inputs(AR = "AR(c(0,1))")
# Regression parameters
model$add_regprm("rls_prm(lambda=0.9)")
# kseq must be added
model$kseq <- 1:4
# In the transformation stage the AR input will be generated
# See that it generates two input matrices, simply with the lagged heat load at t for every k
model$transform_data(subset(D, 1:10))

# Fit with recursive least squares (no parameters prm in the model)
fit <- rls_fit(c(lambda=0.99), model, D, returnanalysis=TRUE)

# Plot the result, see "?plot_ts.rls_fit"
plot_ts(fit, xlim=c(ct("2010-12-20"),max(D$t)))
# Plot for a short period with peaks
plot_ts(fit, xlim=c("2011-01-05","2011-01-07"))

# For online updating, see ??ref{vignette, not yet available}:
# the needed lagged output values are stored in the model for next time new data is available
model$yAR
# The maximum lag needed is also kept
model$maxlagAR
```

---

as.data.frame.data.list

Convert to data.frame

Description

Converts a data.list to a data.frame.

Usage

```
## S3 method for class 'data.list'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

x	The data.list to be converted.
row.names	Not used.
optional	Not used.
...	Not used.

Details

The forecasts in the data.list will result in columns named varname.kxx in the data.frame.

Value

A data.frame

Examples

```
#' # Use the data.list with building heat load
D <- Dbuilding
# Take a subset
D <- subset(D, 1:5, nms=c("t","Taobs","Ta","Iobs","I"), kseq=1:3)

# Convert to a data.frame, note the names of the forecasts are appended .kxx (i.e. for Ta and I)
as.data.frame(D)
```

---

as.data.list *Convert to data.list class*

Description

These functions will convert the object into a data.list.

Convert a data.frame into a data.list

Usage

```
as.data.list(object)

## S3 method for class 'data.frame'
as.data.list(object)
```

Arguments

object	The data.frame to be converted.

Details

A data.list is simply a list of vectors and data.frames. For the use in the onlineforecast package the following format must be kept:

- t: A vector of time.

- vectors with same length as t: Holds observations and values synced to time t.

- data.frames with number of rows as time t: Holds forecasts in each column named by `kxx` where `xx` is the horizon, e.g. `k0` is synced as observations, and `k1` is one-step ahead.

The convention is that columns with forecasts are postfixed with `.kxx` where `xx` is the horizon. See the examples.

Value

a value of class data.list

a data.list

See Also

For specific detailed info see the children, e.g. `as.data.list.data.frame`

as.data.list

Examples

```
# Convert a dataframe with time and two observed variables
X <- data.frame(t=1:10, x=1:10, y=1:10)
as.data.list(X)

# Convert a dataframe with time, forecast and an observed variable
X <- data.frame(t=1:10, x.k1=1:10, x.k2=10:1, yobs=1:10, y.k1=1:10, y.k2=1:10)
as.data.list(X)

# Can be converted back and forth
X
as.data.frame(as.data.list(X))
```

---

aslt *Convertion to POSIXlt*

Description

The argument is converted into POSIXlt with tz="GMT".

Usage

```
aslt(object, ...)

## S3 method for class 'character'
aslt(object, tz = "GMT", ...)

## S3 method for class 'POSIXct'
aslt(object, tz = NA, ...)

## S3 method for class 'POSIXlt'
aslt(object, tz = NA, ...)

## S3 method for class 'numeric'
aslt(object, ...)
```

Arguments

object	The character, POSIXct, POSIClt, or numeric which is converted to POSIXct.
...	Arguments to be passed to methods.
tz	Timezone. If set, then the time zone will be changed of the object.

Value

An object of class POSIXlt

Methods

- aslt.character: Simply a wrapper for `as.POSIXlt`

- aslt.POSIXct: Converts to POSIXct.

- aslt.POSIXlt: Changes the time zone of the object if tz is given.

- aslt.numeric: Converts from UNIX time in seconds to POSIXlt.

Examples

```
# Create a POSIXlt with tz="GMT"
aslt("2019-01-01")
class(aslt("2019-01-01"))
aslt("2019-01-01 01:00:05")

# Convert between time zones
x <- aslt("2019-01-01", tz="CET")
aslt(x,tz="GMT")

# To seconds and back again
aslt(as.numeric(x, units="sec"))
```

bspline *Compute base splines of a variable using the R function* splines::bs*, use in the transform stage.*

Description

Simply wraps the splines::bs, such that it can be used in the transformation stage.

Usage

```
bspline(
  X,
  Boundary.knots = NA,
  intercept = FALSE,
  df = NULL,
  knots = NULL,
  degree = 3,
  bknots = NA,
  periodic = FALSE
)
```

Arguments

X	data.frame (as part of data.list) with horizons as columns named kxx (i.e. one for each horizon)
Boundary.knots	The value is NA: then the boundaries are set to the range of each horizons (columns in X). See ?splines::bs
intercept	See ?splines::bs.
df	See ?splines::bs
knots	See ?splines::bs
degree	See ?splines::bs
bknots	Is just a short for Boundary.knots and replace Boundary.knots (if Boundary.knots is not given)
periodic	Default FALSE. If TRUE, then pbs::pbs is called and periodic splines are generated.

Details

See the help for all arguments with ?splines::bs. NOTE that two arguments have different default values.

See the example https://onlineforecasting.org/examples/solar-power-forecasting.html where the function is used in a model.

Value

List of data frames with the computed base splines, each with columns for the same horizons as in X

See Also

Other Transform stage functions: pbspline()

Examples

```
# How to make a diurnal curve using splines
# Select first 54 hours from the load data
D <- subset(Dbuilding, 1:76, kseq=1:4)
# Make the hour of the day as a forecast input
D$tday <- make_tday(D$t, kseq=1:4)
D$tday

# Calculate the base splines for each column in tday
L <- bspline(D$tday)

# Now L holds a data.frame for each base spline
str(L)
# Hence this will result in four inputs for the regression model

# Plot (note that the splines period starts at tday=0)
plot(D$t, L$bs1$k1, type="s")
for(i in 2:length(L)){
  lines(D$t, L[[i]]$k1, col=i, type="s")
}

# In a model formulation it will be:
model <- forecastmodel$new()
model$add_inputs(mutday = "bspline(tday)")
# We set the horizons (actually not needed for the transform, only required for data checks)
model$kseq <- 1:4
# Such that at the transform stage will give the same as above
model$transform_data(D)

# Periodic splines are useful for modelling a diurnal harmonical functions
L <- bspline(D$tday, bknots=c(0,24), df=4, periodic=TRUE)
# or
L <- pbspline(D$tday, bknots=c(0,24), df=4)
# Note, how it has to have high enough df, else it generates an error

# Plot
plot(D$t, L$bs1$k1, type="s")
for(i in 2:length(L)){
   lines(D$t, L[[i]]$k1, col=i, type="s")
}
```

cache_name *Generation of a name for a cache file for the value of a function.*

Description

Caching of the value returned by a function

Usage

```
cache_name(..., cachedir = "cache")
```

Arguments

`...`	The objects from which to calculate cache file name. If no objects given, then all the objects of the calling function are used for generating the checksum for the file name.
`cachedir`	Path for saving the cache, i.e. prefixed to the generated name, remember to end with '/' to make a directory.

Details

Use it in the beginning of a function, which runs a time consuming calculation, like fitting a model using optimization.

It makes a cache name, which can be used to save a unique cache file (see `cache_save()`).

The `cache_name` function must receive all the objects (in `...`) which influence the value of the function. It simply calculates a checksum using the `digest` package.

Further, it finds the name of the calling function and its definition, such that if anything changes in the function definition, then the cache file name changes too.

Value

A generated cache file name.

Examples

```
# A function for demonstrating the using caching
fun <- function(x, y){
    # Generate the cache name (no argument given, so both x and y is used)
    nm <- cache_name(cachedir=cachedir)
    # If the result is cached, then just return it
    if(file.exists(nm)){ return(readRDS(nm)) }
    # Do the calculation
    res <- x^2 + y + 1
    # Wait 1 sec
    Sys.sleep(1)
    # Save for cache
    cache_save(res, nm)
```

```
    # Return
    return(res)
}

# For this example use a temporary directory
# In real use this should not be temporary! (changes between R sessions with tempdir())
cachedir <- tempdir()

# Uncomment to run:
# First time it takes at least 1 sec.
#fun(x=2,y=2)
# Second time it loads the cache and is much faster
#fun(x=2,y=2)
# Try changing the arguments (x,y) and run again

# See the cache file(s)
#dir(cachedir)
# Delete the cache folder
#unlink(cachedir, recursive=TRUE)

# Demonstrate how cache_name() is functioning
# Cache using the all objects given in the function calling, i.e. both x and y
fun <- function(x,y){
    x^2 + y + 1
    return(cache_name())
}
# These are the same (same values)
fun(x=1,y=2)
fun(1,2)
fun(y=2,x=1)
# But this one is different
fun(x=2,y=1)

# Test: cache using the values specified in the cache_name call
fun2 <- function(x,y){
    x^2 + y + 1
    return(cache_name(x))
}

# So now its only the x value that change the name
fun2(1,2)
fun2(1,3)
# But this one is different
fun2(3,3)
# And the function named changed the name
```

---

cache_save *Save a cache file (name generated with* code_name()

Description

Saves the object as an .RDS file with the filename

Usage

```
cache_save(object, filename)
```

Arguments

object	The object to cache (i.e. the value of the evaluating function).
filename	The cache file name (i.e. use the one generated by cache_name, see examples).

Details

See the examples for `cache_name()`.

---

complete_cases *Find complete cases in forecast matrices*

Description

Returns a logical vector indicating the time points which

Usage

```
complete_cases(object, kseq = NA)

## S3 method for class 'list'
complete_cases(object, kseq = NA)

## S3 method for class 'data.frame'
complete_cases(object, kseq = NA)
```

Arguments

object	A data.frame (with columns named 'kxx') or a list of data.frames.
kseq	integer vector: If given then only these horizons are processed.

Details

Given a forecast matrix the forecasts are lagged "+k" steps to align them and then 'complete.cases()' is run on that .

Gieven a list of forecast matrices the points where all are complete (also all horizons) are complete are TRUE.

Value

A logical vector specifying if there is no missing values across all horizonsd.

Author(s)

Peder Bacher

Examples

```
# Take a small data set
D <- subset(Dbuilding, 1:20, kseq=1:5)
# Check the forecast matrix of ambient temperature
D$Ta
# Which are complete over all horizons? The first are not since not all horizons
# have a value there (after lagging)
complete_cases(D$Ta)
# Same goes if given as a list
complete_cases(D["Ta"])
# and if more than one is given
complete_cases(D[c("Ta","I")])

# Set some NA of some horizon
D$I$k3[8:9] <- NA
# Now they are recognized as not complete
complete_cases(D[c("Ta","I")])

# If we deal with residuals, which are observations and there for have column names "hxx"
Resid <- residuals(D$Ta, D$Taobs)
names(Resid)
# With columns with "h" instead of "k" no lagging occurs in complete_cases
complete_cases(Resid)
#
Resid2 <- Resid
Resid$h3[8:9] <- NA
complete_cases(list(Resid,Resid2))
```

---

ct *Convertion to POSIXct*

Description

The object is converted into POSIXct with tz="GMT".

Usage

```
ct(object, ...)

## S3 method for class 'character'
```

```
ct(object, tz = "GMT", ...)

## S3 method for class 'POSIXct'
ct(object, tz = NA, duplicatedadd = NA, ...)

## S3 method for class 'POSIXlt'
ct(object, tz = NA, duplicatedadd = NA, ...)

## S3 method for class 'numeric'
ct(object, ...)
```

Arguments

`object`	The object to convert can be: character, numeric, POSIXct or POSIXlt
`...`	Arguments to be passed to methods.
`tz`	Timezone. If set, then the time zone will be changed of the object.
`duplicatedadd`	Seconds to be added to duplicated time stamps, to mitigate the problem of duplicated timestamps at the shift to winter time. So the second time a time stamp occurs (identified with `duplicated`) then the seconds will be added.

Details

A simple helper, which wraps `as.POSIXct`‘ and sets the time zone to "GMT" per default.

Value

An object of class POSIXct

Methods

- ct.character: Simply a wrapper for `as.POSIXct` with default `tz`

- ct.POSIXct: Changes the time zone of the object if `tz` is given.

