Package ‘whep’

March 3, 2026

Title Processing Agro-Environmental Data

Version 0.3.0

Description A set of tools for processing and analyzing data developed in the context of the ``Who Has Eaten the Planet'' (WHEP) project, funded by the European Research Council (ERC). For more details on multi-regional input–output model ``Food and Agriculture Biomass Input–Output'' (FABIO) see Bruckner et al. (2019) <doi:10.1021/acs.est.9b03554>.

License MIT + file LICENSE

Imports cli, dplyr, fs, FAOSTAT, httr, nanoparquet, pins, purrr, rappdirs, readr, rlang, stringr, tibble, tidyr, withr, yaml, zoo

Encoding UTF-8

RoxygenNote 7.3.2

Suggests ggplot2, googlesheets4, here, knitr, pointblank, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

URL `https://eduaguilera.github.io/whep/`, `https://github.com/eduaguilera/whep`

BugReports `https://github.com/eduaguilera/whep/issues`

Depends R (>= 4.1.0)

LazyData true

NeedsCompilation no

Author Catalin Covaci [aut, cre] (ORCID: <`https://orcid.org/0009-0005-2186-5972`>), Eduardo Aguilera [aut, cph] (ORCID: <`https://orcid.org/0000-0003-4382-124X`>), Alice Beckmann [aut] (ORCID: <`https://orcid.org/0009-0009-6840-0258`>), Juan Infante [aut] (ORCID: <`https://orcid.org/0000-0003-1446-7181`>), Justin Morgan [aut] (ORCID: <`https://orcid.org/0009-0003-7022-4288`>), João Serra [ctb] (ORCID: <`https://orcid.org/0000-0002-3561-5350`>), European Research Council [fnd]

Maintainer Catalin Covaci <`catalin.covaci@csic.es`>

Repository CRAN

Date/Publication 2026-03-03 21:20:03 UTC

Contents

`add_area_code` *Get area codes from area names*

Description

Add a new column to an existing tibble with the corresponding code for each name. The codes are assumed to be from those defined by the `FABIO` model.

Usage

```
add_area_code(table, name_column = "area_name", code_column = "area_code")
```

Arguments

`table`	The table that will be modified with a new column.
`name_column`	The name of the column in `table` containing the names.
`code_column`	The name of the output column containing the codes.

Value

A tibble with all the contents of `table` and an extra column named `code_column`, which contains the codes. If there is no code match, an `NA` is included.

Examples

```
table <- tibble::tibble(
  area_name = c("Armenia", "Afghanistan", "Dummy Country", "Albania")
)

add_area_code(table)

table |>
  dplyr::rename(my_area_name = area_name) |>
  add_area_code(name_column = "my_area_name")

add_area_code(table, code_column = "my_custom_code")
```

`add_area_name` *Get area names from area codes*

Description

Add a new column to an existing tibble with the corresponding name for each code. The codes are assumed to be from those defined by the `FABIO` model, which them themselves come from `FAOSTAT` internal codes. Equivalences with ISO 3166-1 numeric can be found in the *Area Codes* CSV from the zip file that can be downloaded from FAOSTAT. TODO: Think about this, would be nice to use ISO3 codes but won't be enough for our periods.

Usage

```
add_area_name(table, code_column = "area_code", name_column = "area_name")
```

Arguments

`table`	The table that will be modified with a new column.
`code_column`	The name of the column in `table` containing the codes.
`name_column`	The name of the output column containing the names.

Value

A tibble with all the contents of `table` and an extra column named `name_column`, which contains the names. If there is no name match, an `NA` is included.

Examples

```
table <- tibble::tibble(area_code = c(1, 2, 4444, 3))

add_area_name(table)

table |>
  dplyr::rename(my_area_code = area_code) |>
  add_area_name(code_column = "my_area_code")

add_area_name(table, name_column = "my_custom_name")
```

`add_item_cbs_code` *Get commodity balance sheet item codes from item names*

Description

Add a new column to an existing tibble with the corresponding code for each commodity balance sheet item name. The codes are assumed to be from those defined by FAOSTAT.

Usage

```
add_item_cbs_code(
  table,
  name_column = "item_cbs_name",
  code_column = "item_cbs_code"
)
```

Arguments

`table`	The table that will be modified with a new column.
`name_column`	The name of the column in `table` containing the names.
`code_column`	The name of the output column containing the codes.

Value

A tibble with all the contents of `table` and an extra column named `code_column`, which contains the codes. If there is no code match, an `NA` is included.

Examples

```
table <- tibble::tibble(
  item_cbs_name = c("Cottonseed", "Eggs", "Dummy Item")
)
add_item_cbs_code(table)

table |>
  dplyr::rename(my_item_cbs_name = item_cbs_name) |>
  add_item_cbs_code(name_column = "my_item_cbs_name")

add_item_cbs_code(table, code_column = "my_custom_code")
```

`add_item_cbs_name` *Get commodity balance sheet item names from item codes*

Description

Add a new column to an existing tibble with the corresponding name for each commodity balance sheet item code. The codes are assumed to be from those defined by FAOSTAT.

Usage

```
add_item_cbs_name(
  table,
  code_column = "item_cbs_code",
  name_column = "item_cbs_name"
)
```

Arguments

`table`	The table that will be modified with a new column.
`code_column`	The name of the column in `table` containing the codes.
`name_column`	The name of the output column containing the names.

Value

A tibble with all the contents of `table` and an extra column named `name_column`, which contains the names. If there is no name match, an `NA` is included.

Examples

```
table <- tibble::tibble(item_cbs_code = c(2559, 2744, 9876))
add_item_cbs_name(table)

table |>
  dplyr::rename(my_item_cbs_code = item_cbs_code) |>
  add_item_cbs_name(code_column = "my_item_cbs_code")

add_item_cbs_name(table, name_column = "my_custom_name")
```

`add_item_prod_code` *Get production item codes from item names*

Description

Add a new column to an existing tibble with the corresponding code for each production item name. The codes are assumed to be from those defined by FAOSTAT.

Usage

```
add_item_prod_code(
  table,
  name_column = "item_prod_name",
  code_column = "item_prod_code"
)
```

Arguments

`table`	The table that will be modified with a new column.
`name_column`	The name of the column in `table` containing the names.
`code_column`	The name of the output column containing the codes.

Value

A tibble with all the contents of `table` and an extra column named `code_column`, which contains the codes. If there is no code match, an `NA` is included.

Examples

```
table <- tibble::tibble(
  item_prod_name = c("Rice", "Cabbages", "Dummy Item")
)
add_item_prod_code(table)

table |>
  dplyr::rename(my_item_prod_name = item_prod_name) |>
  add_item_prod_code(name_column = "my_item_prod_name")

add_item_prod_code(table, code_column = "my_custom_code")
```

`add_item_prod_name` *Get production item names from item codes*

Description

Add a new column to an existing tibble with the corresponding name for each production item code. The codes are assumed to be from those defined by FAOSTAT.

Usage

```
add_item_prod_name(
  table,
  code_column = "item_prod_code",
  name_column = "item_prod_name"
)
```

Arguments

`table`	The table that will be modified with a new column.
`code_column`	The name of the column in `table` containing the codes.
`name_column`	The name of the output column containing the names.

