Package 'pems.utils'

July 23, 2025

Title Portable Emissions (and Other Mobile) Measurement System Utilities

Version 0.3.0.8

Date 2025-06-20

Description
Utility functions for the handling, analysis and visualisation of data from portable emissions measurement systems ('PEMS') and other similar mobile activity monitoring devices. The package includes a dedicated 'pems' data class that manages many of the quality control, unit handling and data archiving issues that can hinder efforts to standardise 'PEMS' research.

Type Package

Author Karl Ropkins [aut, cre],
Adrian Felipe Ortega Calle [ctb]

Maintainer Karl Ropkins <karl.ropkins@gmail.com>

URL http://pems.r-forge.r-project.org/

Depends R (>= 2.10.0)

Imports lattice, loa (>= 0.3.1), methods, utils, grid, baseline, ggplot2, rlang, tibble, dplyr

License GPL (>= 2)

LazyLoad yes

LazyData yes

RoxygenNote 7.3.2

Encoding UTF-8

NeedsCompilation no

Repository CRAN

Date/Publication 2025-06-20 21:30:02 UTC

Contents

`pems.utils-package` *pems.utils*

Description

The R package pems.utils contains a range of functions for the routine handling and analysis of data collected by portable emissions measurement systems (PEMS) and other similar mobile monitoring systems.

Details

Package:	pems.utils
Type:	Package
Version:	0.3.0.7
Date:	2024-12-28
License:	GPL (>= 2)
LazyLoad:	yes

The `pems.utils` functions have been arranged according to usage, as follows:

1. Getting data in and out of `pems.utils`.

1.1. Functions for making and importing datasets for use with `pems.utils`: `pems`, `import2PEMS`, etc.

1.2. Exporting data from pems objects and R: `export.data`.

2. Data Structure and General Handling

2.1. The pems object structure: `pems.structure`, `getPEMSElement`, `pemsData`, etc.

3. Generic pems handling

3.1. pems objects, `pems.generics`.

3.2. pems.element objects, `pems.element.generics`.

4. Structure Handling

4.1. Merging pems objects: `merge.pems`, `align`, etc.

4.2. Referencing pems objects: `referencing.pems.data`, `refRow`, etc.

4.3. Time handling functions: `regularize`, etc.

4.4. Unit handler functions: `getUnits`, `setUnits`, `convertUnits`, etc.

5. pems Data Handling

5.1. Plots for pems objects: `pems.plots`, `latticePlot`, `pemsPlot`, etc.

6. Calculations

6.1. Common calculations: `common.calculations`, `calcDistance`, `calcAccel`, etc.

6.2. Common `check...` functions for the routine handling of function arguments/user inputs.

6.3. Other correction code

6.4. Analysing data in pems objects: `summary.reports`

7. Refernce datasets, examples, look-up tables, etc.

7.1. Example datasets: `pems.1`.

7.2. look-up tables: `ref.unit.conversions`, etc.

8. Specialist code

8.1. VSP calculations: `calcVSP`, etc.

8.2. Emissions calculations: `calcEm`, etc.

9. Other Code

9.1. Tidyverse related code... `pems.tidyverse`

Author(s)

Karl Ropkins Maintainer: Karl Ropkins <k.ropkins@its.leeds.ac.uk>

References

Functions in `pems.utils` make extensive use of code developed by others. In particular, I gratefully acknowledge the huge contributions of the R Core Team and numerous contributors in developing and maintaining R:

R Development Core Team (2011). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

See Also

pems, import2PEMS

1.1.make.import.data *making and importing data*

Description

Various pems.utils functions to make and import data as pems objects.

Usage

```
#making pems objects

is.pems(x, full.test = TRUE, ...)

pems(x, units = NULL, constants = NULL, history = NULL,
          ...)

pems.element(x, name = NULL, units = NULL,
          ...)

#associated

isPEMS(...)
makePEMS(...)
makePEMSElement(...)
rebuildPEMS(x, ...)

## S3 method for class 'data.frame'
as.pems(x, ...)

#importing data as pems objects

#general

import2PEMS(file.name = file.choose(), ...,
          file.reader = read.delim,
          output="pems")

importTAB2PEMS(..., file.reader = read.delim)

importCSV2PEMS(..., file.reader = read.csv)

#Horiba OBS
```

```
importOBS2PEMS(file.name = file.choose(),
          pems = "Horiba OBS",
          constants = NULL, history = NULL,
          analytes = c("co", "co2", "nox", "hc"),
          fuel = c("petrol", "diesel", "gasoline"), ...)

importOB12PEMS(file.name = file.choose(),
          pems = "Horiba OBS",
          constants = NULL, history = NULL,
          analytes = c("co", "co2", "nox", "hc"),
          fuel = c("petrol", "diesel", "gasoline"), ...)

#3DATX parSYNC

importParSYNC2PEMS(file.name = file.choose(),
          reset.signals = TRUE,
          history = NULL, constants = NULL,
          pm.analyzer = "parSYNC", ... )

#3DATX CAGE
importCAGE2PEMS(..., calibrator = "CAGE")

#Sensors Inc SEMTECH

importSEMTECH2PEMS(file.name = file.choose(),
          history = NULL, constants = NULL,
          pems = "SEMTECH", ...)


#RoyalTek GPS

importRoyalTek2PEMS(file.name = file.choose(),
          file.type = c("special", "txt", "nmea"),
          vbox = "RoyalTEk", history = NULL,
          constants = NULL, ...)


#KML files

importKML2PEMS(file.name = file.choose(), history = NULL,
          constants = NULL, source = "Unknown", ...)
```

Arguments

x	(A required object) For `is.pems`, any object to be tested as a `pems` object. For `pems`, an object to be used as the starting point to make a `pems` object, so typically a `data.frame` or another `pems` object. For `pems.element`, an object to be used as the starting point to make a `pems.element`. For `rebuildPEMS`, a `pems`

object to be rebuilt.

`full.test` (Logical) For `is.pems`, should the full pems test be applied and the pems structure confirmed?

`...` (Optional) Other arguments, handling varies. For `is.pems` these are ignored. For pems these are added to the pems object unmodified. For `import...` functions, these are passed on and added to the `constants` component of the pems object. Note: This different handling is experimental and may be subject to change in future.

`units`, `constants`, `history`
(Default pems arguments) These are arguments that are routinely generated for pems objects. `units` holds unit ids for unit management, `constants` holds constants that should used specifically with data in the pems object, and `history` holds the pems object modification history.

`name` (Default pems.element argument) `name` (and `units`) are arguments that are routinely generated for `pems.element` objects.

`file.name` (file connection, etc.) For `import...` functions, the file/source to be imported. Note: the default, `file.name = file.choose()`, automatically opens a file browser if this argument is not supplied.

`file.type`, `file.reader`
Data reader parameters for some `import...` functions. `file.type` is the type of file to be imported. Note: Some `import...` functions can handle more than one file type, and `file.type = "[option]"` should be used to identify these. (Note: `file.type` options are typically file type identifiers, such as the file extensions, and a default `'special'`, which leaves the choice to the function. This way this option can typically be ignored unless, e.g. the function does not recognise the file type but the user knows it and wants to force the method.) `file.reader` identifies the R method/function that should be used to read data from the supplied file. For example, for `importTAB2PEMS` and `importCSV2PEMS`, by default, these are the standard R `read...` functions `read.delim` and `read.csv`, respectively.

`output` Where included in formal arguments, an option to control function output.

`pems`, `vbox`, `pm.analyzer`, `calibrator`, `source`
(Character vectors) For some `import...` functions, data source descriptions may be automatically added to the pems object. `pems` and `vbox` are two examples, but others, such as vehicle and fuel descritpions can also be added in a similar fashion. Note: These are for user-reference, so can say whatever you want.

`analytes` (Character vector) For `import...` functions, the names of any pems elements to be tagged as analyte concentrations. Note: If the PEMS unit reports concentrations rather than emissions it is often useful to identify these at import to avoid confusion, and to simplify later handling. So, if encountered, analyte names are prefixed with the term `'conc.'`.

`fuel` Some `import...` functions that handle exhaust monitoring system data may assume fuel types when calibrating inputs or calculating constants. In such cases the `fuel` argument is also included to identify which fuel was used.

`reset.signals` (Logical or Character vector) For `importParSYNC2PEMS`, should any raw signal be reset? The default (`TRUE`) reverses the sign of opacity and ionization signals.

Details

is.pems tests if an object is/is not a pems object.

pems makes a pems object using supplied data and information.

pems.element makes a pems.element object using supplied data and information.

as.pems... functions attempt to convert a supplied object into a pems object. Currently, there is only a data.frame method and (by default) a pems method.

isPEMS, makePEMS and makePEMSElement are historical code, retained for backward compatibility.

rebuildPEMS rebuilds pems object as a different build version.

Crude import... functions import simple file structures, and are useful for getting data quickly into R:pems.utils. importTAB2PEMS imports tab delimited files and clipboard content. importCSV2PEMS imports comma delimited files. Both assume a simple file structure (i.e. data series in columns with names as headers), but require some time data management by the user. Note: These are wrappers for import2PEMS.

Other import... import specific file types.

importOBS2PEMS imports standard Horiba OBS files and converts them to pems objects. See Notes below.

importOB12PEMS imports .OB1 files and converts them to pems objects. OB1 files are generated by a Visual Basic PEMS data handler used during the RETEMM project. Notes below.

importParSYNC2PEMS imports standard parSYNC files and converts them to pems objects. See Notes below.

importCAGE2PEMS imports standard CAGE files and converts them to pems objects. See importParSYNC2PEMS Notes below.

importSEMTECH2PEMS imports Sensors Inc. SEMTECH ECOSTAR files and converts them to pems objects. See Notes below.

importRoyalTek2PEMS imports .txt and .nmea format Royal Tek GPS files and converts them to pems objects. See Notes below.

importKML2PEMS imports .kml format KML files and converts them to pems objects. See Notes below.

Value

is.pems return a logical, TRUE if the supplied object is pems class, otherwise FALSE. If the argument full.test = TRUE is also supplied, additional information about the object is returned as comment(output).

pems and pems.element functions return pems and pems.element objects, respectively, made using the supplied file and any additional information also supplied in the same call.

rebuildPEMS rebuilts the supplied pems object. The default rebuilds as the lastest build structure.

import... functions return a pems object, made using the supplied file and any additional information also supplied in the same call.

Note

isPEMS, makePEMS and makePEMSElement were earlier functions that performed the same functions as is.pems, pems and pems.elements, respectively. The the current versions of these functions are wrappers for their replacements.

With the crude import... functions (import2PEMS, importTAB2PEMS, importCSV2PEMS) modifications are minimal. Unless any additional changes are requested in the import...(...) call, the data is simply read in as a data.frame and converted to a pems object.

With importOBS2PEMS, OBS data is also modified as follows: data series names are simplified and converted to lower case to simplify use in R; the data series time.stamp and local.time are added (generated using the file time stamp, the row counter and the log.rate constant); data series latitude and longitude are resigned according to set N/S and E/W values, if these are present/valid; latitude and longitude units are also reset to 'd.degLat' and 'd.degLon'. Any data series names in analytes is renamed 'conc.[analyte name]'. If not supplied in the importOBS2PEMS, typical OBS constants are currently assumed. Several of these are based on emission source fuel. Defaults for these are generated according to fuel (default 'petrol').

With importOB12PEMS, handling is similar to that with importOBS2PEMS.

With importParSYNC2PEMS, the parSYNC data modifications are as follows: parSYNC Date and Time data-series are merged and converted into *POSIX as time.stamp; local.time is calculated from this; the parSYNC data series Timestamp is retained as parsync.timestamp; by default units are removed from data-series names, but retained as units(pems); (again by default) all names are converted to lower case. The default option reset.signal = TRUE reverses the voltage reading of opacity and ionization time-series (x = - x), if present. Alternatively, imports can be identified specifically by name, e.g. reset.signal = "opacity" to just reset opacity. Typical OBS constants are currently assumed, if not supplied. Several of these are based on emission source fuel. The default assumption is that the fuel is 'petrol' unless fuel has been assigned.

With importCAGE2PEMS, handling is similar to that with importParSYNC2PEMS.

With importSEMTECH2PEMS, SEMTECH data is imported and modified as follows: data series names are simplified and converted to lower case to simplify use in R; the data series time.stamp and local.time are added (generated using the file time stamp). Defaults constants are assigned according to fuel (default 'petrol'). This function in in-development and has so far only been used with files from two sources, so handle with care, and time.format has varied so may need resetting for some files.

With importRoyalTek2PEMS, the Royal Tek data modifications are currently being documented.

With importKML2PEMS, the function attempts to import and time.stamp, latitude, longtiude and altitude data in the suppled KML file. This function in in-development and has so far only been used with KML files from one source, so handle with care.

Author(s)

Karl Ropkins

References

References in preparation.

See Also

See ref.unit.conversions and convertUnits for general unit handling; merge.pems for pems data merging and alignment.

