Package ‘gpindex’

October 13, 2022

Title Generalized Price and Quantity Indexes

Version 0.4.3

Description A small package for calculating lots of different price indexes, and by extension quantity indexes. Provides tools to build and work with any type of bilateral generalized-mean index (of which most price indexes are), along with a few important indexes that don't belong to the generalized-mean family (e.g., superlative quadratic-mean indexes, GEKS). Implements and extends many of the methods in Balk (2008, ISBN:978-1-107-40496-0), von der Lippe (2001, ISBN:3-8246-0638-0), and the CPI manual (2020, ISBN:978-1-51354-298-0) for bilateral price indexes.

Depends R (>= 3.5)

Imports stats, utils

Suggests IndexNumR

License MIT + file LICENSE

Encoding UTF-8

URL `https://github.com/marberts/gpindex`

LazyData true

Collate 'helpers.R' 'means.R' 'weights.R' 'contributions.R' 'price-indexes.R' 'geks.R' 'operators.R' 'offset-prices.R' 'outliers.R'

NeedsCompilation no

Author Steve Martin [aut, cre, cph] (<`https://orcid.org/0000-0003-2544-9480`>)

Maintainer Steve Martin <`stevemartin041@gmail.com`>

Repository CRAN

Date/Publication 2022-05-01 03:50:02 UTC

R topics documented:

`gpindex-package` *Generalized Price and Quantity Indexes*

Description

A small package for calculating lots of different price indexes, and by extension quantity indexes. Provides tools to build and work with any type of bilateral generalized-mean index (of which most price indexes are), along with a few important indexes that don't belong to the generalized-mean family (e.g., superlative quadratic-mean indexes, GEKS). Implements and extends many of the methods in Balk (2008), von der Lippe (2001), and the CPI manual (2020) for bilateral price indexes.

Background

Everything is framed as a price index to avoid duplication; it is trivial to turn a price index into its analogous quantity index by simply switching prices and quantities.

Generalized-mean price indexes (sometimes called generalized price indexes for short) are a large family of price indexes with nice properties, such as the mean-value and identity properties (e.g., Balk, 2008, Chapter 3). When used with value-share weights, these indexes satisfy the key homogeneity properties, commensurability, and are consistent in aggregation. This last feature makes generalized-mean indexes natural candidates for making national statistics, and this justifies the hierarchical structure used by national statistical agencies for calculating and disseminating collections of price indexes.

Almost all bilateral price indexes used in practice are either generalized-mean indexes (like the Laspeyres and Paasche index) or are nested generalized-mean indexes (like the Fisher index).

Usage

A generalized-mean price index is a weighted generalized mean of price relatives. Given a set of price relatives and weights, any generalized-mean price index is easily calculated with the `generalized_mean()` function. What distinguishes different generalized-mean price indexes are the weights and the order of the generalized mean. For example, the standard Laspeyres index uses base-period value-share weights in a generalized mean of order 1 (arithmetic mean). Changing the order of the generalized mean to $1 - \sigma$, where σ is an elasticity of substitution, gives a Lloyd-Moulton index, whereas changing the weights to current-period value-shares gives a Palgrave index. This is the essence of the atomistic approach in chapter 2 of Selvanathan and Rao (1994).

Generalized-mean indexes can also be nested together to get indexes like the Fisher, Drobisch, or AG mean index. The `nested_mean()` function is a simple wrapper for `generalized_mean()` for these cases.

Two important functions for decomposing generalized means are given by `transmute_weights()` and `factor_weights()`. These functions augment the weights in a generalized mean, and can be used to calculate quote contributions (with, e.g., `contributions()`) and price-update weights for generalized-mean indexes. Together these functions provide the key mathematical apparatus to work with any generalized-mean index, and those that nest generalized-mean indexes.

On top of these basic mathematical tools are functions for making standard price indexes when both prices and quantities are known. Weights for a large variety of indexes can be calculated with `index_weights()`, which can be plugged into the relevant generalized mean to calculate most common price indexes, and many uncommon ones. The `price_index` functions provide a simple wrapper, with the `quantity_index()` function turning each of these into its analogous quantity index.

There are utility functions for making price relatives (e.g., `back_period()`) when data are stored in a table, and identifying extreme price relatives (e.g., `resistant_fences()`) that may not be suitable for a price index. All functions can be made to work with grouped data with the `grouped()` operator.

Contribution

There are a number of R packages on the CRAN that implement the standard index-number formulas (e.g., **IndexNumber**, **productivity**, **IndexNumR**, **micEconIndex**, **PriceIndices**). While there is support for a large number of index-number formulas out-of-the box in this package, the focus is on the tools to easily make and work with any type of generalized-mean price index. Consequently, compared to existing packages, this package is suitable for building custom price/quantity indexes, calculating indexes with sample data, decomposing indexes, and learning about or researching different types of index-number formulas.

Author(s)

Maintainer: Steve Martin <`stevemartin041@gmail.com`>

References

Balk, B. M. (2008). *Price and Quantity Index Numbers*. Cambridge University Press.

ILO, IMF, OECD, Eurostat, UN, and World Bank. (2020). *Consumer Price Index Manual: Theory and Practice*. International Monetary Fund.

von der Lippe, P. (2001). *Chain Indices: A Study in Price Index Theory*, Spectrum of Federal Statistics vol. 16. Federal Statistical Office, Wiesbaden.

Selvanathan, E. A. and Rao, D. S. P. (1994). *Index Numbers: A Stochastic Approach*. MacMillan.

See Also

`https://github.com/marberts/gpindex`

`contributions` *Quote contributions*

Description

Calculate additive quote contributions for generalized-mean price indexes, and indexes that nest two levels of generalized means consisting of an outer generalized mean and two inner generalized means (e.g., the Fisher index).

Usage

```
## Function factories
contributions(r)

nested_contributions(r1, r2, t = c(1, 1))

nested_contributions2(r1, r2, t = c(1, 1))

## Special cases
arithmetic_contributions(x, w)

geometric_contributions(x, w)

harmonic_contributions(x, w)

fisher_contributions(x, w1, w2)

fisher_contributions2(x, w1, w2)
```

Arguments

`r, r1`	A finite number giving the order of the generalized mean.
`r2`	A pair of finite numbers giving the order of the inner generalized means.
`t`	A pair of strictly positive weights for the inner generalized means. The default is equal weights.
`x`	A strictly positive numeric vector of price relatives.
`w, w1, w2`	A strictly positive numeric vector of weights, the same length as `x`. The default is to equally weight each element of `x`.

Details

The function `contributions()` is a simple wrapper for `transmute_weights(r, 1)()` to calculate (additive) quote contributions for a price index based on a generalized mean of order `r`. It returns a function to compute a vector `v(x, w)` such that

```
generalized_mean(r)(x, w) - 1 == sum(v(x, w))
```

This generalizes the approach for calculating quote contributions in section 4.2 of Balk (2008) using the method by Martin (2021). The `arithmetic_contributions()`, `geometric_contributions()` and `harmonic_contributions()` functions cover the most important cases (i.e., `r = 1`, `r = 0`, and `r = -1`).

The `nested_contributions()` and `nested_contributions2()` functions are the analog of `contributions()` for an index based on a nested generalized mean with two levels, like a Fisher index. They return a function that calculates the contribution of each element of `x` when a generalized mean of order `r1` aggregates two generalized-mean indexes of `x` with orders `r2`, and weights `w1` and `w2`.

Unlike the case of a generalized-mean index, there are several ways to make contributions for an index based on nested generalized means. `nested_contributions()` uses a generalization of the algorithm in section 6 of Reinsdorf et al. (2002) by Martin (2021). `nested_contributions2()` generalizes the van IJzeren decomposition for the Fisher index (Balk, 2008, section 4.2.2) by constructing a weighted average of the contributions for both of the inner means with the approach by Martin (2021). In most cases the results are broadly similar.

The `fisher_contributions()` and `fisher_contributions2()` functions correspond to `nested_contributions(0, c(1, -1))()` and `nested_contributions2(0, c(1, -1))()`, and are appropriate for calculating quote contributions for a Fisher index.

Value

`contributions()` returns a function:

```
function(x, w){...}
```

This computes the additive contribution for each element of `x` in an index based on the generalized mean of order `r` with weights `w`.

`nested_contributions()` and `nested_contributions2()` return a function:

```
function(x, w1, w2){...}
```

This computes the additive contribution for each element of `x` when a generalized mean of order `r1` aggregates a generalized-mean index of order `r2[1]` with weights `w1` and a generalized-mean index of order `r2[2]` with weights `w2`.

`arithmetic_contributions()`, `geometric_contributions()`, and `harmonic_contributions()` each return a numeric vector, the same length as `x`, giving the contribution of each element of `x` in an arithmetic, geometric, or harmonic index.

`fisher_contributions()` and `fisher_contributions2()` each return a numeric vector, the same length as `x`, giving the contribution of each element of `x` when a geometric mean aggregates an arithmetic mean of `x` with weights `w1` and a harmonic mean of `x` with weights `w2`.

References

Balk, B. M. (2008). *Price and Quantity Index Numbers*. Cambridge University Press.

Martin, S. (2021). A note on general decompositions for price indexes. *Prices Analytical Series*, Statistics Canada catalogue no. 62F0014M. Statistics Canada, Ottawa.