Value

A tibble with all the contents of `table` and an extra column named `name_column`, which contains the names. If there is no name match, an `NA` is included.

Examples

```
table <- tibble::tibble(item_prod_code = c(27, 358, 12345))
add_item_prod_name(table)

table |>
  dplyr::rename(my_item_prod_code = item_prod_code) |>
  add_item_prod_name(code_column = "my_item_prod_code")

add_item_prod_name(table, name_column = "my_custom_name")
```

build_supply_use *Supply and use tables*

Description

Create a table with processes, their inputs (*use*) and their outputs (*supply*).

Usage

```
build_supply_use(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A tibble with the supply and use data for processes. It contains the following columns:

- `year`: The year in which the recorded event occurred.
- `area_code`: The code of the country where the data is from. For code details see e.g. `add_area_name()`.
- `proc_group`: The type of process taking place. It can be one of:
  - `crop_production`: Production of crops and their residues, e.g. rice production, coconut production, etc.
  - `husbandry`: Animal husbandry, e.g. dairy cattle husbandry, non-dairy cattle husbandry, layers chickens farming, etc.
  - `processing`: Derived subproducts obtained from processing other items. The items used as inputs are those that have a non-zero processing use in the commodity balance sheet. See `get_wide_cbs()` for more details. In each process there is a single input. In some processes like olive oil extraction or soyabean oil extraction this might make sense. Others like alcohol production need multiple inputs (e.g. multiple crops work), so in this data there would not be a process like alcohol production but rather a *virtual* process like 'Wheat and products processing', giving all its possible outputs. This is a constraint because of how the data was obtained and might be improved in the future. See `get_processing_coefs()` for more details.
- `proc_cbs_code`: The code of the main item in the process taking place. Together with `proc_group`, these two columns uniquely represent a process. The main item is predictable depending on the value of `proc_group`:
  - `crop_production`: The code is from the item for which seed usage (if any) is reported in the commodity balance sheet (see `get_wide_cbs()` for more). For example, the rice code for a rice production process or the cottonseed code for the cotton production one.
  - `husbandry`: The code of the farmed animal, e.g. bees for beekeeping, non-dairy cattle for non-dairy cattle husbandry, etc.

  - `processing`: The code of the item that is used as input, i.e., the one that is processed to get other derived products. This uniquely defines a process within the group because of the nature of the data that was used, which you can see in `get_processing_coefs()`.

  For code details see e.g. `add_item_cbs_name()`.

- `item_cbs_code`: The code of the item produced or used in the process. Note that this might be the same value as `proc_cbs_code`, e.g., in rice production process for the row defining the amount of rice produced or the amount of rice seed as input, but it might also have a different value, e.g. for the row defining the amount of straw residue from rice production. For code details see e.g. `add_item_cbs_name()`.
- `type`: Can have two values:
  - `use`: The given item is an input of the process.
  - `supply`: The given item is an output of the process.
- `value`: Quantity in tonnes.

Examples

```
build_supply_use(example = TRUE)
```

`calculate_lmdi` *Calculate LMDI decomposition.*

Description

Performs LMDI (Log Mean Divisia Index) decomposition analysis with flexible identity parsing, automatic factor detection, and support for multiple periods and groupings. Supports sectoral decomposition using bracket notation for both summing and grouping operations.

Usage

```
calculate_lmdi(
  data,
  identity,
  identity_labels = NULL,
  time_var = year,
  periods = NULL,
  periods_2 = NULL,
  .by = NULL,
  rolling_mean = 1,
  output_format = "clean",
  verbose = TRUE
)
```

Arguments

`data`	A data frame containing the variables for decomposition. Must include all variables specified in the identity, time variable, and any grouping variables.
`identity`	Character. Decomposition identity in format `"target:factor1*factor2*..."`. The target appears before the colon, factors after, separated by asterisks. Supports explicit ratios with `/` and structural decomposition with `[]`.
`identity_labels`	Character vector. Custom labels for factors to use in output instead of variable names. The first element labels the target, and subsequent elements label each factor in order. Default: NULL uses variable names as-is.
`time_var`	Unquoted name of the time variable column in the data. Default: `year`. Must be numeric or coercible to numeric.
`periods`	Numeric vector. Years defining analysis periods. Each consecutive pair defines one period. Default: NULL uses all available years.
`periods_2`	Numeric vector. Additional period specification for complex multi-period analyses. Default: NULL.
`.by`	Character vector. Grouping variables for performing separate decompositions. Default: NULL (single decomposition for all data).
`rolling_mean`	Numeric. Window size for rolling mean smoothing applied before decomposition. Default: 1 (no smoothing).
`output_format`	Character. Format of output data frame. Options: `"clean"` (default) or `"total"`.
`verbose`	Logical. If TRUE (default), prints progress messages during decomposition.

Details

The LMDI method decomposes changes in a target variable into contributions from multiple factors using logarithmic mean weights. This implementation supports:

Flexible identity specification:

- Automatic factor detection from identity string.
- Support for ratio calculations (implicit division).
- Sectoral aggregation with `[]` notation.
- Sectoral grouping with `{}` notation.

Period analysis: The function can decompose changes over single or multiple periods. Periods are defined by consecutive pairs in the `periods` vector.

Grouping capabilities: Use `.by` to perform separate decompositions for different groups (e.g., countries, regions) while maintaining consistent factor structure.

Value

A tibble with LMDI decomposition results containing:

- Time variables and grouping variables (if specified).
- `additive`: Additive contributions (sum equals total change in target).

- `multiplicative`: Multiplicative indices (product equals target ratio).
- `multiplicative_log`: Log of multiplicative indices.
- Period identifiers and metadata.

Identity Syntax

The identity parameter uses a special syntax to define decomposition:

Basic format: `"target:factor1*factor2*factor3"`

Simple decomposition (no sectors):

- Basic: `"emissions:gdp*(emissions/gdp)"`
- Complete: `"emissions:(emissions/gdp)*(gdp/population)*population"`

Understanding bracket notation:

Square brackets `[]` specify variables to sum across categories, enabling structural decomposition. The bracket aggregates values BEFORE calculating ratios.

Single-level structural decomposition:

- `"emissions:activity*(activity[sector]/activity)*(emissions[sector]/activity[sector])"`
- Creates 3 factors: Activity level, Sectoral structure, Sectoral intensity.

Multi-level structural decomposition:

- Two levels: `"emissions:activity*(activity[sector]/activity)*(activity[sector+fuel]/activity[secto`
- Creates 4 factors: Activity level, Sector structure, Fuel structure, Sectoral-fuel intensity.

Data Requirements

The input data frame must contain:

- All variables mentioned in the identity.
- The time variable (default: "year").
- Grouping variables if using `.by`.
- No missing values in key variables for decomposition periods.

Examples

```
# In these examples, 'activity' is a measure of scale
# (e.g., GDP in million USD) and 'intensity' is the target
# variable per unit activity (e.g., emissions per million USD).
# The units are illustrative; adapt to your context.
# --- Shared sample data ---
data_simple <- tibble::tribble(
  ~year, ~activity, ~intensity, ~emissions,
  2010,  1000,      0.10,       100,
  2011,  1100,      0.12,       132,
  2012,  1200,      0.09,       108,
  2013,  1300,      0.10,       130
```