Examples

```
############
##example 1
############

#make little pems

data <- data.frame(speed=1:10, emissions=1:10)
units <- c("m/s", "g/s")
pems <- pems(x = data, units=units, example="my record")

pems                        #the pems object
summary(pems)               #summary of held data
pems$speed                  #the speed pems.element

#import data file as pems using import... functions

#For example, to import CSV file as pems object
## Not run:
pems <- importCSV2PEMS()

## End(Not run)
```

1.2.export.data *exporting PEMS data*

Description

Some functions for exporting data from R and pems.utils.

Usage

```
exportPEMS(pems, file = "tempfile", file.writer = write.table,
           sep = "\t", ...)

exportPEMS2TAB(pems, file = "tempfile", file.writer = write.table,
           sep = "\t", ...)

exportPEMS2TAB(pems, file = "tempfile", file.writer = write.table,
           sep = "\t", ...)

#exportPEMS2Excel
#currently disabled
```

Arguments

pems	(A required object) The object to export from R, typically a data.frame or pems object.
file	(Character) The name of the file to create when exporting data. This can be 'clipboard', to export to the clipboard assuming the clipboard buffers are not exceeded.
file.writer, sep	(Various arguments) file.writer is the R function used to create the export file. sep is the separator argument passed to file.writer.
...	(Optional) Other arguments, handling varies. For exportPEMS2... functions, these typically passed to exportPEMS or from there to the assigned file.writer.

Details

By default, exportPEMS2TAB and exportPEMS2CSV export the data component of a supplied pems object, to tab-delimited .txt and comma-delimited .csv files, respectively. file sets the file name (default tempfile).

These are typically used in form:

exportPEMS2...(pems, file, ...)

By default, they make the following associated modifications:

If file extensions are not included in file, they add .txt and .csv extensions to tab-delimited and comma-delimited files, respectively. The argument tidy.file=FALSE can be used to disable this modification.

Time stamps, if identified, are exported in "DD/MM/YYYY HH:MM:SS.x" format. Handling can be altered using time.stamp, time.format and tz arguments like import2PEMS or disabled using tidy.time.stamp=FALSE.

Data-series units can also be added to exported file column names in form name(units) by adding the argument units="add.to.names".

Value

exportPEMS2...() functions generate export file from pems data.

Warning

Currently, exportPEMS... functions overwrite without warnings.

Note

exportPEMS2Excel is curently disabled.

These are very crude functions in the most part because they are rarely used. Suggestions for helpful improvements would be very welcome.

Author(s)

Karl Ropkins

References

References in preparation.

See Also

See import2PEMS, etc. for importing data into pems.utils.

Examples

```
############
##example 1
############

#making a comma-delimited copy of pems.1

## Not run:
exportPEMS2CSV(pems.1, "pems.example")
dir()

## End(Not run)
```

2.1.pems.structure *'pems' object structure*

Description

This pages provides a brief outview description of the 'pems' object structure. It also lists some associated functions

Usage

```
getPEMSElement(x, pems = NULL, units = NULL, ...,
            fun.name="getPEMSElement",
            if.missing = "stop", if.null = if.missing,
            track.name = TRUE, .x = enquo(x))

getPEMSData(pems=NULL, ..., fun.name = "getPEMSData",
            if.missing = "stop", .pems = enquo(pems))

getPEMSConstants(pems=NULL, ...,
            fun.name = "getPEMSConstants",
            if.missing = "stop", .pems = enquo(pems))

pemsData(pems=NULL, ...,
          fun.name = "pemsData", if.missing = "stop",
          pems.name = deparse(substitute(pems)))
```

```
pemsConstants(pems=NULL, ...,
          fun.name = "pemsConstants", if.missing = "stop",
          pems.name = deparse(substitute(pems)))

pemsHistory(pems=NULL, ...,
         fun.name = "pemsHistory", if.missing = "stop",
         pems.name = deparse(substitute(pems)))

cpe(...)
```

Arguments

x	(Required vector, typically pems.element) For getPEMSElement, the required data element.
pems	(pems object) If supplied, the pems object to search for x before checking the parent environments and R workspace.
units	(Optional) The units that x should be supplied in (handled by convertUnits).
...	(Optional) Other Arguments.
fun.name, if.missing, if.null, track.name, pems.name, .x, .pems	(Various) Other options using for pems.utils house-keeping. See check... for definitions, although generally these can be ignored by users. See Note below.

Details

The pems object is a managed data.frame. It has five main components: data, units, constants, history and tags. data is the main data.frame. Each element (named data.frame column) is a data-series of the original PEMS data. units are the associated unit definitions. constants is a list of associated constants that are to be used with the pems object. (The preference order is arguments given in a call then constants declared in the pems object then constant defaults held by the pems.utils package.) history is a log of pems object modifications. tags are any other components that the user wishes to add to a pems object as identifiers.

getPEMSElement gets a requested pem.element from pems if supplied or from the local workspace.

pemsData and getPEMSData get the data component of a supplied pems object.

pemsConstants and getPEMSConstants get all constants locally defined for the supplied pems object.

pemsHistory gets the history of supplied pems object.

cpe combines pems.elements. It is intended as an alternative to c(pems.element, ...) while that generic is in-development.

Value

getPEMSElement returns the requested element of a supplied pems object as a managed vector or pems.element, if available. If missing, error handling is by checkIfMissing. See check... for more details.)

pemsData returns the data component of a supplied pems object as a data.frame.

getPEMSData returns the data component of a supplied pems object as a data.frame.

pemsConstants returns the constants component of a supplied pems object as a list.

getPEMSConstants returns the constants component of a supplied pems object as a list.

pemsHistory returns the history component of a supplied pems object as a list.

cpe turns the concatenated form of supplied input.

Note

pems... functions are in development pems object handlers. They are intended for convenient 'front of house' use. As part of this role, their structure will evolve over time, so arguments and operations may change based on user feedback. Those wishing to develop future-proof third party functions should also consider check... functions when developing their code. See common.calculations for some Examples.

getPEMS... functions are a revision of earlier pems... pems object handlers. They are intended to replace pems... code in future package versions.

rlang and dplyr functions now do the heavy lifting for getPEMSElement.

Author(s)

Karl Ropkins

References

rlang and dplyr package functions now do the heavy lifting for getPEMSElement.

Lionel Henry and Hadley Wickham (2018). rlang: Functions for Base Types and Core R and 'Tidyverse' Features. R package version 0.2.0. https://CRAN.R-project.org/package=rlang

Hadley Wickham, Romain Francois, Lionel Henry and Kirill Muller (2017). dplyr: A Grammar of Data Manipulation. R package version 0.7.4. https://CRAN.R-project.org/package=dplyr

See Also

See Also: check... for check... function equivalents; pems.generics for pems object generic functions.

Examples

```
############
##example 1
############

#basic usage

#using example data pems.1
#(supplied as part of pems.utils package)

#pems structure
pems.1
```

```
# extracting the pems.1 element velocity
getPEMSElement(velocity, pems.1)

## Not run:
#generic (SE) equivalents
pems.1$velocity
pems.1["velocity"]

## End(Not run)
```

3.1.generic.pems.handlers

Generic handling of pems objects

Description

pems objects can be manipulated using generic functions like print, plot and summary in a similar fashion to objects of other R classes.

Usage

```
## S3 method for class 'pems'
as.data.frame(x,...)

## S3 method for class 'pems'
 dim(x, ...)

## S3 method for class 'pems'
 x$name, ...

## S3 replacement method for class 'pems'
 x$name, ... <- value

## S3 method for class 'pems'
 x[i, j, ..., force = FALSE, simplify = TRUE]

## S3 replacement method for class 'pems'
 x[i, j, ..., force = FALSE] <- value

## S3 method for class 'pems'
 x[[k, ...]]

## S3 replacement method for class 'pems'
 x[[k, ...]] <- value

## S3 method for class 'pems'
 with(data, expr, ...)
```

```
## S3 method for class 'pems'
 subset(x, ...)

## S3 method for class 'pems'
names(x, ...)

## S3 replacement method for class 'pems'
names(x, ...) <- value

## S3 method for class 'pems'
print(x,..., rows=NULL, cols=NULL, width=NULL)

## S3 method for class 'pems'
plot(x, id = NULL, ignore = "time.stamp", n = 3, ...)

## S3 method for class 'pems'
head(x, n = 6, ...)

## S3 method for class 'pems'
tail(x, n = 6, ...)

## S3 method for class 'pems'
summary(object, ...)

## S3 method for class 'pems'
na.omit(object, ...)

## S3 method for class 'pems'
units(x)

## S3 replacement method for class 'pems'
units(x) <- value
```

Arguments

x, object, data	(An Object of pems class). For direct use with print, plot, summary, etc. NOTE: Object naming (i.e., x or object) is determined in parent or base function in R, so naming can vary by method.
name	Element name, which operates in a similar fashion to data.frame names, e.g. pems$name extracts the element name from the pems object pems.
i, j	Row and column (elements) indices, which operate as a stricter version of data.frame indices. See Note below.
k	Structural indices. See Note below.
expr	(Expression) For with(pems), an expression to be evaluated inside the supplied pems object.

value	(vector, data.frame or pems) An object to be inserted into a pems object in e.g. the form pems[i,j] <- value or pems$name <- value.
...	Addition options, typically passed to associated default method(s).
force	(Logical or character) Data element handling options. force provides various options to forces data to fit it destination. This can either be set as a logical (TRUE/FALSE force/don't force) or one or more character strings to specify particular types of forcing to try when e.g. fitting value into pems[i,j]: 'omit.err.cases', equivalent to TRUE, remove any unknown/unfound i or j terms; 'na.pad.insert' expand the value to fit larger pems[i,j], placing NAs in any holes generated; 'na.pad.target' like previous but expanding pems[i,j] to fit larger value; 'fill.insert' like 'na.pad.inert' but holes are filled by wrapping the supplied value within elements and then by element.
simplify	(Logical) simplify returns a pems.element rather than a pems object if possible.
id, ignore	(local plot parameters). id identifies which data series to plot; ignore identifies which data series to ignore when leaving the choice of id to the function; and, n gives the maximum number of data series to plot when leaving the choice of id to the function.
rows, cols, width	(numerics, optional). For print, if supplied, these reset the number of rows and columns to table when printing a pems object, and character width to print across.
n	(various). For plot, sets the maximum number of data series to plot when leaving the choice of id to the function. For head or tail, sets the number of rows.

Value

Generic functions provide appropriate (conventional) handling of objects of 'pems' class:

as.data.frame(pems) extracts the data.frame component of a pems object.

dim(pems) extracts the dimensions, row count and column count, respectively, of the data.frame component of a pems object. The function also allows nrow(pems) and ncol(pems).

pems$name extracts the named element from a pems objects in a similar fashion to data.frame$name. Likewise, pems$name <- value inserts value into a pems objects in a similar fashion to data.frame$name <- value.

pems.object[i, j] extracts the [i,j] elements of the data held in a pems object. This is returned as either a pems or pems.element object depending on the dimension of the extracted data and the simplify setting.

pems.object[i, j]<- insert value into the [i,j] region of the supplied pems object. By default this action is strict and mismatching pems[i, j] and value dimension produce an error, although mismatching insertions may be forced using the force argument.

pems.object[[k]] extracts structural elements of a pems object: data, the data.frame; units the unit table, etc.

with(pems.object, expression) evaluates the supplied expression using the elements of the supplied pems.object.

subset(pems.object, expression) behaves like subset(data.frame, expression).

`print(pems.object)` provides a (to console) description of a `pems` object. This forshortens large datasets in a similar fashion to a tibble.

`plot(pems.object)` generates a standard R plot using selected data series in a `pems` object.

`names(pems.object)` returns a vector of the names of data series held in a `pems` object when used in the form `names(pems)` or resets names when used in the form `names(pems) <- new.names`.

`na.omit(pems.object)` returns the supplied `pems` object with all rows including NAs removed.

`summary(pems.object)` generates a summary report for data series held in a `pems` object.

`units(pems.object)` extracts the units from a supplied `pems` object when used in the form `units(pems)` or sets/resets units when used in the form `units(pems) <- new.units`.

Note

The `pems` object is intended to be a stricter version of a standard R `data.frame`. Unless the user specifically forces the operation, a `pems[]` or `pems[]<-` call is not allowed unless it fits exactly. So, for example by default the call `pems[,1]<-10` will not place 10 in every row of column one in the same fashion as `data.frame[,1]<-10`.

The logic behind this is that columns (elements) of `pems` objects are time-series. So, users would want to place these exactly and avoid any unintended wrapping. The `force` argument should be used in cases where data padding or wrapping operations are required.

`pems$name` and `pems$name<-` are not are rigorously managed, so behave more like `data.frame$name` and `data.frame$name<-` calls, although even these do wrap by default.

`pems[[]]` provides access to structural components of the `pems` object, e.g. `pems[["data"]]` extracts the `data.frame` component of the `pems` object, `pems[["units"]]` extracts the `units` component, etc. See also `pems.structure`.

Author(s)

Karl Ropkins

References

generics in general:

H. Wickham. Advanced R. CRC Press, 2014.

(Not yet fully implemented within this package.)

