Package 'hicp'

May 8, 2026

Type Package

Title Harmonised Index of Consumer Prices

Version 1.1.0

Description The Harmonised Index of Consumer Prices (HICP) is the key economic figure to measure inflation in the euro area.
The methodology underlying the HICP is documented in the HICP Methodological Manual (<https://ec.europa.eu/eurostat/web/products-manuals-and-guidelines/w/ks-gq-24-003>).
Based on the manual, this package provides functions to access and work with HICP data from Eurostat's public database (<https://ec.europa.eu/eurostat/data/database>).

License EUPL

Encoding UTF-8

Depends R (>= 3.5.0)

Imports restatapi (>= 0.24.0), data.table (>= 1.16.0)

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation no

URL https://github.com/eurostat/hicp

BugReports https://github.com/eurostat/hicp/issues

Author Sebastian Weinand [aut, cre]

Maintainer Sebastian Weinand <sebastian.weinand@ec.europa.eu>

Repository CRAN

Date/Publication 2026-02-03 12:40:20 UTC

Contents

`chaining` *Chain-linking, rebasing and index conversion*

Description

The function `unchain()` unchains an annually chain-linked index series. The unchained indices can be aggregated into higher-level indices, which can be chain-linked using the function `chain()` to obtain a long-term index series. The function `rebase()` sets the reference period of the chain-linked index.

The function `convert()` allows to convert monthly indices into quarterly and yearly indices or 12-month moving averages.

Usage

```
unchain(x, t, by=12, settings=list())

chain(x, t, by=12, settings=list())

rebase(x, t, t.ref="first", settings=list())

convert(x, t, type="year", settings=list())
```

Arguments

`x`	numeric vector of index values.
`t`	date vector of monthly (i.e., one observation per month), quarterly or yearly frequency in format `YYYY-MM-DD`.
`by`	for one-month or one-quarter overlap a single integer between 1 and 12 specifying the price reference period; for annual overlap using a full calendar year `NULL`.
`t.ref`	character specifying the index reference period either in format `YYYY` for a calendar year or `YYYY-MM` for a specific month or quarter. Can also be `first` or `last` to use the first or last available period. If `t.ref` contains multiple entries, these are processed in the order provided, and the first match is used for the rebasing.
`type`	type of converted index. Either `year` (for annual average), `quarter` (for quarterly average), or `12mavg` (for a 12-month moving average).

`settings` list of control settings to be used. The following settings are supported:

- `chatty` : logical indicating if package-specific warnings and info messages should be printed or not. The default is `getOption("hicp.chatty")`.
- `freq` : character specifying the frequency of `t`. Allowed values are `month`, `quarter`, `year`, and `auto` (the default). For `auto`, the frequency is internally derived from `t`.
- `na.rm` : logical indicating if averages for calendar years should also be computed when there are NAs and less than 12 months (or 4 quarters) present (for `na.rm=TRUE`). For the 12-month moving average in `convert()`, the calculations are always based on the last 12 months (or 4 quarters), meaning that only NAs are excluded. The default is `na.rm=FALSE`.

Details

The function `unchain()` sets the value of the first price reference period to `NA` although the value could be set to 100 (if by is not `NULL`) or 100 divided by the average of the year (if `by=NULL`). This is wanted to avoid aggregation of these values. The function `chain()` finally sets the values back to 100.

Value

The functions `unchain()`, `chain()`, `rebase()`, and `convert(..., type="12mavg")` return numeric values of the same length as `x`.

For `type="year"` and `type="quarter"`, the function `convert()` returns a named numeric vector of the length of quarters or years available in `t`, where the names correspond to the last month of the year or quarter.

Author(s)

Sebastian Weinand

References

European Commission, Eurostat, *Harmonised Index of Consumer Prices (HICP) - Methodological Manual - 2024 edition*, Publications Office of the European Union, 2024, doi:10.2785/055028.

