Package 'pems.utils'

March 21, 2012

Type Package

Title tools for the analysis and visualisation of pems data

Version 0.2.1

Date 2012-03-21

Author Karl Ropkins

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

Depends R (>= 2.13.0), lattice, latticeExtra

Description tools for the analysis and visualisation of pems data

License GPL

LazyLoad yes

LazyData yes

R topics documented:

`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:	0.2.1
Date:	2012-03-21
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`: `makePEMS`, `import2PEMS`, etc.

1.2. The pems object structure: `pems.structure`, `getElement`, etc.

1.3. Merging pems objects: `merge.pems`, `bindPEMS`, etc.

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

2. Routine use

2.1. Generic pems handling: `pems.generics`.

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

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

2.3.1. (Other calculations) VSP calculations: `calcVSP`, etc.

2.4. Plots for pems objects: `pems.plots`, `latticePlot`, etc.

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

2.6. Conditioning pems objects: `conditioning.pems.data`, `cutBy`, etc.

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

3.1. Example datasets: `pems.1`.

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

4. Developers tools

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

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

`makePEMS`, `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

isPEMS(x, full.test = TRUE, ...)

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

#importing data as pems objects

#general

import2PEMS(file.name = file.choose(), time.stamp = NULL, local.time = NULL,
          time.format = NULL, units = NULL, constants = NULL, history = NULL,
          ..., file.type = NULL, file.reader = read.delim)

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"), ...)
```

```
#RoyalTek GPS

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

Arguments

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

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

`...` (Optional) Other arguments, handling varies. For `isPEMS` these are ignored. For `makePEMS` 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 experiment and may be subject to change in future.

`units, constants, history`
(Default pems arguments) These are arguments that are routinely included 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.

`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.

`time.stamp, local.time, time.format`
Relatively crude `import...` functions are useful for importing data from the clipboard or simple file types. However, these sometimes need careful handling of time data. If supplied, `time.stamp` and `local.time` are used as indices or ids (column numbers or names) for data series that the user would like to use as the data time stamp and local time records, respectively. If supplied, `time.format` sets the format in which the `time.stamp` should be imported if present/idenified.

`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.

`pems, vbox` (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.

Details

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

`makePEMS` makes a `pems` object using supplied data and information.

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), by 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.

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

Value

`isPEMS` 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)`.

`makePEMS` returns a `pems` object, made using the supplied data and any additional information also supplied in the same call.

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

Warning

Currently, `makePEMS` and `import...` functions handle extra arguments differently. (See Arguments above for details.) This may be subject to change.

Note

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

Author(s)

Karl Ropkins

References

References in preparation.

See Also

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

Examples

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

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

pems                        #the pems object
summary(pems)               #summary of held data
checkInput(speed, pems)     #speed element
```

1.2.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

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

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

Arguments

`input` (A required pems element) For `getElement`, the required data element (data series in data).

`pems` (pems object) If supplied, the `pems` object to search for `element` before checking the parent environments and R workspace.

`...` (Optional) Other Arguments, currently ignored.

`fun.name, if.missing, input.name, pems.name`
(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 defaults set by `pems.utils` < `constants` declared for the `pems` object < constants given in a call.) `history` is a log of `pems` object modifications. `tags` are basically any other components that the user wishes to add to a `pems` object as identifiers.

`getElement` gets a requested data element.

`getData` gets the data component of a supplied `pems` object.

Value

`getElement` returns the requested element as a vector, if available. (If missing, error handling is managed by `if.missing`. See `check...` for more details.)

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

Warning

`get...` arguments and operations may change with `pems.utils` version. Also see Note.

Note

`get...` 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.

Author(s)

Karl Ropkins

References

References in preparation.

See Also

See `check...`.

Examples

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

#basic usage

getElement(velocity, pems.1)

#check... equivalent
#checkInput(veolcity, pems.1)
```

`1.3.merge.data.pems`

Merging data and pems objects

Description

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

Usage

```
#main function(s)

bindPEMS(x = NULL, y = NULL, ..., by = NULL)

#associated

cAlign(x = NULL, y = NULL, y.offset = 0, all = TRUE,
            suffixes = FALSE)

findLinearOffset(x = NULL, y = NULL, offset.range = NULL)
```

Arguments

`x, y` (Required objects, typically pems objects, data.frames or vectors) For `bindPEMS` and `cAlign`, two objects to be bound together. For `findLinearOffset`, two objects to be aligned.