- ct.POSIXlt: Converts to POSIXct.

- ct.numeric: Converts from UNIX time in seconds to POSIXct with `tz` as GMT.

Examples

```
# Create a POSIXct with tz="GMT"
ct("2019-01-01")
class(ct("2019-01-01"))
ct("2019-01-01 01:00:05")


# Convert to POSIXct
class(ct(as.POSIXlt("2019-01-01")))

# To seconds and back again
```

```
ct(as.numeric(1000, units="sec"))


# --------
# Convert character of time which has summer time leaps
# Example from CET (with CEST which is winter time)
#
# The point of shifting to and from summer time:
# DST Start (Clock Forward) DST End (Clock Backward)
# Sunday, March 31, 02:00 Sunday, October 27, 03:00

# --------
# From to winter time to summer time
txt <- c("2019-03-31 01:00",
         "2019-03-31 01:30",
         "2019-03-31 03:00",
         "2019-03-31 03:30")
x <- ct(txt, tz="CET")
x
ct(x, tz="GMT")

# BE AWARE of this conversion of the 02:00: to 02:59:59 (exact time of shift) will lead to a
# wrong conversion
txt <- c("2019-03-31 01:30",
         "2019-03-31 02:00",
         "2019-03-31 03:30")
x <- ct(txt, tz="CET")
x
ct(x, tz="GMT")
# Which a diff on the time can detect, since all steps are not equal
plot(diff(ct(x, tz="GMT")))

# --------
# Shift to winter time is more problematic
# It works like this
txt <- c("2019-10-27 01:30",
         "2019-10-27 02:00",
         "2019-10-27 02:30",
         "2019-10-27 03:00",
         "2019-10-27 03:30")
x <- ct(txt, tz="CET")
x
ct(x, tz="GMT")

# however, timestamps can be given like this
txt <- c("2019-10-27 01:30",
         "2019-10-27 02:00",
         "2019-10-27 02:30",
         "2019-10-27 02:00",
         "2019-10-27 02:30",
         "2019-10-27 03:00",
         "2019-10-27 03:30")
x <- ct(txt, tz="CET")
```

```
x
ct(x, tz="GMT")
# Again can be detected, since all steps are not equal
plot(diff(ct(x, tz="GMT")))
# This can be fixed by (note that it can go wrong, e.g. with gaps around convertion etc.)
ct(x, tz="GMT", duplicatedadd=3600)
```

`data.list` *Make a data.list*

Description

Make a data.list of the vectors and data.frames given.

Usage

```
data.list(...)
```

Arguments

`...`	Should hold: time t, observations as vectors and forecasts as data.frames

Details

See the vignette 'setup-data' on how a data.list must be setup.

It's simply a list of class `data.list` holding:

- vector `t`

- vector(s) of observations

- data.frames (or matrices) of forecast inputs

Value

a data.list.

Examples

```
# Put together a data.list
# The time vector
time <- seq(ct("2019-01-01"),ct("2019-01-02"),by=3600)
# Observations time series (as vector)
xobs <- rnorm(length(time))
# Forecast input as a data.frame with columns names 'kxx', where 'xx' is the horizon
X <- data.frame(matrix(rnorm(length(time)*3), ncol=3))
names(X) <- pst("k",1:3)

D <- data.list(t=time, xobs=xobs, X=X)
```

```
# Check it (see \code{?\link{summary.data.list}})
summary(D)
```

`Dbuilding` *Observations and weather forecasts from a single-family building, weather station and Danish Meteorological Institute (DMI)*

Description

Data of the period from 2010-12-15 to 2011-03-01. The weather station was located within a range of 10 km from the building.

Usage

```
Dbuilding
```

Format

A data list with 1854 rows and 7 variables:

t Time in GMT as POSIXct

heatload The heatload of a single family building in W

heatloadtotal The average heatload of a 16 single family buildings in W

Taobs Observed ambient temperature at the weather station in Celcius

Iobs Observed global radiation at the weather station in W/m^2

Ta Weather forecasts of ambient temperature up to 36 hours ahead from DMI in Celcius

Ta Weather forecasts of global radiation up to 36 hours ahead from DMI in W/m^2

Details

Hourly average values. The time point is set in the end of the hour.

Set in the format of a data.list used as input to forecast models in the onlineforecast package.

Source

See https://onlineforecasting.org/examples/datasets.html.

depth *Depth of a list*

Description

Depth of a list

Usage

```
depth(this)
```

Arguments

this	list

Details

Returns the depth of a list

Value

integer

flattenlist *Flattens list*

Description

Flattens list in a single list of data.frames

Usage

```
flattenlist(x)
```

Arguments

x	List to flatten.

Details

Flattens list. Can maybe be made better. It might end up copying data in memory!? It might change the order of the elements.

Value

A flatten list

`forecastmodel` *Class for forecastmodels*

Description

R6 class for a forecastmodel

Details

This class holds the variables and functions needed for defining and setting up a forecast model - independent of the fitting scheme. See the vignettes on how to setup and use a model and the website https://onlineforecasting.org for more info.

Holds all the information needed independently of the fitting scheme (e.g. lm_fit or rls_fit), see the fields and functions below.

The fields are separated into: - Fields for setting up the model - Fields used when fitting (e.g. which horizons to fit for is set in `kseq`

See the fields description below.

Note, it's an R6 class, hence an object variable is a pointer (reference), which means two important points: - In order to make a copy, the function clone_deep() must be used (usually `clone(deep=TRUE)`, but that will end in an infinite loop). - It can be manimulated directly in functions (without return). The code is written such that no external functions manipulate the model object, except for online updating.

For online updating (i.e. receiving new data and updating the fit), then the model definition and the data becomes entangled, since transformation functions like low-pass filtering with `lp()` requires past values. See the vignette ??(ref to online vignette, not yet available) and note that `rls_fit()` resets the state, which is also done in all `xxx_fit` functions (e.g. `rls_fit`.

Public fields used for setting up the model

- output = NA, character: Name of the output.

- inputs = list(), add them with add_inputs(): List of inputs (which are R6 objects) (note the "cloning of list of reference objects" issue below in deep_clone function)

- regprmexpr = NA: The expression (as character) used for generating the regprm, e.g. "`rls_prm()`" for RLS.

- regprm = list(): Regression parameters calculated by evaluating the `regprmexpr`.

- prmbounds = as.matrix(data.frame(lower=NA, init=NA, upper=NA)): The bounds for optimization of the parameters, e.g. with `rls_optim()`.

- outputrange = NA, numeric vector of length 2: Limits of the predictions cropped in the range, e.g. outputrange = c(0,Inf) removes all negative output predictions.

Public fields used when the model is fitted

- kseq = NA: The horizons to fit for.
- kseqopt = NA: The horizons to fit for when optimizing.
- p = NA: The (transformation stage) parameters used for the fit.
- Lfits = list(): The regression fits, one for each k in kseq (simply a list with the latest fit).
- datatr = NA: Transformed input data (data.list with all inputs for regression)

Public methods

All public functions are described below and in examples a section for each is included:

`$new()`

Create a new 'forecastmodel' object.

Returns a forecastmodel object.

`$add_inputs(...)`

Add inputs to the model.

- `...`: The inputs are given as arguments, see examples.

`$add_regprm(regprm_expr)`

Add expression (as character) which generates regression parameters.

`$add_prmbounds(...)`

Add the transformation parameters and bounds for optimization.

`$get_prmbounds(...)`

Get the transformation parameter bounds, used by optimization functions e.g. `rls_optim()`.

`$insert_prm(prm)`

Insert the transformation parameters prm in the input expressions and regression expressions, and keep them in $prm (simply string manipulation).

`$transform_data(data)`

Function for transforming the input data to the regression stage input data (see `vignette("setup-data", package = "onlineforecast")`).

`$reset_state()`

Resets the input states and stored data for iterative fitting (datatr rows and yAR) (see ??(ref to online updating vignette, not yet available).

`$check(data = NA)`

Check if the model is setup correctly.

Examples

```
# New object
model <- forecastmodel$new()

# Print it
model


# Add model inputs
model$add_inputs(Ta = "lp(Ta)")
# See it
model$inputs
# Update to use no low-pass filter
model$add_inputs(Ta = "Ta")
model$inputs
# Add another
model$add_inputs(I = "lp(I)")
model$inputs

# Simply a list, so manipulate directly
class(model$inputs$Ta)
model$inputs$Ta$expr <- "lp(Ta, a1=0.9)"

# Add the parameters for the regression stage
model$add_regprm("rls_prm(lambda=0.99)")
# The evaluation is a list, which is set in
model$regprm


# Set the lambda to be optimized between 0.9 and 0.999, starting at 0.99
model$add_prmbounds(lambda = c(0.9, 0.99, 0.999))
# Note the "__" syntax to set parameters for inputs: "input__prm"
model$add_prmbounds(Ta__a1 = c(0.8, 0.95, 0.99))

# Get the lower bounds
model$get_prmbounds("lower")

# Insert the init parameters
prm <- model$get_prmbounds("init")
prm
# Before
model$inputs$Ta$expr
# After
model$insert_prm(prm)
model$inputs$Ta$expr

# Check if the model is setup and can be used with a given data.list
# An error is thrown
```

```
try(model$check(Dbuilding))
# Add the model output
model$output <- "heatload"
# Still not error free
try(model$check(Dbuilding))
# Add the horizons to fit for
model$kseq <- 1:4
# Finally, no errors :)
model$check(Dbuilding)
```

`fs` *Generation of Fourrier series.*

Description

Function for generating Fourrier series as a function of x E.g. use for harmonic functions for modelling the diurnal patterns or for basis functions.

Usage

```
fs(X, nharmonics)
```

Arguments

`X`	must be a dataframe with columns k1,k2,..., . One period is from 0 to 1 (so for example if X is hour of day, then divide X by 24 to obtain a daily period).
`nharmonics`	the number of harmonics, so creates double as many inputs! i.e. one sine and one cos for each harmonic.

Value

Returns a list of dataframes (two for each i in `1:nharmonics`) with same number of columns as X.

Examples

```
# Make a data.frame with time of day in hours for different horizons
tday <- make_tday(seq(ct("2019-01-01"), ct("2019-01-04"), by=3600), kseq=1:5)
# See whats in it
str(tday)
head(tday)

# Now use the function to generate Fourier series
L <- fs(tday/24, nharmonics=2)
# See what is in it
str(L)

# Make a plot to see the harmonics
par(mfrow=c(2,1))
# The first harmonic
```

```
plot(L$sin1$k1, type="l")
lines(L$cos1$k1, type="l")
# The second harmonic
plot(L$sin2$k1, type="l")
lines(L$cos2$k1, type="l")
```

---

getse *Getting subelement from list.*

Description

A helping function for getting subelemlts from a list.

Usage

```
getse(L, inm = NA, depth = 2, useregex = FALSE, fun = NA)
```

Arguments

L	The list to get sub elements from.
inm	Either an integer index or a name of the subelements to return.
depth	The depth of the subelements to match names in: - 1: is directly in the list. - 2: is in list of each element in the list. - 3 and more: simply deeper in the sublists.
useregex	logical: should inm be used as regex pattern for returning elements matching, in the specified layer.
fun	function: if given, then it will be applied to all the matched subelements before returning them.

Details

Often it is needed to get a subelement from a list, which can be done using lapply. But to make life easiere here is a small function for getting subelements in a nested list at a certain debth.

Value

A list of the matched elements.

Examples

```
# Make a nested list
L <- list(x1=list(x=list("val11","val112"),
                  y=list("val12"),
                  test=list("testlist2")),
          x2=list(x=list("val21","val212"),
                  y=list("val22"),
                  test=list("testlist2")),
```

```
            x3=list(x=list("val31","val312"),
                    y=list("val32"),
                    test=list("testlist3")))

# Get the subelement "x1"
str(getse(L, "x1", depth=1))
# Same as
str(L[["x1"]])

# Get the element named x in second layer
str(getse(L, "x", depth=2))
# Depth is default to 2
str(getse(L, "y"))

# Nice when splitting string
x <- strsplit(c("x.k1","y.k2"), "\\.")
# Get all before the split "\."
getse(x, 1)
# Get after
getse(x, 2)

# Get an element with an integer index
x <- strsplit(c("x.k1","y.k2","x2"), "\\.")
getse(x, 1)
# if the element is not there, then an error is thrown
try(getse(x, 2))

# Use regex pattern for returning elements matching in the specified layer
getse(L, "^te", depth=2, useregex=TRUE)
```

---

gof *Simple wrapper for graphics.off()*

Description

Simple wrapper for graphics.off()

Usage

```
gof()
```

`input_class` *Class for forecastmodel inputs*

Description

R6 class for for forecastmodel inputs

Details

Holds variables and functions needed for an input, as added by `forecastmodel$add_inputs()`.