See Also

aggregate

Examples

```
### EXAMPLE 1

# sample monthly price index:
t <- seq.Date(from=as.Date("2022-12-01"), to=as.Date("2025-12-01"), by="1 month")
p <- rnorm(n=length(t), mean=100, sd=5)

# rebase index to new reference period:
rebase(x=p, t=t, t.ref=c("1996","2023")) # 1996 not present so 2023 is used
```

```
rebase(x=p, t=t, t.ref=c("1996","first")) # 1996 not present so first period is used

# convert into quarterly index:
convert(x=p, t=t, type="q") # first quarter is not complete so NA

# unchaining and chaining gives initial results:
100*p/p[1]
chain(unchain(p, t, by=12), t, by=12)

# use annual overlap:
100*p/mean(p[1:12])
(res <- chain(unchain(p, t, by=NULL), t, by=NULL))
# note that for backwards compability, each month in the first
# year receives an index value of 100. this allows the same
# computation again:
chain(unchain(res, t, by=NULL), t, by=NULL)

### EXAMPLE 2: Working with published HICP data

library(data.table)
library(restatapi)
options(restatapi_cores=1) # set cores for testing on CRAN
options(hicp.chatty=FALSE) # suppress package messages and warnings

# import monthly price indices for euro area:
dtm <- hicp::data(id="prc_hicp_minr", filters=list(unit="I25", geo="EA"))
dtm[, "time":=as.Date(paste0(time, "-01"))]
setkeyv(x=dtm, cols=c("unit","coicop18","time"))

# unchain, chain, and rebase all euro area indices by COICOP:
dtm[, "dec_ratio" := unchain(x=values, t=time), by="coicop18"]
dtm[, "chained_index" := chain(x=dec_ratio, t=time), by="coicop18"]
dtm[, "index_own" := rebase(x=chained_index, t=time, t.ref="2025"), by="coicop18"]

# convert all monthly indices into annual averages:
dta <- dtm[, as.data.table(
              x=convert(x=values, t=time, type="year"),
              keep.rownames=TRUE), by="coicop18"]
setnames(x=dta, c("coicop18","time","index"))
plot(index~as.Date(time), data=dta[coicop18=="TOTAL",], type="l")
```

`coicop` *COICOP codes and special aggregates*

Description

Valid COICOP codes can be flagged by `is.coicop()`. The function `level()` returns their level (e.g. 2-digit division or 5-digit subclass level).

For HICP data, special aggregates like food or energy have their own codes, which can be flagged by `is.spec.agg()`. The function `spec.agg()` provides the composition of special aggregates.

The function `label()` translates the codes into their descriptions.

Usage

```
# flag COICOP, bundle and special aggregate codes:
is.coicop(id, settings=list())
is.bundle(id, settings=list())
is.spec.agg(id, settings=list())

# derive the level of COICOP codes:
level(id, settings=list())

# label codes:
label(id, settings=list())

# get the composition of a special aggregate:
spec.agg(id=NULL, settings=list())
```

Arguments

`id`	character vector of COICOP or special aggregate codes.
`settings`	list of control settings to be used. The following settings are supported: • `coicop.version` : character specifying the COICOP version to be used. See details for the allowed values. The default is `getOption("hicp.coicop.version")`. • `coicop.prefix` : character specifying a prefix for the COICOP codes. The default is `getOption("hicp.coicop.prefix")`. • `all.items.code` : character specifying the code internally used for the all-items index. The default is taken from `getOption("hicp.all.items.code")`. If the character is named, the function `label()` uses the name as the label of the all-items code.

Details

The following COICOP versions are supported:

- Classification of Individual Consumption According to Purpose (COICOP-1999): `coicop1999`
- European COICOP (version 1, ECOICOP): `ecoicop1`
- ECOICOP adapted to the needs of the HICP (version 1, ECOICOP-HICP): `ecoicop1.hicp`
- COICOP-2018 (including the voluntary 6-digit codes): `coicop2018`
- ECOICOP (version 2, ECOICOP 2): `ecoicop2`
- ECOICOP adapted to the needs of the HICP (version 2, ECOICOP 2 HICP): `ecoicop2.hicp`

The COICOP version can be set temporarily in the function settings or globally via `options("hicp.coicop.version")`. For `ecoicop1.hicp`, some COICOP codes are merged into bundles (e.g. `08X`, `0531_2`), deviating from the expected code structure. Although such bundle codes are no valid COICOP codes, they are internally resolved into their underlying codes and processed in that way if they can be found in the package's internal bundle code dictionary.

None of the COICOP versions include a code for the all-items index. The internal package code for the all-items index is defined by `options("hicp.all.items.code")`. Its level is always 1 and the label `All-items` by default.

For HICP data from Eurostat's database, COICOP codes are usually prefixed by "CP". Whether such a prefix is expected can be defined in `options("hicp.coicop.prefix")`.

Value

The functions `is.coicop()`, `is.bundle()` and `is.spec.agg()` return a logical vector, the function `level()` an integer vector and the function `label()` a character vector. All function outputs have the same length as `id`.

The function `spec.agg()` returns a named list with the special aggregate composition.