`...` (Optional) Other arguments, currently passed on to `cAlign` and `findLinearOffset`.

`by` (Optional numeric or character) When `bindPEMS` is supplied `data.frames` or `pems` objects as `x` and `y` and is asked to use `findLinearOffset` to find `y.offset`, which elements of `x` and `y` should be used? (If this is not supplied and required, the default assumption is the first element of each.)

`y.offset` (Function or numeric) The number of rows to offset data in `y` by relative to data in `x` when combining them. For `bindPEMS`, this can be either a function that deteremines an offset, e.g. `findLinearOffset`, or a numeric value to be used directly as an offset. For `cAlign`, only a numeric is allowed.

`all` (Currently disabled) Argument in revision.

`suffixes` (Logical, or character) Any suffixes to be added to `x` and `y`. If one character terms is supplied this is used as a suffix for all names in `y`. If two character terms are supplied these are used as suffixes for all names in `x` and `y`, respectively. (The default `FALSE` does not add suffixes to either object.)

`offset.range` (Numeric) For `findLinearOffset`, the 'lag window' to compare `x` and `x` across. By default, the function applies the widest possible window.

Details

`bindPEMS` is a column binding function that binds various objects types (e.g. `pems` objects, `data.frames` and `vectors`) and returns results as `pems` objects. It uses `cAlign` to bind associated data. Also, if requested, it uses `findLinearOffset` to handle alignments.

`cAlign` is a modification of the standard `R` column binding function `cbind`. `cAlign` binds `data.frames` (or `vectors`) subject to an applied offset, `y.offset`. The row displacement of the second `data.frame` relative to the first. Unlike `cbind`, `cAlign` does not require `x` and `y` to have the same number of columns. See NOte below.

`findLinearOffset` is a wrapper for the `R` function `ccf`. It is a lag function to find the best linear offset between two data series.

Value

`bindPEMS` returns supplied objects, `x` and `y`, as a single `pems` object, subject to requested alignment and naming modifications.

`cAlign` returns supplied objects, `x` and `y`, as a single `data.frame`, subject to requested alignment and naming modifications.

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

Warning

No warnings

Note

`cAlign` generates offsets and pads out `data.frames` of different column lengths by the addition of `NAs`. So, `data.frames` do not need to be the same column length to be bound, and alignment is subject `y.offset`, a numeric giving the starting row for `y` relative to `x`. `data.frame` names are handled using `make.names`, by the `suffixes` argument to manage names directly.

Author(s)

Karl Ropkins

References

References in preparation.

See Also

See `cbind` for standard column binding in R.

Examples

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

#some data
temp <- rnorm(500)

#get two offset ranges

x <- temp[25:300]
y <- temp[10:200]

plot(x, type="l"); lines(y, col="blue", lty=2)

#estimated offset

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

#applying linear offset

ans <- cAlign(x,y, findLinearOffset(x,y))

plot(ans$x, type="l"); lines(ans$y, col="blue", lty=2)
```

`1.4.export.data` *exporting data data*

Description

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

Usage

```
exportData(data, file="tempfile.txt", ...,
           sep="\t", file.writer = write.table,
           row.names = FALSE)

#exportPEMS2Excel
#currently disabled
```

Arguments

`data` (A required object) The object to export or export data from. For `ExportData`.

`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.

`...` (Optional) Other arguments, handling varies. For `export...` functions, these are currently ignored. Note: This and may be subject to change in future.

`sep, file.writer, row.names`
(Various arguments) `file.writer` is the R function used to create the export file. `sep` and `row.names` are arguments passed to `file.writer`.

Details

`exportData` exports the data component of a `pems` object.

Value

`exportData` generates a tab-delimited file using the supplied data. If a file name is not set using `file`, it is called `tempfile.txt`.

Warning

Currently, the `export...` 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 tab-delimited copy of pems.1

##exportData(pems.1, "pems.example.txt")
##dir()
```

2.2.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"),
         hijack = FALSE)

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

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

`hijack` (Logical) The argument `code` allows functions to run directly. Can be ignored when running functions directly. See `common.calculations` for 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.