Reinsdorf, M. B., Diewert, W. E., and Ehemann, C. (2002). Additive decompositions for Fisher, Tornqvist and geometric mean indexes. *Journal of Economic and Social Measurement*, 28(1-2):51–61.

Webster, M. and Tarnow-Mordi, R. C. (2019). Decomposing multilateral price indexes into the contributions of individual commodities. *Journal of Official Statistics*, 35(2):461–486.

See Also

`transmute_weights` for the underlying implementation.

Examples

```
x <- 2:3

#---- Contributions for a geometric index ----

geometric_mean(x) - 1 # percent change in the Jevons index

geometric_contributions(x)

all.equal(geometric_mean(x) - 1, sum(geometric_contributions(x)))

# This works by first transmuting the weights in the geometric
# mean into weights for an arithmetic mean, then finding the
# contributions to the percent change

scale_weights(transmute_weights(0, 1)(x)) * (x - 1)

# Not the only way to calculate contributions

transmute2 <- function(x) {
  m <- geometric_mean(x)
  (m - 1) / log(m) * log(x) / (x - 1) / length(x)
}

transmute2(x) * (x - 1) # not proportional to the method above
all.equal(sum(transmute2(x) * (x - 1)), geometric_mean(x) - 1)

# But these "transmuted" weights don't recover the geometric mean!
# Not a particularly good way to calculate contributions

isTRUE(all.equal(arithmetic_mean(x, transmute2(x)),
                 geometric_mean(x)))

# There are infinitely many ways to calculate contributions,
# but the weights from transmute_weights(0, 1)() are the *unique*
# weights that recover the geometric mean

perturb <- function(w, e) {
  w + c(e, -e) / (x - 1)
}
```

```
perturb(transmute2(x), 0.1) * (x - 1)
all.equal(sum(perturb(transmute2(x), 0.1) * (x - 1)),
          geometric_mean(x) - 1)
isTRUE(all.equal(arithmetic_mean(x, perturb(transmute2(x), 0.1)),
                 geometric_mean(x)))

#---- Contributions for a Fisher index ----

p1 <- price6[[2]]
p0 <- price6[[1]]
q1 <- quantity6[[2]]
q0 <- quantity6[[1]]

# Quote contributions for the Fisher index in
# section 6 of Reinsdorf et al. (2002)

(con <- fisher_contributions(p1 / p0,
                             index_weights("Laspeyres")(p0, q0),
                             index_weights("Paasche")(p1, q1)))

all.equal(sum(con), fisher_index(p1, p0, q1, q0) - 1)

# Not the only way

(con2 <- fisher_contributions2(p1 / p0,
                               index_weights("Laspeyres")(p0, q0),
                               index_weights("Paasche")(p1, q1)))

all.equal(sum(con2), fisher_index(p1, p0, q1, q0) - 1)

# The same as the van IJzeren decomposition in
# section 4.2.2 of Balk (2008)

Qf <- quantity_index(fisher_index)(q1, q0, p1, p0)
Ql <- quantity_index(laspeyres_index)(q1, q0, p0)
wl <- index_weights("Laspeyres")(p0, q0)
wp <- index_weights("HybridPaasche")(p0, q1)

(Qf / (Qf + Ql) * scale_weights(wl) +
  Ql / (Qf + Ql) * scale_weights(wp)) * (p1 / p0 - 1)

#---- Contributions for other types of indexes ----

# A function to get contributions for any
# superlative quadratic mean of order 'r' index

superlative_contributions <- function(r) {
  nested_contributions(0, c(r / 2, - r / 2))
}

# Works for other types of indexes, like the harmonic
# Laspeyres Paasche index
```

```
hlp_contributions <- nested_contributions(-1, c(1, -1))
hlp_contributions(p1 / p0,
                  index_weights("Laspeyres")(p0, q0),
                  index_weights("Paasche")(p1, q1))

# Or the AG mean index (tau = 0.25)

agmean_contributions <- nested_contributions(1, c(0, 1), c(0.25, 0.75))
agmean_contributions(p1 / p0,
                     index_weights("Laspeyres")(p0, q0),
                     index_weights("Laspeyres")(p0, q0))

# Or the Balk-Walsh index

bw_contributions <- nested_contributions(0, c(0.5, -0.5))
bw_contributions(p1 / p0)
```

geks *GEKS index*

Description

Calculate an inter-temporal GEKS price index over a rolling window, as described in chapter 7 of Balk (2008), by Ivancic et al. (2011), and in chapter 10 of the CPI manual (2020).

Usage

```
## Function factory
geks(f)

## Special cases
tornqvist_geks(p, q, period, product,
                window = nlevels(period), n = window - 1, na.rm = FALSE)

fisher_geks(p, q, period, product,
             window = nlevels(period), n = window - 1, na.rm = FALSE)
```

Arguments

f	A price_index function that uses information on both base and current-period prices and quantities, and satisfies the time-reversal test. Usually a Tornqvist, Fisher, or Walsh index.
p	A numeric vector of prices, the same length as q.
q	A numeric vector of quantities, the same length as p.
period	A factor, or something that can be coerced into one, that gives the corresponding time period for each element in p and q. The ordering of time periods follows the levels of period to agree with cut().

`product`	A factor, or something that can be coerced into one, that gives the corresponding product identifier for each element in `p` and `q`.
`window`	The length of the rolling window. The default is a window that encompasses all periods in `period`. Values that are neither integers nor length 1 are silently truncated to a length 1 integer.
`n`	A number giving the length of the index series for each window, starting from the end of the window. For example, if there are 13 periods in `window`, setting `n = 1` gives the index for period 13. The default gives an index for each period in `window`. Values that are neither integers nor length 1 are silently truncated to a length 1 integer.
`na.rm`	Should missing values for `p` and `q` be removed when calculating the index? By default missing values will return a missing value for the index.

Value

`geks()` returns a function:

```
function(p, q, period, product,
         window = nlevels(period), n = window - 1, na.rm = FALSE){...}
```

This calculates a period-over-period GEKS index with the desired index-number formula, returning a list for each window with a named-numeric vector of index values.

`tornqvist_geks()` and `fisher_geks()` both return a list with a named numeric vector giving the value of the respective period-over-period GEKS index for each window.

Note

Like `back_period`, if multiple prices correspond to a period-product pair, then the back price at a point in time is always the first price for that product in the previous period. Unlike a bilateral index, however, duplicated period-product pairs can have more subtle implications for a multilateral index.

References

Balk, B. M. (2008). *Price and Quantity Index Numbers*. Cambridge University Press.

ILO, IMF, OECD, Eurostat, UN, and World Bank. (2020). *Consumer Price Index Manual: Theory and Practice*. International Monetary Fund.

Ivancic, L., Diewert, W. E., and Fox, K. J. (2011). Scanner data, time aggregation and the construction of price indexes. *Journal of Econometrics*, 161(1): 24–35.

See Also

`price_index` for price-index functions that can be used in `geks()`.

Examples

```
price <- 1:6
quantity <- 6:1
period <- rep(1:3, 2)
product <- rep(letters[1:2], each = 3)

tornqvist_geks(price, quantity, period, product)

tornqvist_geks(price, quantity, period, product, window = 2)

# Missing data

quantity[2] <- NA

# Use all non-missing data

fisher_geks(price, quantity, period, product, na.rm = TRUE)

# Remove records with any missing data

fg <- geks(balanced(fisher_index))
fg(price, quantity, period, product, na.rm = TRUE)
```

`generalized_mean` *Generalized mean*

Description

Calculate a generalized mean.

Usage

```
## Function factory
generalized_mean(r)

## Special cases
arithmetic_mean(x, w, na.rm = FALSE)

geometric_mean(x, w, na.rm = FALSE)

harmonic_mean(x, w, na.rm = FALSE)
```

Arguments

`r`	A finite number giving the order of the generalized mean.
`x`	A strictly positive numeric vector.
`w`	A strictly positive numeric vector of weights, the same length as `x`. The default is to equally weight each element of `x`.

`na.rm`	Should missing values in `x` and `w` be removed? By default missing values in `x` or `w` return a missing value.

Details

The function `generalized_mean()` returns a function to compute the generalized mean of `x` with weights `w` and exponent `r` (i.e., $\prod_{i=1}^{n} x_i^{w_i}$ when $r = 0$ and $\left(\sum_{i=1}^{n} w_i x_i^r\right)^{1/r}$ otherwise). This is also called the power mean, Holder mean, or l_p mean. See Bullen (2003, p. 175) for a definition, or `https://en.wikipedia.org/wiki/Generalized_mean`. The generalized mean is the solution to the optimal prediction problem: choose m to minimize $\sum_{i=1}^{n} w_i \left[\log(x_i) - \log(m)\right]^2$ when $r = 0$, $\sum_{i=1}^{n} w_i \left[x_i^r - m^r\right]^2$ otherwise.

The functions `arithmetic_mean()`, `geometric_mean()`, and `harmonic_mean()` compute the arithmetic, geometric, and harmonic (or subcontrary) means, also known as the Pythagorean means. These are the most useful means for making price indexes, and correspond to setting `r = 1`, `r = 0`, and `r = -1` in `generalized_mean()`.

Both `x` and `w` should be strictly positive (and finite), especially for the purpose of making a price index. This is not enforced, but the results may not make sense if the generalized mean is not defined. There are two exceptions to this.