Details of the class.

Public fields

- expr = NA: The expression as string for transforming the input.

- state_L = list(): The list holding potential state values kept by the function evaluated in the expression.

- state_i = integer(1): index counter for the state list.

Public methods

All public functions are described below and in examples a section for each is included:

`$new(expr)`

Create a new input with the expression `expr`.

`$evaluate(data`

Generate (transform) the input by evaluating the expr with the `data` (data.list) attached.

`$state_reset()`

Each function in the expressions (lp, fs, etc.) have the possibility to save a state, which can be read next time the are called.

Reset the state by deleting `state_L` and setting `state_i` to 0.

After running model$inputs[[1]]$evaluate(D) # the lp() has saved it's state for next time model$inputs[[1]]$state_L # New data arrives Dnew <- subset(Dbuilding, 11, kseq=1:4) # So in lp() the state is read and it continues model$inputs[[1]]$evaluate(Dnew)

If we want to reset the state, which is done in all _fit() functions (e.g. rls_fit), such that all transformations starts from scratch # Reset the state model$inputs[[1]]$evaluate(D) # Test resetting model$inputs[[1]]$state_reset() # Now there is no state model$inputs[[1]]$evaluate(Dnew) # So lp() starts by taking the first data point Dnew$Ta

`$state_getval(initval)`

Get the saved value in state. This function can be used in the beginning of transformation functions to get the current state value. First time called return the `initval`.

Note that since all transformation functions are called in the same order, then the state can be read and saved by keeping a counter internally, the value is saved in the field $state_i.

See example of use in `lp()`.

`$state_setval(val)`

Set the state value, done in the end of a transformation function, see above.

See example of use in `lp()`.

Examples

```
# new:

# An input is created in a forecastmodel
model <- forecastmodel$new()
model$add_inputs(Ta = "lp(Ta, a1=0.9)")
# The 'inputs' is now a list
model$inputs
# With the input object
class(model$inputs[[1]])

# Now the transformation stage can be carried out to create the regression stage data
# Take a data.list subset for the example
D <- subset(Dbuilding, 1:10, kseq=1:4)
# Transform the data
model$inputs[[1]]$evaluate(D)
# What happens is simply that the expression is evaluated with the data
# (Note that since not done in the model some functions are missing)
eval(parse(text=model$inputs[[1]]$expr), D)
```

---

in_range *Selects a period*

Description

Returns a logical vector of boolean values where TRUE indicates if timestamp is within the specified period.

Usage

```
in_range(tstart, time, tend = NA)
```

Arguments

tstart	The start of the period.
time	The timestamps as POSIX.
tend	The end of the period. If not given then the period will have no end.

Details

Returns a logical vector of boolean values where TRUE indicates if timestamp is within the specified period spanned by tstart and tend.

Note the convention of time stamp in the end of the time intervals causes the time point which equals `tstart` not to be included. See last example.

The times can be given as character or POSIX, per default in tz='GMT'.

Value

A logical vector indicating the selected period with TRUE

Examples

```
# Take a subset
D <- subset(Dbuilding, c("2010-12-15", "2011-01-01"))

# Just a logical returning TRUE in a specified period
in_range("2010-12-20", D$t, "2010-12-22")

# Set which period to evaluate when optimizing parameters, like in rls_optim()
# (the points with scoreperiod == false are not included in the score evaluation)
D$scoreperiod <- in_range("2010-12-20", D$t)
D$scoreperiod

# Further, excluding a small period by
D$scoreperiod[in_range("2010-12-26", D$t, "2010-12-27")] <- FALSE
D$scoreperiod

# Note the convention of time stamp in the end of the time intervals
# causes the point with t = 2010-12-26 00:00:00 not to be included
# since it's covering to "2010-12-25 23:00:00" to "2010-12-26 00:00:00"
D$t[in_range("2010-12-26", D$t, "2010-12-27")]

# When characters are given, then they are translated to the time zone of the time vector
D <- subset(Dbuilding, c("2010-12-15", "2010-12-16"))
D$t <- ct(D$t, tz="CET")
D$t[in_range("2010-12-15 02:00", D$t, "2010-12-15 05:00")]
```

lagdf *Lagging which returns a data.frame*

Description

Lagging by shifting the values back or fourth always returning a data.frame.

Lagging of a data.frame

Usage

```
lagdf(x, lagseq)

## S3 method for class 'data.frame'
lagdf(x, lagseq)
```

Arguments

x	The data.frame to have columns lagged
lagseq	The sequence of lags as an integer. Alternatively, as a character "+k", "-k", "+h" or "-h", e.g. "k12" will with "+k" be lagged 12.

Details

This function lags (shifts) the values of the vector. A data.frame is always returned with the columns as the vectors lagged with the values in lagseq. The column names are set to "kxx", where xx are the lag of the column.

This function lags the columns with the integer values specified with the argument `lagseq`.

Value

A data.frame.

A data.frame with columns that are lagged

See Also

`lagdf.data.frame` which is run when x is a data.frame.

Examples

```
# The values are simply shifted
# Ahead in time
lagdf(1:10, 3)
# Back in time
lagdf(1:10, -3)
# Works but returns a numeric column
lagdf(as.factor(1:10), 3)
# Works and returns a character column
```

```
lagdf(as.character(1:10), 3)
# Giving several lag values
lagdf(1:10, c(1:3))
lagdf(1:10, c(5,3,-1))

# See also how to lag a forecast data.frame with: ?lagdf.data.frame



# dataframe of forecasts
X <- data.frame(k1=1:10, k2=1:10, k3=1:10)
X

# Lag all columns
lagdf(X, 1)


# Lag each column different steps
lagdf(X, 1:3)
# Lag each columns with its k value from the column name
lagdf(X, "+k")

# Also works for columns named hxx
names(X) <- gsub("k", "h", names(X))
lagdf(X, "-h")

# If lagseq must have length as columns in X, it doesn't know how to lag and an error is thrown
try(lagdf(X, 1:2))
```

`lagdl` *Lagging which returns a data.list*

Description

Lagging by shifting the values back or fourth always returning a data.list.

Usage

```
lagdl(DL, lagseq)
```

Arguments

`DL`	The data.list to be lagged.
`lagseq`	The integer(s) setting the lag steps.

Details

This function lags (shifts) the values of the vector. A data.list is always returned with each data.frame lagged with `lagdf`.

Value

A data.list.

Examples

```
# The values are simply shifted in each data.frame with lagdf
```

`lagvec` *Lag by shifting*

Description

Lag by shifting the vecter

Usage

```
lagvec(x, lag)
```

Arguments

`x`	The vector to lag
`lag`	(integer) The steps to lag.

Details

A positive value of `lag` shifts the values to the right in the vector.

Value

The shifted vector

Examples

```
# The values are simply shifted
# Ahead in time
lagvec(1:10, 3)
# Back in time
lagvec(1:10, -3)
# Works but returns a numric
lagvec(as.factor(1:10), 3)
# Works and returns a character
lagvec(as.character(1:10), 3)
```

lapply_cbind *Helper which does lapply and then cbind*

Description

Helper which does lapply and then cbind

Usage

```
lapply_cbind(X, FUN, ...)
```

Arguments

X	object to apply on
FUN	function to apply
...	passed on to lapply

lapply_cbind_df *Helper which does lapply, cbind and then as.data.frame*

Description

Helper which does lapply, cbind and then as.data.frame

Usage

```
lapply_cbind_df(X, FUN, ...)
```

Arguments

X	object to apply on
FUN	function to apply
...	passed on to lapply

lapply_rbind *Helper which does lapply and then rbind*

Description

Helper which does lapply and then rbind

Usage

```
lapply_rbind(X, FUN, ...)
```

Arguments

X	object to apply on
FUN	function to apply
...	passed on to lapply

lapply_rbind_df *Helper which does lapply, rbind and then as.data.frame*

Description

Helper which does lapply, rbind and then as.data.frame

Usage

```
lapply_rbind_df(X, FUN, ...)
```

Arguments

X	object to apply on
FUN	function to apply
...	passed on to lapply

`lm_fit` *Fit an onlineforecast model with* `lm`

Description

Fit a linear regression model given a onlineforecast model, seperately for each prediction horizon

Usage

```
lm_fit(
  prm = NA,
  model,
  data,
  scorefun = NA,
  returnanalysis = TRUE,
  printout = TRUE
)
```

Arguments

`prm`	as numeric with the parameters to be used when fitting.
`model`	object of class forecastmodel with the model to be fitted.
`data`	as data.list with the data to fit the model on.
`scorefun`	Optional. If scorefun is given, e.g. `rmse`, then the value of this is also returned.
`returnanalysis`	as logical determining if the analysis should be returned. See below.
`printout`	Defaults to TRUE. Prints the parameters for model.

Value

Depends on:

- If `returnanalysis` is TRUE a list containing:

* `Yhat`: data.frame with forecasts for `model$kseq` horizons.

* `model`: The forecastmodel object cloned deep, so can be modified without changing the original object.

* `data`: data.list with the data used, see examples on how to obtain the transformed data.

* `Lfitval`: a character "Find the fits in model$Lfits", it's a list with the lm fits for each horizon.

* `scoreval`: data.frame with the scorefun result on each horizon (only scoreperiod is included).

- If `returnanalysis` is FALSE (and `scorefun` is given): The sum of the score function on all horizons (specified with model$kseq).

Examples

```
# Take data
D <- subset(Dbuilding, c("2010-12-15", "2011-01-01"))
D$y <- D$heatload
# Define a simple model
model <- forecastmodel$new()
model$output <- "y"
model$add_inputs(Ta = "lp(Ta, a1=0.9)",
                 mu = "one()")

# Before fitting the model, define which points to include in the evaluation of the score function
D$scoreperiod <- in_range("2010-12-20", D$t)
# And the sequence of horizons to fit for
model$kseq <- 1:6

# Now we can fit the model with RLS and get the model validation analysis data
fit <- lm_fit(prm=NA, model=model, data=D)
# What did we get back?
names(fit)
class(fit)
# The one-step forecast
plot(D$y, type="l")
lines(lagvec(fit$Yhat$k1,-1), col=2)
# Get the residuals
plot(residuals(fit)$h1)
# Score for each horizon
score(residuals(fit))

# The lm_fit don't put anything in this field
fit$Lfitval
# Find the lm fits here
model$Lfits
# See result for k=1 horizon
summary(model$Lfits$k1)
# Some diurnal pattern is present
acf(residuals(fit)$h1, na.action=na.pass, lag.max=96)

# Run with other parameters and return the RMSE
lm_fit(c(Ta__a1=0.8), model, D, scorefun=rmse, returnanalysis=FALSE)
lm_fit(c(Ta__a1=0.9), model, D, scorefun=rmse, returnanalysis=FALSE)
```

`lm_optim` *Optimize parameters for onlineforecast model fitted with LM*

Description

Optimize parameters (transformation stage) of LM model

Usage

```
lm_optim(
  model,
  data,
  kseq = NA,
  scorefun = rmse,
  cachedir = "",
  cachererun = FALSE,
  printout = TRUE,
  method = "L-BFGS-B",
  ...
)
```

Arguments

model	The onlineforecast model, including inputs, output, kseq, p
data	The data.list including the variables used in the model.
kseq	The horizons to fit for (if not set, then model$kseq is used)
scorefun	The function to be score used for calculating the score to be optimized.
cachedir	A character specifying the path (and prefix) of the cache file name. If set to "", then no cache will be loaded or written. See https://onlineforecasting.org/vignettes/nice-tricks.html for examples.
cachererun	A logical controlling whether to run the optimization even if the cache exists.
printout	A logical determining if the score function is printed out in each iteration of the optimization.
method	The method argument for optim.
...	Additional parameters to optim

Details

This is a wrapper for optim to enable easy use of bounds and caching in the optimization.