Examples

```
##example 1
##basics pems handling

#extract a subset pems object from pems.1
a <- pems.1[1:4, 1:5]
a

#indices work like data.frame
#a[x] and a[,x] recovers element/column number x
#a[x,] recovers row number x
```

```
#a["name"] and a[,"name"] recovers element/column named "name"
#a[4:5, "name"] recovers rows 4 to 5 of element/column named "name"
#a[x,y] <- z inserts z into a at row x, element y
#etc

#insert 10 in at element 3, row 2
a[2,3] <- 10
a

#replace element conc.co2 with conc.co
a["conc.co2"] <- a$conc.co
a

#Note: by default pems objects subsetting and inserting is
#more rigorous than data.frame subsetting/insertion
#for example, a[1:2, "conc.hc"] <- 8 would generate error
#because the target, a[1:2], and insert, 8, dimensions do not
#match exactly: target 2 x 1; insert 1 x 1

#By default no wrapping is applied.

#the force argument allows the user to control how mismatching
#targets and insertions are handled

#na pad target for larger insert
a[1:2, "conc.hc", force="na.pad.target"] <- 1:5
a

#Note here when the target is padded existing enteries are NOT
#overwritten if they are not declared in a[], and the next
#previously unassigned cells are used for any extra cases in
#the insert.

#wrap insert to fill hole made by a[i,j]
a[1:2, "conc.hc", force="fill.insert"] <- 8
a

#pems$name <- value is equivalent to
#pems[name, force=c("na.pad.target", "na.pad.insert")]
a$new <- 1:4
a
```

`3.2.generic.pems.element.handlers`

Generic handling of pems.element objects

Description

pems elements objects can be manipulated using generic functions like print, plot and summary in a similar fashion to objects of other R classes.

Usage

```
## S3 method for class 'pems.element'
 x[i, ..., force = TRUE, wrap = FALSE]

## S3 replacement method for class 'pems.element'
 x[i, ..., force = TRUE, wrap = FALSE] <- value

## S3 method for class 'pems.element'
as.pems(x, ...)

## S3 method for class 'pems.element'
print(x, ..., n = NULL, rows = NULL, width = NULL)

## S3 method for class 'pems.element'
plot(x, y = NULL, xlab = NULL, ylab = NULL, ...)

## S3 method for class 'pems.element'
units(x)

## S3 replacement method for class 'pems.element'
units(x) <- value

## S3 method for class 'pems.element'
summary(object, ...)

## S3 method for class 'pems.element'
round(x, ...)
```

Arguments

`x`, `object`	(An Object of pems.element class). For direct use with `print`, `plot`, `summary`, etc. NOTE: Object naming (i.e., `x` or `object`) is determined in parent or base function in R, so naming varies by method.
`i`	Element indices, which operate in a similar fashion to `vector` indices.
`...`	Addition options, typically passed to associated default method(s).
`force`, `wrap`	(Logicals) Data element handling options: `force` forces data to fit it destination; `wrap` expands data to fit its destination by wrapping the source `pems.element`.
`value`	(Vector) For calls in `pems.element[1] <- value` or `units(pems.element) <- value`, the value to be inserted.
`n`, `rows`, `width`	(Numerics) For `print(pems)`, number of elements, rows or screen width to foreshorten print output to.
`y`, `xlab`, `ylab`	(other plot arguments). As with the default `plot` method, `y` is an optional second data verctor, typically numeric, and `xlab` and `ylab` are labels to use on x and y axes.

Value

Generic functions provide appropriate (conventional) handling of objects of 'pems.elements' class:

print(pems.element) provides a (to console) description of the supplied pems.element object.

plot(pems.element) generates a standard R plot of the supplied pems.element.

units(pems.element) extracts the units from the supplied pems.element.

Note

A dedicated round(pems.element) is required as a wrapper to round.Date and round.POSIXt handling.

Author(s)

Karl Ropkins

Examples

```
#the velocity pems.element in pems.1
pems.1$velocity
```

4.1.merge.data.pems *Merging data and pems objects*

Description

Various pems.utils functions to merge data of different types.

Usage

```
#basic alignment
align(data1, data2, n = 0, ...)

#alignment based on correlation
cAlign(form, data1 = NULL, data2 = NULL, ...)

#alignment based on time.stamp
tAlign(form, data1, data2 = NULL, order=TRUE, ...)

#basic stacking
stackPEMS(..., key=key, ordered=TRUE)

#historical
findLinearOffset(x = NULL, y = NULL, ...)
```

Arguments

data1, data2	(pems or data.frame; optional for cAlign, required for other alignment functions) pems objects or data.frames to be aligned.
n	(numeric; required) An offset to be applied to data2 when aligning data1 and data2. The default, n = 0, applies no offset and directly aligns the two supplied data sets, first row to first row.
...	(Any other arguments) For stackPEMS this is typically a series of pems objects to be stacked. For other functions, this may be passed on but are typically ignored. See Notes.
form	(formula; required) A formula identifying the elements in the supplied data sets to be used as references for the alignment. This typically takes the form, e.g. cAlign(x~y, d1, d2) where d1$x and d2$y are the data series to be used to correlation align the two data sets.
order	(logical; optional) If TRUE the function orders the data.
key	(character or NSE) For stackPEMS the name to key column that identifies the data sources of elements in a stack of pems objects.
ordered	(logical; default TRUE) For stackPEMS, when creating the source key should the order pems objects were supplied be retained.
x, y	(Required objects, various classes) For bindPEMS, two pems, data.frame, pems.elements or vectors to be bound together. For findLinearOffset, two pems.elements or vectors to be aligned.

Details

The align function accepts two pems objects, data.frame, etc, and returns a single dataset (as a pems object) of the aligned data. An extra argument, n, may be supplied to offset the starting row of the second data set relative to the first. It is intended to be used in the form:

```
aligned.data <- align(data1, data2) #aligned row 1-to-1
```

```
aligned.data <- align(data1, data2, 3) #row 3-to-1, etc
```

The cAlign function accepts a formula and up to two data sets and returns a single data set (as a pems object) of correlation aligned data. This uses the best fit linear offset correlation for the elements identifed in the formula term.

It is intended to be used in the form:

```
aligned.data <- cAlign(name1~name2, data1, data2)
```

The tAlign function accepts a formula and two data sets and returns a single data set (as a pems object) of the time stamp aligned data. This is this done by matching entries in the elements identifed in the formula term.

It is intended to be used in the form:

```
aligned.data <- tAlign(name1~name2, data1, data2)
```

The stackPEMS function stacks two or more pems objects and returns a single pems object. stackPEMS stacks using dplyr function bind_rows so handles pems with column names that do not completely intersect. However it also attempts to units match. It is intended to be used in the form:

```
stacked.data <- stackPEMS(data1, data2)
```

Historical functions:

`findLinearOffset` is historical code.

Value

`align`, `cAlign`, `tAlign`, etc all return a single object of `pem` class containing the aligned data from `data1` and `data2`.

`findLinearOffset` returns the best fit offset for `y` relative to `x`.

Note

These functions are under revision and need to be handled with care.

`cAlign`: By default `cAlign` generates an alignment plot and returns a `pems` object of aligned data. But it also allows several hidden arguments to refine outputs, the logicals `plot`, `offset` and `pems`, which turn off/on plot, offset and pems reporting individually, and `output = c("plot", "offset", "pems")` or combinations thereof also provides a single argument alternative.

`bindPEMS`: The historical function `bindPEMS` has been superceded by `align`.

`findLinearOffset`: `findLinearOffset` is currently retained but will most likely be removed from future versions of `pems.utils`.

The call `cAlign(x~y, output = "offset")` is equivalent to `findLinearOffset(x, y)`.

Author(s)

Karl Ropkins

References

`align` uses the `dplyr` function `full_join`.

`cAlign` function uses the `stats` function `ccf`.

`tAlign` uses the `dplyr` function `full_join`.

See Also

See also: `cbind` for standard column binding in `R`; `dplyr` for `full_join`.

Examples

```
############
##example 1
############

##data vector alignment

#make two offset ranges
temp <- rnorm(500)
x <- temp[25:300]
y <- temp[10:200]
```

```
#plot pre-alignment data
plot(x, type="l"); lines(y, col="blue", lty=2)

#estimated offset
findLinearOffset(x,y)
#[1] -15

#applying linear offset
ans <- align(x, y, findLinearOffset(x,y))
names(ans) <- c("x", "y")

#plot post-alignment data
plot(ans$x, type="l"); lines(ans$y, col="blue", lty=2)

#shortcut using cAlign
## Not run:
plot(x, type="l"); lines(y, col="blue", lty=2)
ans <- cAlign(x~y)
plot(ans$x, type="l"); lines(ans$y, col="blue", lty=2)

## End(Not run)


############
##example 2
############

##aligning data sets
##(pems object example)

#make some offset data
p1 <- pems.1[101:200, 1:5]
p2 <- pems.1[103:350, 1:3]

#correlation alignment using ccf
ans <- cAlign(~conc.co, p1, p2)

#this aligns by comparing p1$conc.co and p2$conc.co
#and aligning at point of best linear regression fit

## Not run:

#compare:

cAlign(~conc.co, p2, p1)
cAlign(conc.co2~conc.co, p1, p2)
#(aligns using p1$conc.co2 and p2$conc.co)
cAlign(conc.co2~conc.co, p1)
#(realigns just conc.co within p1 based on best fit
# with conc.co2 and returns as output ans)

#time stamp alignment
tAlign(~time.stamp, p1, p2)
```

```
#this aligns by pairing elements in p1$time.stamp
#and p2$time.stamp
#(if time stamps have different names
# tAlign(time1~time2, p1, p2), the time stamp name
# from p1 would be retained when merging p1$time1
# and p2$time2, generating [output]$time1)


## End(Not run)

############
##example 3
############

##stacking pems

#make some offset data
p1 <- pems.1[1:2, 1:4]
p2 <- pems.1[3, 2:4]
p3 <- pems.1[4:6, 1:3]

#stack
stackPEMS(p1, p2, p3, key=source)
```

`4.2.referencing.pems.data`

Data Referencing Functions.

Description

Various functions for grouping, subsetting, filtering and conditioning datasets.

Usage

```
refRow(ref = NULL, n = 4, breaks = NULL,
        data = NULL, ..., labels = NULL,
        fun.name = "refRow")

refX(ref = NULL, n = 4, breaks = NULL,
        method = "percentile",
        data = NULL, ..., labels = NULL,
        fun.name = "refX")

refEngineOn(rpm = NULL, data = NULL,
        threshold = 200, ..., labels = NULL,
        fun.name = "refEngineOn")
```

```
refDrivingMode(speed = NULL, accel = NULL,
        time = NULL, data = NULL,
        threshold.speed = 0.1,
        threshold.accel = 0.1,
        ..., labels = NULL,
        fun.name = "refDrivingMode")
```

Arguments

ref	(Data series, typically vector) The reference data-series to consider when making a vector of subset markers/indices. See Details.
n, breaks	(numerics) With refRow and refX, n sets the number of equal intervals to attempt to cut the data into. With refRow, breaks sets the rows at which to cut the data at. With refX, breaks sets the values of ref to cut the data at. In both cases, if both n and breaks are set, breaks is applied.
data	(Optional data.frame or pems object) The data source if ref is supplied in either a data.frame or pems object.
...	(Optional) Other arguments, currently passed on to pems.utils management functions.
labels	(Vector, typically Character) a vector of labels to be assigned to the reference regions.
fun.name	(function managment argument) fun.name is the name of the parent function, to be used in error messaging.
method	(Various) For refX, the method to use when cutting data. If character vector, 'percentile' or 'range'. If function, it should be in form function(ref, n), and return breaks.
rpm	For refEngineOn, the input, assumed to be engine speed and expected to have units of rpm.
threshold	For refEngineOn, the signal threshold above which the vehicle engine is assumed to be on.
speed, accel, time	For refDrivingMode, possible inputs. Strictly, refDrivingMode needs speed and accel but can use speed and time to build accel.
threshold.speed, threshold.accel	For refDrivingMode, the speed and acceleration signal thresholds. Below these thresholds the signals are assumed to be noise and the vehicle is not moving or accelerating, respectively.

Details

ref... functions generate a vector of subset markers or indices based of the referencing method applied and the length of ref. See Value regarding outputs.

refRow assigns reference regions based on row number. Because row depends on the length of the ref element independent of values, this is a unique case where ref can be either a vector or

a data set (pems, data.frame). It accepts n to set the number of cases to make or breaks to set break-points at specific rows.

refX assigns reference regions based on the value of a supplied data-series.It accepts n to set the number of cases to make or breaks to set the ref values to make break-points. If using n, method used to assign cut method, e.g. 'percentile' or 'range'.

refEngineOn assigns reference regions based on engine operation status. It uses the input, which it assumes is engine speed, and assumes reported engine speeds larger than the supplied threshold, by default 200 rpm, indicate that the engine is on.

refDrivingMode assigns reference regions based on vehicle driving mode. It uses inputs, assumed to be speed, accel and/or accel, and associated threshold to characterize activity as decel, idle, cruise or accel.

Value

By default results are returned as pems.elements.