Author(s)

Sebastian Weinand

References

European Commission, Eurostat, *Harmonised Index of Consumer Prices (HICP) - Methodological Manual - 2024 edition*, Publications Office of the European Union, 2024, Annex 11.1, doi:10.2785/055028.

See Also

`child`, `parent`, `tree`

Examples

```
# example codes:
ids <- c("TOTAL","CP01","CP011","CP99","FOOD","NRG")
ids[is.coicop(ids)] # all-items is no valid COICOP code
ids[is.spec.agg(ids)]
level(ids) # special aggregates without level
label(ids)
```

---

coicop.relatives *COICOP relatives*

Description

The functions `parent()` and `child()` derive the higher-level parents or lower-level children of a COICOP code.

Usage

```
child(id, usedict=TRUE, closest=TRUE, k=1, settings=list())

parent(id, usedict=TRUE, closest=TRUE, k=1, settings=list())
```

Arguments

id	character vector of COICOP codes.
usedict	logical indicating if parents or children should be derived from the full code dictionary defined by settings$coicop.version (if set to TRUE) or only from the codes present in id.
closest	logical indicating if the closest parents or children should be derived or the *k*-th ones defined by k. For example, if set to TRUE, the closest parent could be the direct parent for one code (e.g. 031->03) and the grandparent for another (e.g. 0321->03).
k	integer specifying the *k*-th relative (e.g., 1 for direct parents or children, 2 for grandparents and grandchildren, ...). Multiple values allowed, e.g., k=c(1,2). Only relevant if closest=FALSE.
settings	list of control settings to be used. The following settings are supported: • coicop.version : character specifying the COICOP version to be used. See coicop for the allowed values. The default is getOption("hicp.coicop.version"). • coicop.prefix : character specifying a prefix for the COICOP codes. The default is getOption("hicp.coicop.prefix"). • all.items.code : character specifying the code internally used for the all-items index. The default is taken from getOption("hicp.all.items.code"). • simplify : logical indicating if the output should be simplified into a vector if possible. The default is FALSE. If both a COICOP bundle code and the underlying codes are the parent or child, only the latter codes are kept (e.g., c(08X,082)->082). Note that simplification usually only works for parent().

Value

A list with the same length as id.

Author(s)

Sebastian Weinand

See Also

coicop, tree

Examples

```
# no children of CP01 present in id:
child(id="CP01", usedict=FALSE)

# still no direct child present:
child(id=c("CP01","CP0111"), usedict=FALSE, closest=FALSE, k=1)

# but a grandchild of CP01 is found:
child(id=c("CP01","CP0111"), usedict=FALSE, closest=TRUE)
```

```
# now, get the children directly from the code dictionary:
child(id=c("CP01","CP0111"), usedict=TRUE)

# which works analogously for parents:
parent(id=c("CP01","CP0111"), usedict=TRUE)
```

`countries` *European aggregates and countries*

Description

The function `countries()` lists all countries with available HICP data in their protocol order.

The functions `is.ea()`, `is.eu()` and `is.eea()` indicate if a country belongs to one of the European aggregate at a specific date.

Usage

```
countries(group="All", t=Sys.Date())

is.ea(id, t=Sys.Date())

is.eu(id, t=Sys.Date())

is.eea(id, t=Sys.Date())
```

Arguments

`group`	character specifying the scope. Allowed values are `All` for all countries, `EA` for euro area countries, `EU` for the Member States of the European Union and `EEA` for members of the European Economic Area.
`id`	character vector of country codes.
`t`	date vector in format `YYYY-MM-DD`. Must be of length one for `countries()`.

Details

The functions `is.ea()`, `is.eu()` and `is.eea()` can be used to flag the evolving composition of European aggregates by letting `t` advance in time. By contrast, for the fixed composition of European aggregates at a specific date, `t` can be fixed at this date.

Value

For `countries()`, a named vector of countries.

For `is.ea()`, `is.eu()` and `is.eea()`, a logical vector of the same length as `id`.

Author(s)

Sebastian Weinand

References

European Commission, Eurostat, *Harmonised Index of Consumer Prices (HICP) - Methodological Manual - 2024 edition*, Publications Office of the European Union, 2024, Annex 11.2, doi:10.2785/055028.

Examples

```
# EU member states in 2025:
countries(group="EU", t=as.Date("2025-01-01"))

# check which of the following countries belong to euro area in 2025:
is.ea(id=c("HR","BG"), t=as.Date("2025-01-01"))

# check over time:
is.ea(id="BG", t=as.Date(c("2024-01-01","2025-01-01","2026-01-01")))
```

`hicp.data` *Download HICP data*

Description

These functions are simple wrappers of functions in the `restatapi` package.