Value

Result object of optim(). Parameters resulting from the optimization can be found from result$par

See Also

link{optim} for how to control the optimization and rls_optim which works very similarly.

Examples

```
# Take data
D <- subset(Dbuilding, c("2010-12-15", "2011-01-01"))
D$y <- D$heatload
# Define a simple model
```

```
model <- forecastmodel$new()
model$add_inputs(Ta = "lp(Ta, a1=0.9)",
                 mu = "one()")
# Before fitting the model, define which points to include in the evaluation of the score function
D$scoreperiod <- in_range("2010-12-20", D$t)
# And the sequence of horizons to fit for
model$kseq <- 1:6

# Now we can fit the model and get the score, as it is
lm_fit(model=model, data=D, scorefun=rmse, returnanalysis=FALSE)
# Or we can change the low-pass filter coefficient
lm_fit(c(Ta__a1=0.99), model, D, rmse, returnanalysis=FALSE)

# This could be passed to optim() (or any optimizer).
# See \code{forecastmodel$insert_prm()} for more details.
optim(c(Ta__a1=0.98), lm_fit, model=model, data=D, scorefun=rmse, returnanalysis=FALSE,
      lower=c(Ta__a1=0.4), upper=c(Ta__a1=0.999), method="L-BFGS-B")

# lm_optim is simply a helper it makes using bounds easiere and enables caching of the results
# First add bounds for lambda (lower, init, upper)
model$add_prmbounds(Ta__a1 = c(0.4, 0.98, 0.999))

# Now the same optimization as above can be done by
val <- lm_optim(model, D)
val
```

`lm_predict` *Prediction with an lm forecast model.*

Description

Use a fitted forecast model to predict its output variable with transformed data.

Usage

```
lm_predict(model, datatr)
```

Arguments

`model`	Onlineforecast model object which has been fitted.
`datatr`	Transformed data.

Details

See the ??ref(recursive updating vignette, not yet available).

Value

The Yhat forecast matrix with a forecast for each model$kseq and for each time point in `datatr$t`.

Examples

```
# Take data
D <- subset(Dbuilding, c("2010-12-15", "2011-01-01"))
D$y <- D$heatload
# Define a model
model <- forecastmodel$new()
model$add_inputs(Ta = "lp(Ta, a1=0.7)", mu = "one()")

# Before fitting the model, define which points to include in the evaluation of the score function
D$scoreperiod <- in_range("2010-12-20", D$t)
# And the sequence of horizons to fit for
model$kseq <- 1:6

# Transform using the mdoel
datatr <- model$transform_data(D)

# See the transformed data
str(datatr)

# The model has not been fitted
model$Lfits

# To fit
lm_fit(model=model, data=D)

# Now the fits for each horizon are there (the latest update)
# For example
summary(model$Lfits$k1)

# Use the fit for prediction
D$Yhat <- lm_predict(model, datatr)

# Plot it
plot_ts(D, c("y|Yhat"), kseq=1)
```

`long_format` *Long format of prediction data.frame*

Description

Creates a long format of the predictions

Usage

```
long_format(fit, Time = NULL)
```

Arguments

fit	The result from either lm_fit or rls_fit
Time	If the timestamps are missing from the fit object

Details

This functions creates a useful prediction data.frame which can be useful for analysis and plotting.

Value

Data.frame of when the prediction where made, also the prediction value and timestamp.

Examples

```
# Take data
D <- subset(Dbuilding, c("2010-12-15", "2011-01-01"))
D$y <- D$heatload
D$scoreperiod <- in_range("2010-12-20", D$t)
# Define a model
model <- forecastmodel$new()
model$add_inputs(Ta = "Ta",
                 mu = "one()")
model$add_regprm("rls_prm(lambda=0.99)")
model$kseq <- 1:6
# Fit it
fit <- rls_fit(prm=c(lambda=0.99), model, D)

# Get the forecasts (in fit$Yhat) on long format
long_format(fit)
```

---

lp *First-order low-pass filtering*

Description

First-order low-pass filtering of a time series vector.

Usage

```
lp(X, a1, usestate = TRUE)
```

Arguments

X	Dataframe or matrix (or list of them) of forecasts in columns to be low-pass filtered.
a1	The low-pass filter coefficient.
usestate	logical: Use the state kept in the model$input? if lp() is used outside model$transform_data(), then it must be set to FALSE, otherwise the input$state (which is not there) will be read leading to an error.

Details

This function applies a first order unity gain low-pass filter to the columns of X. The low-pass filter is applied to each column seperately. The stationary gain of the filter i one.

If a list of dataframes (or matrices) is given, then the low-pass filtering is recursively applied on each.

Value

The low-pass filtered dataframe (as a matrix)

Examples

```
# Make a dataframe for the examples
X <- data.frame(k1=rep(c(0,1),each=5))
X$k2 <- X$k1
Xf <- lp(X, 0.5, usestate=FALSE)
Xf

# See the input and the low-pass filtered result
plot(X$k1)
lines(Xf[ ,"k1"])
# Slower response with higher a1 value
lines(lp(X, 0.8, usestate=FALSE)[ ,"k1"])

# If a list of dataframes is given, then lp() is recursively applied on each
lp(list(X,X), 0.5, usestate=FALSE)
```

---

lp_vector_cpp *Low pass filtering of a vector.*

Description

This function returns a vector which is x through a unity gain first-order low-pass filter.

Arguments

x	A numeric vector
a1	the first order low-pass filter coefficient

`make_input` *Make a forecast matrix (as data.frame) from observations.*

Description

This function creates a data.frame with columns for each horizon such that it can be added to a data.list and used in a forecast model.

Usage

```
make_input(observations, kseq)
```

Arguments

observations	vector of observations.
kseq	vector of integers, respresenting the desired "k-steps ahead".

Value

Returns a forecast matrix (as a data.frame) with simply the observation vector copied to each column.

Examples

```
# Data for example
D <- subset(Dbuilding, c("2010-12-15","2010-12-20"))

# Generate the input
make_input(D$heatload, 1:4)

# Set is in D, such that it can be used in input expressions (e.g. by model$add_inputs(AR = "Ar0")
D$AR0 <- make_input(D$heatload, 1:36)
```

make_periodic *Make an forecast matrix with a periodic time signal.*

Description

This function creates a data.frame with k-steps-ahead values of a periodic time signal, such that it can be added to a data.list and used inputs to a forecast model.

Usage

```
make_periodic(time, kseq, period, offset = 0, tstep = NA)
```

Arguments

time	vector of times of class "POSIXct" "POSIXt".
kseq	vector of integers, representing the desired "k-steps ahead".
period	a numeric setting the length of the period in seconds.
offset	a numeric setting an offset in the period start in seconds.
tstep	step time of k in seconds.

Value

Returns a forecast matrix (data.frame) with rownames = times, colnames = k1, k2, k3, ... The content of the data frame is the hour of day.

See Also

make_tday

Examples

```
# Create a time sequence of 30 min sample period
tseq <- seq(ct("2019-01-01"), ct("2019-02-01 12:00"), by=1800)

# Make the three hourly periodic sequence
make_periodic(tseq, 1:10, 3*3600)

# With an offset of one hour
make_periodic(tseq, 1:10, 3*3600, 3600)
```

make_tday *Make an hour-of-day forecast matrix*

Description

This function creates a data.frame with k-steps-ahead values of hour of day, such that it can be added to a data.list and used inputs to a forecast model.

Usage

```
make_tday(time, kseq, tstep = NA)
```

Arguments

time	vector of times of class "POSIXct" "POSIXt".
kseq	vector of integers, representing the desired "k-steps ahead".
tstep	step time of k in seconds.

Value

Returns a forecast matrix (data.frame) with rownames = times, colnames = k1, k2, k3, ... The content of the data frame is the hour of day.

See Also

make_periodic

Examples

```
# Create a time sequence of 30 min sample period
tseq <- seq(ct("2019-01-01"), ct("2019-02-01 12:00"), by=1800)

# Make the time of day sequence (assuming time between k steps is same as for tseq)
make_tday(tseq, 1:10)

# With 0.5 hour steps, kstep in hours
make_tday(tseq, 1:10, tstep=3600)
```

nams *Return the column names*

Description

Return the column names of a dataframe or a matrix.

Usage

```
nams(x)

nams(x) <- value
```

Arguments

x	The matrix or data.frame to set the column names for.
value	The names to be given.

Details

Simply to have a single function for returning the column names, instead of `colnames()` for a `matrix` and `names()` for a `data.frame`).

Examples

```
# Generate a matrix
X <- matrix(1, nrow=2, ncol=3)
colnames(X) <- c("c1","c2","c3")
D <- as.data.frame(X)

# Annoyingly this fails (for a matrix)
## Not run: names(X)
# Could use this everywhere
colnames(D)
# but this is shorter
nams(X)
nams(D)

# Also for assignment
nams(D) <- c("x1","x2","x3")
nams(D)
```

one *Create ones for model input intercept*

Description

Returns a data.frame of ones which can be used in forecast model inputs

Usage

```
one()
```

Details

The function returns ones which can be used to generate ones, e.g. to be used as a intercept for a model.

See vignettes 'setup-data' and 'setup-and-use-model'.

Value

A data.frame of ones

Examples

```
# A model
model <- forecastmodel$new()
# Use the function in the input definition
model$add_inputs(mu = "one()")
# Set the forecast horizons
model$kseq <- 1:4
# During the transformation stage the one will be generated for the horizons
model$transform_data(subset(Dbuilding, 1:7))
```

onlineforecast *Functions for online forecasting*

Description

Functions for online forecasting

Details

This package provides functions to for setting up forecast models which run in an online setting, e.g. like demand, solar and wind power forecasts updated regularly - often hourly, and forecasts up to 48 hours ahead.

See the website https://onlineforecasting.org for more information.

`pairs.data.list` *Generation of pairs plot for a data.list.*

Description

Generate a pairs plot for the vectors in the data.list.

Usage

```
## S3 method for class 'data.list'
pairs(
  x,
  subset = NA,
  nms = NA,
  kseq = NA,
  lagforecasts = TRUE,
  pattern = NA,
  lower.panel = NULL,
  panel = panel.smooth,
  pch = 20,
  cex = 0.7,
  ...
)
```

Arguments

`x`	The data.list from which to plot.
`subset`	The subset to be included. Passed to `subset.data.list()`.
`nms`	The names of the variables to be included. Passed to `subset.data.list()`.
`kseq`	The horizons to be included. Passed to `subset.data.list()`.
`lagforecasts`	Lag the forecasts such that they are synced with obervations. Passed to `subset.data.list()`.
`pattern`	Regex pattern to select the included variables. Passed to `subset.data.list()`.
`lower.panel`	Passed to `pairs()`.
`panel`	Passed to `pairs()`.
`pch`	Passed to `pairs()`.
`cex`	Passed to `pairs()`.
`...`	Passed to `pairs()`.

Details

A very useful plot for checking what is in the forecasts, how they are synced and match the observations.

Examples

```
# Take a subset for the example
D <- subset(Dbuilding, c("2010-12-15","2011-01-15"), pattern="^Ta|^I", kseq=1:3)
pairs(D)

# If the forecasts and the observations are not aligned in time,
# which is easy to see by comparing to the previous plot.
pairs(D, lagforecasts=FALSE)
# Especially for the solar I syncronization is really important!
# Hence if the forecasts were not synced properly, then it can be detected using this type of plot.

# Alternatively, lag when taking the subset
D <- subset(Dbuilding, c("2010-12-15","2011-01-15"), pattern="^Ta|^I", kseq=1:3, lagforecasts=TRUE)
pairs(D, lagforecasts=FALSE)
```

---

par_ts *Set parameters for* plot_ts()

Description

Set parameters for plot_ts() globally

Usage

```
par_ts(fromoptions = FALSE, p = NA, ...)
```

Arguments

fromoptions	logical: Read the list of parameters set in options("par_ts")$par_ts, then the additional parameters set in ... are replaced before the list is returned.
p	List of the parameters, as returned by the function itself. If given, the additional parameters set in ... are replaced before the list is returned.
...	any of the following parameters can be set replacing the default values: xnm "t": The name of the time legendspace 10: Horizontal space for the lengend in character spaces legendcex 1: Scaling of the legend legendrangeshow TRUE: Include the range for each variable in the legend ylimextend c(lower,upper): Extend the ylim for each plot with a proportion, seperately for the lower and upper limit yaxisextend c(lower,upper): Extend the yaxis for each plot with a proportion, seperately for the lower and upper limit mainsline (numeric): with the line for the main in the plots. cex (numeric): The cex to use for the plot_ts plots. plotfun: The function used for plotting, as default lines. xaxisformat (character): The format of the xaxis, see strptime(). colorramp colorRampPalette: The colorramp used for setting multiple colors in each plot

Details

Often in a report some plot parameters must be set for all plots, which is done with `par()`.