```
)

# --- 1. Year-over-year decomposition (default) ---
# Decompose annual emission changes into activity and intensity effects.
# The additive column sums to the total change in emissions each period.
calculate_lmdi(
  data_simple,
  identity = "emissions:activity*intensity",
  time_var = year,
  verbose = FALSE
) |>
  dplyr::select(
    period,
    component_type,
    factor_label,
    additive,
    multiplicative
  )

# --- 2. Single baseline-to-end period ---
# Pass a two-element periods vector to get a single cumulative period
# instead of year-over-year results.
calculate_lmdi(
  data_simple,
  identity = "emissions:activity*intensity",
  time_var = year,
  periods = c(2010, 2013),
  verbose = FALSE
) |>
  dplyr::select(
    period,
    component_type,
    factor_label,
    additive,
    multiplicative
  )

# --- 3. Year-over-year AND one cumulative summary period ---
# Use periods_2 to append an extra comparison period alongside the
# year-over-year results.
calculate_lmdi(
  data_simple,
  identity = "emissions:activity*intensity",
  time_var = year,
  periods = c(2010, 2011, 2012, 2013),
  periods_2 = c(2010, 2013),
  verbose = FALSE
) |>
  dplyr::select(
    period,
    component_type,
    factor_label,
    additive,
    multiplicative
  )

# --- 4. Per-country decomposition with .by ---
# Separate LMDI runs per country; results are stacked with a country column.
data_countries <- tibble::tribble(
  ~year, ~country, ~activity, ~intensity, ~emissions,
  2010, "ESP", 1000, 0.10, 100,
  2011, "ESP", 1100, 0.11, 121,
  2012, "ESP", 1200, 0.10, 120,
  2010, "FRA", 2000, 0.05, 100,
  2011, "FRA", 2200, 0.05, 110,
  2012, "FRA", 2400, 0.05, 120
)

calculate_lmdi(
  data_countries,
  identity = "emissions:activity*intensity",
  time_var = year,
  .by = "country",
  verbose = FALSE
) |>
  dplyr::select(
    country,
    period,
    component_type,
    factor_label,
    additive,
    multiplicative
  )

# --- 5. Ratio notation ---
# Express factors as explicit ratios (e.g. intensity = emissions/activity).
# Factor labels in the output preserve the ratio form for clarity.
calculate_lmdi(
  data_simple,
  identity = "emissions:(emissions/activity)*activity",
  time_var = year,
  verbose = FALSE
) |>
  dplyr::select(
    period,
    component_type,
    factor_label,
    additive,
    multiplicative
  )

# --- 6. Structural (sectoral) decomposition with [] notation ---
# Decomposes emissions into:
#   total_activity * sector_structure * sector_intensity
# [] sums the bracketed variable across sector before forming ratios,
# enabling proper structural decomposition.
```

```
data_sectors <- tibble::tribble(
  ~year, ~sector,      ~activity, ~emissions,
  2010, "industry",   600,        60,
  2010, "transport",  400,        40,
  2011, "industry",   700,        63,
  2011, "transport",  500,        55
) |>
  dplyr::group_by(year) |>
  dplyr::mutate(total_activity = sum(activity)) |>
  dplyr::ungroup()

calculate_lmdi(
  data_sectors,
  identity = paste0(
    "emissions:",
    "total_activity*",
    "(activity[sector]/total_activity)*",
    "(emissions[sector]/activity[sector])"
  ),
  time_var = year,
  verbose = FALSE
) |>
  dplyr::select(
    period,
    component_type,
    factor_label,
    additive,
    multiplicative
  )

# --- 7. Custom factor labels ---
# Replace raw variable names with readable labels for reporting.
# Supply one label per term (target first, then each factor in order).
calculate_lmdi(
  data_simple,
  identity = "emissions:activity*intensity",
  identity_labels = c(
    "Total Emissions",
    "Activity Effect",
    "Intensity Effect"
  ),
  time_var = year,
  verbose = FALSE
) |>
  dplyr::select(
    period,
    component_type,
    factor_label,
    additive,
    multiplicative
  )

# --- 8. Rolling mean smoothing before decomposition ---
# A 3-year rolling mean reduces noise in volatile series before
# computing LMDI weights. Edge years use partial windows (fewer
# than k observations) so no periods are lost.
data_smooth <- tibble::tibble(
  year      = 2010:2020,
  activity  = seq(1000, 2000, length.out = 11),
  intensity = rep(0.1, 11),
  emissions = seq(1000, 2000, length.out = 11) * 0.1
)

calculate_lmdi(
  data_smooth,
  identity = "emissions:activity*intensity",
  time_var = year,
  rolling_mean = 3,
  verbose = FALSE
) |>
  dplyr::select(
    period,
    component_type,
    factor_label,
    additive,
    multiplicative
  )
```

`calculate_nue_crops` *N soil inputs and Nitrogen Use Efficiency (NUE) for crop*

Description

N inputs (deposition, fixation, synthetic fertilizers, urban sources, manure) and N production in Spain from 1860 to the present for the GRAFS model at the provincial level. The crop NUE is defined as the percentage of produced nitrogen relative to the total nitrogen inputs to the soil. Total soil inputs are calculated as: inputs = deposition + fixation + synthetic + manure + urban

Usage

```
calculate_nue_crops(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A tibble containing nitrogen use efficiency (NUE) for crops. It includes the following columns:

- `year`: Year.

- province_name: The Spanish province.
- item: The item which was produced, defined in names_biomass_cb.
- box: One of the two systems of the GRAFS model: cropland or semi-natural agroecosystems.
- nue: Nitrogen Use Efficiency as a percentage (%).

Examples

```
calculate_nue_crops(example = TRUE)
```

---

calculate_nue_livestock

NUE for Livestock

Description

Calculates Nitrogen Use Efficiency (NUE) for livestock categories (excluding pets).

The livestock NUE is defined as the percentage of nitrogen in livestock products relative to the nitrogen in feed intake: nue = prod_n / feed_n * 100

Additionally, a mass balance is calculated to check the recovery of N in products and excretion relative to feed intake: mass_balance = (prod_n + excretion_n) / feed_n

Usage

```
calculate_nue_livestock(example = FALSE)
```

Arguments

example	If TRUE, return a small example output without downloading remote data. Default is FALSE.

Value

A tibble containing:

- year: Year
- province_name: Spanish province
- livestock_cat: Livestock category
- item: Produced item
- prod_n: Nitrogen in livestock products (Mg)
- feed_n: Nitrogen in feed intake (Mg)
- excretion_n: Nitrogen excreted (Mg)
- nue: Nitrogen Use Efficiency (%)
- mass_balance: Mass balance ratio (%)

Examples

```
calculate_nue_livestock(example = TRUE)
```

`calculate_system_nue` *System NUE*

Description

Calculates the NUE for Spain at the provincial level. The system NUE is defined as the percentage of total nitrogen production (`total_prod`) relative to the sum of all nitrogen inputs (`inputs`) into the soil system.