The reference vector generated by ref... functions can then be used to group, subset, filter or condition data in pems objects.

refRow assigns reference according to row number, and, by default, reference labels show start row and end row of the referenced case.

refX assigns reference according to value of supplied input, and, by default, reference labels show lower value and higher value of the referenced case.

refEngineOn assigns reference according to engine operation status based on engine speed, and, by default, reference labels are 'on' or 'off'.

refDrivingMode assigns reference according to vehicle driving mode, based on vehicle speed, acceleration and associated thresholds, and, by default, reference labels are decel, idle, cruise and accel.

Note

With refRow, If n is applied and the length of ref is not exactly divisible by n a best attempt is made.

With refX, if breaks are at values that are duplicated, all same values are assigned to the same (lower) value case, so e.g. 'percentile' may vary significantly if break-point values are highly duplicated in ref

Author(s)

Karl Ropkins

References

References in preparation.

See Also

cut, etc. in the main R package.

Examples

```
###########
##example 1
###########

#basic usage

#working with a temporary pems

temp <- pems.1

#cut into equal subsets

temp$ref <- refRow(velocity, n= 5, data=temp)
pemsPlot(local.time, velocity, cond=ref, data=temp,
             type="l", layout=c(1,5))

#cut at three points

temp <- pems.1
temp$ref <- refRow(velocity, breaks=c(180,410,700),
             data=temp)

pemsPlot(local.time, velocity, cond=ref, data=temp,
             type="l", layout=c(1,5))
```

4.3.time.handlers *pems Data Time Handlers*

Description

Time handlers are subset of `pems.utils` functions that work on or with time records (`time.stamp` and `local.time`).

Usage

```
regularize(data, Hz=1, method=1, ...)

repairLocalTime(data, local.time, ref, ..., reset.count = TRUE,
      fun.name = "repairLocalTime")
```

Arguments

`data`	(Required, typically `pems`) The dataset to be worked with. For `regularize`, the dataset to regularize (see below).

Hz	(For regularize) (Required numeric) The time resolution to regularize the data to in Hertz. So, the default, Hz=1 is one measurement (or row of data) per second.
method	(For regularize) (Required numeric) The regularization method to apply. The default, method=1 uses approx to linearly interpolate regular time-series for all supplied time-series. The alternative method=2 uses a bin-and-average strategy.
...	(Optional) Other arguments, typically passed on.
local.time	(For repairLocalTime) (Required pems.element) The local.time pems.element to work been repaired.
ref, reset.count	(For repairLocalTime) (Other arguments) ref is a second source that local.time can be inferred from in cases where local.time records are missing. If TRUE, reset.count resets local.time so it starts at 0.
fun.name	(character) (pems handler) argument used by pems.utils. These can typically be ignored.

Details

regularize attempts to extrapolate a regular series, generated at the time resolution requested, from the supplied data. Both methods can be used for the regularization of irregularly time-logged data, but differ in their data handling. Method 1 estimates measurements at regular intervals by linearly interpolating between the last valid point and the next valid point in supplied time-series. It therefore hole-fills gaps in time-series and is perhaps best suited for use with sparser data-sets. It can also be used to interolate time-series to higher time-resolutions, but should not be used aggressively, e.g. to convert 1Hz data data to 10Hz. By contrast, method 2 uses data binning to aggregate all supplied measurements (e.g. all measurements between -0.5 and +0.5 seconds of requested times when returning 1 Hz data) and (mean) average these. It is better suited for use with higher resolution time-series (e.g. gong from about 10Hz to 1Hz) and does not hole-fill empty time intervals. If you want mean binning and hole-filling, apply method 2 then method 1, e.g.:

```
new.data <- regularize(regularize(my.data, method=2), method=1)
```

repairLocalTime attempts to repair an incomplete local.time record. For example, if you merge two datasets with overlapping but different time ranges, one may not track the time range of the other and this can generate incomplete time records. This function attempts to hole-fill such cases.

Value

regularize returns the supplied dataset (data) with time-series (time.stamp and local.time) are regularized at the requestion time resolution, based on Hz value. It uses approx or data binning to estimate associated changes for other data-series.

repairLocalTime returns a repaired local.time pem.element, typically the supplied local.time with any holes (NAs) it can fill filled.

Note

All time handlers should be used with care.

Author(s)

Karl Ropkins

References

`regularize(..., method=1)` uses `approx`:

Base R Stats package function based on Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) The New S Language. Wadsworth & Brooks/Cole.

A lot of leg-work testing `regularize` was done by then Leeds MSc Student Adrian Felipe Ortega Calle.

See Also

`approx` regarding data regularization methods.

4.4.unit.handlers *data unit handlers*

Description

Various pems.utils functions for the management of data units.

Usage

```
getUnits(input = NULL, data = NULL, ...,
         if.missing = c("stop", "warning", "return"))

setUnits(input = NULL, units = NULL, data = NULL, ...,
         if.missing = c("stop", "warning", "return"),
         output = c("input", "data.frame", "pems", "special"),
         force = FALSE, overwrite = FALSE)

convertUnits(input = NULL, to = NULL, from = NULL, data = NULL, ...,
         if.missing = c("stop", "warning", "return"),
         output = c("input", "data.frame", "pems", "special"),
         unit.conversions = NULL, force = FALSE, overwrite = FALSE)

#local unit.conversion method handling

addUnitConversion(to = NULL, from = NULL, conversion = NULL,
         tag = "undocumented",
         unit.conversions = ref.unit.conversions, ...,
         overwrite = FALSE)

addUnitAlias(ref = NULL, alias = NULL,
         unit.conversions = ref.unit.conversions, ...)
```

```
listUnitConversions(unit.conversions = ref.unit.conversions, ...,
         verbose = FALSE, to = NULL, from = NULL)
```

Arguments

input	(vector, object or object element) An input, e.g. a vector of speed measurements.
data	(data.frame, pems object) If supplied, the assumed source for an input. This can currently be a standard data.frame or a 'pems' object. Note: if an input is not found in data, the parent environment is then also checked before returning an error message.
units, to, from, ref, alias, tag	(Character vectors). Unit ids. units sets the units of input in setUnits. to sets the units to convert input to when using convertUnits. The additional arguments from can be used to apply unit conversions to inputs with un-defined or mismatched units, but requires the extra argument force = TRUE to confirm action. When working with local unit conversions to and from should be used to identify specific conversions, e.g. when using addUnitConversion to add a new unit conversion method, and ref and alias should be used to identify a current unit id and new alias, respectively, when using addUnitAlias. tag is an optional more detailed conversion description, intended for use in method documentation. (See Below for further details.)
...	(Optional) Other arguments, currently ignored.
if.missing	(Optional character vector) What the function should do if things do not go as expected. Current options include: "stop" to stop the function with an error message; "warning" to warn users that expected information was missing but to continue running the parent code; or "return" to continue running the parent code without any warnings.
output	(Character vector) Output mode for function results. Options currently include: special, input, data.frame, and pems. See force, overwrite and Values below for further details.
force	(Logical) Should a unit change to attempted even if checking indicates a mismatch, e.g. an attempt to set the units of an input that already has units assigned.
overwrite	(Logical) If 'same name' cases are encountered when packing/repacking an output into a data.frame or pems object, should the function overwrite the case in the target data.frame or pems object with the modified input? (If FALSE, a new element is generated with a unique name in the form [input.name].number.)
unit.conversions	(Optional list) If supplied, unit.conversions is a 'look up' table for unit conversion methods. By default, functions in pems.utils use the reference ref.unit.conversions, but this can be copied to the workspace and updated to provide the user with a means of updating and expanding the method set.
conversion	(Numeric or function) When adding or updating a conversion method using addUnitConversion, the conversion method. This can be a numeric, in which case it is assumed to be a multiplication factor (and converted to a function in

	the form `function(x) x * conversion`) or a function to be applied directly to an `input`.
`verbose`	(Logical) For `listUnitConversions`. Should `unit.conversions` be reported in detail? By default (`verbose = FALSE`) only unit conversion `tags` are reported.

Details

`getUnits` returns the units of an `input`.

`setUnits` sets/resets the units of an `input`.

`convertUnits` converts the units of an `input`.

`addUnitConversion` adds a conversion method to a local version of the unit conversion look-up table. Methods should be supplied as `to` and `from` unit ids and an associated `conversion`. A `tag` can also be supplied to provide a more detailed description of the conversion for use in documentation.

`addUnitAlias` adds an alias for an existing unit id in a local version of the unit conversion look-up table. The existing unit id should be identified using `ref` and the new alias should be assinged using `alias`. The `alias` is added to all `to` and `from` elements containing `ref` to allow users to work with alternative unit abbreviations.

`listUnitConversions` lists the methods a supplied unit conversion look-up table. If `to` and/or `from` arguments are also supplied, these are used to subsample relevant methods.

Value

`getUnits` returns the units of an `input` as a character vector if available, else it returns `NULL`.

`setUnits` sets the units of an `input` to a supplied value, `units`, if they have not already be set or if `force = TRUE`. The result is returned as the modified `input` alone, the modified `input` as an element in a `data.frame`, or the modifed `input` as an element in a `pems` object (depending on `output` setting). If either a `data.frame` or `pems` object is supplied as `data`, this is used as the target when repacking the `output`. (Note: `output = "special"` is a special case which allows the function to select the `output` mode based on the type of `data` supplied.

`convertUnits` converts the units of an `input`. Typically, this is done by setting the required new units, using `to`, and letting the function select a suitable conversion method. However, conversions can be forced by setting `from` and `force = TRUE` to apply a specifc `to`/`from` method to an `input` regardless of the actual `units` of `input`. As with `setUnits`, results can be `output` as `input`, `data.frame` or `pems` objects.

`addUnitConversion` returns a supplied unit conversion look-up table (or in its absence the reference `ref.unit.conversions`) subject to the requested addition or update. Note: modifications that change exist information require the extra argument `overwrite = TRUE` as confirmation.

`addUnitAlias` returns a supplied unit conversion look-up table (or in its absence the reference `ref.unit.conversions`) subject to the requested alias addition.

`listUnitConversions` returns summary descriptions of methods in the supplied unit conversion look-up table (or in its absence the reference `ref.unit.conversions`). Additional arguments, `to` and `from`, can be used to select unit conversions of particular relevance.

Note

This set of functions is intended to provide a flexible framework for the routine handling of data units.

Author(s)

Karl Ropkins

References

References in preparation

See Also

pems.element

Examples

```
############
##example 1
############

#work with data units

#getting units (where assigned)
getUnits(velocity, pems.1) #km/h

#setting units
a <- 1:10
a <- setUnits(a, "km/h") #add unit

#alternaltive
#using pems.element
#a <- pems.element(a, units="km/h", name = "a")

#changing units
convertUnits(a, "mi/h")

# [1] 0.6213712 1.2427424 1.8641136 2.4854848 3.1068560 3.7282272 4.3495983
# [8] 4.9709695 5.5923407 6.2137119
# pems.element; [unnamed] [mi/h] [n = 10]

############
##example 2
############

#working with local unit conversions
#adding/updating unit conversion methods

#make a local reference
ref.list <- ref.unit.conversions

#add a miles/hour alias to mi/h
ref.list <- addUnitAlias("mi/h", "miles/hour", ref.list)

#add a new conversion
ref.list <- addUnitConversion(to = "silly", from = "km/h",
```

```
                              conversion = function(x) 12 + (21 * x),
                              tag = "kilometers/hour to some silly scale",
                              unit.conversions = ref.list)

#use these
convertUnits(a, "miles/hour", unit.conversions = ref.list)

# [1] 0.6213712 1.2427424 1.8641136 2.4854848 3.1068560 3.7282272 4.3495983
# [8] 4.9709695 5.5923407 6.2137119
# units: "miles/hour" (as above but using your unit abbreviations)

convertUnits(a, "silly", unit.conversions = ref.list)

# [1]  33  54  75  96 117 138 159 180 201 222
# units: "silly" (well, you get what you ask for)
```

5.1.pems.plots *Various plots for pems.utils*

Description

Various plot functions and visualization tools for use with pems objects.