The parameters which are general for `plot_ts()` can be set and saved in `options()`, and they will then be applied as default in all calls to plot_ts(). See the examples how to do this.

If any of these parameters are given to `plot_ts()`, then it will be used over the default.

Value

A list of the parameters above, which can be set globally (see examples) or passed to `plot_ts`.

Examples

```
# Data for plots
D <- subset(Dbuilding, 1:192)

# See the parameters which can be set
p <- par_ts()
names(p)
p$xnm

# Using the default values
plot_ts(D, c("heatload","Ta"), kseq=1:24)

# Set the parameters directly
plot_ts(D, c("heatload","Ta"), kseq=1:24, legendcex=0.8, legendspace=8)

# Set parameters to be given in a list
p <- par_ts()
p$legendcex <- 0.8
p$legendspace <- 8

# Use those parameters
plot_ts(D, c("heatload","Ta"), kseq=1:24, p=p)

# Set globally (if not set specifed the default values will be used)
options(par_ts=p)

# Now the global parameters will be used
plot_ts(D, c("heatload","Ta"), kseq=1:24)

# Still providing a parameter directly it will used, e.g. change the plotting function
plot_ts(D, c("heatload","Ta"), kseq=1:24, plotfun=points)

# Control more precisely the plotting function
plot_ts(D, c("heatload","Ta"), kseq=1:24, plotfun=function(x, ...){ points(x, type="b", ...)})

# Another colorramp function
p$colorramp <- rainbow
options(par_ts=p)
plot_ts(D, c("heatload","Ta"), kseq=1:24)
```

pbspline *Wrapper for* bspline *with* periodic=TRUE

Description

Wrapper for bspline with periodic=TRUE

Usage

```
pbspline(
  X,
  Boundary.knots = NA,
  intercept = FALSE,
  df = NULL,
  knots = NULL,
  degree = 3,
  bknots = NA
)
```

Arguments

X	see ?bspline
Boundary.knots	see ?bspline
intercept	see ?bspline
df	see ?bspline
knots	see ?bspline
degree	see ?bspline
bknots	see ?bspline

Details

Simply a wrapper.

See Also

Other Transform stage functions: bspline()

`persistence` *Generate persistence forecasts*

Description

Generate persistence and periodic persistence forecasts

Usage

```
persistence(y, kseq, perlen = NA)
```

Arguments

`y`	(numeric) The model output to be forecasted.
`kseq`	(integer) The horizons to be forecasted.
`perlen`	(integer) The period length for seasonal persistence.

Details

Generate a forecast matrix using persistence. The simple persistence is with the current value of y, i.e. the value at time t is used as forecast

A seasonal persistence with a specific period can be generated by setting the argument `perlen` to the length of the period in steps. The value used for the forecast is then the latest available, which is matches the seasonality for time t+k, see the examples.

Value

Forecast matrix as a `data.frame` (named `Yhat` in similar functions)

Examples

```
# Simple persistence just copies the current value for the forecasts
persistence(1:10, kseq=1:4)

# Seasonal persistence takes the value perlen steps back
persistence(1:10, kseq=1:4, perlen=4)

# If the horizons are longer than perlen, then the perlen*i steps back is taken (i is an integer)
persistence(1:10, kseq=1:12, perlen=4)
```

plot_ts *Time series plotting*

Description

Plot time series of observations and forecasts (lagged to be aligned in time).

Plot forecasts, residuals, cumulated residuals and RLS coefficients

Simply the same as `plot_ts()` with `usely=TRUE`, such that plotly is used.

Usage

```
plot_ts(
  object,
  patterns = ".*",
  xlim = NA,
  ylims = NA,
  xlab = "",
  ylabs = NA,
  mains = "",
  mainouter = "",
  legendtexts = NA,
  colormaps = NA,
  xat = NA,
  usely = FALSE,
  plotit = TRUE,
  p = NA,
  ...
)

## S3 method for class 'data.list'
plot_ts(
  object,
  patterns = ".*",
  xlim = NA,
  ylims = NA,
  xlab = "",
  ylabs = NA,
  mains = "",
  mainouter = "",
  legendtexts = NA,
  colormaps = NA,
  xat = NA,
  usely = FALSE,
  plotit = TRUE,
  p = NA,
  kseq = NA,
```

```
  ...
)

## S3 method for class 'data.frame'
plot_ts(
  object,
  patterns = ".*",
  xlim = NA,
  ylims = NA,
  xlab = "",
  ylabs = NA,
  mains = NA,
  mainouter = "",
  legendtexts = NA,
  colormaps = NA,
  xat = NA,
  usely = FALSE,
  plotit = TRUE,
  p = NA,
  namesdata = NA,
  ...
)

## S3 method for class 'matrix'
plot_ts(
  object,
  patterns = ".*",
  xlim = NA,
  ylims = NA,
  xlab = "",
  ylabs = NA,
  mains = NA,
  mainouter = "",
  legendtexts = NA,
  colormaps = NA,
  xat = NA,
  usely = FALSE,
  plotit = TRUE,
  p = NA,
  namesdata = NA,
  ...
)

## S3 method for class 'rls_fit'
plot_ts(
  object,
  patterns = c("^y$|^Yhat$", "^Residuals$", "CumAbsResiduals$", pst("^",
    names(fit$Lfitval[[1]]), "$")),
```

```
  xlim = NA,
  ylims = NA,
  xlab = "",
  ylabs = NA,
  mains = "",
  mainouter = "",
  legendtexts = NA,
  colormaps = NA,
  xat = NA,
  usely = FALSE,
  plotit = TRUE,
  p = NA,
  kseq = NA,
  ...
)

plotly_ts(
  object,
  patterns = ".*",
  xlim = NA,
  ylims = NA,
  xlab = "",
  ylabs = NA,
  mains = "",
  mainouter = "",
  legendtexts = NA,
  colormaps = NA,
  xat = NA,
  usely = FALSE,
  p = NA,
  ...
)
```

Arguments

object	A data.list or data.frame with observations and forecasts, note diffe
patterns	See plot_ts. The default pattern finds the generated series in the function, '!!RLSinputs!!' will be replaced with the names of the RLS inputs (regression stage inputs).
xlim	The time range as a character of length 2 and form "YYYY-MM-DD" or POSIX. Date to start and end the plot.
ylims	The ylim for each plot given in a list.
xlab	A character with the label for the x-axis.
ylabs	A character vector with labels for the y-axes.
mains	A character vector with the main for each plot.
mainouter	A character with the main at the top of the plot (can also be added afterwards with title(main, outer=TRUE)).

legendtexts	A list with the legend texts for each plot (replaces the names of the variables).
colormaps	A list of colormaps, which will be used in each plot.
xat	POSIXt specifying where the ticks on x-axis should be put.
usely	If TRUE then plotly will be used.
plotit	If FALSE then the plot will not be generated, only data returned.
p	The plot_ts parameters in a list, as generated with the function par_ts().
...	Parameters passed to par_ts, see the list of parameters in ?par_ts.
kseq	For class(object)=="data.list" an integer vector, default = NA. Control which forecast horizons to include in the plots. If NA all the horizons will be included.
namesdata	For class(object)=="data.frame" a character vector. Names of columns in object to be searched in, instead of names(object).
fit	An rls_fit.

Details

Generates time series plots depending on the variables matched by each regular expression given in the patterns argument.

The forecasts matrices in the data.list given in object will be lagged to be aligned in time (i.e. k-step forecasts will be lagged by k).

Use the plotly package if argument usely is TRUE, see plotly_ts().

A useful plot for residual analysis and model validation of an RLS fitted forecast model.

All parameters, except those described below, are simply passed to plot_ts().

The plotly package must be installed and loaded.

Note that the plot parameters set with par_ts() have no effect on the plotly plots.

See https://onlineforecasting.org/vignettes/nice-tricks.html.

Value

A list with a data.frame with the data for each plot, if usely=TRUE, then a list of the figures (drawn with print(subplot(L, shareX=TRUE, nrows=length(L), titleY = TRUE))).

The plotted data in a data.list.

See Also

par_ts for setting plot control parameters.

regex for regular expressions to select which variables to plot.

plot_ts.

Examples

```
# Time series plots for \code{data.list}, same as for \code{data.frame} except use of \code{kseq}
D <- Dbuilding
plot_ts(D, c("heatload","Ta"), kseq=c(1,24))
# Make two plots (and set the space for the legend)
plot_ts(D, c("heatload","Ta"), kseq=c(1,24), legendspace=11)
# Only the Ta observations
plot_ts(D, c("heatload","Taobs$"), kseq=c(1,24), legendspace=11)

# Give labels
plot_ts(D, c("heatload","Ta"), kseq=c(1,24), xlab="Time", ylabs=c("Heat (kW)","Temperature (C)"))
# Mains (see mainsline in par_ts())
plot_ts(D, c("heatload","Ta"), kseq=c(1,24), mains=c("Heatload","Temperature"), mainsline=c(-1,-2))

# Format of the xaxis (see par_ts())
plot_ts(D, c("heatload","Ta"), kseq=c(1,24), xaxisformat="%Y-%m-%d %H:%m")

# Return the data, for other plots etc.
L <- plot_ts(D, c("heatload","Ta"), kseq=c(1,24))
names(L[[1]])
names(L[[2]])



# Fit a model (see vignette 'setup-and-use-model'
D <- Dbuilding
D$scoreperiod <- in_range("2010-12-22", D$t)
model <- forecastmodel$new()
model$output = "heatload"
model$add_inputs(Ta = "Ta",
                 mu = "one()")
model$add_regprm("rls_prm(lambda=0.9)")
model$kseq <- c(3,18)
fit1 <- rls_fit(NA, model, D, returnanalysis = TRUE)

# Plot it
plot_ts(fit1)

# Return the data
Dplot <- plot_ts(fit1)

# The RLS coefficients are now in a nice format
head(Dplot$mu)


# See the website link above
```

`print.forecastmodel` *Print forecast model*

Description

Prints a forecast model

Usage

```
## S3 method for class 'forecastmodel'
print(x, ...)
```

Arguments

x	A forecastmodel object
...	Not used.

Details

A simple print out of the model output and inputs

---

print_to_message *Simple function for capturing from the print function and send it in a message().*

Description

Simple function for capturing from the print function and send it in a message().

Usage

```
print_to_message(...)
```

Arguments

...	Passed to print which passed to message.

---

pst *Simple wrapper for paste0().*

Description

Simple wrapper for paste0().

Usage

```
pst(...)
```

Arguments

...	Passed to paste0().

`resample` *Resampling to equidistant time series*

Description

Make a downsampling to a lower sampling frequency

Usage

```
resample(
  object,
  ts,
  tstart = NA,
  tend = NA,
  timename = "t",
  fun = mean,
  quantizetime = TRUE,
  ...
)
```

Arguments

`object`	Can be data.frame
`ts`	(numeric) New sample period in seconds
`tstart`	A POSIXxx (or charater or numeric), which indicates the first time point in the series returned
`tend`	A POSIXxx (or charater or numeric), which indicates the last time point in the series returned
`timename`	(character) The name of the time column in object
`fun`	(function) The function of apply. Default is mean, such that average values are obtained
`quantizetime`	(logical) Should the new time points be set to the end of the time intervals, or should they also be the result of the fun function
`...`	Passed on to the fun function

Details

Given an object with a column indicating the time points of the observations the function returns a similar object, where the function is applied for each new (and longer) interval.

Typically it is used if for example 15 minute values should be made into 1 hour values.

NOTE that it is always assumed that the time point is at the end of the time interval, e.g. if hourly values are returned, then "2019-01-01 01:00" indicates the first hour in 2019.