Usage

```
calculate_system_nue(n_soil_inputs = create_n_soil_inputs(), example = FALSE)
```

Arguments

`n_soil_inputs`	A tibble of nitrogen soil input (deposition, fixation, synthetic, manure, urban). If not provided and `example = FALSE`, it will be computed from `create_n_soil_inputs()`.
`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A tibble with the following columns:

- `year`: Year
- `province_name`: Spanish province
- `total_prod`: Total nitrogen production (Mg)
- `inputs`: Total nitrogen inputs (Mg)
- `nue_system`: System-level Nitrogen Use Efficiency (%)

Examples

```
calculate_system_nue(example = TRUE)
```

`create_n_nat_destiny` *GRAFS Nitrogen (N) flows at Spain national level*

Description

Provides N flows of the Spanish agro-food system on a national level between 1860 and 2020. This dataset is the national equivalent of the provincial GRAFS model and represents Spain as a single system without internal trade between provinces. All production, consumption and soil inputs are aggregated nationally before calculating trade with the outside.

Usage

```
create_n_nat_destiny(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A final tibble containing national N flow data by origin and destiny. It includes the following columns:

- `year`: The year in which the recorded event occurred.
- `item`: The item which was produced, defined in `names_biomass_cb`.
- `irrig_cat`: Irrigation form (irrigated or rainfed)
- `box`: One of the GRAFS model systems: cropland, Semi-natural agroecosystems, Livestock, Fish, or Agro-industry.
- `origin`: The origin category of N: Cropland, Semi-natural agroecosystems, Livestock, Fish, Agro-industry, Deposition, Fixation, Synthetic, People (waste water), Livestock (manure).
- `destiny`: The destiny category of N: population_food, population_other_uses, livestock_mono, livestock_rum (feed), export, Cropland (for N soil inputs).
- `mg_n`: Nitrogen amount in megagrams (Mg).
- `province_name`: Set to "Spain" for all national-level rows.

Examples

```
create_n_nat_destiny(example = TRUE)
```

`create_n_production` *N production for Spain*

Description

Calculates N production at the provincial level in Spain. Production is derived from consumption, export, import, and other uses.

Usage

```
create_n_production(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A tibble containing:

- `year`: Year
- `province_name`: Spanish province
- `item`: Product item
- `box`: Ecosystem box
- `prod`: Produced N (Mg)

Examples

```
create_n_production(example = TRUE)
```

`create_n_prov_destiny` *GRAFS Nitrogen (N) flows*

Description

Provides N flows of the spanish agro-food system on a provincial level between 1860 and 2020. This dataset is the the base of the GRAFS model and contains data in megagrams of N (MgN) for each year, province, item, origin and destiny. Thereby, the origin column represents where N comes from, which includes N soil inputs, imports and production. The destiny column shows where N goes to, which includes export, population food, population other uses and feed or cropland (in case of N soil inputs). Processed items, residues, woody crops, grazed weeds are taken into account.

Usage

```
create_n_prov_destiny(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A final tibble containing N flow data by origin and destiny. It includes the following columns:

- `year`: The year in which the recorded event occurred.
- `province_name`: The Spanish province where the data is from.
- `item`: The item which was produced, defined in `names_biomass_cb`.
- `irrig_cat`: Irrigation form (irrigated or rainfed)
- `box`: One of the GRAFS model systems: cropland, Semi-natural agroecosystems, Livestock, Fish, or Agro-industry.

- `origin`: The origin category of N: Cropland, Semi-natural agroecosystems, Livestock, Fish, Agro-industry, Deposition, Fixation, Synthetic, People (waste water), Livestock (manure).
- `destiny`: The destiny category of N: population_food, population_other_uses, livestock_mono, livestock_rum (feed), export, Cropland (for N soil inputs).
- `mg_n`: Nitrogen amount in megagrams (Mg).

Examples

```
create_n_prov_destiny(example = TRUE)
```

`create_n_soil_inputs` *Nitrogen (N) soil inputs for Spain*

Description

Calculates total nitrogen inputs to soils in Spain at the provincial level. This includes contributions from:

- Atmospheric deposition (`deposition`)
- Biological nitrogen fixation (`fixation`)
- Synthetic fertilizers (`synthetic`)
- Manure (excreta, solid, liquid) (`manure`)
- Urban sources (`urban`)

Special land use categories and items are aggregated:

- Semi-natural agroecosystems (e.g., Dehesa, Pasture_Shrubland)
- Firewood biomass (e.g., Conifers, Holm oak)

Usage

```
create_n_soil_inputs(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A tibble containing:

- `year`: Year
- `province_name`: Spanish province
- `item`: Crop, land use, or biomass item
- `irrig_cat`: Irrigation form (irrigated or rainfed)

- `box`: Land use or ecosystem box for aggregation
- `deposition`: N input from atmospheric deposition (Mg)
- `fixation`: N input from biological N fixation (Mg)
- `synthetic`: N input from synthetic fertilizers (Mg)
- `manure`: N input from livestock manure (Mg)
- `urban`: N input from urban sources (Mg)

Examples

```
create_n_soil_inputs(example = TRUE)
```

`expand_trade_sources` *Trade data sources*

Description

Create a new dataframe where each row has a year range into one where each row is a single year, effectively 'expanding' the whole year range.

Usage

```
expand_trade_sources(trade_sources)
```

Arguments

`trade_sources`	A tibble dataframe where each row contains the year range.

Value

A tibble dataframe where each row corresponds to a single year for a given source.

Examples

```
trade_sources <- tibble::tibble(
  Name = c("a", "b", "c"),
  Trade = c("t1", "t2", "t3"),
  Info_Format = c("year", "partial_series", "year"),
  Timeline_Start = c(1, 1, 2),
  Timeline_End = c(3, 4, 5),
  Timeline_Freq = c(1, 1, 2),
  `Imp/Exp` = "Imp",
  SACO_link = NA,
)
expand_trade_sources(trade_sources)
```

fill_linear *Fill gaps by linear interpolation, or carrying forward or backward.*

Description

Fills gaps (`NA` values) in a time-dependent variable by linear interpolation between two points, or carrying forward or backwards the last or initial values, respectively. It also creates a new variable indicating the source of the filled values.

Usage

```
fill_linear(
  data,
  value_col,
  time_col = year,
  interpolate = TRUE,
  fill_forward = TRUE,
  fill_backward = TRUE,
  value_smooth_window = NULL,
  .by = NULL
)
```

Arguments

`data`	A data frame containing one observation per row.
`value_col`	The column containing gaps to be filled.
`time_col`	The column containing time values. Default: `year`.
`interpolate`	Logical. If `TRUE` (default), performs linear interpolation.
`fill_forward`	Logical. If `TRUE` (default), carries last value forward.
`fill_backward`	Logical. If `TRUE` (default), carries first value backward.
`value_smooth_window`	An integer specifying the window size for a centered moving average applied to the variable before gap-filling. Useful for variables with high inter-annual variability. If `NULL` (default), no smoothing is applied.
`.by`	A character vector with the grouping variables (optional).

Value

A tibble data frame (ungrouped) where gaps in value_col have been filled, and a new "source" variable has been created indicating if the value is original or, in case it has been estimated, the gapfilling method that has been used.