1. The convention in Hardy et al. (1952, p. 13) is used in cases where `x` has zeros: the generalized mean is 0 whenever `w` is strictly positive and `r` < 0. (The analogous convention holds whenever at least one element of `x` is `Inf`: the generalized mean is `Inf` whenever `w` is strictly positive and `r > 0`.)
2. Some authors let `w` be non-negative and sum to 1 (e.g., Sydsaeter et al., 2005, p. 47). If `w` has zeros, then the corresponding element of `x` has no impact on the mean whenever `x` is strictly positive. Unlike `weighted.mean()`, however, zeros in `w` are not strong zeros, so infinite values in `x` will propagate even if the corresponding elements of `w` are zero.

The weights are scaled to sum to 1 to satisfy the definition of a generalized mean. There are certain price indexes where the weights should not be scaled (e.g., the Vartia-I index); use `sum()` for these cases.

The underlying calculation returned by `generalized_mean()` is mostly identical to `weighted.mean()`, with one important exception: missing values in the weights are not treated differently than missing values in `x`. Setting `na.rm = TRUE` drops missing values in both `x` and `w`, not just `x`. This ensures that certain useful identities are satisfied with missing values in `x`. In most cases `arithmetic_mean()` is a drop-in replacement for `weighted.mean()`.

Value

`generalized_mean()` returns a function:

```
function(x, w, na.rm = FALSE){...}
```

This computes the generalized mean of order `r` of `x` with weights `w`.

`arithmetic_mean()`, `geometric_mean()`, and `harmonic_mean()` each return a numeric value for the generalized means of order 1, 0, and -1.

Warning

Passing very small values for `r` can give misleading results, and warning is given whenever `abs(r)` is sufficiently small. In general, `r` should not be a computed value.

Note

`generalized_mean()` can be defined on the extended real line, so that `r = -Inf/Inf` returns `min()`/`max()`, to agree with the definition in, e.g., Bullen (2003). This is not implemented, and `r` must be finite.

There are a number of existing functions for calculating *unweighted* geometric and harmonic means, namely the `geometric.mean()` and `harmonic.mean()` functions in the **psych** package, the `geomean()` function in the **FSA** package, the `GMean()` and `HMean()` functions in the **DescTools** package, and the `geoMean()` function in the **EnvStats** package. Similarly, the `ci_generalized_mean()` function in the **Compind** package calculates an *unweighted* generalized mean.

References

Bullen, P. S. (2003). *Handbook of Means and Their Inequalities*. Springer Science+Business Media.

Fisher, I. (1922). *The Making of Index Numbers*. Houghton Mifflin Company.

Hardy, G., Littlewood, J. E., and Polya, G. (1952). *Inequalities* (2nd edition). Cambridge University Press.

ILO, IMF, OECD, Eurostat, UN, and World Bank. (2020). *Consumer Price Index Manual: Theory and Practice*. International Monetary Fund.

Lord, N. (2002). Does Smaller Spread Always Mean Larger Product? *The Mathematical Gazette*, 86(506): 273-274.

Sydsaeter, K., Strom, A., and Berck, P. (2005). *Economists' Mathematical Manual* (4th edition). Springer.

See Also

`generalized_logmean` for the generalized logarithmic mean.

`lehmer_mean` for the Lehmer mean, an alternative to the generalized mean.

`nested_mean` for nesting generalized means.

`transmute_weights` transforms the weights to turn a generalized mean of order r into a generalized mean of order s.

`factor_weights` calculates the weights to factor a mean of products into a product of means.

`price_index` and `quantity_index` for simple wrappers that use `generalized_mean()` to calculate common indexes.

`back_period`/`base_period` for a simple utility function to turn prices in a table into price relatives.

Examples

```
x <- 1:3
w <- c(0.25, 0.25, 0.5)
```

```
#---- Common generalized means ----

# Arithmetic mean

arithmetic_mean(x, w) # same as weighted.mean(x, w)

# Geometric mean

geometric_mean(x, w) # same as prod(x^w)

# Using prod() to manually calculate the geometric mean
# can give misleading results

z <- 1:1000
prod(z)^(1 / length(z)) # overflow
geometric_mean(z)

z <- seq(0.0001, by = 0.0005, length.out = 1000)
prod(z)^(1 / length(z)) # underflow
geometric_mean(z)

# Harmonic mean

harmonic_mean(x, w) # same as 1 / weighted.mean(1 / x, w)

# Quadratic mean / root mean square

generalized_mean(2)(x, w)

# Cubic mean
# Notice that this is larger than the other means so far because
# the generalized mean is increasing in r

generalized_mean(3)(x, w)

#---- Comparing the Pythagorean means ----

# The dispersion between the arithmetic, geometric, and harmonic
# mean usually increases as the variance of 'x' increases

x <- c(1, 3, 5)
y <- c(2, 3, 4)

var(x) > var(y)

arithmetic_mean(x) - geometric_mean(x)
arithmetic_mean(y) - geometric_mean(y)

geometric_mean(x) - harmonic_mean(x)
geometric_mean(y) - harmonic_mean(y)

# But the dispersion between these means is only bounded by
# the variance (Bullen, 2003, p. 156)
```

```
arithmetic_mean(x) - geometric_mean(x) >= 2 / 3 * var(x) / (2 * max(x))
arithmetic_mean(x) - geometric_mean(x) <= 2 / 3 * var(x) / (2 * min(x))

# Example by Lord (2002) where the dispersion decreases as
# the variance increases, counter to the claims by
# Fisher (1922, p. 108) and the CPI manual (par. 1.14)

x <- (5 + c(sqrt(5), -sqrt(5), -3)) / 4
y <- (16 + c(7 * sqrt(2), -7 * sqrt(2), 0)) / 16

var(x) > var(y)

arithmetic_mean(x) - geometric_mean(x)
arithmetic_mean(y) - geometric_mean(y)

geometric_mean(x) - harmonic_mean(x)
geometric_mean(y) - harmonic_mean(y)

# The "bias" in the arithmetic and harmonic indexes is also
# smaller in this case, counter to the claim
# by Fisher (1922, p. 108)

arithmetic_mean(x) * arithmetic_mean(1 / x) - 1
arithmetic_mean(y) * arithmetic_mean(1 / y) - 1

harmonic_mean(x) * harmonic_mean(1 / x) - 1
harmonic_mean(y) * harmonic_mean(1 / y) - 1

#---- Missing values ----

w[2] <- NA

arithmetic_mean(x, w)

arithmetic_mean(x, w, na.rm = TRUE) # drop the second observation
weighted.mean(x, w, na.rm = TRUE) # still returns NA
```

---

lehmer_mean *Lehmer mean*

Description

Calculate a Lehmer mean.

Usage

```
## Function factory
lehmer_mean(r)
```

```
## Special case
contraharmonic_mean(x, w, na.rm = FALSE)
```

Arguments

`r`	A finite number giving the order of the Lehmer mean.
`x`	A strictly positive numeric vector.
`w`	A strictly positive numeric vector of weights, the same length as `x`. The default is to equally weight each element of `x`.
`na.rm`	Should missing values in `x` and `w` be removed? By default missing values in `x` or `w` return a missing value.

Details

The function `lehmer_mean()` returns a function to compute the Lehmer mean of order `r` of `x` with weights `w`, which is calculated as the arithmetic mean of `x` with weights wx^{r-1}. This is also called the counter-harmonic mean or generalized anti-harmonic mean. See Bullen (2003, p. 245) for a definition, or `https://en.wikipedia.org/wiki/Lehmer_mean`.

The Lehmer mean of order 2 is sometimes called the contraharmonic (or anti-harmonic) mean. The function `contraharmonic_mean()` simply calls `lehmer_mean(2)()`. Like the generalized mean, the contraharmonic mean is the solution to an optimal prediction problem: choose m to minimize $\sum_{i=1}^{n} w_i \left(\frac{x_i}{m} - 1\right)^2$. The Lehmer mean of order -1 has a similar interpretation, replacing $\frac{x_i}{m}$ with $\frac{m}{x_i}$, and together these bound the harmonic and arithmetic means.

Both `x` and `w` should be strictly positive. This is not enforced, but the results may not make sense if the generalized mean in not defined.

The Lehmer mean is an alternative to the generalized mean that generalizes the Pythagorean means. The function `lehmer_mean(1)()` is identical to `arithmetic_mean()`, `lehmer_mean(0)()` is identical to `harmonic_mean()`, and `lehmer_mean(0.5)()` is identical to `geometric_mean()` with two values and no weights. See von der Lippe (2015) for more details on the use of these means for making price indexes.

Value

`lehmer_mean()` returns a function:

```
function(x, w, na.rm = FALSE){...}
```

This computes the Lehmer mean of order `r` of `x` with weights `w`.

`contraharmonic_mean()` returns a numeric value for the Lehmer mean of order 2.

Note

`lehmer_mean()` can be defined on the extended real line, so that `r = -Inf/Inf` returns `min()`/`max()`, to agree with the definition in, e.g., Bullen (2003). This is not implemented, and `r` must be finite.

References

Bullen, P. S. (2003). *Handbook of Means and Their Inequalities*. Springer Science+Business Media.