Usage

```
#pemsPlot

pemsPlot(x, y = NULL, z = NULL, groups = NULL,
         cond = NULL, ..., data = NULL,
         units = TRUE, multi.y = "special",
         fun.name="pemsPlot")

#associated functions

pemsXYZCondUnitsHandler(x, y = NULL, z = NULL,
         cond = NULL, groups = NULL, data = NULL,
         units = TRUE, ...,
         fun.name = "pemsXYZCondHandler")

preprocess.pemsPlot(lattice.like = lattice.like,
         units = units, ...)

panel.pemsPlot(..., loa.settings = FALSE)

panel.routePath(..., loa.settings = FALSE)

#WatsonPlot
```

```
WatsonPlot(speed, accel = NULL, z = NULL, ...,
         data = NULL, cond = NULL, units = TRUE,
         plot.type = 2, fun.name="WatsonPlot")

#associated functions

preprocess.WatsonPlot(lattice.like = lattice.like, ...)

panel.WatsonBinPlot(..., ref.line = TRUE,
         process.panel = NULL, plot.panel = NULL,
         omit.stopped = FALSE, process = TRUE,
         plot = TRUE, loa.settings = FALSE)

panel.WatsonContourPlot(...,
         plot.panel=NULL, process = TRUE,
         plot = TRUE, loa.settings = FALSE)

panel.WatsonSmoothContourPlot(...,
         plot.panel=NULL, process = TRUE,
         plot = TRUE, loa.settings = FALSE)

#old plots

latticePlot(x = NULL, data = NULL, plot = lattice::xyplot,
         panel = NULL,  ..., greyscale = FALSE,
         fun.name = "latticePlot")

panel.PEMSXYPlot(..., grid=NULL)

XYZPlot(x = NULL, ..., data = NULL, statistic = NULL,
         x.res = 10, y.res = 20, plot = lattice::levelplot,
        fun.name = "XYZPlot")
```

Arguments

`x`, `y`, `z`, `groups`, `cond`

(Various) The main plot elements. `x` and `y` are typically plot coordinates. `z` is any additional information that could be used , e.g. to modify points plotted at `(x,y)` coordinates or generate a third axis for a surface plot. `groups` and `cond` are plot grouping and conditioning terms that can be used to subset the supplied data and/or generate multiple plot panels. See Below.

For new plots, these should be supplied individually, e.g. for `pemsPlot`:

`pemsPlot(x, y, z, groups, cond)`

For old plots, these must be formulae.

For `latticePlot` the basic formula structure is `y ~ x | cond`.

For `XYZPlot` the basic formula structure is `z ~ y * x | cond`. `z` is optional, but

when it is not supplied z it is treated as the bin count.

See Notes and Examples.

`data` (Optional `data.frame` or `pems` object) For most plots, the data source for plot elements, e.g x, y, z, etc, if these are not supplied directly or accessible from the current workspace.

`units` (Optional `logical` or `list`) for `pemsPlot` only, unit handling information. By default, `pems.utils` adds any known units to plot labels and allows in-plot unit management. Unit management is handled by `convertUnits`, and requested conversions need to assigned to an axis. So, for example, the call `pemsPlot(..., x.to="m/s")` would generate a plot with the x-axis in units of m/s (assuming `pems.utils` knows the unit conversion and the x-axis data series is in units that can be converted). All unit management and associated figure labelling can disabled using `units = FALSE`, or unit suffixes can be removed but unit management retained using `units.add.to.labels = FALSE`.

`multi.y` (character) `pems.plots` accepted multiple y data-series if passed using `cpe`, e.g. `y = cpe(a,b)`. `multi.y` sets how these are handled, options include `'groups'`, `'cond'` and `'special'` (the default), which selects groups if not used in plot call else cond.

`...` (Optional) Other arguments, typically passed on. This includes, `scheme` which sets the default coloring scheme for the plot. See Note below.

`fun.name` (Function management argument) `pems.utils` management settings, can typically be ignored by most users.

`panel`, `plot.panel`, `process.panel`

(Functions) These functions are used to generate the content of individual plot panels. Typically, all in-panel data processing and plotting is carried out using `panel`. However, these steps can be handled by separate functions if these are supplied as `plot.panel` and `process.panel`.

`lattice.like`, `plot`, `process`, `loa.settings`

(Various) Plot management elements. These can typically be ignored by most users, but allow plot developers to fine-tune plots. See Details below.

`speed`, `accel` (Various) For `WatsonPlot`, the x and y terms, speed and acceleration, respectively.

`ref.line` (Logical or list) For `WatsonPlot`. This argument manages the speed = 0 reference added to WatsonPlots. It can be either a logical (`TRUE`/`FALSE`), to turn the line on or off, or a list of parameters to set the properties of the line.

`omit.stopped` (Logical or character) For `WatsonPlot`, how to handle idling data: `FALSE` include in plot; `TRUE` or `'points'` removes idle points before plotting; `'cell'` or `'cells'` removes any cells that include idling points before the data is binned.

`plot.type` (numeric) For `WatsonPlot`, pre-set plot types: `1` scatter plot; `2` bin plot; `3` contour plot of binned data; `4` smoothed surface of binned data. See also `statistic`

`greyscale` (Logical) For older plots only, should the plot be greyscale by default? This option resets the `lattice` color themes to greyscale while the plot is beging generated. So: (1) This only effects the plot itself, not subsequent plots; and, (2) any user resets overwrite this, e.g. `latticePlot(..., greyscale=TRUE, col="red")` will place red symbols on an overwise greyscale plot. Newer plots use the alternative `plot(..., scheme = "greyscale")`.

grid	(List) If supplied, a list of plot parameters to be used to control the appearance of the grid component of the plot. See Below.
statistic	(Function) when binning data with XYZPlot and WatsonPlot, the function to use when evaluating the elements of each data bin. By default, length is used if z is not supplied to generate a frequency plot and mean is used if z is supplied to generate a bin average plot.
x.res, y.res	(Numerics) when binning data with XYZPlot, the number of x- and y-axis bins to generate.

Details

pems.utils includes conventional (generic) plot methods for pems and pems.element objects. See plot.pems and plot.pems.element for further details.

However, it also includes a range of higher-level plotting functions developed for use with pems data.

Early plots, e.g. latticePlot, only allowed plot arguments using the lattice formula format. While this is flexible and very powerful system, some users preferred the more conventional plot(x,y,..) call format. So, newer plots, e.g. pemsPlot, allow both conventional plot and lattice-style formula calls.

pemsXYZCondUnitsHandler handles the pems information associated with the plots. This routine is included as a discrete function within this package and others are welcome to use elsewhere for similar purposes. edit(pemsPlot) to see it.

Newer plots use a combination of lattice and loa functions to provide a range of additional plotting options, such as integrated panel and key management. See loa documentation for further details.

preprocess... and panel... functions handle pre-plot and in-plot elements of plot generation. These use the loa modification of the lattice plotting framework.

See Notes, Examples and extra documentation: pems.plots.

Value

By default, pemsPlot generates a bubble plot, so it plots (x,y) points, and by default color-grades and size-scales them according to z if also supplied.

When supplied speed and accel data-series as x and y cases, the WatsonPlot generates various forms of Watson's classic speed/accel frequency distribution plot.

latticePlot and XYZPlot are general purpose 'xy' and 'xyz' data plotting functions.

fortify is intended for use by ggplot2 functions when users are plotting data in pems objects. See Notes.

Warning

IMPORTANT: Conditioning is currently disabled on XYZPlot.

XYZPlot is a short-term replace for previous function quickPlot. It will most likely be replaced when pems.utils.0.3 is released.

pemsPlot and WatsonPlot no longer accept formula x, y, z inputs.

With all these functions I have tried to make the default plotting options as robust as possible. However, it is not always possible to test all the plot combines that users might try. So, please let me know if anything is not working nicely for you. Thanks.

Note

General:

Like most other plot functions in R, lattice functions use a number of common parameter terms. For example, xlab and ylab reset the x and y labels of a graph; xlim and ylim to set the x- and y-scales of a graph; col sets the color of a plot element; type sets the type ('p' for points, 'l' for lines, etc); pch and cex set plot symbol type and size, respectively; and, lty and lwd set plot line type and thickness, respectively; etc. These terms are passed onto and evaluated by all these plot functions to provide standard plot control.

latticePlot:

The default plot option for latticePlot is xyplot.

panel options for latticePlot: Default panel.xyplot. The alternative panel, panel.PEMSXYPlot supplied as part of this package, adds a grid layer to a standard xy panel. The extra code just allows you to pass specific plot parameters to the grid panel using the argument grid. You can build almost any plot layout using these and other panels in lattice and loa as building blocks.

XYZPlot:

The default plot option for latticePlot is levelplot.

pemsPlot:

pemsPlot and subsequent plot functions use an alternative convention. Here, plots include separate process and plot steps. This option allows the plot to pass on share the results of in-panel calculations with other panels and keys. The handling mechanism is part of the loa package.

The reason for latticePlot, etc:

latticePlot combines a number of lattice and latticeExtra function modifications I regularly use when plotting data. So, it is basically a short cut to save having to write out a lot of code I regularly use. I would encourage anyone to at the very least have a look at lattice.

I also hope those learning lattice, find these functions a helpful introduction and handy 'stop gap' while they are getting to grips with the code behind trellis and panel structures.

Author(s)

Karl Ropkins

References

lattice:

Sarkar, Deepayan (2008) Lattice: Multivariate Data Visualization with R. Springer, New York. ISBN 978-0-387-75968-5

latticeExtra:

Deepayan Sarkar and Felix Andrews (2011). latticeExtra: Extra Graphical Utilities Based on Lattice. R package version 0.6-18. http://CRAN.R-project.org/package=latticeExtra

lattice is one of number of really nice graphical tools in R. Others, like ggplot2 and iplot, help you to very quickly explore your data. But, for me the trellis framework of lattice has always felt the most flexible.

See Also

See lattice, latticeExtra, loa.

Examples

```
############
##example 1
############

## Not run:
#plotting pems with other packages

#base
plot(pems.1)
plot(pems.1$velocity)

#with lattice
require(lattice)
xyplot(velocity~local.time, data = pems.1, type = "l")

#with ggplot2
require(ggplot2)
qplot(time.stamp, velocity, data=pems.1)
ggplot(pems.1, aes(x = time.stamp, y = velocity)) + geom_line()
#etc


## End(Not run)

############
##example 2
############

#basic usage of latticePlot

latticePlot(velocity~local.time, data = pems.1, type = "l")

latticePlot(velocity~local.time, data = pems.1, col = "red",
            pch = 20, panel = panel.PEMSXYPlot,
            grid = list(col ="black", lty=2))

############
##example 3
############

#basic usage of XYZPlot
```

```
a <- calcAccel(velocity, local.time, data = pems.1, output="pems")

XYZPlot(~accel*velocity, data=a)

XYZPlot(~accel*velocity, data=a, plot = lattice::wireframe, shade=TRUE)


############
##example 4
############

#basic usage of pemsPlot

pemsPlot(local.time, velocity, data=pems.1, type="l")


############
##example 5
############

#basic usage of WatsonPlot

#Note: using 'a' generated in example 3
WatsonPlot(velocity, accel, data=a)

## Not run:
#omit.stopped for different handling of idling data
WatsonPlot(velocity, accel, data=a, omit.stopped="points")
WatsonPlot(velocity, accel, data=a, omit.stopped="cells")

#plot.type for different plot methods
WatsonPlot(velocity, accel, data=a, plot.type=1)
WatsonPlot(velocity, accel, data=a, plot.type=2)
WatsonPlot(velocity, accel, data=a, plot.type=3)
WatsonPlot(velocity, accel, data=a, plot.type=4)

## End(Not run)
```

`6.1.common.calculations`

Common calculations

Description

Various common calculations associated with PEMS data.

Usage

```
calcDistance(speed = NULL, time = NULL, data = NULL,
```

```
          ..., fun.name = "calcDistance")

calcSpeed(distance = NULL, time = NULL, data = NULL,
          ..., fun.name = "calcSpeed")

calcAccel(speed = NULL, time = NULL, data = NULL,
          ..., method = 2, fun.name = "calcAccel")

calcAcceleration(speed = NULL, time = NULL, data = NULL,
          ..., method = 2, fun.name = "calcAccel")

calcJerk(accel = NULL, time = NULL, data = NULL,
          ..., fun.name = "calcJerk")

#associated

calcChecks(fun.name = "calcChecks", ..., data = NULL,
          if.missing = c("stop", "warning", "return"),
          output = c("input", "data.frame", "pems", "special"),
          unit.conversions = NULL, overwrite = FALSE)

calcPack(output = NULL, data = NULL, settings = NULL,
          fun.name = "calcPack", this.call = NULL)
```

Arguments

speed, time, distance, accel	(Required data series typically vectors) The inputs to use when doing a calculation. These can typically be vectors or elements in either a data.frame or pems object.
data	(Optional data.frame or pems object) The data source if either a data.frame or pems object is being used.
...	(Optional) Other arguments, currently passed on to calcChecks.
fun.name	(Optional character) The name of the parent function, to be used in error messaging.
if.missing, output, unit.conversions, overwrite, settings, this.call	(Various) Along with data and fun.name, arguments used by calcCheck and calcPack to manage error and unit handling and workhorse calc... operations. These are typically passed to the appropriate check... or ...Units function for evaluation. See Details, Note and Examples below.
method	(Character) Currently for calcAccel only. The method to use if options exist.

Details

With the exception of calcChecks and calcPack, calc... functions are common calculations.

calcDistance calculates distance (in m) using speed and time.

calcSpeed calculates speed (in m/s) using distance and time.

calcAccel calculates acceleration (in m/s/s) using speed and time.

calcJerk calculates jerk (rate of change of acceleration in m/s/s/s) using acceleration and time.

By default results are returned as pems.elements. Other options include returning as the supplied data plus the results as either a data.frame or a pems object. See Note below.

Unit management is by convertUnits. See Note below.

The extra functions calcChecks and calcPack are add-ins that anyone can use to develop other similiar functions. They are added at the start and end of standard calc... functions to provide an option to use with third-party code. See Note.