Examples

```
sample_tibble <- tibble::tibble(
  category = c("a", "a", "a", "a", "a", "a", "b", "b", "b", "b", "b", "b"),
  year = c(
    "2015", "2016", "2017", "2018", "2019", "2020",
    "2015", "2016", "2017", "2018", "2019", "2020"
  ),
  value = c(NA, 3, NA, NA, 0, NA, 1, NA, NA, NA, 5, NA),
)
fill_linear(sample_tibble, value, .by = c("category"))
fill_linear(
  sample_tibble,
  value,
  interpolate = FALSE,
  .by = c("category"),
)
```

`fill_proxy_growth` *Fill gaps using growth rates from proxy variables*

Description

Fills missing values using growth rates from a proxy variable (reference series). Supports regional aggregations, weighting, and linear interpolation for small gaps.

Usage

```
fill_proxy_growth(
  data,
  value_col,
  proxy_col,
  time_col = year,
  .by = NULL,
  max_gap = Inf,
  max_gap_linear = 3,
  fill_scope = NULL,
  value_smooth_window = NULL,
  proxy_smooth_window = 1,
  output_format = "clean",
  verbose = TRUE
)
```

Arguments

`data`	A data frame containing time series data.
`value_col`	The column containing values to fill.
`proxy_col`	Character or vector. Proxy variable(s) for calculating growth rates. Supports multiple syntax formats:

- **Simple numeric proxy** (e.g., `"population"`): Auto-detects numeric columns and uses them as proxy variable. Inherits the `.by` parameter to compute proxy values per group.
- **Simple categorical proxy** (e.g., `"region"`): Auto-detects categorical columns and interprets as `value_col:region`. Aggregates `value_col` by the specified groups.
- **Advanced syntax** (e.g., `"gdp:region"`): Format is `"variable:group1+group2"`. Aggregates variable by specified groups.
- **Hierarchical fallback** (e.g., `c("population", "gdp:region")`): Tries first proxy, falls back to second if first fails.
- **Weighted aggregation** (e.g., `"gdp[population]"`): Weight variable by specified column during aggregation.

`time_col`	The column containing time values. Default: `year`.
`.by`	A character vector with the grouping variables (optional).
`max_gap`	Numeric. Maximum gap size to fill using growth method. Default: Inf.
`max_gap_linear`	Numeric. Maximum gap size for linear interpolation fallback. Default: 3.
`fill_scope`	Quosure. Filter expression to limit filling scope. Default: NULL.
`value_smooth_window`	Integer. Window size for a centered moving average applied to the value column before gap-filling. Useful for variables with high inter-annual variability. If `NULL` (default), no smoothing is applied.
`proxy_smooth_window`	Integer. Window size for moving average smoothing of proxy reference values before computing growth rates. Default: 1.
`output_format`	Character. Output format: "clean" or "detailed". Default: "clean".
`verbose`	Logical. Print progress messages. Default: TRUE.

Details

Combined Growth Sequence (Hierarchical Interpolation):

When using multiple proxies with hierarchical fallback, the function implements an intelligent combined growth sequence strategy:

1. Better proxies (earlier in hierarchy) are tried first for each gap.
2. If a better proxy has partial coverage within a gap, those growth rates are used for the covered positions.
3. Fallback proxies fill only the remaining positions where better proxies are not available.
4. Values filled by better proxies are protected from being overwritten.

Value

A data frame with filled values. If output_format = "clean", returns original columns with updated value_col and added source column. If "detailed", includes all intermediate columns.

See Also

`fill_linear()`, `fill_sum()`

Examples

```
# Fill GDP using population as proxy
data <- tibble::tibble(
  country = rep("ESP", 4),
  year = 2010:2013,
  gdp = c(1000, NA, NA, 1200),
  population = c(46, 46.5, 47, 47.5)
)

fill_proxy_growth(
  data,
  value_col = gdp,
  proxy_col = "population",
  .by = "country"
)
```

`fill_sum` *Fill gaps summing the previous value of a variable to the value of another variable.*

Description

Fills gaps in a variable with the sum of its previous value and the value of another variable. When a gap has multiple observations, the values are accumulated along the series. When there is a gap at the start of the series, it can either remain unfilled or assume an invisible 0 value before the first observation and start filling with cumulative sum.

Usage

```
fill_sum(
  data,
  value_col,
  change_col,
  time_col = year,
  start_with_zero = TRUE,
  .by = NULL
)
```

Arguments

`data`	A data frame containing one observation per row.
`value_col`	The column containing gaps to be filled.
`change_col`	The column whose values will be used to fill the gaps.

`time_col`	The column containing time values. Default: `year`.
`start_with_zero`	Logical. If TRUE (default), assumes an invisible 0 value before the first observation and fills with cumulative sum starting from the first change_col value. If FALSE, starting NA values remain unfilled.
`.by`	A character vector with the grouping variables (optional).

Value

A tibble dataframe (ungrouped) where gaps in value_col have been filled, and a new "source" variable has been created indicating if the value is original or, in case it has been estimated, the gapfilling method that has been used.

Examples

```
sample_tibble <- tibble::tibble(
  category = c("a", "a", "a", "a", "a", "a", "b", "b", "b", "b", "b", "b"),
  year = c(
    "2015", "2016", "2017", "2018", "2019", "2020",
    "2015", "2016", "2017", "2018", "2019", "2020"
  ),
  value = c(NA, 3, NA, NA, 0, NA, 1, NA, NA, NA, 5, NA),
  change_variable = c(1, 2, 3, 4, 1, 1, 0, 0, 0, 0, 0, 1)
)
fill_sum(
  sample_tibble,
  value,
  change_variable,
  start_with_zero = FALSE,
  .by = c("category")
)
fill_sum(
  sample_tibble,
  value,
  change_variable,
  start_with_zero = TRUE,
  .by = c("category")
)
```

`get_bilateral_trade` *Bilateral trade data*

Description

Reports trade between pairs of countries in given years.