Warning

None currently

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

None currently

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")

#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
# units: "mi/h"

###########
##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)
```

2.3.1.vsp.calculations *Vehicle Specific Power (VSP) calculations*

Description

Functions associated with VSP calculations.

Usage

```
calcVSP(speed = NULL, accel = NULL, slope = NULL,
          time = NULL, distance = NULL, data = NULL,
          calc.method = calcVSPJimenezPalaciosCMEM,
          ..., fun.name = "calcVSP", hijack= FALSE)

calcVSPJimenezPalaciosCMEM(speed = NULL, accel = NULL,
          slope = NULL, m = NULL, a = NULL, b = NULL,
          c = NULL, g = NULL, ..., data = NULL,
          fun.name = "calcVSPJimenezPalaciosCMEM",
          hijack= FALSE)
```

Arguments

`speed, accel, slope, time, distance`
(Data series, typically vectors) The inputs to use when doing a calculation. See Note about `calcVSP` usage and `calc.method`.

`data` (Optional `data.frame` or `pems` object) The data source if either a `data.frame` or `pems` object is being used.

`calc.method` (Required function) The function to use to calculate VSP. (default `calcVSPJimenezPalaciosCM` See Note about `calcVSP` usage and `calc.method`.

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

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

`hijack` (Logical) Is this function being locally 'hijacked' by a user/function developer? See Note on `hijack` below.

`m, a, b, c, g`
(Numerics) VSP constants. If not supplied or preset in the associated `pems` object, defaults are applied. See Below.

Details

`calcVSP...` functions calculate VSP using.

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

... and passes on speed and accel to the method defined by `calc.method`. (This means all VSP functions run via `calcVSP(..., calc.method = function)` share this option without needed dedicated code and only required speed and accel as inputs.)

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

Value

`calcVSPJimenezPalaciosCMEM` and `calcVSP` by default use Jimenez Palacios and CMEM methods to calculate VSP (in kW/metric ton).

Warning

`calcVSPJimenezPalaciosCMEM` does not currently have special case for buses as of Giannelli et al (2005) encoded. (Please let me know if you need to use them.)

Note

`calcVSP...` constants can be set/modified in the calculation call, e.g. `calcVSP(..., a = [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 VSP constants are to be added to a `pems` object, these should have the prefix 'vsp.', so for, e.g., `a` is stored in pems constants are `vsp.a`. This is because the common VSP designations (`a`, `b`, `c`, etc.) can be very easily wrongly assigned.

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

`hijack` is an in-development argument, supplied to allow code developers to run multiple functions in different function environments. When developers 'mix and match' code from several sources it can become unstable, especially if functions are run within functions within functions, etc. `hijack = TRUE` and associated code makes a function local to 'side-step' this issue. This work by assuming/expecting all inputs to be local, i.e. supplied directly by the code user.

Author(s)

Karl Ropkins

References

`calcVSPJimenezPalaciosCMEM` 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.

Giannelli, R.A., Nam, E.K., Helmer, K., Younglove, T., Scora, G., and Barth, M. (2005) Heavy-Duty Diesel Vehicle Fuel Consumption Modelling Based on Road Load and Power Train Parameters. SAE Technical Papers, No, 05CV-3.

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

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

For vehicles < 3.855 metric tons; a = 1.1, and c = 0.000302 (as of Jimenez-Palacios, 1999).

For vehicles 3.855 to 6.350 metric tone; a = 0.0996m/2204.6, c = 1.47 + 5.22e-5m/2205 (as of Giannelli et al, 2005).

For vehicles 6.350 to 14.968 metric tone; a = 0.0875m/2204.6, c = 1.93 + 5.90e-5m/2205 (as of Giannelli et al, 2005).

For vehicles > 14.968 metric tone; a = 0.0661m/2204.6, c = 2.89 + 4.21e-5m/2205 (as of Giannelli et al, 2005).

In all cases, by default b = 0.132, g = 9.81, and if not supplied `slope` is assumed to be zero and `m` is assumed to 1.5 metric tons.

See Also

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

Examples

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

#basic usage

ans <- calcVSP(velocity, time = local.time, data = pems.1)

#ans <- pems.1 + calculated vsp
```

2.3.common.calculations *Common calculations*

Description

Various common calculations associated with PEMS data.