Value

With the exception of calcChecks and calcPack, all calc... functions return either a pems.element vector, data.frame or pems object, depending on output setting and data supplied.

Note

Unit handling in pems.utils is via checkUnits, getUnits, setUnits and convertUnits. Allowed unit conversion methods have to be defined in ref.unit.conversions or a locally defined alternative supplied by the user. See convertUnits for an example of how to locally work with unit conversions.

calc.dist and calc.accel are alternatives to calcDistance and calcAccel.

The functions calcChecks and calcPack are currently under revision and likely to be replaced in future versions of pems.utils.

Author(s)

Karl Ropkins

References

References in preparation.

See Also

calcVSP for VSP calculations. calcEm for emissions calculations.

Examples

```
###########
##example 1
###########

#basic usage

#calculated accel as pems.element

calcAccel(velocity, local.time, pems.1)
```

```
#answer returned as suppied pems + calculated accel

calcAccel(velocity, local.time, pems.1, output = "pems")

#or, if you would rather...
## Not run:
pems.1$accel <- calcAccel(velocity, local.time, pems.1)

## End(Not run)

############
#example 2
############

#making wrappers for routine data processing

my.pems <- list(pems.1, pems.1)

sapply(my.pems, function(x)
                  calcAccel(velocity, local.time, data=x))

#ans = accel data series for each pems in my.pems list

#             [,1]        [,2]
# [1,]          NA          NA
# [2,]  0.00000000  0.00000000
# [3,]  0.05555556  0.05555556
# [4,]  0.00000000  0.00000000
# [5,] -0.02777778 -0.02777778
# [6,]  0.05555556  0.05555556
#       ....

#note:
#sapply if you can/want to simiplify outputs
#lapply if you want to keep output as a list of answers
```

6.2.common.check.functions

common check... functions

Description

Various pems.utils workhorse functions for input checking and routine data handling.

Usage

```
checkOption(option=NULL, allowed.options=NULL,
           option.name = "option", allowed.options.name = "allowed.options",
```

```
            partial.match=TRUE, fun.name = "checkOption",
            if.missing = c("stop", "warning", "return"),
            output = c("option", "test.result"), ...)

checkPEMS(data = NULL, fun.name = "checkPEMS",
            if.missing = c("return", "warning", "stop"),
            output = c("pems", "data.frame", "test.result"),
            ...)

checkUnits(input = NULL, units = NULL, data = NULL,
            input.name = NULL, fun.name = "checkUnits",
            if.missing = c("stop", "warning", "return"),
            output = c("special", "units", "input", "test.result"),
            ..., unit.conversions = NULL)

checkOutput(input = NULL, data = NULL,
            input.name = NULL, fun.name = "checkOutput",
            if.missing = c("stop", "warning", "return"),
            output = c("pems", "data.frame", "input", "test.result"),
            overwrite = FALSE, ...)

checkIfMissing(..., if.missing = c("stop", "warning", "return"),
            reply = NULL, suggest = NULL, if.warning = NULL,
            fun.name = NULL)
```

Arguments

input	(vector, object or object element) An input to be tested or recovered for subsequent use by another function, e.g. a speed measurement from a pems object.
data	(data.frame, pems object) If supplied, the assumed source for an input. This can currently be a standard data.frame or a 'pems' object. Note: if an input is not found in data, the parent environment is then also checked before returning an error message.
input.name, option.name	(Optional character vectors) If a check... function is used as a workhorse by another function, the name it is given in any associated error messaging. See Note below.
fun.name	(Optional character vector) If a check... function is used as a workhorse routine within another function, the name of that other function to be used in any associated error messaging. See Note below.
if.missing	(Optional character vector) How to handle an input, option, etc, if missing, not supplied or NULL. Current options include: "stop" to stop the check... function and any parent function using it with an error message; "warning" to warn users that expected information was missing but to continue running the parent code; or "return" to continue running the parent code without any warnings.
output	(Character vector) Output mode for check... function results. Options typically include the check type and "test.results". See Value below.

`...` (Optional) Other arguments, currently ignored by all `check...` functions expect `checkIfMissing`.

`option`, `allowed.options`, `allowed.options.name`
(Character vectors) For `checkOption`, `option` and `allowed.options` are the supplied option, and the allowed options it should be one of, respectively, and `allowed.options.name` if way these allowed options should be identified in any associated error messaging. See Note below.

`partial.match` (Logical) For `checkOption`, should partial matching be used when comparing `option` and `allowed.options`.

`units` (Character vector) For `checkUnits`, the units to return `input` in, if requested (`output = "input"`). Note: The default, `output = "special"`, is a special case which allows `checkUnits` to return either the units if they are not set in the call (equivalent to `output = "units"`) or the `input` in the requested units if they are set in the call (equivalent to `output = "input"`).

`unit.conversions`
(List) For `checkUnits`, the conversion method source. See `ref.unit.conversions` and `convertUnits` for further details.

`overwrite` (Logical) For `checkOutput`, when packing/repacking a `data.frame` or `pems` object, should 'same name' cases be overwritten? If `FALSE` and 'same names' are encountered, e.g. when modifying an existing `data.frame` or `pems` element, a new element if generated with a unique name in the form `[name].[number]`.

`reply`, `suggest`, `if.warning`
(Character vectors) For `checkIfMissing`, when generating error or warning messages, the main reply/problem description, any suggestions what users can try to fix this, and the action taken by the function if just warning (e.g. setting the missing value to `NULL`), respectively. All are options.

Details

The `check...` functions are intended as a means of future-proofing `pems.utils` data handling. They provide routine error/warning messaging and consistent 'front-of-house' handling of function arguments regardless of any underlying changes in the structure of the `pems` objects and/or `pems.utils` code. This means third-party function developed using these functions should be highly stable.

`checkOption` checks a supplied option against a set of allowed options, and then if present or matchable returns the assigned option. It is intended as a workhorse for handling optional function arguments.

`checkPEMS` checks a supplied data source and provides a short-cut for converting this to and from `data.frames` and `pems` object classes. It is intended as a 'best-of-both-worlds' mechanism, so users can supply data in various different formats, but function developers only have to work with that data in one (known) format.

`checkUnits` checks the units of a previously recovered `input`, and then, depending on the `output` setting, returns either the `units` of the `input` or the `input` in the required `units` (assuming the associated conversion is known).

`checkOutput` packs/repacks a previously recovered `input`. Depending on the `output` setting, this can be as the (standalone) `input`, an element of a `data.frame` or an element of a `pems` object.

checkIfMissing if a workhorse function for the if.missing argument. If any of the supplied additional arguments are NULL, it stops, warns and continues or continues a parent function according to the if.missing argument. If supplied, reply, suggest and if.warning arguments are used to generate the associated error or warning message.

Value

All check... functions return a logical if output = "test.result", TRUE if the input, option, etc., is suitable for use in that fashion or FALSE if not.

Otherwise,

checkOption return the option argument if valid (on the basis of if.missing) or an error, warning and/or NULL (on the basis of if.missing) if not. If partial.match = TRUE and partial matching is possible this is in the full form given in allowed.options regardless of the degree of abbreviation used by the user.

checkPEMS returns the data argument if valid or an error, warning and/or NULL (on the basis of if.missing) if not. Depending on output setting, the valid return is either a data.frame or pems object.

checkUnits returns the units of the input argument if no other information is supplied and units have previously been assigned to that input. If units are assigned in the call or output is forced (output = "input"), the input is returned in the requested units. If this action is not possible (e.g. pems.utils does not know the conversion), the function returns an error, a warning and the unchanged input or the unchanged input alone depending on if.missing setting.

Depening on if.missing argument, checkIfMissing either stops all parent functions with an error message, warns of a problem but allows parent functions to continue running, or allows parent functions to continue without informing the user.

Note

The ...name arguments allow the check... functions to be used silently. If a parent function is identified as fun.name and the check case (input, option, etc.) is identified with the associated ...name argument these are used in any associated error messaging.

Author(s)

Karl Ropkins

References

[TO DO]

See Also

See ref.unit.conversions and convertUnits for general unit handling.

6.3.corrections *Corrections*

Description

Corrections are a special subset of functions which by default write over the elements that they recalculate.

Usage

```
correctInput(input = NULL, ..., data = NULL,
         correction = NULL)

zeroNegatives(input = NULL, ..., data = NULL,
         screen = FALSE)

correctBaseline(x, ..., data = NULL, output = "ans")

#associated

calcPack2(input, ..., settings = NULL, data = NULL)
```

Arguments

input	(Required data series typically vectors) The input to use when makin a correction. This is typically a vector or element in either a data.frame or pems object.
x	(For correctBaseline) (Required data series typically vectors) The input to use when makin a correction. This is typically a vector or element in either a data.frame or pems object.
...	(Optional) Other arguments, typically passed on.
data	(Optional data.frame or pems object) The data source if either a data.frame or pems object is being used.
correction	(For correctInput, required function) The correction operation to apply to input. This is typically a function or function name (as character string).
screen	(For zeroNegatives, logical) If the user intends screening the correction before applying it, this should be set to TRUE.
output	(character) Where options exists for the function output, the required output. For correctBaseline, current options: 'ans' (the default) and 'diagnostic'.
settings	(For calcPack2, list) Any arguments to be used as settings when handling pems.elements. Unless developing functions, this can typically be ignored.

Details

correctInput is a general correction handlers. It accepts an input and a function, correction, which it applies to input.

zeroNegatives resets any negative values in an input to zero.

correctBaseline attempts to correct the baseline of a supplied data (pems.element vector) time-series. Baseline corrections are carried out using methods from the baseline package. See Below.

calcPack2 is an alternative version of calcPack. See associated help for details.

Value

With the exception of calcPack2, all the above functions generate input corrections.

correctBaseline returns the supplied data time-series (x) after applying the requested baseline correction (see below).

Note

By default, corrections return results in the format of the input. So: If inputs are supplied as vectors, the answer is returned as a vector; If inputs are supplied in a pems object, that pems object is returned with the answer added in. This behaviour is enabled by the default output = "special". Output type can be forced by declaring ouput in the function call. Options "input", "data.frame" and "pems" return vectors, data.frames and pems objects, respectively.

Unlike other calculations, corrections automatically replace the associated input, unless prevented (using overwrite = FALSE).

This function management is handled by calcChecks and calcPack. These are front and back end calc... function add-ins that anyone can use as a 'minimal code' means of integrating third-party and pems.utils functions.

See calcChecks documentation for details.

correctBaseline is a recent transfer from sleeper.service. It uses baseline functions to provide 'best guess' baseline corrections.

By default, it applies:

baseline(..., method="rollingBall", wm=50, ws=50)

Please Note the 'best guess': As baseline corrections are based statistical estimates of likely baselines rather than actual measures of drift these should be treated as estimates.

Unit management is by convertUnits. See associated help documentation for details.

Author(s)

Karl Ropkins

References

baseline:

Kristian Hovde Liland and Bjorn-Helge Mevik (2015). baseline: Baseline Correction of Spectra. R package version 1.2-1. https://CRAN.R-project.org/package=baseline

See Also

baseline regarding baseline corrections.

common.calculations, calcVSP or calcEm for calculation functions.

getElement (checkUnits, etc and convertUnits for data management.

Examples

```
############
##example 1
############

#basic usage

zeroNegatives(-10:10) #etc
```

6.4.analysis.summary.reports

Generating summary reports

Description

Various functions for generating summary reports for pems objects.

Usage

```
summaryReport(speed = NULL, time = NULL, accel = NULL,
          distance = NULL, data = NULL, ...,
          lod.speed = 0.1, lod.accel = 0.1,
          fun.name = "summaryReport")
```

Arguments

speed, accel, time, distance	(Data series typically vectors) The inputs to use when doing a calculation. These can typically be vectors or elements in either a data.frame or pems object if supplied as data. See Details below regarding requirements.
data	(Optional data.frame or pems object) The data source if either a data.frame or pems object is being used.
...	(Optional) Other arguments, currently passed on to calcChecks which in turn provides access to pems.utils management arguments such as if.missing and unit handlers such as unit.conversions.

`lod.speed`, `lod.accel`
: (numerics) The limits of detection for speed and accel measurements, respectively. [Note: if only one value is given for accel, the accel limits are assumed to be c(-lod.accel, + lod.accel)].

`fun.name`
: (Optional character) The name of the parent function, to be used in error messaging.

Details

`summaryReport` does not strictly require all the arguments `speed`, `accel`, `time` and `distance` as inputs. It calculates as many of the missing cases as it can using the `common.calculations` before halting an analysis or warning the user of any problems.

Unit management is by `convertUnits`. See Note below.

Value

`summaryReport` returns a one-row `data.frame` with twelve elements:

`distance.travelled.km` this total distance travelled (in km)

`time.total.s` the total time taken (in s)

`avg.speed.km.h` the mean speed as averaged across the total journey/dataset (in km/h)

`avg.running.speed.km.h` the mean speed while the vehicle was in motion (in km/h), assuming a 0.01 km/h accuracy for `speed` measurements.

