Package 'AirSensor'

October 8, 2020

Type Package

Version 1.0.2

Title Process and Display Data from Air Quality Sensors

Author Jonathan Callahan [aut, cre],
Hans Martin [aut],
Kayleigh Wilson [aut],
Tate Brasel [aut],
Helen Miller [aut],
Vasileios Papapostolou [ctb],
Ashley Collier-Oxandale [ctb],
Brandon Feenstra [ctb]

Maintainer Jonathan Callahan <jonathan.s.callahan@gmail.com>

Depends R (>= 3.5.0)

Imports countrycode, cowplot, dplyr (>= 1.0.0), dygraphs, geodist, geosphere, GGally, ggmap, ggplot2, gridExtra, httpcode, httr, jsonlite, leaflet, lubridate, magrittr, MazamaCoreUtils (>= 0.4.2), MazamaLocationUtils (>= 0.1.6), MazamaSpatialUtils (>= 0.6.3), openair, PWFSLSmoke (>= 1.2.100), readr, RColorBrewer, rlang, scales, seismicRoll, sp, stringr, tibble, tidyr, tidyselect, worldmet (>= 0.9.2), xts, zoo

Suggests knitr, markdown, testthat (>= 2.1.0), rmarkdown, roxygen2

Description Process and display data from air quality sensors. Initial focus is on PM2.5 measurements from sensors produced by 'PurpleAir' <https://www2.purpleair.com>.

URL https://github.com/MazamaScience/AirSensor

BugReports https://github.com/MazamaScience/AirSensor/issues

VignetteBuilder knitr

Repository CRAN

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

Language en-US

NeedsCompilation no

Date/Publication 2020-10-08 12:10:02 UTC

R topics documented:

`AirSensor` *Data access and analysis functions for PurpleAir sensor data*

Description

This package contains code to access current synoptic data from Purple Air as well as time series data for individual sensors from Thing Speak.

Functions for downloading and enhancing sensor data return one of two types of object:

- `pas` – PurpleAirSynoptic dataframe of uniformly named properties
- `pat` – PurpleAirTimeseries list of dataframes containing sensor metadata and data

Analysis and visualization functions provide basic functionality for working with PurpleAir sensor data and comparing it with national monitoring data retrieved with the **PWFSLSmoke** package.

`ArchiveBaseDir` *Base directory for pre-generated data*

Description

If an archive of pre-generated data files is available locally, users can set the location of this directory with`setArchiveBaseDir()`. Otherwise, users must specify an external source of pre-generated datafiles with `setArchiveBaseUrl()`.

To avoid internet latency, specification of BASE_DIR will always take precedence over specification of BASE_URL.

Package functions that load pre-generated data files will load data from this directory. These functions include:

- `pas_load()`
- `pat_load()`
- `pat_loadLatest()`
- `pat_loadMonth()`
- `sensor_load()`
- `sensor_loadLatest()`
- `sensor_loadMonth()`

Format

Directory string.

See Also

getArchiveBaseDir

setArchiveBaseDir

setArchiveBaseUrl

`ArchiveBaseUrl` *Base URL for pre-generated data*

Description

This package maintains an internal archive base URL which users can set using `setArchiveBaseUrl()`. Alternatively, if an archive of pre-generated data files is availalbe locally, users can set the location of this directory with`setArchiveBaseDir()`.

To avoid internet latency, specification of BASE_DIR will always take precedence over specification of BASE_URL. Known base URLs include:

- http://data.mazamascience.com/PurpleAir/v1

Package functions that load pre-generated data files download data from this URL. These functions include:

- `pas_load()`
- `pat_load()`
- `pat_loadLatest()`
- `pat_loadMonth()`
- `sensor_load()`
- `sensor_loadLatest()`
- `sensor_loadMonth()`

Format

URL string.

See Also

getArchiveBaseUrl

setArchiveBaseUrl

setArchiveBaseDIR

`example_pas` *Example Purple Air Synoptic dataset*

Description

The `example_pas` dataset provides a quickly loadable version of a *pa_synoptic* object for practicing and code examples. This dataset was generated on 2020-09-15 by running:

```
library(AirSensor)

initializeMazamaSpatialUtils()

example_pas <- pas_createNew(countryCodes = "US")

save(example_pas, file = "data/example_pas.rda")
```

Usage

```
example_pas
```

Format

A tibble with 16584 rows and 44 columns of data.

Source

https://www.purpleair.com/json

See Also

example_pas_raw

example_pas_raw *Example raw Purple Air Synoptic dataset*

Description

The example_pas_raw dataset provides a quickly loadable version of raw Purple Air synoptic data JSON for practicing and code examples. This dataset contains data for sensors in Washington and Oregon and was generated on 2020-09-15 by running:

```
library(AirSensor)

initializeMazamaSpatialUtils()

example_pas_raw <-
  pas_downloadParseRawData()
  dplyr::filter(Lon > -125.0 & Lon < -117.0 & Lat > 42.0 & Lat < 49.0)

save(example_pas_raw, file = "data/example_pas_raw.rda")
```

This dataset can be converted into a standard *pas* dataset with:

```
pas <- pas_enhanceData(example_pas_raw)
```

Usage

```
example_pas_raw
```

Format

A tibble with 1233 rows and 32 columns of data.