Lehmer, D. H. (1971). On the Compounding of Certain Means. *Journal of Mathematical Analysis and Applications*, 36(1): 183-200.

von der Lippe, P. (2015). Generalized Statistical Means and New Price Index Formulas, Notes on some unexplored index formulas, their interpretations and generalizations. Munich Personal RePEc Archive paper no. 64952.

See Also

`generalized_mean` for the generalized mean, an alternative to the Lehmer mean.

`generalized_logmean` for the generalized logarithmic mean.

Examples

```
x <- 2:3
w <- c(0.25, 0.75)

#---- The Pythagorean means are special cases of the Lehmer mean ----

all.equal(lehmer_mean(1)(x, w), arithmetic_mean(x, w))
all.equal(lehmer_mean(0)(x, w), harmonic_mean(x, w))
all.equal(lehmer_mean(0.5)(x), geometric_mean(x))

#---- Comparing Lehmer means and generalized means ----

# When r < 1, the generalized mean is larger than the
# corresponding Lehmer mean

lehmer_mean(-1)(x, w) < generalized_mean(-1)(x, w)

# The reverse is true when r > 1

lehmer_mean(3)(x, w) > generalized_mean(3)(x, w)

# This implies the contraharmonic mean is larger than the
# quadratic mean, and therefore the Pythagorean means

contraharmonic_mean(x, w) > arithmetic_mean(x, w)
contraharmonic_mean(x, w) > geometric_mean(x, w)
contraharmonic_mean(x, w) > harmonic_mean(x, w)

# and the logarithmic mean

contraharmonic_mean(2:3) > logmean(2, 3)

# The difference between the arithmetic mean and
# contraharmonic mean is proportional to the variance of x
```

```
weighted_var <- function(x, w) {
  arithmetic_mean(x^2, w) - arithmetic_mean(x, w)^2
}

arithmetic_mean(x, w) + weighted_var(x, w) / arithmetic_mean(x, w)
contraharmonic_mean(x, w)

#---- Changing the order of the mean ----

# It is easy to modify the weights to turn a Lehmer mean of order r
# into a Lehmer mean of order s because the Lehmer mean can be
# expressed as an arithmetic mean

r <- 2
s <- -3
lehmer_mean(r)(x, w)
lehmer_mean(s)(x, w * x^(r - 1) / x^(s - 1))

# The weights can also be modified to turn a Lehmer mean of order r
# into a generalized mean of order s

lehmer_mean(r)(x, w)
generalized_mean(s)(x, transmute_weights(1, s)(x, w * x^(r - 1)))

# and vice versa

lehmer_mean(r)(x, transmute_weights(s, 1)(x, w) / x^(r - 1))
generalized_mean(s)(x, w)

#---- Quote contributions ----

# Quote contributions for a price index based on the Lehmer mean
# are easy to calculate

scale_weights(w * x^(r - 1)) * (x - 1)
```

`logarithmic_means` *Logarithmic means*

Description

Calculate a generalized logarithmic mean / extended mean.

Usage

```
## Function factories
extended_mean(r, s)

generalized_logmean(r)
```

```
## Special case
logmean(a, b, tol = .Machine$double.eps^0.5)
```

Arguments

`r, s`	A finite number giving the order of the generalized logarithmic mean / extended mean.
`a, b`	A strictly positive numeric vector.
`tol`	The tolerance used to determine if `a == b`.

Details

The function `extended_mean()` returns a function to compute the component-wise extended mean of `a` and `b` of orders `r` and `s`. See Bullen (2003, p. 393) for a definition. This is also called the difference mean, Stolarsky mean, or extended mean-value mean.

The function `generalized_logmean()` returns a function to compute the component-wise generalized logarithmic mean of `a` and `b` of order `r`. See Bullen (2003, p. 385) for a definition, or `https://en.wikipedia.org/wiki/Stolarsky_mean`. The generalized logarithmic mean is a special case of the extended mean, corresponding to `extended_mean(r, 1)()`, but is more commonly used for price indexes.

The function `logmean()` returns the ordinary component-wise logarithmic mean of `a` and `b`, and corresponds to `generalized_logmean(1)()`.

Both `a` and `b` should be strictly positive. This is not enforced, but the results may not make sense when the generalized logarithmic mean / extended mean is not defined. The usual recycling rules apply when `a` and `b` are not the same length.

By definition, the generalized logarithmic mean / extended mean of `a` and `b` is `a` when `a == b`. The `tol` argument is used to test equality by checking if `abs(a - b) <= tol`. The default value is the same as `all.equal()`. Setting `tol = 0` will test for exact equality, but can give misleading results when `a` and `b` are computed values.

Value

`generalized_logmean()` and `extended_mean()` return a function:

```
function(a, b, tol = .Machine$double.eps^0.5){...}
```

This computes the component-wise generalized logarithmic mean of order `r`, or the extended mean of orders `r` and `s`, of `a` and `b`.

`logmean()` returns a numeric vector, the same length as `max(length(a), length(b))`, giving the component-wise logarithmic mean of `a` and `b`.

Warning

Passing very small values for `r` or `s` can give misleading results, and warning is given whenever `abs(r)` or `abs(s)` is sufficiently small. Similarly, values for `r` and `s` that are very close, but not equal, can give misleading results. In general, `r` and `s` should not be computed values.

Note

`generalized_logmean()` can be defined on the extended real line, so that `r = -Inf/Inf` returns `pmin()`/`pmax()`, to agree with the definition in, e.g., Bullen (2003). This is not implemented, and `r` must be finite as in the original formulation by Stolarsky (1975).

References

Balk, B. M. (2008). *Price and Quantity Index Numbers*. Cambridge University Press.

Bullen, P. S. (2003). *Handbook of Means and Their Inequalities*. Springer Science+Business Media.

Stolarsky, K. B. (1975). Generalizations of the Logarithmic Mean. *Mathematics Magazine*, 48(2): 87-92.

See Also

`generalized_mean` for the generalized mean.

`transmute_weights` uses the extended mean to turn a generalized mean of order r into a generalized mean of order s.

Examples

```
x <- 8:5
y <- 1:4

#---- Comparing logarithmic means and generalized means ----

# The arithmetic and geometric means are special cases of the
# generalized logarithmic mean

all.equal(generalized_logmean(2)(x, y), (x + y) / 2)
all.equal(generalized_logmean(-1)(x, y), sqrt(x * y))

# The logarithmic mean lies between the arithmetic and geometric means
# because the generalized logarithmic mean is increasing in r

all(logmean(x, y) < (x + y) / 2) &
  all(logmean(x, y) > sqrt(x * y))

# The harmonic mean cannot be expressed as a logarithmic mean,
# but can be expressed as an extended mean

all.equal(extended_mean(-2, -1)(x, y), 2 / (1 / x + 1 / y))

# The quadratic mean is also a type of extended mean

all.equal(extended_mean(2, 4)(x, y), sqrt(x^2 / 2 + y^2 / 2))

# As are heronian and centroidal means

all.equal(extended_mean(0.5, 1.5)(x, y),
          (x + sqrt(x * y) + y) / 3)
all.equal(extended_mean(2, 3)(x, y),
          2 / 3 * (x^2 + x * y + y^2) / (x + y))

#---- Approximating the logarithmic mean ----

# The logarithmic mean can be approximated as a convex
# combination of the arithmetic and geometric means that gives
# more weight to the geometric mean

approx1 <- 1 / 3 * (x + y) / 2 + 2 / 3 * sqrt(x * y)
approx2 <- ((x + y) / 2)^(1 / 3) * (sqrt(x * y))^(2 / 3)

approx1 - logmean(x, y) # always a positive approximation error
approx2 - logmean(x, y) # a negative approximation error

# A better approximation

correction <- (log(x / y) / pi)^4 / 32
approx1 / (1 + correction) - logmean(x, y)

#---- Some identities ----

# A useful identity for turning an additive change into
# a proportionate change

all.equal(logmean(x, y) * log(x / y), x - y)

# Works for other orders, too

r <- 2

all.equal(generalized_logmean(r)(x, y)^(r - 1) * (r * (x - y)),
          (x^r - y^r))

# Some other identities

all.equal(generalized_logmean(-2)(1, 2),
          (harmonic_mean(1:2) * geometric_mean(1:2)^2)^(1 / 3))

all.equal(generalized_logmean(0.5)(1, 2),
          (arithmetic_mean(1:2) + geometric_mean(1:2)) / 2)

all.equal(logmean(1, 2),
          geometric_mean(1:2)^2 * logmean(1, 1/2))

#---- Integral representations of the logarithmic mean ----

logmean(2, 3)

integrate(function(t) 2^(1 - t) * 3^t, 0, 1)$value
1 / integrate(function(t) 1 / (2 * (1 - t) + 3 * t), 0, 1)$value
```

`nested_mean` *Nested generalized mean*

Description

Calculate the (outer) generalized mean of two (inner) generalized means (i.e., crossing generalized means).

Usage

```
## Function factory
nested_mean(r1, r2, t = c(1, 1))

## Special case
fisher_mean(x, w1, w2, na.rm = FALSE)
```

Arguments

`r1`	A finite number giving the order of the outer generalized mean.
`r2`	A pair of finite numbers giving the order of the inner generalized means.
`t`	A pair of strictly positive weights for the inner generalized means. The default is equal weights.
`x`	A strictly positive numeric vector.
`w1, w2`	A strictly positive numeric vector of weights, the same length as `x`. The default is to equally weight each element of `x`.
`na.rm`	Should missing values in `x`, `w1`, and `w2` be removed? By default missing values in `x`, `w1`, or `w2` return a missing value.