The function `datasets()` lists all available HICP data sets in Eurostat's public database, while the function `datafilters()` gives the allowed values that can be used for filtering a data set. The function `data()` downloads a specific data set with filtering on key parameters and time, if supplied.

Usage

```
datasets(pattern="^prc_hicp", ...)

datafilters(id, ...)

data(id, filters=list(), date.range=NULL, flags=FALSE, ...)
```

Arguments

`pattern`	character for pattern matching on data set identifier. See also `grepl`.
`id`	data set identifier, which can be obtained from `datasets()`.
`filters`	named list of filters to be applied to the data request. Allowed values for filtering can be retrieved from `datafilters()`. For HICP data, typical filter variables are the index reference period (`unit: I15, I25`), the country (`geo: EA, DE, FR, ...`) or the COICOP code (`coicop18: CP01, CP02, SERV, ...`).
`date.range`	character vector of start and end date used for filtering on time dimension. These must follow the pattern `YYYY(-MM)?`. An open interval can be defined by setting one date to `NA`.
`flags`	logical indicating if data flags should be returned or not.

... further arguments that can be passed to the functions:

- get_eurostat_toc for datasets()
- get_eurostat_dsd for datafilters()
- get_eurostat_data for data()

Value

A data.table.

Author(s)

Sebastian Weinand

Source

See Eurostat's public database at https://ec.europa.eu/eurostat/web/main/data/database.

Examples

```
# set cores for testing on CRAN:
library(restatapi)
options(restatapi_cores=1)

# view available HICP data sets:
datasets()

# get allowed filters for item weights:
datafilters(id="prc_hicp_iw")

# download item weights since 2015 for euro area:
data(id="prc_hicp_iw", filters=list("geo"="EA"), date.range=c("2015", NA))
```

---

index.aggregation *Index number functions and aggregation*

Description

Lower-level price indices can be aggregated into higher-level indices in a single step using the bilateral index formulas below or gradually following the COICOP tree structure with the function aggregate.tree().

The functions aggregate() and disaggregate() can be used for the calculation of user-defined aggregates (e.g., HICP special aggregates). For aggregate(), lower-level indices are aggregated into the respective total. For disaggregate(), they are deducted from the total to receive a subaggregate.

Usage

```
# bilateral price index formulas:
jevons(x)
carli(x)
harmonic(x)
laspeyres(x, w0)
paasche(x, wt)
fisher(x, w0, wt)
toernqvist(x, w0, wt)
walsh(x, w0, wt)

# aggregation into user-defined aggregates:
aggregate(x, w0, wt, id, formula=laspeyres, agg=list(), settings=list())

# disaggregation into user-defined aggregates:
disaggregate(x, w0, id, agg=list(), settings=list())

# gradual aggregation following the COICOP tree:
aggregate.tree(x, w0, wt, id, formula=laspeyres, settings=list())
```

Arguments

`x`	numeric vector of price relatives between two periods, typically obtained from `unchain()`.
`w0`, `wt`	numeric vector of weights in the base period `w0` (e.g., for the Laspeyres index) or current period `wt` (e.g., for the Paasche index).
`id`	character vector of aggregate codes. For `aggregate.tree()`, only valid COICOP (bundle) codes are processed.
`formula`	function or named list of functions specifying the index formula(s) used for aggregation. Each function must return a scalar and have the argument `x`. For weighted index formulas, the arguments `w0` and/or `wt` must be available as well.
`agg`	list of user-defined aggregates to be calculated. For `disaggregate()`, the list must have names specifying the aggregate from which indices are deducted. Each list element is a vector of codes that can be found in `id`. See `settings$exact` for further specification of this argument.
`settings`	list of control settings to be used. The following settings are supported:

- `chatty` : logical indicating if package-specific warnings and info messages should be printed or not. The default is `getOption("hicp.chatty")`.
- `coicop.version` : character specifying the COICOP version to be used. See `coicop` for the allowed values. The default is `getOption("hicp.coicop.version")`.
- `coicop.prefix` : character specifying a prefix for the COICOP codes. The default is `getOption("hicp.coicop.prefix")`.
- `all.items.code` : character specifying the code internally used for the all-items index. The default is taken from `getOption("hicp.all.items.code")`.
- `exact` : logical indicating if the codes in `agg` must all be present in `id` for aggregation or not. If `FALSE`, aggregation is carried out using the codes

present in agg. If TRUE and some codes cannot be found in id, NA is returned. The default is TRUE.