Usage

```
get_bilateral_trade(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A tibble with the reported trade between countries. For efficient memory usage, the tibble is not exactly in tidy format. It contains the following columns:

- `year`: The year in which the recorded event occurred.
- `item_cbs_code`: FAOSTAT internal code for the item that is being traded. For code details see e.g. `add_item_cbs_name()`.
- `bilateral_trade`: Square matrix of `NxN` dimensions where `N` is the total number of countries being considered. The matrix row and column names are exactly equal and they represent country codes.
  - Row name: The code of the country where the data is from. For code details see e.g. `add_area_name()`.
  - Column name: FAOSTAT internal code for the country that is importing the item. See row name explanation above.

  If `m` is the matrix, the value at `m["A", "B"]` is the trade in tonnes from country `"A"` to country `"B"`, for the corresponding year and item. The matrix can be considered *balanced*. This means:

  - The sum of all values from row `"A"`, where `"A"` is any country, should match the total exports from country `"A"` reported in the commodity balance sheet (which is considered more accurate for totals).
  - The sum of all values from column `"A"`, where `"A"` is any country, should match the total imports into country `"A"` reported in the commodity balance sheet (which is considered more accurate for totals).

  The sums may not be exactly the expected values because of precision issues and/or the iterative proportional fitting algorithm not converging fast enough, but should be relatively very close to the desired totals.

The step by step approach to obtain this data tries to follow the FABIO model and is explained below. All the steps are performed separately for each group of year and item.

- From the FAOSTAT reported bilateral trade, there are sometimes two values for one trade flow: the exported amount claimed by the reporter country and the import amount claimed by the partner country. Here, the export data was preferred, i.e., if country `"A"` says it exported `X` tonnes to country `"B"` but country `"B"` claims they got `Y` tonnes from country `"A"`, we trust the export data `X`. This choice is only needed if there exists a reported amount from both sides. Otherwise, the single existing report is chosen.
- Complete the country data, that is, add any missing combinations of country trade with NAs, which will be estimated later. In the matrix form, this doesn't increase the memory usage since we had to build a matrix anyway (for the balancing algorithm), and the *empty* parts also take up memory. This is also done for total imports/exports from the commodity balance sheet, but these are directly filled with 0s instead.

- The total imports and exports from the commodity balance sheet are balanced by downscaling the largest of the two to match the lowest. This is done in the following way:
  - If `total_imports` > `total_exports`: Set `import` as `total_exports * import / total_import`.
  - If `total_exports` > `total_exports`: Set `export` as `total_exports * export / total_export`.
- The missing data in the matrix must be estimated. It's done like this:
  - For each pair of exporter `i` and importer `j`, we estimate a bilateral trade `m[i, j]` using the export shares of `i` and import shares of `j` from the commodity balance sheet:
    - `est_1 <- exports[i] * imports[j] / sum(imports)`, i.e., total exports of country `i` spread among other countries' import shares.
    - `est_2 <- imports[j] * exports[i] / sum(exports)`, i.e. total imports of country `j` spread among other countries' export shares.
    - `est <- (est_1 + est_2) / 2`, i.e., the mean of both estimates.

    In the above computations, exports and imports are the original values before they were balanced.
  - The estimates for data that already existed (i.e. non-NA) are discarded. For the ones left, for each row (i.e. exporter country), we get the difference between its balanced total export and the sum of original non-estimated data. The result is the `gap` we can actually fill with estimates, so as to not get past the reported total export. If the sum of non-discarded estimates is larger, it must be downscaled and spread by computing `gap * non_discarded_estimate / sum(non_discarded_estimates)`.
  - The estimates are divided by a *trust factor*, in the sense that we don't rely on the whole value, thinking that a non-present value might actually be because that specific trade was 0, so we don't overestimate too much. The chosen factor is 10%, so only 10% of the estimate's value is actually used to fill the NA from the original bilateral trade matrix.
- The matrix is balanced, as mentioned before, using the iterative proportional fitting algorithm. The target sums for rows and columns are respectively the balanced exports and imports computed from the commodity balance sheet.

Examples

```
get_bilateral_trade(example = TRUE)
```

`get_faostat_data` *Scrapes activity_data from FAOSTAT and slightly post-processes it*

Description

Important: Dynamically allows for the introduction of subsets as `"..."`.

Note: overhead by individually scraping FAOSTAT code QCL for crop data; it's fine.

Usage

```
get_faostat_data(activity_data, ...)
```

Arguments

`activity_data`	activity data required from FAOSTAT; needs to be one of `c('livestock','crop_area','crop_yield',`
`...`	can be whichever column name from `get_faostat_bulk`, particularly `year`, `area` or `ISO3_CODE`.

Value

`data.frame` of FAOSTAT for `activity_data`; default is for all years and countries.

Examples

```
## Not run:
get_faostat_data("livestock", year = 2010, area = "Portugal")

## End(Not run)
```

`get_feed_intake` *Livestock feed intake*

Description

Get amount of items used for feeding livestock.

Usage

```
get_feed_intake(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A tibble with the feed intake data. It contains the following columns:

- `year`: The year in which the recorded event occurred.
- `area_code`: The code of the country where the data is from. For code details see e.g. `add_area_name()`.
- `live_anim_code`: Commodity balance sheet code for the type of livestock that is fed. For code details see e.g. `add_item_cbs_name()`.
- `item_cbs_code`: The code of the item that is used for feeding the animal. For code details see e.g. `add_item_cbs_name()`.
- `feed_type`: The type of item that is being fed. It can be one of:
  - `animals`: Livestock product, e.g. `Bovine Meat`, `Butter, Ghee`, etc.
  - `crops`: Crop product, e.g. `Vegetables, Other`, `Oats`, etc.

  - `residues`: Crop residue, e.g. `Straw`, `Fodder legumes`, etc.
  - `grass`: Grass, e.g. `Grassland`, `Temporary grassland`, etc.
  - `scavenging`: Other residues. Single `Scavenging` item.
- `supply`: The computed amount in tonnes of this item that should be fed to this animal, when sharing the total item `feed` use from the Commodity Balance Sheet among all livestock.
- `intake`: The actual amount in tonnes that the animal needs, which can be less than the theoretical used amount from `supply`.
- `intake_dry_matter`: The amount specified by `intake` but only considering dry matter, so it should be less than `intake`.
- `loss`: The amount that is not used for feed. This is `supply - intake`.
- `loss_share`: The percent that is lost. This is `loss / supply`.

Examples

```
get_feed_intake(example = TRUE)
```

`get_primary_production` *Primary items production*

Description

Get amount of crops, livestock and livestock products.

Usage

```
get_primary_production(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A tibble with the item production data. It contains the following columns:

- `year`: The year in which the recorded event occurred.
- `area_code`: The code of the country where the data is from. For code details see e.g. `add_area_name()`.
- `item_prod_code`: FAOSTAT internal code for each produced item.
- `item_cbs_code`: FAOSTAT internal code for each commodity balance sheet item. The commodity balance sheet contains an aggregated version of production items. This field is the code for the corresponding aggregated item.

- `live_anim_code`: Commodity balance sheet code for the type of livestock that produces the livestock product. It can be:
  - `NA`: The entry is not a livestock product.
  - Non-`NA`: The code for the livestock type. The name can also be retrieved by using `add_item_cbs_name()`.
- `unit`: Measurement unit for the data. Here, keep in mind three groups of items: crops (e.g. `Apples and products`, `Beans`...), livestock (e.g. `Cattle, dairy`, `Goats`...) and livestock products (e.g. `Poultry Meat`, `Offals, Edible`...). Then the unit can be one of:
  - `tonnes`: Available for crops and livestock products.
  - `ha`: Hectares, available for crops.
  - `t_ha`: Tonnes per hectare, available for crops.
  - `heads`: Number of animals, available for livestock.
  - `LU`: Standard Livestock Unit measure, available for livestock.
  - `t_head`: tonnes per head, available for livestock products.
  - `t_LU`: tonnes per Livestock Unit, available for livestock products.
- `value`: The amount of item produced, measured in `unit`.

Examples

```
get_primary_production(example = TRUE)
```

`get_primary_residues` *Crop residue items*

Description

Get type and amount of residue produced for each crop production item.

Usage

```
get_primary_residues(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A tibble with the crop residue data. It contains the following columns:

- `year`: The year in which the recorded event occurred.
- `area_code`: The code of the country where the data is from. For code details see e.g. `add_area_name()`.

- `item_cbs_code_crop`: FAOSTAT internal code for each commodity balance sheet item. This is the crop that is generating the residue.
- `item_cbs_code_residue`: FAOSTAT internal code for each commodity balance sheet item. This is the obtained residue. In the commodity balance sheet, this can be three different items right now:
  - `2105`: `Straw`
  - `2106`: `Other crop residues`
  - `2107`: `Firewood`

  These are actually not FAOSTAT defined items, but custom defined by us. When necessary, FAOSTAT codes are extended for our needs.
- `value`: The amount of residue produced, measured in tonnes.