Value

`nested_mean()` returns a function:

```
function(x, w1, w2, na.rm = FALSE){...}
```

This computes the generalized mean of order `r1` of the generalized mean of order `r2[1]` of `x` with weights `w1` and the generalized mean of order `r2[2]` of `x` with weights `w2`.

`fisher_mean()` returns a numeric value for the geometric mean of the arithmetic and harmonic means (i.e., `r1 = 0` and `r2 = c(1, -1)`).

Note

There is some ambiguity about how to remove missing values in `w1` or `w2` when `na.rm = TRUE`. The approach here is to remove missing values when calculating each of the inner means individually, rather than removing all missing values prior to any calculations. This means that a different number of data points could be used to calculate the inner means. Use the `balanced()` operator to balance missing values across `w1` and w2 prior to any calculations.

References

Diewert, W. E. (1976). Exact and superlative index numbers. *Journal of Econometrics*, 4(2): 114–145.

ILO, IMF, OECD, Eurostat, UN, and World Bank. (2004). *Producer Price Index Manual: Theory and Practice*. International Monetary Fund.

Lent, J. and Dorfman, A. H. (2009). Using a weighted average of base period price indexes to approximate a superlative index. *Journal of Official Statistics*, 25(1):139–149.

See Also

`generalized_mean` for the generalized mean.

`nested_contributions` for quote contributions for indexes based on nested generalized means, like the Fisher index.

Examples

```
x <- 1:3
w1 <- 4:6
w2 <- 7:9

#---- Making superlative indexes ----

# A function to make the superlative quadratic mean price index
# by Diewert (1976) as a product of generalized means

quadratic_mean_index <- function(x, w0, w1, r) {
  x <- sqrt(x)
  generalized_mean(r)(x, w0) * generalized_mean(-r)(x, w1)
}

quadratic_mean_index(x, w1, w2, 2)

# Same as the nested generalized mean (with the order halved)

quadratic_mean_index2 <- function(r) nested_mean(0, c(r / 2, -r / 2))

quadratic_mean_index2(2)(x, w1, w2)

# The arithmetic AG mean index by Lent and Dorfman (2009)

agmean_index <- function(tau) nested_mean(1, c(0, 1), c(tau, 1 - tau))

agmean_index(0.25)(x, w1, w1)

#---- Walsh index ----

# The (arithmetic) Walsh index is the implicit price index when using a
# superlative quadratic mean quantity index of order 1

p1 <- price6[[2]]; p0 <- price6[[1]]
```

```
q1 <- quantity6[[2]]; q0 <- quantity6[[1]]

walsh <- quadratic_mean_index2(1)

sum(p1 * q1) / sum(p0 * q0) / walsh(q1 / q0, p0 * q0, p1 * q1)

sum(p1 * sqrt(q1 * q0)) / sum(p0 * sqrt(q1 * q0))

# Counter to the PPI manual (par. 1.105), it is not a
# superlative quadratic mean price index of order 1

walsh(p1 / p0, p0 * q0, p1 * q1)

#---- Missing values ----

x[1] <- NA
w1[2] <- NA

fisher_mean(x, w1, w2, na.rm = TRUE)

# Same as using obs 2 and 3 in an arithmetic mean,
# and obs 3 in a harmonic mean

geometric_mean(c(arithmetic_mean(x, w1, na.rm = TRUE),
                 harmonic_mean(x, w2, na.rm = TRUE)))

# Use balanced() to use only obs 3 in both inner means

balanced(fisher_mean)(x, w1, w2, na.rm = TRUE)
```

offset_prices *Offset prices*

Description

Utility functions to offset vectors of prices and quantities. For each product, these compute either the index for the previous period (back period), or the index for the first period (base period). Useful when price information is stored in a table.

Usage

```
back_period(period, product = gl(1, length(period)))

base_period(period, product = gl(1, length(period)))

back_price(x, period, product = gl(1, length(x))) # deprecated

base_price(x, period, product = gl(1, length(x))) # deprecated
```

Arguments

`period`	A factor, or something that can be coerced into one, that gives the time period for each transaction. The ordering of time periods follows the levels of `period` to agree with `cut()`.
`product`	A factor, or something that can be coerced into one, that gives the product identifier for each transaction. The default is to assume that all transactions are for the same product.
`x`	An atomic vector of prices.

Value

`back_period()` and `base_period()` return a numeric vector of indices for the back/base periods. With `back_period()`, for all periods after the first, the resulting vector gives the location of the corresponding product in the previous period. The locations are unchanged for the first time period. With `base_period()`, the resulting vector gives the location of the corresponding product in the first period.

`back_price()` and `base_price()` return an offset copy of `x`.

Note

By definition, there must be at most one transaction for each product in each time period to determine a back period. If multiple transactions correspond to a period-product pair, then the back period at a point in time is always the first location for that product in the previous period.

See Also

`outliers` for common methods to detect outliers for price relatives.

`generalized_mean` for the generalized mean.

Examples

```
df <- data.frame(price = 1:6,
                 product = factor(c("a", "b")),
                 period = factor(c(1, 1, 2, 2, 3, 3)))

with(df, back_period(period, product))

# Make period-over-period price relatives

with(df, price / price[back_period(period, product)])

# Make fixed-base price relatives

with(df, price / price[base_period(period, product)])

# Change the base period with relevel()

with(df, price / price[base_period(relevel(period, "2"), product)])
```

```
# Warning is given if the same product has multiple prices
# at any point in time

with(df, back_period(period))
```

`operators` *Function operators*

Description

Operators to augment the way a function behaves.

- `quantity_index()` remaps price arguments into quantity argument (and vice versa) to turn a price index into a quantity index.
- `grouped()` makes a function applicable to grouped data.
- `balanced()` makes a function balance the removal of `NA`s across multiple input vectors.

Usage

```
quantity_index(f)

grouped(f, ...)

balanced(f, ...)
```

Arguments

`f`	A function.
`...`	Additional arguments to `f` that should *not* be treated as grouped / should *not* be balanced.

Value

`quantity_index()` returns a function like `f`, except that the role of prices/quantities is reversed.

`grouped()` returns a function like `f` with a new argument `group`. This accepts a factor to split all other arguments in `f` (except those specified in `...`) into groups before applying `f` to each group and combining the results. It is similar to `ave()`, but more general.

`balanced()` returns a function like `f`, except that missing values are balanced across *all* inputs when `na.rm = TRUE` (except those specified in `...`). This is like using `complete.cases()` on the inputs of `f`.

Note

Additional arguments passed to `...` are evaluated, so the return function doesn't evaluate these arguments lazily. In most cases these additional arguments are parameters like `elasticity` or switches like `na.rm`, and eager evaluation prevents these from being changed in, e.g., a loop. Lazy evaluation can be had by passing an anonymous function that partials out these arguments.

See Also

price_index for the possible functions that can serve as inputs for `quantity_index()` and `balanced()`.

Examples

```
p1 <- price6[[3]]
p0 <- price6[[2]]
q1 <- quantity6[[3]]
q0 <- quantity6[[2]]

#---- Quantity index ----
# Remap argument names to be quantities rather than prices

quantity_index(laspeyres_index)(q1 = q1, q0 = q0, p0 = p0)

laspeyres_index(p1 = q1, p0 = q0, q0 = p0)

# Works with the index_weights() functions, too

quantity_index(index_weights("Laspeyres"))(q0 = q0, p0 = p0)

#---- Grouped ----
# Calculate Tornqvist weights for two groups

f <- factor(rep(letters[1:2], each = 3))
tornqvist_weights <- grouped(index_weights("Tornqvist"))
tornqvist_weights(p1, p0, q1, q0, group = f)

# Calculate a mean like ave(), but with weights

x <- 1:6
w <- c(1:5, NA)
grouped_mean <- grouped(geometric_mean, na.rm = TRUE)
grouped_mean(x, w, group = f)

# Redistribute weights

w1 <- c(2, 4)
w2 <- 1:6

harmonic_mean(mapply(harmonic_mean, split(x, f), split(w2, f)), w1)

wr <- grouped(scale_weights)(w2, group = f) * w1[f]
harmonic_mean(x, wr)

#---- Balanced ----
# Balance missing values for a Fisher index

fisher <- balanced(fisher_index)
fisher(p1, p0, q1, replace(q0, 3, NA), na.rm = TRUE)
fisher_index(p1[-3], p0[-3], q1[-3], q0[-3])
```

```
# Operators can be combined, but some care may be needed

grouped(balanced(fisher_mean), na.rm = TRUE)(x, w, group = f)
balanced(grouped(fisher_mean))(x, w, group = f, na.rm = TRUE)
```

outliers *Outlier methods for price relatives*

Description

Standard cutoff-based methods for detecting outliers with price relatives.