All time points at the time point (border) of between two intervals is assigned to the first interval of the two.

Value

A downsampled data.frame

Examples

```
# Generate some test data with 10 minutes sampling frequency for one day
X <- data.frame(t=seq(ct("2019-01-01 00:10"),ct("2019-01-02"), by=10*60))

# A single sine over the day
X$val <- sin(as.numeric(X$t)/3600*2*pi/(24))

# Resample to hourly average values
Xre <- resample(X, 3600)
plot(X$t, X$val)
lines(Xre$t, Xre$val, type="b", col=2)

# Resample to hourly max values
Xre <- resample(X, 3600, fun=max)
lines(Xre$t, Xre$val, type="b", col=3)

# Another starting time point
Xre <- resample(X, 3600, tstart="2019-01-01 00:30")
lines(Xre$t, Xre$val, type="b", col=4)
```

---

residuals.data.frame *Calculate the residuals given a forecast matrix and the observations.*

Description

Calculate the residuals given a forecast matrix and the observations.

Usage

```
## S3 method for class 'data.frame'
residuals(object, y, ...)

## S3 method for class 'matrix'
residuals(object, y, ...)

## S3 method for class 'list'
residuals(object, y, ...)

## S3 method for class 'forecastmodel_fit'
residuals(object, ...)
```

Arguments

object	The forecast matrix (a data.frame with kxx as column names, Yhat in returned fits).
y	The observations vector.
...	Not used.

Details

Simply give the forecast matrix and the observations to get the residuals for each horizon in the forecast matrix.

The residuals returned are synced with the observations (i.e. k0) and the columns are names "hxx" (not kxx) to indicate this and will not be lagged in plot_ts().

Value

If object is a matrix or data.frame: a data.frame with the residuals for each horizon. If object is a list: A list with residuals from each element.

Examples

```
# ?? list example
# Just a vector to be forecasted
n <- 100
D <- data.list()
D$t <- 1:n
D$y <- c(filter(rnorm(n), 0.95, "recursive"))
plot(D$y, type="l")

# Generate a forecast matrix with a simple persistence model
D$Yhat <- persistence(D$y, kseq=1:4)

# The residuals for each horizon
D$Resid <- residuals(D$Yhat, D$y)
D$Resid
# Note the names of the columns
names(D$Resid)
# which means that they are aligned with the observations and will not be lagged in the plot
plot_ts(D, c("y|Yhat","Resid"))

# Check that it matches (the forecasts is lagged in the plot_ts
# such that the forecast for t+k is at t+k (and not t))
plot_ts(D, c("y|Yhat","Resid"), xlim=c(1,10), kseq=1,
        plotfun=function(x,...){lines(x,...,type="b")})

# Just for fun, see the auto-correlation function of the persistence
acf(D$Resid$h1, na.action=na.pass)
acf(D$Resid$h4, na.action=na.pass)
```

`rls_fit` *Fit an onlineforecast model with Recursive Least Squares (RLS).*

Description

This function fits the onlineforecast model to the data and returns either: model validation data or just the score value.

Usage

```
rls_fit(
  prm = NA,
  model,
  data,
  scorefun = NA,
  returnanalysis = TRUE,
  runcpp = TRUE,
  printout = TRUE
)
```

Arguments

`prm`	vector with the parameters for fitting. Deliberately as the first element to be able to use `optim` or other optimizer. If NA then the model will be fitted with the current values in the input expressions, see examples.
`model`	as an object of class forecastmodel: The model to be fitted.
`data`	as a data.list with the data to fit the model on.
`scorefun`	as a function (optional), default is `rmse`. If the score function is given it will be applied to the residuals of each horizon (only data$scoreperiod is included).
`returnanalysis`	as a logical. If FALSE then the sum of the scoreval on all horizons are returned, if TRUE a list with values for analysis.
`runcpp`	logical: If true the c++ implementation of RLS is run, if false the R implementation is run (slower).
`printout`	logical: If TRUE the offline parameters and the score function value are printed.

Details

This function has three main purposes (in the examples these three are demonstrated in the examples):

- Returning model validation data, such as residuals and recursive estimated parameters.

- For optimizing the parameters using an R optimizer function. The parameters to optimize for is given in `prm`

- Fitting a model to data and saving the final state in the model object (such that from that point the model can be updated recursively as new data is received).

Note, if the `scorefun` is given the `data$scoreperiod` must be set to (int or logical) define which points to be evaluated in the scorefun.

Value

Depends on:

- If `returnanalysis` is TRUE a list containing:

* `Yhat`: data.frame with forecasts for `model$kseq` horizons.

* `model`: The forecastmodel object cloned deep, so can be modified without changing the original object.

* `data`: data.list with the data used, see examples on how to obtain the transformed data.

* `Lfitval`: list with RLS coefficients in a data.frame for each horizon, use `plot_ts.rls_fit` to plot them and to obtain them as a data.frame for each coefficient.

* `scoreval`: data.frame with the scorefun result on each horizon (only scoreperiod is included).

- If `returnanalysis` is FALSE (and `scorefun` is given): The sum of the score function on all horizons (specified with model$kseq).

See Also

For optimizing parameters `rls_optim()`, for summary `summary.rls_fit`, for plotting `plot_ts.rls_fit()`, and the other functions starting with 'rls_'.

Examples

```
# Take data
D <- subset(Dbuilding, c("2010-12-15", "2011-01-01"))
D$y <- D$heatload
# Define a simple model
model <- forecastmodel$new()
model$output <- "y"
model$add_inputs(Ta = "Ta",
    mu = "one()")
model$add_regprm("rls_prm(lambda=0.99)")

# Before fitting the model, define which points to include in the evaluation of the score function
D$scoreperiod <- in_range("2010-12-20", D$t)
# And the sequence of horizons to fit for
model$kseq <- 1:6

# Now we can fit the model with RLS and get the model validation analysis data
fit <- rls_fit(model = model, data = D)
# What did we get back?
names(fit)
# The one-step forecast
plot(D$y, type="l")
lines(fit$Yhat$k1, col=2)
# The one-step RLS coefficients over time (Lfitval is a list of the fits for each horizon)
plot(fit$Lfitval$k1$Ta, type="l")

# A summary
summary(fit)
```

```
# Plot the fit
plot_ts(fit, kseq=1)

# Fitting with lower lambda makes the RLS coefficients change faster
fit2 <- rls_fit(prm = c(lambda=0.9), model, D)
plot_ts(fit2, kseq=1)


# It can return a score
rls_fit(c(lambda=0.9), model, D, scorefun=rmse, returnanalysis=FALSE)

# Such that it can be passed to an optimzer (see ?rls_optim for a nice wrapper of optim)
val <- optim(c(lambda=0.99), rls_fit, model = model, data = D, scorefun = rmse,
             returnanalysis=FALSE)
val$par
# Which can then simply be applied
rls_fit(val$par, model, D, scorefun=rmse, returnanalysis=FALSE)
# see ?rls_optim, how optim is wrapped for a little easiere use

# See rmse as a function of horizon
fit <- rls_fit(val$par, model, D, scorefun = rmse)
plot(fit$scoreval, xlab="Horizon k", ylab="RMSE")
# See ?score for a little more consistent way of calculating this


# Try adding a low-pass filter to Ta
model$add_inputs(Ta = "lp(Ta, a1=0.92)")
# To obtain the transformed data, i.e. the data which is used as input to the RLS
model$reset_state()
# Generate the the transformed data
datatr <- model$transform_data(D)
# What did we get?
str(datatr)
# See the effect of low-pass filtering
plot(D$Ta$k1, type="l")
lines(datatr$Ta$k1, col=2)
# Try changing the 'a1' coefficient and rerun
# ?rls_optim for how to optimize also this coefficient
```

`rls_optim` *Optimize parameters for onlineforecast model fitted with RLS*

Description

Optimize parameters (transformation stage) of RLS model

Usage

```
rls_optim(
  model,
  data,
  kseq = NA,
  scorefun = rmse,
  cachedir = "",
  cachererun = FALSE,
  printout = TRUE,
  method = "L-BFGS-B",
  ...
)
```

Arguments

model	The onlineforecast model, including inputs, output, kseq, p
data	The data.list which holds the data on which the model is fitted.
kseq	The horizons to fit for (if not set, then model$kseq is used)
scorefun	The function to be score used for calculating the score to be optimized.
cachedir	A character specifying the path (and prefix) of the cache file name. If set to "", then no cache will be loaded or written. See https://onlineforecasting.org/vignettes/nice-tricks.html for examples.
cachererun	A logical controlling whether to run the optimization even if the cache exists.
printout	A logical determining if the score function is printed out in each iteration of the optimization.
method	The method argument for optim.
...	Additional parameters to optim

Details

This is a wrapper for optim to enable easy use of bounds and caching in the optimization.

One smart trick, is to cache the optimization results. Caching can be done by providing a path to the cachedir argument (relative to the current working directory). E.g. rls_optim(model, D, cachedir="cache") will write a file in the folder 'cache', such that next time the same call is carried out, then the file is read instead of running the optimization again. See the example in https://onlineforecasting.org/vignettes/nice-tricks.html.

Value

Result object of optim(). Parameters resulting from the optimization can be found from result$par

See Also

link{optim} for how to control the optimization.

Examples

```
# Take data
D <- subset(Dbuilding, c("2010-12-15", "2011-01-01"))
D$y <- D$heatload
# Define a simple model
model <- forecastmodel$new()
model$add_inputs(Ta = "Ta", mu = "one()")
model$add_regprm("rls_prm(lambda=0.99)")

# Before fitting the model, define which points to include in the evaluation of the score function
D$scoreperiod <- in_range("2010-12-20", D$t)
# And the sequence of horizons to fit for
model$kseq <- 1:6
# Now we can fit the model and get the score, as it is
rls_fit(model=model, data=D, scorefun=rmse, returnanalysis=FALSE)
# Or we can change the lambda
rls_fit(c(lambda=0.9), model, D, rmse, returnanalysis=FALSE)

# This could be passed to optim() (or any optimizer, see forecastmodel$insert_prm()).
optim(c(lambda=0.98), rls_fit, model=model, data=D, scorefun=rmse, returnanalysis=FALSE,
      lower=c(lambda=0.9), upper=c(lambda=0.999), method="L-BFGS-B")

# rls_optim is simply a helper, it's makes using bounds easiere and enables caching of the results
# First add bounds for lambda (lower, init, upper)
model$add_prmbounds(lambda = c(0.9, 0.98, 0.999))

# Now the same optimization as above can be done by
val <- rls_optim(model, D)
val
```

`rls_predict` *Prediction with an rls model.*

Description

Use a fitted forecast model to predict its output variable with transformed data.

Usage

```
rls_predict(model, datatr = NA)
```

Arguments

`model`	Onlineforecast model object which has been fitted.
`datatr`	Transformed data.

Details

See the ??ref(recursive updating vignette, not yet available).

Value

The Yhat forecast matrix with a forecast for each model$kseq and for each time point in `datatr$t`.

Examples

```
# Take data
D <- subset(Dbuilding, c("2010-12-15", "2011-01-01"))
D$y <- D$heatload
# Define a simple model
model <- forecastmodel$new()
model$add_inputs(Ta = "Ta", mu = "one()")
model$add_regprm("rls_prm(lambda=0.99)")

# Before fitting the model, define which points to include in the evaluation of the score function
D$scoreperiod <- in_range("2010-12-20", D$t)
# And the sequence of horizons to fit for
model$kseq <- 1:6

# Transform using the mdoel
datatr <- model$transform_data(D)

# See the transformed data
str(datatr)

# The model has not been fitted
model$Lfits

# To fit
rls_fit(model=model, data=D)

# Now the fits for each horizon are there (the latest update)
# For example the current parameter estimates
model$Lfits$k1$theta

# Use the current values for prediction
D$Yhat <- rls_predict(model, datatr)

# Plot it
plot_ts(D, c("y|Yhat"), kseq=1)

# Recursive updating and prediction
Dnew <- subset(Dbuilding, c("2011-01-01", "2011-01-02"))

for(i in 1:length(Dnew$t)){
    # New data arrives
    Dt <- subset(Dnew, i)
    # Remember that the transformation must only be done once if some transformation
```

```
    # which is has a state, e.g. lp(), is used
    datatr <- model$transform_data(Dt)
    # Update, remember that this must only be once for each new point
    # (it updates the parameter estimates, i.e. model$Lfits)
    rls_update(model, datatr, Dt$heatload)
    # Now predict to generate the new forecast
    print(rls_predict(model, datatr))
}
```

`rls_prm` *Function for generating the parameters for RLS regression*

Description

Function for generating the parameters for RLS regression

Usage

```
rls_prm(lambda)
```

Arguments

`lambda`	The forgetting factor

Details

The RLS needs only a forgetting factor parameter.