Examples

```
get_primary_residues(example = TRUE)
```

get_processing_coefs *Processed products share factors*

Description

Reports quantities of commodity balance sheet items used for `processing` and quantities of their corresponding processed output items.

Usage

```
get_processing_coefs(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A tibble with the quantities for each processed product. It contains the following columns:

- `year`: The year in which the recorded event occurred.
- `area_code`: The code of the country where the data is from. For code details see e.g. `add_area_name()`.
- `item_cbs_code_to_process`: FAOSTAT internal code for each one of the items that are being processed and will give other subproduct items. For code details see e.g. `add_item_cbs_name()`.
- `value_to_process`: tonnes of this item that are being processed. It matches the amount found in the `processing` column from the data obtained by `get_wide_cbs()`.
- `item_cbs_code_processed`: FAOSTAT internal code for each one of the subproduct items that are obtained when processing. For code details see e.g. `add_item_cbs_name()`.

- `initial_conversion_factor`: estimate for the number of tonnes of `item_cbs_code_processed` obtained for each tonne of `item_cbs_code_to_process`. It will be used to compute the `final_conversion_factor`, which leaves everything balanced. TODO: explain how it's computed.
- `initial_value_processed`: first estimate for the number of tonnes of `item_cbs_code_processed` obtained from `item_cbs_code_to_process`. It is computed as `value_to_process` * `initial_conversion_factor`.
- `conversion_factor_scaling`: computed scaling needed to adapt `initial_conversion_factor` so as to get a final balanced total of subproduct quantities. TODO: explain how it's computed.
- `final_conversion_factor`: final used estimate for the number of tonnes of `item_cbs_code_processed` obtained for each tonne of `item_cbs_code_to_process`. It is computed as `initial_conversion_factor` * `conversion_factor_scaling`.
- `final_value_processed`: final estimate for the number of tonnes of `item_cbs_code_processed` obtained from `item_cbs_code_to_process`. It is computed as `initial_value_processed` * `final_conversion_factor`.

For the final data obtained, the quantities `final_value_processed` are balanced in the following sense: the total sum of `final_value_processed` for each unique tuple of `(year, area_code, item_cbs_code_processed)` should be exactly the quantity reported for that year, country and `item_cbs_code_processed` item in the `production` column obtained from `get_wide_cbs()`. This is because they are not primary products, so the amount from 'production' is actually the amount of subproduct obtained. TODO: Fix few data where this doesn't hold.

Examples

```
get_processing_coefs(example = TRUE)
```

`get_wide_cbs` *Commodity balance sheet data*

Description

States supply and use parts for each commodity balance sheet (CBS) item.

Usage

```
get_wide_cbs(example = FALSE)
```

Arguments

`example`	If `TRUE`, return a small example output without downloading remote data. Default is `FALSE`.

Value

A tibble with the commodity balance sheet data in wide format. It contains the following columns:

- `year`: The year in which the recorded event occurred.
- `area_code`: The code of the country where the data is from. For code details see e.g. `add_area_name()`.
- `item_cbs_code`: FAOSTAT internal code for each item. For code details see e.g. `add_item_cbs_name()`.

The other columns are quantities (measured in tonnes), where total supply and total use should be balanced.

For supply:

- `production`: Produced locally.
- `import`: Obtained from importing from other countries.
- `stock_retrieval`: Available as net stock from previous years. For ease, only one stock column is included here as supply. If the value is positive, there is a stock quantity available as supply. Otherwise, it means a larger quantity was stored for later years and cannot be used as supply, having to deduce it from total supply. Since in this case it is negative, the total supply is still computed as the sum of all of these.

For use:

- `food`: Food for humans.
- `feed`: Food for animals.
- `export`: Released as export for other countries.
- `seed`: Intended for new production.
- `processing`: The product will be used to obtain other subproducts.
- `other_uses`: Any other use not included in the above ones.

There is an additional column `domestic_supply` which is computed as the total use excluding `export`.

Examples

```
get_wide_cbs(example = TRUE)
```

`harmonize_interpolate` *Harmonize advanced cases with interpolation for 1:N groups*

Description

Harmonize data containing `"simple"` and `"1:n"` mappings. `"simple"` covers both 1:1 and N:1 relationships (values are summed). For `"1:n"` groups (one original item splits into several harmonized items) this function computes value shares across the full year range, interpolates missing shares, and applies them to split values.

Important for 1:n mappings: For each original item that splits into multiple harmonized items (e.g., "wheatrice" into "wheat" and "rice"), provide **one row per target** `item_code_harm`. Each row should have the same `item`, `year`, and `value`, differing only in `item_code_harm`. For example, to disaggregate "wheatrice":

- Row 1: `item` = "wheatrice", `item_code_harm` = 1
- Row 2: `item` = "wheatrice", `item_code_harm` = 2

Do not provide a single row; the function will not create duplicates automatically.

Usage

```
harmonize_interpolate(data, ...)
```

Arguments

`data`	A data frame containing at least columns: • `item`: String, original item name. • `item_code_harm`: Numeric, code for harmonized item. • `year`: Numeric, year of observation. • `value`: Numeric, value of observation. • `type`: String, `"simple"` or `"1:n"`.
`...`	Additional grouping columns provided as bare names.

Value

A tibble with columns:

- `item_code`: Numeric, code for harmonized item.
- `year`: Numeric, year of observation.
- `value`: Numeric, summed value of observation.
- and any additional grouping columns.

Examples

```
# Simple-only data (no 1:n rows)
df_simple <- tibble::tribble(
  ~item,      ~item_code_harm, ~year, ~value, ~type,
  "wheat",    1,               2000,   5,     "simple",
  "barley",   2,               2000,   3,     "simple",
  "oats",     2,               2000,   2,     "simple"
)
harmonize_interpolate(df_simple)

# Mixed simple + 1:n data
df_mixed <- tibble::tribble(
  ~item,       ~item_code_harm, ~year, ~value, ~type,
  "wheatrice", 1,               2000,  20,     "1:n",
  "wheatrice", 2,               2000,  20,     "1:n",
  "wheat",     1,               2000,   8,     "simple",
  "rice",      2,               2000,  12,     "simple"
)
harmonize_interpolate(df_mixed)

# Multiple years with share interpolation
# Shares are known in 2000 and 2002; 2001 is interpolated.
df_years <- tibble::tribble(
  ~item,       ~item_code_harm, ~year, ~value, ~type,
  "wheat",     1,               2000,   6,     "simple",
  "rice",      2,               2000,   4,     "simple",
  "wheatrice", 1,               2001,  10,     "1:n",
  "wheatrice", 2,               2001,  10,     "1:n",
  "wheat",     1,               2002,   8,     "simple",
  "rice",      2,               2002,   2,     "simple"
)
harmonize_interpolate(df_years)