Usage

```
calcDistance(speed = NULL, time = NULL, data = NULL,
           ..., fun.name = "calcDistance", hijack= FALSE)

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


calcAccel(speed = NULL, time = NULL, data = NULL,
           ..., fun.name = "calcAccel", hijack= FALSE)
```

```
calcAcceleration(...)

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

#associated

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

`hijack`
(Logical) Is this function being locally 'hijacked' by a user/function developer? See Note on `hijack` below.

`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.

Details

With the exception of `calcChecks`, `calc...` functions do 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 in the supplied format. 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"`.

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 add at the start and end of standard `calc...` functions to provide a 'minimal code' mechanism for the integrating of third-party code. See Note and Example 3 below.

Value

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

Warning

No warnings

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.

hijack is an in-development argument, supplied to allow code developers to run multiple functions in different function environments. When developers 'mix and match' code from several sources it can become unstable, especially if functions are run within functions within functions, etc. hijack = TRUE and associated code makes a function local to 'side-step' this issue. This work by assuming/expecting all inputs to be local, i.e. supplied directly by the code user. See Example 3 below.

Author(s)

Karl Ropkins

References

References in preparation.

See Also

calcVSP for VSP calculations.

getElement (checkInput if passing elements as inputs), checkUnits and convertUnits for data management.

Examples

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

#basic usage

ans <- calcAccel(velocity, local.time, pems.1)

#ans = pems.1 + calculated accel

ans <- calcAccel(velocity, local.time, pems.1, output = "input")

#ans = calculated accel


###########
```

```
#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, output="input"))

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

#             [,1]        [,2]
# [1,]  0.00000000  0.00000000
# [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
# ...

#or
#lapply(my.pems, function(x)
#                  calcAccel(velocity, local.time, data=x))
#for output as a list of pems objects with accel added to each

#note:
#sapply if you can/want to simiplify output to data.frame
#lapply if you want to keep output as a list of answers


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

#making a function that allows third party hijack
#and pems management

my.fun <- function(speed, time, data = NULL, hijack = FALSE,
                   ..., fun.name = "my.function"){

    #setup
    this.call <- match.call()

    #run checks
    settings <- calcChecks(fun.name, ..., data = data)

    #get pems elements if not already got
    if(!hijack){

        #the check handle errors and error messaging
        #checkInput
        speed <- checkInput(speed, data, fun.name = fun.name,
                            if.missing = settings$if.missing,
                            unit.conversions = settings$unit.conversions)

        time <- checkInput(time, data, fun.name = fun.name,
                            if.missing = settings$if.missing,
                            unit.conversions = settings$unit.conversions)
    }

    #any extra error/missing case handling?

    #run 'hijacked' code

    #reset units to what you want
    #... allows you to pass local unit.conversions
    speed <- convertUnits(speed, to = "km/h", hijack = TRUE,
                          if.missing = settings$if.missing,
                          unit.conversions = settings$unit.conversions)

    #use someone else's calculation
    distance <- calcDistance(speed, time, hijack = TRUE,
                          if.missing = settings$if.missing,
                          unit.conversions = settings$unit.conversions)

    #do other stuff?

    #reset units again?

    #output
    #calcPack handling the output type
    #and my pems history tracking if data modified
    calcPack(output = distance, data = data, settings = settings,
             fun.name = fun.name, this.call = this.call)

}

ans <- my.fun(velocity, local.time, pems.1)

#seems long winded but gives you control of
##the error handling, unit management, and output type
##and (if working in pems objects) history logging

##and lets you do this

ans <- lapply(my.pems, function(x)
                  my.fun(velocity, local.time, data=x))

#which will not always work if you are running
#functions in functions, especially if those functions
#are also running functions in functions...
```

`2.4.pems.plots` *Various plots for pems.utils*

Description

Various plot functions and visualization tools for `pems` objects.

Usage

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

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

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

Arguments

`x` (Required formula) `pems.utils` makes extensive use of the `lattice` package. This employs a highly flexible formula based plotting framework.

For `latticePlot` the basic formula structure is `y ~ x | cond`, where `y` is the data series to use as the y-axis, `x` is the data series to as the x-axis and `cond` is an addition 'conditioning' data series which is used to separate the data into different subplots.

For `XYZPlot` the basic formula structure is `Z~ y * x | cond`, `z` is the data series to use as the z-axis or z element of the plot, `y` is the data series to use as the y-axis, `x` is the data series to as the x-axis and `cond` is an addition 'conditioning' data series which is used to separate the data into different subplots. `z` is optional, but When it is not supplied `z` it is treated as the bin count.

See Notes, Warnings and Examples.

`data` (Optional `data.frame` or `pems` object) The data source elements in `x` if not the current environment or a parent.