Value

A list of the parameters

Examples

```
# Take data
D <- subset(Dbuilding, c("2010-12-15", "2011-01-01"))
D$y <- D$heatload
D$scoreperiod <- in_range("2010-12-20", D$t)
# Define a simple model
model <- forecastmodel$new()
model$add_inputs(Ta = "Ta", mu = "one()")
model$kseq <- 1:6

# Here the expression which sets the parameters is defined
model$add_regprm("rls_prm(lambda=0.99)")
model$regprmexpr

# These will fit with lambda=0.99
```

```
rls_fit(prm=NA, model, D)
rls_fit(prm=c(lambda=0.99), model, D)

# The expression is evaluated when the model is fitted
rls_fit(prm=c(lambda=0.85), model, D)

# What happens is simply that the expression was manipulated
model$regprmexpr
model$regprm

# Same change could be done by
model$regprm <- list(lambda=0.3)
model$regprm
val <- rls_fit(prm=NA, model, D)
```

`rls_summary` *Print summary of an onlineforecast model fitted with RLS*

Description

The summary of an onlineforecast model fitted with RLS with simple stats providing a simple overview.

Usage

```
rls_summary(object, scoreperiod = NA, scorefun = rmse, printit = TRUE, ...)
```

Arguments

`object`	of class `rls_fit`, so a fit calculated by `rls_fit`.
`scoreperiod`	logical (or index). If this scoreperiod is given, then it will be used over the one in the fit.
`scorefun`	The score function to be applied on each horizon.
`printit`	Print the result.
`...`	Not used.

Details

The following is printed:

* The model.

* Number of observations included in the scoreperiod.

* RLS coefficients summary statistics for the estimated coefficient time series (since observations are correlated, then usual statistics cannot be applied directly):

- mean: the sample mean of the series.

- sd: sample standard deviation of the series.

- min: minimum of the series.

- max: maximum of the series.

* Scorefunction applied for each horizon, per default the RMSE.

Value

A list of:

- scorefun.

- scoreval (value of the scorefun for each horizon).

- scoreperiod is the scoreperiod used.

Examples

```
# Take data
D <- subset(Dbuilding, c("2010-12-15", "2011-01-01"))
D$y <- D$heatload
D$scoreperiod <- in_range("2010-12-20", D$t)
# Define a model
model <- forecastmodel$new()
model$add_inputs(Ta = "Ta",
                 mu = "one()")
model$add_regprm("rls_prm(lambda=0.99)")
model$kseq <- 1:6
# Fit it
fit <- rls_fit(prm=c(lambda=0.99), model, D)

# Print the summary
summary(fit)
# We see:
#   - The model (output, inputs, lambda)
#  - The Ta coefficient is around -0.12 in average (for all horizons) with a standard dev. of 0.03,
#     so not varying extremely (between -0.18 and -0.027).
#   - The intercept mu is around 5.5 and varying very little.
#   - The RMSE is around 0.9 for all horizons.

# The residuals and coefficient series can be seen by
plot_ts(fit)
```

`rls_update` *Updates the model fits*

Description

Calculates the RLS update of the model coefficients with the provived data.

Usage

```
rls_update(model, datatr = NA, y = NA, runcpp = TRUE)
```

Arguments

model	A model object
datatr	a data.list with transformed data (from model$transform_data(D))
y	A vector of the model output for the corresponding time steps in datatr
runcpp	Optional, default = TRUE. If TRUE, a c++ implementation of the update is run, otherwise a slower R implementation is used.

Details

See vignette ??ref(recursive updating, not yet finished) on how to use the function.

Value

Returns a named list for each horizon (model$kseq) containing the variables needed for the RLS fit (for each horizon, which is saved in model$Lfits):

It will update variables in the forecast model object.

See Also

See rls_predict.

Examples

```
# See rls_predict examples
```

---

rls_update_cpp *Calculating k-step recursive least squares estimates*

Description

This function applies the k-step recursive least squares scheme to estimate parameters in a linear regression model.

Arguments

y	Vector of observation
X	Matrix of input variables (design matrix)
theta	Vector of parameters (initial value)
P	Covariance matrix (initial value)
lambda	Forgetting factor
k	Forecast horizon
n	Length of the input
np	Dimension of P (np x np)
istart	Start index
kmax	Keep only the last kmax rows for next time

---

rmse *Computes the RMSE score.*

Description

Returns the RMSE.

Usage

```
rmse(x)
```

Arguments

x	a numerical vector of residuals.

Details

Used for forecast evaluation evaluation and optimization of parameters in model fitting.

Note that NAs are ignored (i.e. mean is called with na.rm=TRUE).

Value

The RMSE score.

See Also

score() for calculation of a score for the k'th horizon

Examples

```
 # Just a vector to be forecasted
 y <- c(filter(rnorm(100), 0.95, "recursive"))
 # Generate a forecast matrix with a simple persistence model
 Yhat <- persistence(y, kseq=1:4)
 # The residuals for each horizon
 Resid <- residuals(Yhat, y)

# Calculate the score for the k1 horizon
rmse(Resid$h1)

# For all horizons
apply(Resid, 2, rmse)
```

---

score *Calculate the score for each horizon.*

Description

Calculates the score for each horizon for a matrix with residuals for each horizon.

Usage

```
score(object, scoreperiod = NA, usecomplete = TRUE, scorefun = rmse, ...)

## S3 method for class 'list'
score(object, scoreperiod = NA, usecomplete = TRUE, scorefun = rmse, ...)

## S3 method for class 'data.frame'
score(object, scoreperiod = NA, usecomplete = TRUE, scorefun = rmse, ...)
```

Arguments

object	??list or A matrix with residuals (columns named hxx) for which to calculate the score for each horizon.
scoreperiod	as a logical vector controlling which points to be included in the score calculation. If NA then all values are included (depeding on 'complete').
usecomplete	TRUE then only the values available for all horizons are included (i.e. if at one time point there is a missing value, then values for this time point is removed for all horizons in the calculation).
scorefun	The score function.
...	is passed on to the score function.

Details

Applies the scorefun on all horizons (each column) of the residuals matrix. See the description of each parameter for more details.

Value

A list with the a numeric vector with the score value for each horizon and the applied scoreperiod (note can be different from the given scoreperiod, if only complete observations are used (as per default)).

Examples

```
# Just a vector to be forecasted
y <- c(filter(rnorm(100), 0.95, "recursive"))
# Generate a forecast matrix with a simple persistence model
Yhat <- persistence(y, kseq=1:4)
# The residuals for each horizon
Resid <- residuals(Yhat, y)

# Calculate the score for the k1 horizon
score(Resid)

# In the beginning the horizons have NAs
head(Resid)
# Default is that only complete cases over all horizons are included
score(Resid)
# So including all cases for each horizon will give different values
score(Resid, usecomplete=FALSE)

# Given a list
# The residuals for each horizon
Resid2 <- residuals(persistence(y,kseq=1:4)+rnorm(100), y)

score(list(Resid,Resid2))
```

---

setpar *Setting par() plotting parameters*

Description

Setting par() plotting parameters to a set of default values

Usage

```
setpar(tmpl = "ts", mfrow = c(1, 1), ...)
```

Arguments

tmpl	The name of the parameter template, give "ts" as default
mfrow	The mfrow for par.
...	More parameters for par.

Details

A simple function, which sets the par() plotting parameters to a default set of values.

Actually, only really used for setting useful par values for multiple time series plots with same x-axis. Give tmpl="ts" and mfrow=c(x,1), where x is the number of plots.

Value

Return the original set of parameters, such that they can be reset after plotting.

Examples

```
# Make some data
D <- data.frame(t=seq(ct("2020-01-01"),ct("2020-01-10"),len=100), x=rnorm(100), y=runif(100))
# Remember the currect par values
oldpar <- setpar()

# Generate two stacked plots with same x-axis
setpar("ts", mfrow=c(2,1))
plot(D$t, D$x, type="l")
plot(D$t, D$y, type="l")
# Note xaxt="s" must be set
axis.POSIXct(1, D$t, xaxt="s", format="%Y-%m-%d")

# Set back the par to the previous
par(oldpar)

# In a function, where this is used and a plot is generated,
# then do like this in order to automatically reset on exit
oldpar <- setpar(mfrow=c(2,1))
on.exit(par(oldpar))
```

---

stairs *Plotting stairs with time point at end of interval.*

Description

Plotting steps with time point at end of interval

Usage

```
stairs(x, y, type = "b", preline = FALSE, pch = 19, ...)
```

Arguments

x	x values for plot.
y	y values for plot.
type	if 'b' then include points.
preline	if TRUE, then a line backwards from the first point is added.
pch	Passed to points().
...	Passed to lines() and points() when they are called in the function.

Details

It's easy to plot stairs with plot(x,y,type="s"), however that makes the steps forward from x, for time series this works if the time points are at the beginning of the intervals.

Often with time series the time points are in the end of the intervals, so the steps should go backaward, this is achieved with this function.

Examples

```
# Usual stairs plot has steps forward from x
x <- rnorm(10)
plot(1:10, x, type="s")

# Stairs with step backward from x
plot(1:10, x, type="n")
stairs(1:10, x)

# Use for time series plotting
plot_ts(Dbuilding, "heatload", c("2010-12-15","2010-12-16"), plotfun=stairs)

# Set it globally for all plot_ts
p <- par_ts()
p$plotfun <- stairs
options(par_ts=p)
plot_ts(Dbuilding, "heatload", c("2010-12-15","2010-12-16"))

# Modify it to only lines
plot_ts(Dbuilding, "heatload", c("2010-12-15","2010-12-16"),
        plotfun=function(x,y,...){stairs(x,y, type="l")})
```

---

state_getval *Get the state value kept in last call.*

Description

Get the state value kept in last call to the transformation function.

Usage

```
state_getval(initval)
```

Arguments

initval	If no state was kept, then this init value is returned.

Details

Transformation functions (e.g. lp, fs, bspline) can need to keep a state value between calls, e.g. when new data arrives and must be transformed. This function is used to getting the state values set in last call to the function.

Uses the input_class$state_getval().

Value

The state value, but if not found, then the initval.

See Also

state_setval() for setting the state value and input_class.

Examples

```
# See how it can be used in lp, which needs to save the state of the filter
# Note how it is not needed to do anything else than getting and setting the state
# in transformations (model$transform_data()), then multiple transformation functions can be called,
# but they are always in the same order, so the state (set,get) functions keep a counter internally
# to make sure that the correct values are set and returned when called again.
lp
```

---

state_setval *Set a state value to be kept for next the transformation function is called.*

Description

Set a state value to be kept for next the transformation function is called.

Usage

```
state_setval(val)
```

Arguments

val	The value to set and kept for next call.

Details

Transformation functions (e.g. lp, fs, bspline) can need to keep a state value between calls, e.g. when new data arrives and must be transformed. This function is used to setting the state values set in last call to the function.

Uses the `input_class$state_getval()`.

See Also

`state_setval()` for setting the state value and `input_class`.