- names : character of names for the aggregates in agg. If not supplied, the aggregates are numbered.

Details

The bilateral index formulas currently available are intended for the aggregation of (unchained) price relatives x. The Dutot index is therefore not implemented.

Value

The functions jevons(), carli(), harmonic(), laspeyres(), paasche(), fisher(), toernqvist(), and walsh() return a single aggregated value.

The functions aggregate(), disaggregate() and aggregate.tree() return a data.table with the sum of weights w0 and wt (if supplied) and the computed aggregates for each index formula specified by formula.

Author(s)

Sebastian Weinand

References

European Commission, Eurostat, *Harmonised Index of Consumer Prices (HICP) - Methodological Manual - 2024 edition*, Publications Office of the European Union, 2024, doi:10.2785/055028.

See Also

unchain, tree, spec.agg

Examples

```
library(data.table)

### EXAMPLE 1

# example data with unchained prices and weights:
dt <- data.table("coicop"=c("CP0111","CP0112","CP012","CP021","CP022"),
                 "price"=c(102,105,99,109,115)/100,
                 "weight"=c(0.2,0.15,0.4,0.2,0.05))

# aggregate directly into overall index:
dt[, laspeyres(x=price, w0=weight)]

# same result at top level with gradual aggregation:
(dtagg <- dt[, aggregate.tree(x=price, w0=weight, id=coicop)])

# compute user-defined aggregates by disaggregation:
dtagg[, disaggregate(x=laspeyres, w0=w0, id=id,
                     agg=list("TOTAL"=c("CP01"), "TOTAL"=c("CP022")),
```

```
                    settings=list(names=c("A","B")))]

# which can be similarly derived by aggregation:
dtagg[, aggregate(x=laspeyres, w0=w0, id=id,
                  agg=list(c("CP021","CP022"), c("CP011","CP012","CP021")),
                  settings=list(names=c("A","B")))]

# same aggregates by several index formulas:
dtagg[, aggregate(x=laspeyres, w0=w0, id=id,
                  agg=list(c("CP021","CP022"), c("CP011","CP012","CP021")),
                  formula=list("lasp"=laspeyres, "jev"=jevons, "mean"=mean),
                  settings=list(names=c("A","B")))]

# no aggregation if one index is missing:
dtagg[, aggregate(x=laspeyres, w0=w0, id=id,
                  agg=list(c("CP01","CP02","CP03")),
                  settings=list(exact=TRUE))]

# or just use the available ones:
dtagg[, aggregate(x=laspeyres, w0=w0, id=id,
                  agg=list(c("CP01","CP02","CP03")),
                  settings=list(exact=FALSE))]

### EXAMPLE 2: Index aggregation using published HICP data

library(restatapi)
options(restatapi_cores=1)  # set cores for testing on CRAN
options(hicp.chatty=FALSE)  # suppress package messages and warnings

# import monthly price indices for euro area since 2014:
dtp <- hicp::data(id="prc_hicp_minr",
                  date.range=c("2014-12", NA),
                  filters=list(unit="I25", geo="EA"))
dtp[, "time":=as.Date(paste0(time, "-01"))]
dtp[, "year":=as.integer(format(time, "%Y"))]
setnames(x=dtp, old="values", new="index")

# unchain all indices for aggregation:
dtp[, "dec_ratio" := unchain(x=index, t=time), by="coicop18"]

# import euro area item weights since 2014:
dtw <- hicp::data(id="prc_hicp_iw",
                  date.range=c("2014",NA),
                  filters=list(geo="EA"))
dtw[, "time":=as.integer(time)]
setnames(x=dtw, old=c("time","values"), new=c("year","weight"))

# merge price indices and item weights:
dtall <- merge(x=dtp, y=dtw, by=c("geo","coicop18","year"), all.x=TRUE)
dtall <- dtall[year <= year(Sys.Date())-1,]

# derive COICOP tree at lowest possible level:
dtall[weight>0 & !is.na(dec_ratio),
```

```
      "tree":=tree(id=coicop18, w=weight, flag=TRUE, settings=list(w.tol=0.1)),
      by="time"]

# except for rounding, we receive total weight of 1000 in each period:
dtall[tree==TRUE, sum(weight), by="time"]

# (1) compute all-items HICP in one step using only lowest-level indices:
hicp.own <- dtall[tree==TRUE,
                  list("laspey"=laspeyres(x=dec_ratio, w0=weight)),
                  by="time"]
setorderv(x=hicp.own, cols="time")
hicp.own[, "chain_laspey" := chain(x=laspey, t=time, by=12)]
hicp.own[, "chain_laspey_25" := rebase(x=chain_laspey, t=time, t.ref="2025")]

# (2) compute all-items HICP gradually through all higher-levels:
hicp.own.all <- dtall[weight>0 & !is.na(dec_ratio),
                      aggregate.tree(x=dec_ratio, w0=weight, id=coicop18),
                      by="time"]
setorderv(x=hicp.own.all, cols="time")
hicp.own.all[, "chain_laspey" := chain(x=laspeyres, t=time, by=12), by="id"]
hicp.own.all[, "chain_laspey_25" := rebase(x=chain_laspey, t=time, t.ref="2025"), by="id"]

# (3) compare all-items HICP from direct and gradual aggregation:
all(abs(hicp.own.all[id=="TOTAL", chain_laspey_25]-hicp.own$chain_laspey_25)<0.1)
# no differences -> consistent in aggregation
```