`time.idle.s` and `time.idle.pc`, the time the vehicle was idling (in s and as a percentage, respectively), also assuming a 0.01 km/h cutoff for `speed` measurements.

`avg.accel.m.s.s` the mean (positive component of) acceleration (in m/s/s), assuming a 0.1 m/s/s cutoff for `accel` measurements.

`time.accel.s` and `time.accel.pc`, the time the vehicle was accelerating (in s and as a percentage, respectively), also assuming a 0.1 m/s/s cutoff for `accel` measurements.

`avg.decel.m.s.s` the mean deceleration (negative component of acceleration in m/s/s), assuming a -0.1 m/s/s cutoff for `accel` measurements.

`time.decel.s` and `time.decel.pc`, the time the vehicle was decelerating (in s and as a percentage, respectively), also assuming a -0.1 m/s/s cutoff for `accel` measurements.

Warning

Currently, `summaryReport` outputs have units incorporated into their names because the outputs themselves are unitless `data.frames`.

Note

Unit handling in `pems.utils` is via `checkUnits`, `getUnits`, `setUnits` and `convertUnits`. Allowed unit conversion methods have to be defined in `ref.unit.conversions` or a locally defined alternative supplied by the user. See `convertUnits` for an example of how to locally work with unit conversions.

Author(s)

Karl Ropkins

References

References in preparation.

See Also

checkUnits and convertUnits for data management.

Examples

```
###########
##example 1
###########

#basic usage

summaryReport(velocity, local.time, data=pems.1)

#   distance.travelled.km time.total.s avg.speed.km.h avg.running.speed.km.h
# 1              6.186056         1000        22.2698               28.78538
#   time.idle.s time.idle.pc avg.accel.m.s.s time.accel.s time.accel.pc
# 1          40            4       0.7921279          271          27.1
#   avg.decel.m.s.s time.decel.s time.decel.pc
# 1      -0.9039449          238          23.8

#apply to multiple cases

my.pems <- list(pems.1, pems.1)

sapply(my.pems, function(x)
                    summaryReport(velocity, local.time, data = x))


#                        [,1]       [,2]
# distance.travelled.km  6.186056   6.186056
# time.total.s           1000       1000
# avg.speed.km.h         22.2698    22.2698
# avg.running.speed.km.h 28.78538   28.78538
# time.idle.s            40         40
# time.idle.pc           4          4
# avg.accel.m.s.s        0.7921279  0.7921279
# time.accel.s           271        271
# time.accel.pc          27.1       27.1
# avg.decel.m.s.s        -0.9039449 -0.9039449
# time.decel.s           238        238
# time.decel.pc          23.8       23.8
```

7.1.vsp.code *Vehicle Specific Power (VSP) related code*

Description

Functions associated with VSP calculations.

Usage

```
#calculation

calcVSP(speed = NULL, accel = NULL, slope = NULL,
          time = NULL, distance = NULL, data = NULL,
          calc.method = calcVSP_JimenezPalacios,
          ..., fun.name = "calcVSP", this.call = NULL)

calcVSP_JimenezPalacios(speed = NULL, accel = NULL,
          slope = NULL, vehicle.weight = NULL, vsp.a = NULL,
          vsp.b = NULL, vsp.c = NULL, vsp.g = NULL, ...,
          data = NULL,
          fun.name = "calcVSP_JimenezPalacios",
          this.call = NULL)

#VSP binning

refVSPBin(..., bin.method="ncsu.14")

refVSPBin_NCSU.14(vsp = NULL, data = NULL,
          ..., fun.name="refVSPBin_NSCU.14")

refVSPBin_MOVES.23(vsp = NULL, speed = NULL, data = NULL,
          ..., fun.name="refVSPBin_MOVES.23")

#vsp plotting

VSPPlot(vsp, em = NULL, ..., data = NULL, plot.type = 1,
          fun.name="VSPPlot")

VSPBinPlot(vspbin, em = NULL, ..., data = NULL,
          plot.type = 1, stat = NULL, fun.name="VSPBinPlot")
```

Arguments

`speed`, `accel`, `slope`, `time`, `distance`, `vsp`, `vspbin`, `em`
: (Typically pems.element vectors) `speed`, `accel`, `slope`, `time` and `distance` are possible inputs for VSP calculation. `vsp` and `speed` are possible inputs for VSP binning methods. `vsp`, `vspbin` and `em` are `x` and `y` inputs for associated plots. (See Notes about inputs and methods.)

`data`
: (Optional, typically `pems`) The data source for inputs.

`calc.method`, `bin.method`
: (Required functions) `calc.method` is the function used to calculate VSP (default `calcVSP_JimenezPalaciosCMEM`). `bin.method` is the methods used when binning VSP measurements. (See Notes.)

`...`
: (Optional) Other arguments, currently passed on as supplied to associated calculation or binning method, or back to pemsPlot.

`fun.name`
: (Optional character) The name of the parent function, to be used in error messaging.

`this.call`
: (Optional) Initial call, should generally be ignored. See `common.calculations` for further details.

`vehicle.weight`, `vsp.a`, `vsp.b`, `vsp.c`, `vsp.g`
: (Numerics) VSP constants. If not supplied, defaults are applied. See Below.

`plot.type`
: (Optional numeric) For `VSPPlot` and `VSPBinPlot`, the type of plot to generate. For `VSPPlot`, 1 a conventional scatter plot; or 2 a box-and-whisker plot. For `VSPBinPlot`, 1 a bar plot; or 2 a box-and-whisker plot.

`stat`
: (Function) For `VSPBinPlot`, the statistic to use when calculating bar scales for `plot.type` 1. By default this is mean if `em` is supplied or count if not. NOTE: stat is ignore when `plot.type` is used

Details

`calcVSP...` functions calculate VSP:

`calcVSP` is a wrapper function which allows users to supply different combinations of inputs. VSP calculations typically require speed, acceleration and slope inputs. However, This wrapper allows different input combinations, e.g.:

time and distance (time and distance -> speed, time and speed -> accel)

time and speed (time and speed -> accel)

speed and accel

This then passes on `speed`, `accel` and (if supplied) `slope` to the method defined by `calc.method`. (This means other VSP functions run via `calcVSP(..., calc.method = function)` share this option without needed dedicated code.)

`calcVSP_JimenezPalacios` calculates VSP according to Jimenez Palacios methods. See References and Note below.

`refVSPBin...` functions generate a reference list of VSP bins:

`refVSPBin` is a wrapper that generates VSP Mode bins depending on method applied.

`binVSP_NCSU.14` bins supplied `vsp` using the 14 bin method described in Frey et al 2002.

binVSP_MOVES.23 bins supplied vsp using that and speed and the 23 bin MOVES method (See Note).

VSPPlot generates various plots of VSP (x-axis) and emission (y-axis) data.

VSPBinPlot generates various plots of VSP binned data.

Value

calcVSP by default uses the Jimenez Palacios method to calculate VSP in kW/metric ton.

refVSPBin generates a pems.element factor vector of VSP Mode bin assignments.

VSPPlot and VSPBinPlot generate plots as lattice objects.

Note

calcVSP... constants can be set/modified in the calculation call, e.g. calcVSP(..., vsp.a = [new.value]). If not supplied, defaults are used. (See References.)

binVSP_MOVES.23 is in-development. Do not use without independent confirmation of values.

Unit handling in pems.utils is via checkUnits, getUnits, setUnits and convertUnits. See common.calculations for details.

Author(s)

Karl Ropkins

References

calcVSP_JimenezPalacios uses methods described in:

Jimenez-Palacios, J.L. (1999) Understanding and Quantifying Motor Vehicle Emissions with Vehicle Specific Power and TILDAS Remote Sensing. PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA.

vehicle.weight is the vehicle mass (in metric tons), and vsp.a, vsp.b, vsp.c and vsp.g are the calculations constants for:

vsp = speed * (a * accel + (g * slope) + b) + (c * speed^3)

By default: a = 1.1, b = 0.132, c = 0.000302 and g = 0.132 (as of Jimenez-Palacios, 1999).

Method ONLY INTENDED FOR vehicles < 3.855 metric tons.

refVSPBin_NCSU.14 VSP binning as described in:

Frey, H.C., Unal, A., Chen, J., Li, S. and Xuan, C., 2002. Methodology for developing modal emission rates for EPA's multi-scale motor vehicle & equipment emission system. Ann Arbor, Michigan: US Environmental Protection Agency. (EPA420-R-02-027)

See Also

See common.calculations (and checkUnits and convertUnits) for details of data management.

Examples

```
###########
##example 1
###########

#basic usage

vsp <- calcVSP(velocity, time = local.time, data = pems.1)
#where the returned object, vsp, is vsp values as pems.element

ncsu.14 <- refVSPBin(vsp)
#where the returned object, nscu.14, is the associated modal bin
# assignments based on the Frey et al (2002) 14 bin method.
```

7.2.emissions.calculations

Emission calculations

Description

Functions associated with emissions calculations.

Usage

```
calcEm(conc = NULL, calc.method = calcEm_HoribaPitot,
          analyte = NULL, ..., data = NULL, fun.name = "calcEm",
          force = FALSE, this.call = NULL)

calcEm_HoribaPitot(conc = NULL, time = local.time, exflow = exh.flow.rate,
          extemp = exh.temp, express = exh.press, analyte = NULL,
          delay = NULL, mm = NULL, ..., force = force, data = NULL,
          fun.name = "calcEm_HoribaPitot", this.call = NULL)
```

Arguments

conc	(Data series, typically pems.element vector) Analyte/species concentrations, the main input for calculating emissions. If conc is a concentration data series from a standard pems source it should be named conc.[analyte] and pems.utils will manage it accordingly. See below for further details.
time, exflow, extemp, express	(Data series, typically pems.element vectors) Other inputs used when calculating emissions. The combination depending on the calcuation method used (and set by calc.method).
calc.method	(Required function) The function to use to calculate emissions. (Default calcEm_HoribaPitot). See below for further details.

`analyte` (Optional character vector) The analyte emissions are to be calculated for. If supplied, this is used as a reference when assigning molecule weight and other analyte properties if these are not provided as part of calculate call. If not supplied, `pems.utils` attempts to recover these from available sources, e.g. `data` if supplied as part of the calculation call or package references such as `ref.chem`.

`...` (Optional) Other arguments, currently passed on to function provided as `calc.method` (default `calcEm_HoribaPitot`) and appropriate `pems.utils` functions.

`data` (Optional pems object) The data source for inputs.

`fun.name`, `this.call`, `force` (Various pems management functions) `fun.name` (character vector) the name of the parent function, to be used in error messaging. `this.call` the initial call (can generally be ignored). `force` (Logical) Should `calcEm` and `calc.method` ignore any error checking, e.g. units assignments, and do calculations anyway?

`delay`, `mm` (Optional numerics) Emissions calculation constants. `delay` is the time delay between `conc` measurements and other timeseries. `mm` is the molecular mass of the analyte. If supplied, these in-call values supercede any preset in e.g. package `look-up.tables`.

Details

`calcEm...` functions calculate emissions.

`calcEm` is a wrapper function which is intended to provide a conventient front for emissions calculation methods. It accepts an input `conc` which it checks and passes on to `calc.method`, along with other supplied arguments.

`calcEm_HoribaPitot` calculates emissions using methods described in the Horiba OBS Operators Manual. In addition to `conc`, the function requires the time, and exhaust flow data series (measured by the OBS Pitot flow meter). By default, the function assumes that these are default names that are generated for these when standard OBS files are imported into `R` using the `pems.utils` import function `importOBS2PEMS`. See References and Note below.

Value

`calcEm_HoribaPitot` (and `calcEm` by default) use Horiba Manual methods to calculate emissions (in g/s).

Note

`calcEm...` constants can be set/modified in the calculation call, e.g. `calcEm(..., delay = [new.value])`. If not supplied, these are first checked for in the associated `pems` object (if supplied), or set to default values. See References. If analyte-related constants are to be added to a `pems` object, these should be named in the format '[type].[analyte]', e.g. `delay.co` for the `delay` constant to be used for the analyte CO.

Unit handling in `pems.utils` is via `checkUnits`, `getUnits`, `setUnits` and `convertUnits`. See `common.calculations` for details.

Author(s)

Karl Ropkins

References

calcEm_HoribaPitot uses methods described in:

The Horiba Operators Manual.

See Also

See common.calculations.

Examples

```
############
##example 1
############

#basic usage

em.co <- calcEm(conc.co, data = pems.1)

#where the returned object, em.co, is a pems.element
```

7.3.coldstart.code *Cold Start Emissions related code*

Description

Functions associated with Cold Start Emissions calculations.