Examples

```
# See how it can be used in lp, which needs to save the state of the filter
# Note how it is not needed to do anything else than getting and setting the state
# in transformations (model$transform_data()), then multiple transformation functions can be called,
# but they are always in the same order, so the state (set,get) functions keep a counter internally
# to make sure that the correct values are set and returned when called again.
lp
```

---

step_optim *Forward and backward model selection*

Description

Forward and backward model selection

Usage

```
step_optim(
  modelfull,
  data,
  prm = list(NA),
  direction = c("both", "backward", "forward", "backwardboth", "forwardboth"),
  modelstart = NA,
  keepinputs = FALSE,
  optimfun = rls_optim,
  fitfun = NA,
  scorefun = rmse,
  printout = FALSE,
  mc.cores = getOption("mc.cores", 2L),
  ...
)
```

Arguments

modelfull	The full forecastmodel containing all inputs which will be can be included in the selection.
data	The data.list which holds the data on which the model is fitted.
prm	A list of integer parameters to be stepped. Given using the same syntax as parameters for optimization, e.g. 'list(I__degree = c(min=3, max=7))' will step the "degree" for input "I".
direction	The direction to be used in the selection process.
modelstart	A forecastmodel. If it's set then it will be used as the selected model from the first step of the stepping. It should be a sub model of the full model.
keepinputs	If TRUE no inputs can be removed in a step, if FALSE then any input can be removed. If given as a character vector with names of inputs, then they cannot be removed in any step.
optimfun	The function which will carry out the optimization in each step.
fitfun	A fit function, should be the same as used in optimfun(). If provided, then the score is caculated with this function (instead of the one called in optimfun(), hence the default is rls_fit(), which is called in rls_optim()). Furthermore, information on complete cases are printed and returned.
scorefun	The score function used.
printout	Logical. Passed on to fitting functions.
mc.cores	The mc.cores argument of mclapply. If debugging it can be necessary to set it to 1 to stop execution.
...	Additional arguments which will be passed on to optimfun. For example control how many steps

Details

This function takes a model and carry out a model selection by stepping backward, forward or in both directions.

Note that mclapply is used. In order to control the number of cores to use, then set it, e.g. to one core 'options(mc.cores=1)', which is needed for debugging to work.

The full model containing all inputs must be given. In each step new models are generated, with either one removed input or one added input, and then all the generated models are optimized and their scores compared. If any new model have an improved score compared to the currently selected model, then the new is selected and the process is repeated until no new improvement is obtained.

In addition to selecting inputs, then integer parameters can also be stepped through, e.g. the order of basis splined or the number of harmonics in Fourier series.

The stepping process is different depending on the direction. In addition to the full model, a starting model can be given, then the selection process will start from that model.

If the direction is "both", which is default (same as "backwardboth") then the stepping is: - In first step inputs are removed one-by-one - In following steps, inputs still in the model are removed one-by-one, and inputs not in the model are added one-by-one

If the direction is "backwards": - Inputs are only removed in each step

If the direction is "forwardboth": - In the first step all inputs are removed - In following steps (same as "both")

If the direction is "forward": - In the first step all inputs are removed and from there inputs are only added

For stepping through integer variables in the transformation stage, then these have to be set in the "prm" argument. The stepping process will follow the input selection described above.

In case of missing values, especially in combination with auto-regressive models, it can be very important to make sure that only complete cases are included when calculating the score. By providing the 'fitfun' argument then the score will be calculated using only the complete cases across horizons and models in each step, see the last examples.

Note, that either kseq or kseqopt must be set on the modelfull object. If kseqopt is set, then it is used no matter the value of kseq.

Value

A list with the result of each step: - '$model' is the model selected in each step - '$score' is the score for the model selected in each step - '$optimresult' the result return by the the optimfun - '$completecases' a logical vector (NA if fitfun argument is not given) indicating which time points were complete across all horizons and models for the particular step.

Examples

```
# The data, just a rather short period to keep running times short
D <- subset(Dbuilding, c("2010-12-15", "2011-02-01"))
# Set the score period
D$scoreperiod <- in_range("2010-12-22", D$t)
#
D$tday <- make_tday(D$t, 1:36)
# Generate an input which is just random noise, i.e. should be removed in the selection
set.seed(83792)
D$noise <- make_input(rnorm(length(D$t)), 1:36)

# The full model
model <- forecastmodel$new()
# Set the model output
model$output = "heatload"
# Inputs (transformation step)
model$add_inputs(Ta = "Ta",
                 noise = "noise",
                 mu_tday = "fs(tday/24, nharmonics=5)",
                 mu = "one()")
# Regression step parameters
model$add_regprm("rls_prm(lambda=0.9)")
# Optimization bounds for parameters
model$add_prmbounds(lambda = c(0.9, 0.99, 0.9999))

# Select a model, in the optimization just run it for a single horizon
# Note that kseqopt could also be set
```

```
model$kseq <- 5

# Set the parameters to step on, note the
prm <- list(mu_tday__nharmonics = c(min=3, max=7))

# Note the control argument, which is passed to optim, it's now set to few
# iterations in the offline parameter optimization (MUST be increased in real applications)
control <- list(maxit=1)

# On Windows multi cores are not supported, so for the examples use only one core
mc.cores <- 1

# Run the default selection scheme, which is "both"
# (same as "backwardboth" if no start model is given)
L <- step_optim(model, D, prm, control=control, mc.cores=mc.cores)

# The optim value from each step is returned
getse(L, "optimresult")
getse(L,"score")

# The final model
L$final$model

# Other selection schemes
Lforward <- step_optim(model, D, prm, "forward", control=control, mc.cores=mc.cores)
Lbackward <- step_optim(model, D, prm, "backward", control=control, mc.cores=mc.cores)
Lbackwardboth <- step_optim(model, D, prm, "backwardboth", control=control, mc.cores=mc.cores)
Lforwardboth <- step_optim(model, D, prm, "forwardboth", control=control, mc.cores=mc.cores)

# It's possible avoid removing specified inputs
L <- step_optim(model, D, prm, keepinputs=c("mu","mu_tday"), control=control, mc.cores=mc.cores)

# Give a starting model
modelstart <- model$clone_deep()
modelstart$inputs[2:3] <- NULL
L <- step_optim(model, D, prm, modelstart=modelstart, control=control, mc.cores=mc.cores)

# If a fitting function is given, then it will be used for calculating the forecasts.
# Below it's the rls_fit function, so the same as used internally in rls_fit, so only
# difference is that now ONLY on the complete cases for all models in each step are used
# when calculating the score in each step
L1 <- step_optim(model, D, prm, fitfun=rls_fit, control=control, mc.cores=mc.cores)

# The easiest way to conclude if missing values have an influence is to
# compare the selection result running with and without
L2 <- step_optim(model, D, prm, control=control, mc.cores=mc.cores)

# Compare the selected models
tmp1 <- capture.output(getse(L1, "model"))
tmp2 <- capture.output(getse(L2, "model"))
identical(tmp1, tmp2)
```

```
# Note that caching can be really smart (the cache files are located in the
# cachedir folder (folder in current working directory, can be removed with
# unlink(foldername)) See e.g. `?rls_optim` for how the caching works
# L <- step_optim(model, D, prm, "forward", cachedir="cache", cachererun=FALSE, mc.cores=mc.cores)
```

`subset.data.list` *Take a subset of a data.list.*

Description

Take a subset of a data.list.

Usage

```
## S3 method for class 'data.list'
subset(
  x,
  subset = NA,
  nms = NA,
  kseq = NA,
  lagforecasts = FALSE,
  pattern = NA,
  ...
)
```

Arguments

`x`	The data.list to take a subset of.
`subset`	Given as the integer indexes or a logical vector, or alternatively `c(tstart,tend)`, where tstart and tend are either as POSIX or characters.
`nms`	The names of the variables in `x` to be included.
`kseq`	The k horizons of forecasts to be included.
`lagforecasts`	Should the forecasts be lagged k steps (thus useful for plotting etc.).
`pattern`	Regex pattern applied to select the variables in x to be included.
`...`	Not implemented.

Details

Different arguments can be given to select the subset. See the examples.

Value

a data.list with the subset.

Examples

```
# Use the data.list with building heat load
D <- Dbuilding
# Take a subset for the example
D <- subset(D, 1:10, nms=c("t","Taobs","Ta","Iobs","I"), kseq=1:3)

# Take subset index 2:4
subset(D, 2:4)

# Take subset for a period
subset(D, c("2010-12-15 02:00","2010-12-15 04:00"))

# Cannot request a variable not there
try(subset(D, nms=c("x","Ta")))

# Take specific horizons
subset(D, nms=c("I","Ta"), kseq = 1:2)
subset(D, nms=c("I","Ta"), kseq = 1)

# Lag the forecasts such that they are aligned in time with observations
subset(D, nms=c("Taobs","Ta"), kseq = 2:3, lagforecasts = TRUE)

# The order follows the order in nms
subset(D, nms=c("Ta","I"), kseq = 2)

# Return variables mathing a regex
subset(D, kseq=2, pattern="^I")

# Take data for Ta and lag the forecasts (good for plotting and fitting a model)
X <- subset(Dbuilding, 1:1000, pattern="^Ta", kseq = 10, lagforecasts = TRUE)

# A scatter plot between the forecast and the observations
# (try lagforecasts = FALSE and see the difference)
plot(X$Ta$k10, X$Taobs)

# Fit a model for the 10-step horizon
abline(lm(Taobs ~ Ta.k10, as.data.frame(X)), col=2)
```

`summary.data.list` *Summary with checks of the data.list for appropriate form.*

Description

Summary including checks of the data.list for appropriate form.

Usage

```
## S3 method for class 'data.list'
```

```
summary(
  object,
  printit = TRUE,
  stopit = TRUE,
  nms = names(object),
  msgextra = "",
  ...
)
```

Arguments

object	The object to be summarized and checked
printit	A boolean deciding if check results tables are printed
stopit	A boolean deciding if the function stop with an error if the check is not ok
nms	A character vector. If given specifies the variables (vectors or matrices) in object to check
msgextra	A character which is added in the printout of an (potential) error message
...	Not used

Details

Prints on table form the result of the checks.

Value

The tables generated.

Checking the data.list for appropriate form:

A check of the time vector t, which must have equidistant time points and no NAs.

Then the results of checks of vectors (observations):

- NAs: Proportion of NAs

- length: Same length as the 't' vector?

- class: The class of the vector

Then the results of checking data.frames and matrices (forecasts):

- maxHorizonNAs: The proportion of NAs for the horizon (i.e. column) with the highest proportion of NAs

- meanNAs: The proportion of NAs of the entire matrix

- nrow: Same length as the 't' vector?

- colnames: Columns must be names 'kx', where 'x' is the horizon (e.g. k12 is 12-step horizon)

- sameclass: Error if not all columns are the same class

- class: Prints the class of the columns if they are all the same

Examples

```
summary(Dbuilding)

# Some NAs in k1 forecast
D <- Dbuilding
D$Ta$k1[1:1500] <- NA
summary(D)

# Vector with observations not same length as t throws error
D <- Dbuilding
D$heatload <- D$heatload[1:10]
try(summary(D))

# Forecasts wrong count
D <- Dbuilding
D$Ta <- D$Ta[1:10, ]
try(summary(D))

# Wrong column names
D <- Dbuilding
names(D$Ta)[4] <- "xk"
names(D$Ta)[2] <- "x2"
try(summary(D))

# No column names
D <- Dbuilding
names(D$Ta) <- NULL
try(summary(D))

# Don't stop or only print if stopped
onlineforecast:::summary.data.list(D, stopit=FALSE)
try(onlineforecast:::summary.data.list(D, printit=FALSE))

# Only check for specified variables
# (e.g. do like this in model functions to check only variables used in model)
onlineforecast:::summary.data.list(D, nms=c("heatload","I"))
```

%**% *Multiplication of list with y, elementwise*

Description

Multiplication of each element in a list (x) with y

Usage

```
x %**% y
```

Arguments

x	a list of matrices, data.frames, etc.
y	a vector, data.frame or matrix

Details

Each element of x is multiplied with y using the usual elementwise '*' operator.

Typical use is when a function, e.g. `bspline()`, returns a list of matrices (e.g. one for each base spline) and they should individually be multiplied with y (a vector, matrix, etc.).

Since this is intended to be used for forecast models in the transformation stage then there are some percularities:

If the number of columns or the names of the columns are not equal for one element in x and y, then only the columns with same names are used, hence the resulting matrices can be of lower dimensions.

See the example https://onlineforecasting.org/examples/solar-power-forecasting.html where the operator is used.

Value

A list of same length of x

Examples

```
x <- list(matrix(1:9,3), matrix(9:1,3))
x

y <- matrix(2,3,3)
y

x %**% y

y <- 1:3

x %**% y

# Naming percularity
nams(x[[1]]) <- c("k1","k2","k3")
nams(x[[2]]) <- c("k2","k3","k4")
y <- matrix(2,3,3)
nams(y) <- c("k1","k3","k7")

# Now the only the horizons matching will be used
x %**% y
```

Index