`plot, panel` (Optional functions) The functions to use to generate the plot framework and the individual plot panels. For `latticePlot`, these are by default the `lattice` functions `xyplot` and `panel.xyplot`. For `XYZPlot`, currently only `plot` is forced, and this is by default the `lattice` function `levelplot`

`...` (Optional) Other arguments, currently passed on to `plot` and `panel`.

`greyscale` (Logical) 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. See Warning.

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

`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`, the function to use when evaluation the elements of each data bin.

`x.res, y.res` (Numerics) when binning data with `XYZPlot`, the number of x- and y-axis bins to generate.

Details

latticePlot is a wrapper for a number of lattice and latticeExtra plot function modifications that simplify routine handling of plotted data. See Examples.

panel.PEMSXYPlot is a simple gridded panel function intended for use with the panel argument of latticePlot or lattice plot functions directly.

XYZPlot is a wrapper for a number of lattice plot functions that provide 'xyz' data visualisations.

See Examples, Warnings and Note.

Value

latticePlot is a wrapper for various lattice and latticeExtra functions that make nice graphs very quickly. It generates trellis-style graphical outputs based on 'xy' data sets.

XYZPlot generates trellis-style graphical outputs based on 'xyz' data sets.

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.

The greyscale argument is a recent addition to latticePlot. I think I have reset all default colors, but may have missing something. Please let me know if you spot anything still colored and I'll get it fixed as soon as possible. Thanks.

Note

plot options for latticePlot: The default option is xyplot.

panel options for latticePlot: The default option is panel.xyplot. The addition panel, panel.PEMSXYPlot supplied as part of this package adds a grid layer to a standard xy panel. It is simply made using two panels, panel.grid and panel.xyplot, both in lattice. edit{panel.PEMSXYPlot} to have a look at it. 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 as building blocks.

plot options for XYZPlot: The default option is levelplot.

Other arguments: 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 grpah; xlim and ylim 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 think control.

The reason for latticePlot: latticePlot combines a number of lattice and latticeExtra functions of 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 latticePlot 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`, `hexplot` and `iplot`, help you to very quickly explore your data. But, for me the trellis framework of `lattice` has always been the most flexible.

See Also

See `lattice`.

Examples

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

#basic usage of latticePlot

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

#in lattice, xyplot(velocity~local.time, data = getData(pems.1), type = "l")
#Note: to use lattice functions directly with pems objects
#       just pass data component with data = getData(pems)
#       (NOT data = pems)


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


#basic usage of XYZPlot

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

XYZPlot(~accel*velocity, data=a)

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

2.5.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, ...,
          fun.name = "summaryReport", hijack= FALSE)
```

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.

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

hijack
: (Logical) Is this function being locally 'hijacked' by a user/function developer? See Note on hijack below.

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.

hijack is an in-development argument, supplied to allow code developers to run multiple functions in different function environments. When developers 'mix and match' code from several sources it can become unstable, especially if functions are run within functions within functions, etc. hijack = TRUE and associated code makes a function local to 'side-step' this issue. This work by assuming/expecting all inputs to be local, i.e. supplied directly by the code user.

Author(s)

Karl Ropkins

References

References in preparation.

See Also

checkInput, 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
```

2.6.conditioning.pems.data

Data conditioning for pems data

Description

Various functions for the conditioning of `pems` objects.

Usage

```
cutBy(ref = NULL, ..., data = NULL, cut.method = NULL,
         labels = NULL, fun.name = "cutBy", hijack= FALSE)

cutByRow(ref = NULL, n = 4, rows = NULL, ..., data = NULL,
         fun.name = "cutByRow", hijack= FALSE)
```

Arguments

`ref` (Data series typically vector) The reference data series to consider when making a vector of subset markers/indices. See Details.

`...` (Optional) Other arguments, currently passed on to `pems.utils` management functions.

`data` (Optional `data.frame` or `pems` object) The data source if `ref` is supplied in either a `data.frame` or `pems` object.

`cut.method` (Optional function) For `cutBy` only, the method to use when cutting `ref`. If not supplied, this is supply `cutByRow` by default.

`labels` (Character vector) For `cutBy` only, a vector of names to be assigned to the cut regions.

`fun.name, hijack`
(Various managment arguments) `fun.name` is the name of the parent function, to be used in error messaging. `hijack` is a logical) that developers can use to locally 'hijack' this function. See Note on `hijack` below.

`n, rows` (numerics) `n` sets the number of equal intervals to attempt to cut the data into. `rows` sets the exact rows at which to cut the data at. If `n` is applied and the length of `ref` is not exactly divisible by `n` a best attempt is made. If both `n` and `rows` are set, `rows` is applied.