Usage

```
#calculations

fitColdStart(em, time, engine.on = NULL,
             data = NULL, method = 2, ...,
             fun.name="fitColdStart")

#Cold Start Plots

coldStartPlot(time, em = NULL,
             ..., data = NULL, engine.on = NULL,
             plot.type = 1, method = 2,
             fun.name="coldStartPlot")

panel.coldStartPlot1(..., loa.settings = FALSE)
panel.coldStartPlot2(..., loa.settings = FALSE)
```

Arguments

em, time	(Typically pems.element vectors) em is the emissions data-series that the cold start contribution should be estimated for; time is the associated time-series, typically a local time measurement in seconds.
engine.on	(Optimal, single Numeric) The time the emission source, e.g. monitored vehicle engine, was started. If not supplied, this is assumed to be start of the supplied em and time data-series. See also Notes.
data	(Optional, typically pems) The data source for em and time.
method	(Optinal, Numeric) The method to use when fitting and calculating the cold start contribution: method 1 Single break point fit of accumulated emissions; method 2 modified break-point. If not supplied, method 2 is used by default.See also Notes and References.
...	(Optional) Other arguments, currently passed on as supplied to assoicated calculation or plotting function, or passed back to pemsPlot.
fun.name	(Optional character) The name of the parent function, to be used in error messaging.
plot.type	(Optional numeric) For coldStartPlot, the type of cold start plot to generate: 1 a conventional accumulation profile; or 2 an emission time-series. If not supplied, plot type 1 is selected by default.
loa.settings	(Logical) For coldStartPlot panel functions, a loa plot argument that can typically be ignored by plot users.

Details

fitColdStart fits a cold start model to the supplied emissions and time-series data.

coldStartPlot generates a plot of the cold start model.

panel.coldStartPlot1 and panel.coldStartPlot1 are plot panels used by coldStartPlot when generating plot.types 1 and 2, respectively.

Value

fitColdStart generates a cold start contribution report as a pems dataset.

coldStartPlot generates a cold start contribution report as a lattice plot.

Note

Regarding engine.on: This is specifically the time the engine is turned on rather than the row of data set where this happens. In some cases, they are same, e.g. when the data is logged at a regular 1-Hz and data capture is complete.

Regarding method: Method 1 (break-point) and method 2 (modified break-point) are based on the identification of a change point in the accumulated emissions profile.

[Doc further]

(See References.)

Author(s)

Karl Ropkins

References

`fitColdStart` uses methods described in:

[Heeb]

[Ropkins cold start]

See Also

See `common.calculations` (and `checkUnits` and `convertUnits`) for details of data management.

Examples

```
############
##example 1
############

#basic usage

#to do/maybe not run... time to compile...
```

7.4.speed.em.code *Speed Emissions related code*

Description

Functions associated with Speed/Emissions terms.

Usage

```
#calculations

fitSpeedEm(em, time, speed, engine.on = NULL,
           data = NULL, method = 1, min.speed = 5,
           bin.size = NULL, ...,
           fun.name="fitEmSpeed")

#speed/emissions Plots

speedEmPlot(speed, em = NULL, time = NULL,
         ..., data = NULL, engine.on = NULL,
         min.speed = 5, bin.size = NULL,
         plot.type = 1, method = 1,
         fun.name="speedEmPlot")
```

Arguments

em, time, speed	(Typically pems.element vectors) em is the (g/s) emissions data-series; speed and time are the associated speed profile and time-series.
engine.on	(Optimal, single Numeric) The time the emission source, e.g. monitored vehicle engine, was started. If not supplied, this is assumed to be start of the supplied em and time data-series. See also Notes.
data	(Optional, typically pems) The data source for em, speed and time.
method	(Optinal, Numeric) The method to use when calculating and binning data: method 1 calculate g/km emissions and bin by row. See also bin.size, Notes and References.
min.speed	(Optinal, Numeric) measurements when speeds were less than this value are exluded, default value 5. See also Notes.
bin.size	(Optinal, Numeric) The data binning scale to use. For method 1, this is the number of rows of measurements to merge.
...	(Optional) Other arguments, currently passed on as supplied to associated calculation or plotting function, or back to pemsPlot.
fun.name	(Optional character) The name of the parent function, to be used in error messaging.
plot.type	(Optional numeric) For speedEmPlot, the type of speed/emission plot to generate: 1 a conventional scatter plot; or 2 a box-and-whisker plot. If not supplied, plot type 1 is selected by default.

Details

fitSpeedEm builds a speed and g/km emissions data sets for the supplied emissions, speed and time-series data.

speedEmPlot generates a plot of one or more data set generated by fitSpeedEm.

Value

fitSpeedEm generates a speed/emissions contribution report as a pems dataset.

speedEmPlot generates a speed/emissions contribution report as a lattice plot.

Note

Regarding engine.on: This is specifically the time the engine is turned on rather than the row of data set where this happens. In some cases, they are same, e.g. when the data is logged at a regular 1-Hz and data capture is complete.

Regarding method: Method 1 [Doc further].

[Doc further]

(See References.)

Author(s)

Karl Ropkins

References

`fitColdStart` uses methods described in:

[COPERT on speed/emission terms]

[Ropkins speed/emissions]

See Also

See `common.calculations` (and `checkUnits` and `convertUnits`) for details of data management.

Examples

```
############
##example 1
############

#basic usage

#to do/maybe not run... time to compile...
```

`8.1.pems.tidyverse.tools`

Functions to use tidyverse code with pems.utils outputs

Description

Various codes and methods.

Usage

```
#ggplot2

## S3 method for class 'pems'
fortify(model, data, ...)

#dplyr (1) standard methods

## S3 method for class 'pems'
select(.data, ...)
## S3 method for class 'pems'
rename(.data, ...)
## S3 method for class 'pems'
filter(.data, ...)
## S3 method for class 'pems'
arrange(.data, ...)
## S3 method for class 'pems'
slice(.data, ...)
```

```
## S3 method for class 'pems'
mutate(.data, ..., units=NULL, warn=TRUE)
## S3 method for class 'pems'
group_by(.data, ..., .add=FALSE)
## S3 method for class 'pems'
groups(x)
## S3 method for class 'pems'
ungroup(x, ...)
## S3 method for class 'pems'
group_size(x)
## S3 method for class 'pems'
n_groups(x)
## S3 method for class 'pems'
summarise(.data, ...)
## S3 method for class 'pems'
pull(.data, ...)

#dplyr (2) related underscore methods

## S3 method for class 'pems'
select_(.data, ..., warn=TRUE)
## S3 method for class 'pems'
rename_(.data, ..., warn=TRUE)
## S3 method for class 'pems'
filter_(.data, ..., warn=TRUE)
## S3 method for class 'pems'
arrange_(.data, ..., warn=TRUE)
## S3 method for class 'pems'
slice_(.data, ..., warn=TRUE)
## S3 method for class 'pems'
mutate_(.data, ..., units=NULL, warn=TRUE)
## S3 method for class 'pems'
group_by_(.data, ..., .add=FALSE, warn=TRUE)
## S3 method for class 'pems'
summarise_(.data, ..., warn=TRUE)

#dplyr (3) joining methods
## S3 method for class 'pems'
inner_join(x, y, by = NULL, copy = FALSE, ...)
## S3 method for class 'pems'
left_join(x, y, by = NULL, copy = FALSE, ...)
## S3 method for class 'pems'
right_join(x, y, by = NULL, copy = FALSE, ...)
## S3 method for class 'pems'
full_join(x, y, by = NULL, copy = FALSE, ...)
## S3 method for class 'pems'
semi_join(x, y, by = NULL, copy = FALSE, ...)
## S3 method for class 'pems'
```

```
anti_join(x, y, by = NULL, copy = FALSE, ...)
```

Arguments

model, data	(pems.object) In fortify, the pems object to be used as a data source when plotting using ggplot2 code. The method is rotuinely applied by ggplot2, so users can typically ignore this. See below.
...	(Optional) Other arguments, typically passed on to equivalent tidyverse function or method.
.data	(pems.object) For dplyr functions, the pems object to be used with, e.g. dplyr code.
warn	(Optional) Give warnings? For an underscore methods: a warning that an underscore method was used (See Below). For mutate: if new elements are generated without unit assignments.
units	(Character) In mutate, the units to assign to new elements created by call. See Below.
x, y	(Various) For group... functions, x is the pems dataset to be grouped. For ...join functions, x and y are the two datasets (pems, data.frame, etc) to be joined together.
.add	(Optional) Argument used by group_by and related dplyr grouping functions.
by, copy	(Various) For ...join functions, consistent with dplyr, by and copy are optional arguments. See Below.

Details

fortify is used by ggplot2 functions when these are used to plot data in a pems dataset. Most users will never have to use this directly.

The pems object methods select, rename, filter, arrange, slice, mutate, group_by and summarise are similar to data.frame methods of the same names in dplyr, but (hopefully) they also track units, etc, like a pems object. Work in progress. See below, especially Note.

Equivalent underscore methods (select_, etc) are also provided, although it should be noted that they are probably going when dplyr drops these.

Data joining methods include inner_join, left_join, right_join, full_join, semi_join and anti_join. Like above these are similar data.frame equivalents in dplyr, but (hopefully) also track units, etc, like a pems object. Same 'work in progress' caveat. See Note.

Value

select returns the requested part of the supplied pems object, e.g.: select(pems.1, velocity) returns the velocity element of pems.1 as a single column pems.object, consistent with the data.frame handling of select.data.frame.

rename returns the supplied pems object with the requested name change, e.g.: rename(pems.1, speed=velocity) returns pems.1 with the velocity column renamed speed.

`filter` returns the supplied pems object after the requested filter operation has been applied, e.g.: `filter(pems.1, velocity>0.5)` returns pems.1 after excluding all rows where the velocity value was less than or equal to 0.5.

`arrange` returns the supplied pems object reordered based on order of values in an identified element, e.g.: `arrange(pems.1, velocity)` returns pems.1 with its row reordered lowest to highest velocity entry.

`slice` returns requested rows of the supplied pems object, e.g.: `slice(pems.1, 1:10)` returns rows 1 to 10 of pems.1 as a new pems object.

`mutate` returns the supplied pems object with extra elements calculated as requested, e.g.: `mutate(pems.1, new=velocity*2)` returns the pems object with additional column, called new, which is twice the values in the velocity column. The units of the new column can be set using the additional argument units, e.g. `mutate(pems.1, new=velocity*2, units="ick")`.

`group_by` returns a `grouped_df` object, which allowed by-group handling in subsequent `dplyr` code.

`summarise` works like `summarise(data.frame, ...)` and allows dataset calculations, e.g. `summarise(pems, mean(velocity))` calculates the mean of the `velocity` of a supplied `pems` object. Units cannot be tracked during such calls and outputs are returned as a `tibble` as with `summarise.data.frame`.

The `..._join` joining methods, join two supplied datasets. The first, `x`, must be a `pems` to employ `..._join.pems` but the second, `y` can be e.g. a `data.frame`, etc.

Warning

This currently work in progress - handle with care.

Note

Currently not sure what I think about tidyverse, but it is always interesting, and ideas like `fortify` are nice.

The `fortify` method was developed by Hadley Wickham to simplify the integration of `ggplot2` functions and special object classes.

It is a really nice idea for multiple reasons, the main one being that package users will probably never have to worry about it. However, packaging it means you can use a `pems` object directly as the data argument with `ggplot2` code.

Author(s)

Karl Ropkins

References

Generics in general:

H. Wickham. Advanced R. CRC Press, 2014.

(Not yet fully implemented within this package.)

ggplot2:

H. Wickham. ggplot2: elegant graphics for data analysis. Springer New York, 2009.

(See Chapter 9, section 9.3, pages 169-175, for discussion of fortify)

dplyr:

Hadley Wickham, Romain Francois, Lionel Henry and Kirill Muller (2020). dplyr: A Grammar of Data Manipulation. R package version 1.0.2. https://CRAN.R-project.org/package=dplyr

---

`9.1.example.data` *example data for use with pems.utils*

Description

Example data intended for use with functions in `pems.utils`.

Usage

```
pems.1
```

Format

`pems.1` is a example `pems` object.

Details

`pems.1` is supplied as part of the `pems.utils` package.

Note

None at present

Source

Reference in preparation

References

None at present

See Also

See examples in `pems.structure`.

`9.2.look-up.tables` *reference data for use with pems.utils*

Description

Various reference and example datasets intended for use with functions in `pems.utils`.

Usage

```
ref.unit.conversions

ref.chem

ref.petrol

ref.diesel

pems.scheme
```

Format

`ref.unit.conversions`: Unit conversion methods stored as a list of lists. See Details.

`ref.chem`, `ref.petrol`, `ref.diesel`: Common chemical and fuel constants stored as lists.

`pems.scheme`: Default scheme for `pems.utils` plots.

Details

`unit.conversions` is basically a 'look-up' for unit conversion methods. Each element of the list is another list. These lists are each individual conversion methods comprising four elements: `to` and `from`, character vectors given the unit ids and alias of the unit types that can be converted using the method; `conversion`, a function for the associated conversion method; and (possibly) `tag`, a more detailed description of the conversion intended for use in documentation.

Other `ref...` are sets of constants or reference information stored as lists. `ref.chem` contains atomic weights of some elements and molecular weights of some species. `ref.petrol` and `ref.diesel` contain default properties for typical fuels.

Note

`ref.unit.conversions` can be updated locally. See `convertUnits`, `addUnitConversion`, etc.

Examples

```
#basic structure
ref.unit.conversions[[1]]
```

Index