`linking` *Linking-in new index series*

Description

The function `link()` links a new index series (`x.new`) to an existing one (`x`) using the overlap periods in `t.overlap`. In the resulting linked index series, the new index series starts after the existing one.

The function `lsf()` computes the level-shift factors for linking via the overlap periods in `t.overlap` in comparison to the standard one-month overlap method using December of year `t-1`. The level-shift factors can then be used to shift the index level of a HICP index series.

Usage

```
link(x, x.new, t, t.overlap=NULL, settings=list())

lsf(x, x.new, t, t.overlap=NULL, settings=list())
```

Arguments

`x`, `x.new`	numeric vector of index values. `NA`-values in the vectors indicate when the index series discontinues (for `x`) or starts (for `x.new`).

t — date vector of monthly (i.e., one observation per month), quarterly or yearly frequency in format YYYY-MM-DD.

t.overlap — character specifying the overlap period either in format YYYY for a calendar year or YYYY-MM for a specific month or quarter. Multiple periods can be provided. If NULL, all intersecting periods in x and x.new are used.

settings — list of control settings to be used. The following settings are supported:

- chatty : logical indicating if package-specific warnings and info messages should be printed or not. The default is getOption("hicp.chatty").
- freq : character specifying the frequency of t. Allowed values are month, quarter, year, and auto (the default). For auto, the frequency is internally derived from t.
- na.rm : logical indicating if averages for calendar years should also be computed when there are NAs and less than 12 months (or 4 quarters) present (for na.rm=TRUE).

Value

The function link() returns a numeric vector or a matrix of the same length as t, while lsf() provides a named numeric vector of the same length as t.overlap.

Author(s)

Sebastian Weinand

See Also

chain

Examples

```
# input data:
set.seed(1)
t <- seq.Date(from=as.Date("2015-01-01"), to=as.Date("2024-05-01"), by="1 month")
x.new <- rnorm(n=length(t), mean=100, sd=5)
x.new <- rebase(x=x.new, t=t, t.ref="2019-12")
x.old <- x.new + rnorm(n=length(x.new), sd=5)
x.old <- rebase(x=x.old, t=t, t.ref="2015")
x.old[t>as.Date("2021-12-01")] <- NA # current index discontinues in 2021
x.new[t<as.Date("2020-01-01")] <- NA # new index starts in 2019-12

# linking in new index in different periods:
matplot(x=t,
        y=link(x=x.old, x.new=x.new, t=t, t.overlap=c("2021-12","2020","2021")),
        col=c("red","blue","green"), type="l", lty=1,
        xlab=NA, ylab="Index", ylim=c(80,120))
lines(x=t, y=x.old, col="black")
abline(v=as.Date("2021-12-01"), lty="dashed")
legend(x="topleft",
       legend=c("One-month overlap using December 2021",
                "Annual overlap using 2021",
```

```
                "Annual overlap using 2020"),
       fill=c("red","green","blue"), bty = "n")

# compute level-shift factors:
lsf(x=x.old, x.new=x.new, t=t, t.overlap=c("2020","2021"))

# level-shift factors can be applied to already chain-linked index series
# to obtain linked series using another overlap period:
x.new.chained <- link(x=x.old, x.new=x.new, t=t, t.overlap="2021-12")

# level-shift adjustment:
x.new.adj <- ifelse(test=t>as.Date("2021-12-01"),
                    yes=x.new.chained*lsf(x=x.old, x.new=x.new, t=t, t.overlap="2020"),
                    no=x.new.chained)

# compare:
all.equal(x.new.adj, link(x=x.old, x.new=x.new, t=t, t.overlap="2020"))
```

`rates` *Change rates and contributions*

Description

The function `rates()` derives monthly, quarterly and annual rates of change from an index series.