Usage

```
fixed_cutoff(x, cu = 2.5, cl = 1 / cu)

robust_z(x, cu = 2.5, cl = cu)

quartile_method(x, cu = 2.5, cl = cu, a = 0, type = 7)

resistant_fences(x, cu = 2.5, cl = cu, a = 0, type = 7)

tukey_algorithm(x, cu = 2.5, cl = cu, type = 7)

hb_transform(x)
```

Arguments

`x`	A strictly positive numeric vector of price relatives. These can be made with, e.g., `back_period()`.
`cu, cl`	A numeric vector giving the upper and lower cutoffs for each element of `x`. The usual recycling rules apply.
`a`	A numeric vector between 0 and 1 giving the scale factor for the median to establish the minimum dispersion between quartiles for each element of `x`. The default does not set a minimum dispersion. The usual recycling rules apply.
`type`	See `quantile`.

Details

Each of these functions constructs an interval of the form $[b_l(x) - c_l \times l(x), b_u(x) + c_u \times u(x)]$ and assigns a value in `x` as `TRUE` if that value does not belong to the interval, `FALSE` otherwise. The methods differ in how they construct the values $b_l(x)$, $b_u(x)$, $l(x)$, and $u(x)$. Any missing values in `x` are ignored when calculating the cutoffs, but will return `NA`.

The fixed cutoff method is the simplest, and just uses the interval $[c_l, c_u]$.

The quartile method and Tukey algorithm are described in paragraphs 5.113 to 5.135 of the CPI manual (2020), as well as by Rais (2008) and Hutton (2008). The resistant fences method is an alternative to the quartile method, and is described by Rais (2008) and Hutton (2008). Quantile-based methods often identify price relatives as outliers because the distribution is concentrated around 1; setting `a > 0` puts a floor on the minimum dispersion between quantiles as a fraction of the median. See the references for more details.

The robust Z-score is the usual method to identify relatives in the (asymmetric) tails of the distribution, simply replacing the mean with the median, and the standard deviation with the median absolute deviation.

These methods often assume that price relatives are symmetrically distributed (if not Gaussian). As the distribution of price relatives often has a long right tail, the natural logarithm can be used to transform price relative before identifying outliers (sometimes under the assumption that price relatives are distributed log-normal). The Hidiroglou-Berthelot transformation is another approach, described in the CPI manual (par. 5.124).

Value

A logical vector, the same length as `x`, that is `TRUE` if the corresponding element of `x` is identified as an outlier, `FALSE` otherwise.

References

Hutton, H. (2008). Dynamic outlier detection in price index surveys. *Proceedings of the Survey Methods Section: Statistical Society of Canada Annual Meeting*.

ILO, IMF, OECD, Eurostat, UN, and World Bank. (2020). *Consumer Price Index Manual: Theory and Practice*. International Monetary Fund.

Rais, S. (2008). Outlier detection for the Consumer Price Index. *Proceedings of the Survey Methods Section: Statistical Society of Canada Annual Meeting*.

See Also

`grouped` to make each of these functions operate on grouped data.

`back_period`/`base_period` for a simple utility function to turn prices in a table into price relatives.

Examples

```
set.seed(1234)

x <- rlnorm(10)

fixed_cutoff(x)
robust_z(x)
quartile_method(x)
resistant_fences(x) # always identifies fewer outliers than quartile method
tukey_algorithm(x)

log(x)
hb_transform(x)

# Works the same for grouped data
```

```
f <- c("a", "b", "a", "a", "b", "b", "b", "a", "a", "b")
grouped(quartile_method)(x, group = f)
```

price_data *Sample price/quantity data*

Description

Prices and quantities for six products over five periods.

Usage

```
price6
quantity6
```

Format

Each data frame has 6 rows and 5 columns, with each row corresponding to a product and each column corresponding to a time period.

Note

Adapted from tables 3.1 and 3.2 in Balk (2008), which were adapted from tables 19.1 and 19.2 in the PPI manual.

Source

Balk, B. M. (2008). *Price and Quantity Index Numbers*. Cambridge University Press.

ILO, IMF, OECD, Eurostat, UN, and World Bank. (2004). *Producer Price Index Manual: Theory and Practice*. International Monetary Fund.

Examples

```
# Recreate tables 3.4, 3.6, and 3.12 from Balk (2008)

index_formulas <- function(p1, p0, q1, q0) {
  c(harmonic_laspeyres = harmonic_index("Laspeyres")(p1, p0, q0),
    geometric_laspeyres = geometric_index("Laspeyres")(p1, p0, q0),
    laspeyres = arithmetic_index("Laspeyres")(p1, p0, q0),
    paasche = harmonic_index("Paasche")(p1, p0, q1),
    geometric_paasche = geometric_index("Paasche")(p1, p0, q1),
    palgrave = arithmetic_index("Palgrave")(p1, p0, q1),
    fisher = fisher_index(p1, p0, q1, q0),
    tornqvist = geometric_index("Tornqvist")(p1, p0, q1, q0),
    marshall_edgeworth = arithmetic_index("MarshallEdgeworth")(p1, p0, q1, q0),
    walsh1 = arithmetic_index("Walsh1")(p1, p0, q1, q0),
    vartia2 = geometric_index("Vartia2")(p1, p0, q1, q0),
    vartia1 = geometric_index("Vartia1")(p1, p0, q1, q0),
    stuvel = stuvel_index(2, 2)(p1, p0, q1, q0)
  )
}

round(t(mapply(index_formulas, price6, price6[1], quantity6, quantity6[1])), 4)
```

`price_index` *Price indexes*

Description

Calculate a variety of price indexes using information on prices and quantities at two points in time.

Usage

```
## Function factories
arithmetic_index(type)

geometric_index(type)

harmonic_index(type)

lm_index(elasticity)

arithmetic_agmean_index(elasticity)

geometric_agmean_index(elasticity)

stuvel_index(a, b)

index_weights(type)

## Special cases
laspeyres_index(p1, p0, q0, na.rm = FALSE)

paasche_index(p1, p0, q1, na.rm = FALSE)

jevons_index(p1, p0, na.rm = FALSE)

lowe_index(p1, p0, qb, na.rm = FALSE)

young_index(p1, p0, pb, qb, na.rm = FALSE)

fisher_index(p1, p0, q1, q0, na.rm = FALSE)

hlp_index(p1, p0, q1, q0, na.rm = FALSE)

cswd_index(p1, p0, na.rm = FALSE)
```

```
cswdb_index(p1, p0, q1, q0, na.rm = FALSE)

bw_index(p1, p0, na.rm = FALSE)

lehr_index(p1, p0, q1, q0, na.rm = FALSE)
```

Arguments

`type`	The name of the index. See details for the possible types of indexes.
`elasticity`	The elasticity of substitution for the Lloyd-Moulton and AG mean indexes.
`a, b`	Parameters for the generalized Stuvel index.
`p1`	Current-period prices.
`p0`	Base-period prices.
`q1`	Current-period quantities.
`q0`	Base-period quantities.
`pb`	Period-b prices for the Lowe/Young index.
`qb`	Period-b quantities for the Lowe/Young index.
`na.rm`	Should missing values be removed? By default missing values for prices or quantities return a missing value.

Details

The `arithmetic_index()`, `geometric_index()`, and `harmonic_index()` functions return a function to calculate a given type of arithmetic, geometric (logarithmic), and harmonic index. Together, these functions produce functions to calculate the following indexes.

- **Arithmetic indexes**
- Carli
- Dutot
- Laspeyres
- Palgrave
- Unnamed index (arithmetic mean of Laspeyres and Palgrave)
- Drobisch (arithmetic mean of Laspeyres and Paasche)
- Walsh-I (arithmetic Walsh)
- Marshall-Edgeworth
- Geary-Khamis
- Lowe
- Young
- **Geometric indexes**
- Jevons
- Geometric Laspeyres

- Geometric Paasche
- Geometric Young
- Tornqvist (or Tornqvist-Theil)
- Montgomery-Vartia / Vartia-I
- Sato-Vartia / Vartia-II
- Walsh-II (geometric Walsh)
- Theil
- Rao
- **Harmonic indexes**
- Coggeshall (equally weighted harmonic index)
- Paasche
- Harmonic Laspeyres
- Harmonic Young

Along with the `lm_index()` function to calculate the Lloyd-Moulton index, these are just convenient wrappers for `generalized_mean()` and `index_weights()`.

The Laspeyres, Paasche, Jevons, Lowe, and Young indexes are among the most common price indexes, and so they get their own functions. The `laspeyres_index()`, `lowe_index()`, and `young_index()` functions correspond to setting the appropriate `type` in `arithmetic_index()`; `paasche_index()` and `jevons_index()` instead come from the `harmonic_index()` and `geometric_index()` functions.

In addition to these indexes, there are also functions for calculating a variety of indexes not based on generalized means. The Fisher index is the geometric mean of the arithmetic Laspeyres and Paasche indexes; the Harmonic Laspeyres Paasche index is the harmonic analog of the Fisher index (8054 on Fisher's list). The Carruthers-Sellwood-Ward-Dalen and Carruthers-Sellwood-Ward-Dalen-Balk indexes are sample analogs of the Fisher index; the Balk-Walsh index is the sample analog of the Walsh index. The AG mean index is the arithmetic or geometric mean of the geometric and arithmetic Laspeyres indexes, weighted by the elasticity of substitution. The `stuvel_index()` function returns a function to calculate a Stuvel index of the given parameters. The Lehr index is an alternative to the Geary-Khamis index, and is the implicit price index for Fisher's index 4153.

The `index_weights()` function returns a function to calculate weights for a variety of price indexes. Weights for the following types of indexes can be calculated.