Details

`cutBy` and `cutBy...` functions generate a vector of subset markers or indices based of the type of cut applied and the range/size of the reference, `ref`. As elsewhere in `pems.utils`, by default inputs are returned in the state them are supplied. See Value.

`cutBy` is a wrapper for other `cutBy...` functions. It provides additional options for `cut.marker` naming.

`cutByRow` assigns cut regions based in row number.

Value

By default results are returned in the supplied format. 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"`.

The `cut.marker` vector generated by `cutBY` and `cutBy...` functions can then be used to condition and subsample data in `pems` objects.

Warning

Currently, no warnings.

Note

`hijack` is an in-development argument, supplied to allow code developers to run multiple functions in different function environments. When developers 'mix and match' code from several sources it can become unstable, especially if functions are run within functions within functions, etc. `hijack = TRUE` and associated code makes a function local to 'side-step' this issue. This work by assuming/expecting all inputs to be local, i.e. supplied directly by the code user.

Various other `cutBy...` options can be very simply encoded.

Author(s)

Karl Ropkins

References

References in preparation.

See Also

cut, etc. in the main R package.

Examples

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

#basic usage

#cut into equal subsets
a <- cutBy(velocity, n= 5, data=pems.1)

latticePlot(velocity~local.time|cut, data=a,
             type="l", layout=c(1,5))

#cut at three points
a <- cutBy(velocity, rows=c(180,410,700), data=pems.1)

latticePlot(velocity~local.time|cut, data=a,
             type="l", layout=c(1,4))
```

3.1.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

Examples

```
#to be confirmed
```

3.2.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
```

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.

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.

Source

TO BE COMPLETED

Examples

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

4.1.common.check.functions

common check... functions

Description

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

Usage

```
checkInput(input = NULL, data = NULL, input.name = NULL,
           fun.name = NULL, if.missing = c("stop", "warning", "return"),
           output = c("input", "test.result"), ...)

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.

checkInput checks/gets a supplied input. It is intended for use with standard function arguments, e.g. the speed time-series a user supplies for acceleration calculation in calcAccel, and supplied the input and any associated information, e.g. a source name and units, if available.

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,

checkInput returns the input argument if valid with any associated information added as attributes or an error, warning and/or NULL (on the basis of if.missing) if not.

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.

Warning

None currently

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 (codeinput, `option`, etc.) is identified with the associated `...name` argument these are used in any associated error messaging.

For example, if `checkInput` is used to get the `x` values for a standard plot, and `input.name = "x"` and `fun.name = "standard.plot"` and `x` is not found, the associated error message is `"Error: In standard.plot(...) input 'x' not found"` rather than `"Error: In checkInput(...) input 'input' not found"`, although it is `checkInput` that terminates the parent function.

Author(s)

Karl Ropkins

References

[TO DO]

See Also

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

Examples

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

#some different data

a <- 1
b <- data.frame(v=10, c=1)

#using checkInput

checkInput(a)           # 1  (attr name = "a")
checkInput(v, data = b) # 10 (attr name = "v")

#checkInput(v)
#Error:   In checkInput(...) input 'v' not found

###########
##example 2
###########
```

```
#using checkInput to plot with user labels

standard.plot <- function(x, y, data = NULL){

    #getting x and y
    my.x <- checkInput(x, data = data, input.name = "x",
                       fun.name = "standard.plot")
    my.y <- checkInput(y, data = data, input.name = "y",
                       fun.name = "standard.plot")

    #plotting (and labelling it)
    plot(my.x, my.y, xlab = attributes(my.x)$name,
         ylab = attributes(my.y)$name)
}

standard.plot(a,v, data=b) #etc
```

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'
names(x, ...)

## S3 method for class 'pems'
print(x, verbose = FALSE, ...)

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

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

Arguments

`x, object`	(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.
`...`	Addition options, typically passed to associated default method(s).
`verbose`	(Logical, TRUE/FALSE). Should the longer form of the output be returned? By default, functions with short and long output versions return the short form.

`id, ignore, n`
: (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.

Value

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

`print(pems.object)` provides a (to console) description of that `pems` object.

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

`names(pems.object)` returns a vector of the names of data series held in a `pems` object.

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

Author(s)

Karl Ropkins

Examples

```
## Not run:

#make object
print(pems.object)
names(pems.object)


## End(Not run)
```

Index