The function `contrib()` computes the contributions of a subcomponent (e.g., food, energy) to the change rate of the overall index (for chained indices with price reference period December of the previous year).

Usage

```
rates(x, t, type="year", settings=list())

contrib(x, w, t, x.all, w.all, type="year", settings=list())
```

Arguments

`x`, `x.all`	numeric vector of index values of the subcomponent (`x`) and the overall index (`x.all`).
`w`, `w.all`	numeric vector of weights of the subcomponent (`w`) and the overall index (`w.all`).
`t`	date vector of monthly (i.e., one observation per month), quarterly or yearly frequency in format `YYYY-MM-DD`.
`type`	character specifying the type of change rate. Allowed values are `month` for monthly change rates, `quarter` for quarterly change rates, and `year` for annual change rates. See also details.
`settings`	list of control settings to be used. The following settings are supported: • `chatty` : logical indicating if package-specific warnings and info messages should be printed or not. The default is `getOption("hicp.chatty")`.

- `freq` : character specifying the frequency of `t`. Allowed values are `month`, `quarter`, `year`, and `auto` (the default). For `auto`, the frequency is internally derived from `t`.
- `method` : character specifying the method for decomposing the change rates. Allowed values are `ribe` (the default) and `kirchner`.

Details

For monthly frequency, the change rates show the percentage change of `x` in the current month compared to the previous month (monthly change rates, *m-1*), compared to three months ago (quarterly change rates, *m-3*), or compared to the same month one year before (annual change rates, *m-12*).

For quarterly frequency, the change rates show the percentage change of `x` in the current quarter compared to the previous quarter (quarterly change rates, *q-1*) or compared to the same quarter one year before (annual change rates, *q-4*).

For yearly frequency, the change rates show the percentage change of `x` in the current year compared to the previous year (annual change rates, *y-1*). If `x` is an annual index produced by `convert()`, the annual change rates correspond to annual average change rates.

Value

A numeric vector of the same length as `x`.

Author(s)

Sebastian Weinand

References

European Commission, Eurostat, *Harmonised Index of Consumer Prices (HICP) - Methodological Manual - 2024 edition*, Publications Office of the European Union, 2024, doi:10.2785/055028.

Examples

```
### EXAMPLE 1

# sample monthly price index:
t <- seq.Date(from=as.Date("2022-12-01"), to=as.Date("2025-12-01"), by="1 month")
p <- rnorm(n=length(t), mean=100, sd=5)

# compute change rates:
rates(x=p, t=t, type="month") # one month to the previous month
rates(x=p, t=t, type="year")  # month to the same month of previous year

# compute annual average rate of change:
pa <- convert(x=p, t=t, type="y") # now annual frequency
rates(x=pa, t=as.Date(names(pa)), type="year")

# compute 12-month average rate of change:
pmvg <- convert(x=p, t=t, type="12mavg") # still monthly frequency
rates(x=pmvg, t=t, type="year")
```

```
### EXAMPLE 2: Ribe contributions using published HICP data

library(data.table)
library(restatapi)
options(restatapi_cores=1)  # set cores for testing on CRAN
options(hicp.chatty=FALSE)  # suppress package messages and warnings

# import monthly price indices:
dtp <- hicp::data(id="prc_hicp_minr", filters=list(unit="I25", geo="EA"))
dtp[, "time":=as.Date(paste0(time, "-01"))]
dtp[, "year":=as.integer(format(time, "%Y"))]
setnames(x=dtp, old="values", new="index")

# import item weights:
dtw <- hicp::data(id="prc_hicp_iw", filters=list(geo="EA"))
dtw[, "time":=as.integer(time)]
setnames(x=dtw, old=c("time","values"), new=c("year","weight"))

# merge price indices and item weights:
dtall <- merge(x=dtp, y=dtw, by=c("geo","coicop18","year"), all.x=TRUE)

# add all-items hicp:
dtall <- merge(x=dtall,
               y=dtall[coicop18=="TOTAL", list(geo,time,index,weight)],
               by=c("geo","time"), all.x=TRUE, suffixes=c("","_all"))

# Ribe contributions by COICOP:
dtall[, "ribe" := contrib(x=index, w=weight, t=time,
                          x.all=index_all, w.all=weight_all,
                          type="year", settings=list(method="ribe")), by="coicop18"]

# plot annual change rates over time:
plot(rates(x=index, t=time, type="year")~time,
     data=dtall[coicop18=="TOTAL",],
     type="l", ylim=c(-2,12))

# add contribution of energy to plot:
lines(ribe~time, data=dtall[coicop18=="NRG"], col="red")
```

`tree` *COICOP tree*

Description

Following the tree structure of COICOP, the function `tree()` derives from a given set of COICOP codes those at the lowest possible level. This can be particularly useful for aggregating from the lowest to the highest level in a single step.