# With extra grouping columns
df_grouped <- tibble::tribble(
  ~item,       ~item_code_harm, ~year, ~value, ~type,    ~country,
  "wheat",     1,               2000,   6,     "simple", "usa",
  "rice",      2,               2000,   4,     "simple", "usa",
  "wheatrice", 1,               2001,  10,     "1:n",    "usa",
  "wheatrice", 2,               2001,  10,     "1:n",    "usa",
  "wheat",     1,               2002,   8,     "simple", "usa",
  "rice",      2,               2002,   2,     "simple", "usa",
  "wheat",     1,               2002,   8,     "simple", "germany"
)
harmonize_interpolate(df_grouped, country)
```

`harmonize_simple` *Harmonize rows labeled "simple" by summing values*

Description

Sum `value` for rows where `type == "simple"`. This covers both 1:1 and N:1 item mappings, since in both cases the values are simply summed. The results are grouped by `item_code_harm`, `year` and any additional grouping columns supplied via `...`.

Usage

```
harmonize_simple(data, ...)
```

Arguments

`data`	A data frame containing at least columns: • `item_code_harm`: Numeric, code for harmonized item. • `year`: Numeric, year of observation. • `value`: Numeric, value of observation. • `type`: String, harmonization type. Only `"simple"` rows are used.
`...`	Additional grouping columns supplied as bare names.

Value

A tibble with columns:

- `item_code_harm`: Numeric, code for harmonized item.
- `year`: Numeric, year of observation.
- `value`: Numeric, summed value of observation.
- and any additional grouping columns.

Examples

```
# 1:1 mapping: one original item -> one harmonized code
df_one_to_one <- tibble::tribble(
  ~item_code_harm, ~year, ~value, ~type,
  1,               2000,  10,     "simple",
  2,               2000,   3,     "simple",
  1,               2001,  12,     "simple",
  2,               2001,   5,     "simple"
)
harmonize_simple(df_one_to_one)

# N:1 mapping: multiple items map to the same code
df_many_to_one <- tibble::tribble(
  ~item_code_harm, ~year, ~value, ~type,
  1,               2000,   4,     "simple",
  1,               2000,   6,     "simple",
  2,               2000,   3,     "simple"
)
harmonize_simple(df_many_to_one)

# With an extra grouping column (e.g. country)
```

```
df_grouped <- tibble::tribble(
  ~item_code_harm, ~year, ~value, ~type,    ~country,
  1,               2000,   4,     "simple", "usa",
  1,               2000,   6,     "simple", "usa",
  1,               2000,   9,     "simple", "germany",
  2,               2000,   3,     "simple", "usa"
)
harmonize_simple(df_grouped, country)

# Rows with type != "simple" are ignored
df_mixed <- tibble::tribble(
  ~item_code_harm, ~year, ~value, ~type,
  1,               2000,  10,     "simple",
  1,               2000,  99,     "1:n",
  2,               2000,   3,     "simple"
)
harmonize_simple(df_mixed)
```

items_cbs *Commodity balance sheet items*

Description

Defines name/code correspondences for commodity balance sheet (CBS) items.

Usage

```
items_cbs
```

Format

A tibble where each row corresponds to one CBS item. It contains the following columns:

- `item_cbs_code`: A numeric code used to refer to the CBS item.
- `item_cbs_name`: A natural language name for the item.
- `item_type`: An ad-hoc grouping of items. This is a work in progress evolving depending on our needs, so for now it only has two possible values:
  - `livestock`: The CBS item represents a live animal.
  - `other`: Not any of the previous groups.

Source

Inspired by FAOSTAT data.

`items_prod` *Primary production items*

Description

Defines name/code correspondences for production items.

Usage

```
items_prod
```

Format

A tibble where each row corresponds to one production item. It contains the following columns:

- `item_prod_code`: A numeric code used to refer to the item.
- `item_prod_name`: A natural language name for the item.
- `item_type`: An ad-hoc grouping of items. This is a work in progress evolving depending on our needs, so for now it only has two possible values:
  - `crop_product`: The CBS item represents a crop product.
  - `other`: Not any of the previous groups.

Source

Inspired by FAOSTAT data.

`polities` *Polities*

Description

Defines name/code correspondences for polities (political entities).

Usage

```
polities
```

Format

A tibble where each row corresponds to one polity. It contains the following columns: TODO: On polities Pull Request, coming soon

`whep_inputs` *External inputs*

Description

The information needed for accessing external datasets used as inputs in our modeling.

Usage

```
whep_inputs
```

Format

A tibble where each row corresponds to one external input dataset. It contains the following columns:

- `alias`: An internal name used to refer to this dataset, which is the expected name when trying to get the dataset with `whep_read_file()`.
- `board_url`: The public static URL where the data is found, following the concept of a *board* from the `pins` package, which is what we use for storing these input datasets.
- `version`: The specific version of the dataset, as defined by the `pins` package. The version is a string similar to `"20250714T123343Z-114b5"`. This version is the one used by default if no `version` is specified when calling `whep_read_file()`. If you want to use a different one, you can find the available versions of a file by using `whep_list_file_versions()`.

Source

Created by the package authors.

`whep_list_file_versions` *Input file versions*

Description

Lists all existing versions of an input file from `whep_inputs`.

Usage

```
whep_list_file_versions(file_alias)
```

Arguments

`file_alias`	Internal name of the requested file. You can find the possible values in the `whep_inputs` dataset.

Value

A tibble where each row is a version. For details about its format, see `pins::pin_versions()`.

Examples

```
whep_list_file_versions("read_example")
```

`whep_read_file` *Download, cache and read files*

Description

Used to fetch input files that are needed for the package's functions and that were built in external sources and are too large to include directly. This is a public function for transparency purposes, so that users can inspect the original inputs of this package that were not directly processed here.

If the requested file doesn't exist locally, it is downloaded from a public link and cached before reading it. This is all implemented using the `pins` package. It supports multiple file formats and file versioning.

Usage

```
whep_read_file(file_alias, type = "parquet", version = NULL)
```

Arguments

`file_alias`	Internal name of the requested file. You can find the possible values in the `alias` column of the `whep_inputs` dataset.
`type`	The extension of the file that must be read. Possible values: • `parquet`: This is the default value for code efficiency reasons. • `csv`: Mainly available for those who want a more human-readable option. If the `parquet` version is available, this is useless because this function already returns the dataset in an `R` object, so the origin is irrelevant, and `parquet` is read faster. Saving each file in both formats is for transparency and accessibility purposes, e.g., having to share the data with non-programmers who can easily import a CSV into a spreadsheet. You will most likely never have to set this option manually unless for some reason a file could not be supplied in e.g. `parquet` format but was in another one.
`version`	The version of the file that must be read. Possible values: • `NULL`: This is the default value. A frozen version is chosen to make the code reproducible. Each release will have its own frozen versions. The version is the string that can be found in `whep_inputs` in the `version` column. • `"latest"`: This overrides the frozen version and instead fetches the latest one that is available. This might or might not match the frozen version. • Other: A specific version can also be used. For more details read the `version` column information from `whep_inputs`.

Value

A tibble with the dataset. Some information about each dataset can be found in the code where it's used as input for further processing.

Examples

```
whep_read_file("read_example")
whep_read_file("read_example", type = "parquet", version = "latest")
whep_read_file(
  "read_example",
  type = "csv",
  version = "20250721T152646Z-ce61b"
)
```

Index