- Carli / Jevons / Coggeshall
- Dutot
- Laspeyres / Lloyd-Moulton
- Hybrid Laspeyres (for use in a harmonic mean)
- Paasche / Palgrave
- Hybrid Paasche (for use in an arithmetic mean)
- Tornqvist / Unnamed
- Drobisch
- Walsh-I (for an arithmetic Walsh index)

- Walsh-II (for a geometric Walsh index)
- Marshall-Edgeworth
- Geary-Khamis
- Montgomery-Vartia / Vartia-I
- Sato-Vartia / Vartia-II
- Theil
- Rao
- Lowe
- Young

The weights need not sum to 1, as this normalization isn't always appropriate (i.e., for the Vartia-I weights).

Naming for the indexes and weights generally follows the CPI manual (2020), Balk (2008), and Selvanathan and Rao (1994). In several cases two or more names correspond to the same weights (e.g., Paasche and Palgrave, or Sato-Vartia and Vartia-II). The calculations are given in the examples.

Value

`arithmetic_index()`, `geometric_index()`, `harmonic_index()`, `stuvel_index()`, and `index_weights()` each return a function to compute the relevant price indexes; `lm_index()`, `arithmetic_agmean_index()`, and `geometric_agmean_index()` each return a function to calculate the relevant index for a given elasticity of substitution. The others return a numeric value giving the change in price between the base period and current period.

Note

There are different ways to deal with missing values in a price index, and care should be taken when relying on these functions to remove missing values. Setting `na.rm = TRUE` removes price relatives with missing information, either because of a missing price or a missing weight, while using all available non-missing information to make the weights.

Certain properties of an index-number formula may not work as expected when removing missing values if there is ambiguity about how to remove missing values from the weights (as in, e.g., a Tornqvist or Sato-Vartia index). The `balanced()` operator may be helpful, as it balances the removal of missing values across prices and quantities prior to making the weights.

References

Balk, B. M. (2008). *Price and Quantity Index Numbers*. Cambridge University Press.

Fisher, I. (1922). *The Making of Index Numbers*. Houghton Mifflin Company.

ILO, IMF, OECD, Eurostat, UN, and World Bank. (2020). *Consumer Price Index Manual: Theory and Practice*. International Monetary Fund.

von der Lippe, P. (2001). *Chain Indices: A Study in Price Index Theory*, Spectrum of Federal Statistics vol. 16. Federal Statistical Office, Wiesbaden.

von der Lippe, P. (2015). Generalized Statistical Means and New Price Index Formulas, Notes on some unexplored index formulas, their interpretations and generalizations. Munich Personal RePEc Archive paper no. 64952.

Selvanathan, E. A. and Rao, D. S. P. (1994). *Index Numbers: A Stochastic Approach*. MacMillan.

See Also

generalized_mean for the generalized mean that powers most of these functions.

contributions for calculating quote contributions.

update_weights for price-updating weights.

quantity_index to remap the arguments in these functions for a quantity index.

geks for making a GEKS index based on, e.g., the Fisher index.

price6 for an example of how to use these functions with more than two time periods.

The **piar** package has more functionality working with price indexes for multiple groups of products over many time periods.

Examples

```
p0 <- price6[[2]]
p1 <- price6[[3]]
q0 <- quantity6[[2]]
q1 <- quantity6[[3]]
pb <- price6[[1]]
qb <- quantity6[[1]]

#---- Calculating price indexes ----

# Most indexes can be calculated by combining the appropriate
# weights with the correct type of mean

geometric_index("Laspeyres")(p1, p0, q0)
geometric_mean(p1 / p0, index_weights("Laspeyres")(p0, q0))

# Arithmetic Laspeyres index

laspeyres_index(p1, p0, q0)
arithmetic_mean(p1 / p0, index_weights("Laspeyres")(p0, q0))

# Harmonic calculation for the arithmetic Laspeyres

harmonic_mean(p1 / p0, index_weights("HybridLaspeyres")(p1, q0))

# Same as transmuting the weights

all.equal(
  scale_weights(index_weights("HybridLaspeyres")(p1, q0)),
  scale_weights(transmute_weights(1, -1)(p1 / p0, index_weights("Laspeyres")(p0, q0)))
)

# This strategy can be used to make more exotic indexes,
# like the quadratic-mean index (von der Lippe, 2001, p. 71)

generalized_mean(2)(p1 / p0, index_weights("Laspeyres")(p0, q0))

# Or the exponential mean index (p. 64)
```

```
log(arithmetic_mean(exp(p1 / p0), index_weights("Laspeyres")(p0, q0)))

# Or the arithmetic hybrid index (von der Lippe, 2015, p. 5)

arithmetic_mean(p1 / p0, index_weights("HybridLaspeyres")(p1, q0))
contraharmonic_mean(p1 / p0, index_weights("Laspeyres")(p0, q0))

# Unlike its arithmetic counterpart, the geometric Laspeyres
# can increase when base-period prices increase if some of these
# prices are small

gl <- geometric_index("Laspeyres")
p0_small <- replace(p0, 1, p0[1] / 5)
p0_dx <- replace(p0_small, 1, p0_small[1] + 0.01)
gl(p1, p0_small, q0) < gl(p1, p0_dx, q0)

#---- Price updating the weights in a price index ----

# Chain an index by price updating the weights

p2 <- price6[[4]]
laspeyres_index(p2, p0, q0)

I1 <- laspeyres_index(p1, p0, q0)
w_pu <- update_weights(p1 / p0, index_weights("Laspeyres")(p0, q0))
I2 <- arithmetic_mean(p2 / p1, w_pu)
I1 * I2

# Works for other types of indexes, too

harmonic_index("Laspeyres")(p2, p0, q0)

I1 <- harmonic_index("Laspeyres")(p1, p0, q0)
w_pu <- factor_weights(-1)(p1 / p0, index_weights("Laspeyres")(p0, q0))
I2 <- harmonic_mean(p2 / p1, w_pu)
I1 * I2

#---- Quote contributions ----

# Quote contributions for the Tornqvist index

w <- index_weights("Tornqvist")(p1, p0, q1, q0)
(con <- geometric_contributions(p1 / p0, w))

all.equal(sum(con), geometric_index("Tornqvist")(p1, p0, q1, q0) - 1)

#---- Missing values ----

# NAs get special treatment

p_na <- replace(p0, 6, NA)
```

```
# Drops the last price relative

laspeyres_index(p1, p_na, q0, na.rm = TRUE)

# Only drops the last period-0 price

sum(p1 * q0, na.rm = TRUE) / sum(p_na * q0, na.rm = TRUE)

#---- von Bortkiewicz decomposition ----

paasche_index(p1, p0, q1) / laspeyres_index(p1, p0, q0) - 1

wl <- scale_weights(index_weights("Laspeyres")(p0, q0))
pl <- laspeyres_index(p1, p0, q0)
ql <- quantity_index(laspeyres_index)(q1, q0, p0)

sum(wl * (p1 / p0 / pl - 1) * (q1 / q0 / ql - 1))

# Similar decomposition for geometric Laspeyres/Paasche

wp <- scale_weights(index_weights("Paasche")(p1, q1))
gl <- geometric_index("Laspeyres")(p1, p0, q0)
gp <- geometric_index("Paasche")(p1, p0, q1)

log(gp / gl)

sum(scale_weights(wl) * (wp / wl - 1) * log(p1 / p0 / gl))

#---- Consistency in aggregation ----

p0a <- p0[1:3]; p0b <- p0[4:6]
p1a <- p1[1:3]; p1b <- p1[4:6]
q0a <- q0[1:3]; q0b <- q0[4:6]
q1a <- q1[1:3]; q1b <- q1[4:6]

# Indexes based on the generalized mean with value share weights
# are consistent in aggregation

lm_index(0.75)(p1, p0, q0)

w <- index_weights("LloydMoulton")(p0, q0)
Ia <- generalized_mean(0.25)(p1a / p0a, w[1:3])
Ib <- generalized_mean(0.25)(p1b / p0b, w[4:6])
generalized_mean(0.25)(c(Ia, Ib), c(sum(w[1:3]), sum(w[4:6])))

# Agrees with group-wise indexes

all.equal(lm_index(0.75)(p1a, p0a, q0a), Ia)
all.equal(lm_index(0.75)(p1b, p0b, q0b), Ib)

# Care is needed with more complex weights, e.g., Drobisch,
# as this doesn't fit Balk's (2008) definition (p. 113) of
# a generalized-mean index (it's the arithmetic mean of a Laspeyres
# and Paasche index)

arithmetic_index("Drobisch")(p1, p0, q1, q0)

w <- index_weights("Drobisch")(p1, p0, q1, q0)
Ia <- arithmetic_mean(p1a / p0a, w[1:3])
Ib <- arithmetic_mean(p1b / p0b, w[4:6])
arithmetic_mean(c(Ia, Ib), c(sum(w[1:3]), sum(w[4:6])))

# Does not agree with group-wise indexes

all.equal(arithmetic_index("Drobisch")(p1a, p0a, q1a, q0a), Ia)
all.equal(arithmetic_index("Drobisch")(p1b, p0b, q1b, q0b), Ib)

#---- Making the weights for different indexes ----

# Explicit calculation for each of the different weights
# Carli/Jevons/Coggeshall

all.equal(index_weights("Carli")(p1), rep(1, length(p0)))

# Dutot

all.equal(index_weights("Dutot")(p0), p0)

# Laspeyres / Lloyd-Moulton

all.equal(index_weights("Laspeyres")(p0, q0), p0 * q0)

# Hybrid Laspeyres

all.equal(index_weights("HybridLaspeyres")(p1, q0), p1 * q0)

# Paasche / Palgrave

all.equal(index_weights("Paasche")(p1, q1), p1 * q1)

# Hybrid Paasche

all.equal(index_weights("HybridPaasche")(p0, q1), p0 * q1)

# Tornqvist / Unnamed

all.equal(index_weights("Tornqvist")(p1, p0, q1, q0),
          0.5 * p0 * q0 / sum(p0 * q0) + 0.5 * p1 * q1 / sum(p1 * q1))

# Drobisch

all.equal(index_weights("Drobisch")(p1, p0, q1, q0),
          0.5 * p0 * q0 / sum(p0 * q0) + 0.5 * p0 * q1 / sum(p0 * q1))

# Walsh-I
```

```
all.equal(index_weights("Walsh1")(p0, q1, q0),
          p0 * sqrt(q0 * q1))

# Marshall-Edgeworth

all.equal(index_weights("MarshallEdgeworth")(p0, q1, q0),
          p0 * (q0 + q1))

# Geary-Khamis

all.equal(index_weights("GearyKhamis")(p0, q1, q0),
          p0 / (1 / q0 + 1 / q1))

# Montgomery-Vartia / Vartia-I

all.equal(index_weights("MontgomeryVartia")(p1, p0, q1, q0),
          logmean(p0 * q0, p1 * q1) / logmean(sum(p0 * q0), sum(p1 * q1)))

# Sato-Vartia / Vartia-II

all.equal(index_weights("SatoVartia")(p1, p0, q1, q0),
          logmean(p0 * q0 / sum(p0 * q0), p1 * q1 / sum(p1 * q1)))

# Walsh-II

all.equal(index_weights("Walsh2")(p1, p0, q1, q0),
          sqrt(p0 * q0 * p1 * q1))

# Theil

all.equal(index_weights("Theil")(p1, p0, q1, q0),
          {w0 <- scale_weights(p0 * q0);
           w1 <- scale_weights(p1 * q1);
           (w0 * w1 * (w0 + w1) / 2)^(1 / 3)})

# Rao

all.equal(index_weights("Rao")(p1, p0, q1, q0),
          {w0 <- scale_weights(p0 * q0);
           w1 <- scale_weights(p1 * q1);
           w0 * w1 / (w0 + w1)})

# Lowe

all.equal(index_weights("Lowe")(p0, qb), p0 * qb)

# Young

all.equal(index_weights("Young")(pb, qb), pb * qb)
```