Usage

```
tree(id, by=NULL, w=NULL, flag=FALSE, settings=list())
```

Arguments

id	character vector of COICOP codes.
by	vector specifying the variable to be used for merging the derived COICOP codes, e.g., a vector of dates to obtain the same composition of COICOP codes over time. If by=NULL, no merging is performed.
w	numeric weight of id. If supplied, it is checked that the weights of children add up to the weight of their parent (allowing for tolerance settings$w.tol). If w=NULL, no checking of weight aggregation is performed.
flag	logical specifying the function output. For FALSE, a list with the COICOP codes at each level. For TRUE, a logical vector of the same length as id indicating which COICOP codes in id define the lowest level.
settings	list of control settings to be used. The following settings are supported:

- chatty : logical indicating if package-specific warnings and info messages should be printed or not. The default is getOption("hicp.chatty").
- coicop.version : character specifying the COICOP version to be used. See coicop for the allowed values. The default is getOption("hicp.coicop.version").
- coicop.prefix : character specifying a prefix for the COICOP codes. The default is getOption("hicp.coicop.prefix").
- all.items.code : character specifying the code internally used for the all-items index. The default is taken from getOption("hicp.all.items.code").
- max.lvl : integer specifying the maximum (or deepest) COICOP level allowed. If NULL (the default), the maximum level found in id is used.
- w.tol : numeric tolerance for checking of weights. Only relevant if w is not NULL. The default is 1/100.

Details

The derivation of COICOP codes at the lowest level follows a top-down-approach. Starting from the top level of the COICOP tree (usually the all-items code), it is checked if

1. the code has children in id,
2. the children's weights correctly add up to the weight of the parent (if w provided),
3. all children can be found in all the groups in by (if by provided).

Only if all three conditions are met, the children are stored and further processed following the same three steps. Otherwise, the parent is kept and the processing stops in the respective node. This process is followed until the lowest level of all codes is reached.

If by is provided, the function tree() first subsets all codes in id to the intersecting levels. This ensures that the derivation of the COICOP codes does not directly stops if, for example, the all-items code is missing in one of the groups in by. For example, assume the codes (00,01,02,011,012,021) for by=1 and (01,011,012,021) for by=2. In this case, the code 00 would be dropped internally first because its level is not available for by=2. The other codes would be processed since their levels intersect across by. However, since (01,02) do not fulfill the third check, the derivation would stop and no merged tree would be available though codes (011,012,021) seem to be a solution.

Value

Either a list (for `flag=FALSE`) or a logical vector of the same length as `id` (for `flag=TRUE`).

Author(s)

Sebastian Weinand

See Also

`child`, `coicop`, `parent`

Examples

```
# example codes:
ids <- c("CP01","CP011","CP012","CP0111","CP0112")

# derive lowest level of COICOP tree from top to bottom:
tree(ids) # (CP0111,CP0112,CP012) at lowest level

# or just flag lowest level:
tree(ids, flag=TRUE)

# still same codes because weights add up:
tree(id=ids, w=c(0.2,0.08,0.12,0.05,0.03))

# now (CP011,CP012) because weights do not correctly add up at lower levels:
tree(id=ids, w=c(0.2,0.08,0.12,0.05,0.01))

# again (CP011,CP012) because maximum COICOP level limited to 3 digits:
tree(id=c(ids,"01121"),
     w=c(0.2,0.08,0.12,0.02,0.06,0.06),
     settings=list(max.lvl=3))

# merge (or fix) COICOP tree over groups:
tree(id=c("TOTAL","CP01","CP02","CP011","CP012",
          "TOTAL","CP01","CP02","CP011"),
     by=c(1,1,1,1,1, 2,2,2,2),
     w=c(1,0.3,0.7,0.12,0.18, 1,0.32,0.68,0.15))
# for by=1, the lowest level would be (CP011,CP012,CP02).
# however, CP012 is missing for by=2. therefore, the merged
# COICOP tree consists of (CP01,CP02) at the lowest level.
```

Index