`transform_weights` *Transform weights*

Description

Useful transformations for the weights in a generalized mean.

- Transmute weights to turn a generalized mean of order r into a generalized mean of order s. Useful for calculating additive and multiplicative decompositions for a generalized-mean index, and those made of nested generalized means (e.g., Fisher index).
- Factor weights to turn the generalized mean of a product into the product of generalized means. Useful for price-updating the weights in a generalized-mean index.
- Scale weights so they sum to 1.

Usage

```
transmute_weights(r, s)

nested_transmute(r1, r2, s, t = c(1, 1))

nested_transmute2(r1, r2, s, t = c(1, 1))

factor_weights(r)

update_weights(x, w)

scale_weights(x)
```

Arguments

`r, s`	A finite number giving the order of the generalized mean. See details.
`r1, r2, t`	The same arguments as in `nested_mean()`. See details.
`x`	A strictly positive numeric vector.
`w`	A strictly positive numeric vector of weights, the same length as `x`. The default is to equally weight each element of `x`.

Details

The function `transmute_weights()` returns a function to compute a vector of weights `v(x, w)` such that

```
generalized_mean(r)(x, w) == generalized_mean(s)(x, v(x, w))
```

`nested_transmute()` and `nested_transmute2()` do the same for nested generalized means, so that

```
nested_mean(r1, r2, t)(x, w1, w2) == generalized_mean(s)(x, v(x, w1, w2))
```

This generalizes the result for turning a geometric mean into an arithmetic mean (and vice versa) in section 4.2 of Balk (2008), and a Fisher mean into an arithmetic mean in section 6 of Reinsdorf et al. (2002), although these are usually the most important cases. See Martin (2021) for details. `nested_transmute2()` takes a slightly different approach than `nested_transmute()`, generalizing the van IJzeren arithmetic decomposition for the Fisher index in section 4.2.2 of Balk (2008) using the approach by Martin (2021), although in most cases the results are broadly similar.

The function `factor_weights()` returns a function to compute weights `u(x, w)` such that

```
generalized_mean(r)(x * y, w) == generalized_mean(r)(x, w) *
                                 generalized_mean(r)(y, u(x, w))
```

This generalizes the result in section C.5 of Chapter 9 of the PPI Manual for chaining the Young index, and gives a way to chain generalized-mean price indexes over time. Factoring weights with `r = 1` sometimes gets called price-updating weights; `update_weights()` simply calls `factor_weights(1)()`.

The function `scale_weights()` scales a vector of weights so they sum to 1 by calling `x / sum(x, na.rm = TRUE)`.

Both `x` and `w` should be strictly positive. This is not enforced, but the results may not make sense in cases where the generalized mean and generalized logarithmic mean are not defined.

Value

`transmute_weights()` and `factor_weights()` return a function:

```
function(x, w){...}
```

In each case this function augments the weights `w`.

`nested_transmute()` and `nested_transmute2()` similarly return a function:

```
function(x, w1, w2){...}
```

`update_weights()` and `scale_weights()` return a numeric vector the same length as `x`.

Note

Transmuting, factoring, and scaling weights will return a value that is the same length as `x`, so any missing values in `x` or `w`/`w1`/`w2` will return `NA`. Unless all values are `NA`, however, the result for transmuting or factoring will still satisfy the above identities when `na.rm = TRUE` in `generalized_mean()` and `nested_mean()`. Similarly, the result of scaling will sum to 1 when missing values are removed.

References

Balk, B. M. (2008). *Price and Quantity Index Numbers*. Cambridge University Press.

ILO, IMF, OECD, Eurostat, UN, and World Bank. (2004). *Producer Price Index Manual: Theory and Practice*. International Monetary Fund.

Martin, S. (2021). A note on general decompositions for price indexes. *Prices Analytical Series*, Statistics Canada catalogue no. 62F0014M. Statistics Canada, Ottawa.

Reinsdorf, M. B., Diewert, W. E., and Ehemann, C. (2002). Additive decompositions for Fisher, Tornqvist and geometric mean indexes. *Journal of Economic and Social Measurement*, 28(1-2):51–61.

Sydsaeter, K., Strom, A., and Berck, P. (2005). *Economists' Mathematical Manual* (4th edition). Springer.

See Also

`generalized_mean` for the generalized mean.

`nested_mean` for the nested mean.

`extended_mean` for the extended mean that underlies `transmute_weights()`.

`contributions` for calculating additive quote contributions.

`grouped` to make each of these functions operate on grouped data.

Examples

```
x <- 1:3
y <- 4:6
w <- 3:1

#---- Transforming generalized means ----

# Calculate the geometric mean as an arithmetic mean and
# harmonic mean by transmuting the weights

geometric_mean(x)
arithmetic_mean(x, transmute_weights(0, 1)(x))
harmonic_mean(x, transmute_weights(0, -1)(x))

# Transmuting the weights for a harmonic mean into those
# for an arithmetic mean is the same as using weights w / x

all.equal(
    scale_weights(transmute_weights(-1, 1)(x, w)),
    scale_weights(w / x)
)

# Works for nested means, too

w1 <- 3:1
w2 <- 1:3

fisher_mean(x, w1, w2)

arithmetic_mean(x, nested_transmute(0, c(1, -1), 1)(x, w1, w2))
arithmetic_mean(x, nested_transmute2(0, c(1, -1), 1)(x, w1, w2))

#---- Quote contributions ----

# Transmuted weights can be used to calculate quote contributions
# for, e.g., a geometric price index

scale_weights(transmute_weights(0, 1)(x)) * (x - 1)
geometric_contributions(x) # the more convenient way

#---- Basket representation of a price index ----

# Any generalized-mean index can be represented as a basket-style
# index by transmuting the weights, which is how some authors
# define a price index (e.g., Sydsaeter et al., 2005, p. 174)

p1 <- 2:6
p0 <- 1:5

qs <- transmute_weights(-1, 1)(p1 / p0) / p0
all.equal(harmonic_mean(p1 / p0), sum(p1 * qs) / sum(p0 * qs))

#---- Factoring the product of generalized means ----

# Factor the harmonic mean by chaining the calculation

harmonic_mean(x * y, w)
harmonic_mean(x, w) * harmonic_mean(y, factor_weights(-1)(x, w))

# The common case of an arithmetic mean

arithmetic_mean(x * y, w)
arithmetic_mean(x, w) * arithmetic_mean(y, update_weights(x, w))

# In cases where x and y have the same order, Chebyshev's
# inequality implies that the chained calculation is too small

arithmetic_mean(x * y, w) > arithmetic_mean(x, w) * arithmetic_mean(y, w)
```

Index