Source

https://www.purpleair.com/json

See Also

example_pas

example_pat *Example PurpleAir Timeseries dataset*

Description

The example_pat dataset provides a quickly loadable version of a *pa_timeseries* object for practicing and code examples. This dataset was was generated on 2020-09-15 by running:

```
library(AirSensor)

initializeMazamaSpatialUtils()

example_pat <- pat_createNew(
  id = "ebcb53584e44bb6f_3218",
  pas = example_pas,
  startdate = "2018-08-01",
  enddate = "2018-08-28",
  verbose = TRUE
)

save(example_pat, file = "data/example_pat.rda")
```

Usage

```
example_pat
```

Format

An S3 object composed of "meta" and "data" data.

See Also

example_pat_failure_A

example_pat_failure_B

example_pat_failure_A *Example PurpleAir Timeseries dataset exhibiting moderate errors*

Description

The example_pat_failure_A dataset provides a quickly loadable version of a *pa_timeseries* object for practicing and code examples. This dataset was was generated on 2020-09-15 by running:

```
library(AirSensor)

initializeMazamaSpatialUtils()

example_pat_failure_A <- pat_createNew(
  label = "SCNP_20",
  pas = example_pas,
  startdate = "2019-04-01",
  enddate = "2019-04-18",
  verbose = "TRUE"
)

save(example_pat_failure_A, file = "data/example_pat_failure_A.rda")
```

Usage

```
example_pat_failure_A
```

Format

An S3 object composed of "meta" and "data" data.

See Also

example_pat

example_pat_failure_B

---

example_pat_failure_B *Example PurpleAir Timeseries dataset exhibiting severe errors*

Description

The `example_pat_failure_B` dataset provides a quickly loadable version of a *pa_timeseries* object for practicing and code examples. This dataset was was generated on 2020-09-15 by running:

```
library(AirSensor)

initializeMazamaSpatialUtils()

example_pat_failure_B <- pat_createNew(
  label = "SCTV_16",
  pas = example_pas,
  startdate = "2019-06-01",
  enddate = "2019-06-18",
  verbose = TRUE
)

save(example_pat_failure_B, file = "data/example_pat_failure_B.rda")
```

Usage

```
example_pat_failure_B
```

Format

An S3 object composed of "meta" and "data" data.

See Also

example_pat

example_pat_failure_A

---

example_sensor *Example AirSensor Timeseries dataset*

Description

The `example_sensor` dataset provides a quickly loadable version of an *airsensor* object for practicing and code examples. This dataset was was generated on 2020-09-15 by running:

```
library(AirSensor)

initializeMazamaSpatialUtils()

example_sensor <- pat_createNew(
  label = "SCAN_14",
  pas = example_pas,
  startdate = "2018-08-14",
  enddate = "2018-09-07"
)
pat_createAirSensor(parameter = 'pm25', FUN = AirSensor::PurpleAirQC_hourly_AB_01)

save(example_sensor, file = "data/example_sensor.rda")
```

Usage

```
example_sensor
```

Format

An S3 object composed of "meta" and "data" data.

`example_sensor_scaqmd` *Example AirSensor Timeseries dataset*

Description

The `example_sensor_scaqmd` dataset provides a quickly loadable version of a multi-sensor *airsensor* object for practicing and code examples. This dataset was was generated on 2020-09-15 by running:

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

example_sensor_scaqmd <-
  sensor_load("scaqmd", startdate = 20190701, enddate = 20190708)

save(example_sensor_scaqmd, file = "data/example_sensor_scaqmd.rda")
```

Usage

```
example_sensor_scaqmd
```

Format

An S3 object composed of "meta" and "data" data.

`getArchiveBaseDir` *Get data archive base directory*

Description

Returns the package base directory pointing to an archive of pre-generated data files.

Usage

```
getArchiveBaseDir()
```

Value

directory string.

See Also

archiveBaseDir

setArchiveBaseDir

getArchiveBaseUrl *Get data archive base URL*

Description

Returns the package base URL pointing to an archive of pre-generated data files.

Usage

```
getArchiveBaseUrl()
```

Value

URL string.

See Also

archiveBaseUrl

setArchiveBaseUrl

initializeMazamaSpatialUtils *Initialize MazamaSpatialUtils package*

Description

Convenience function that wraps:

```
data("SimpleCountriesEEZ", package = "MazamaSpatialUtils")
data("SimpleTimezones", package = "MazamaSpatialUtils")
MazamaSpatialUtils::setSpatialDataDir('~/Data/Spatial')
MazamaSpatialUtils::loadSpatialData('NaturalEarthAdm1')
```

This function should be run before using `pas_load()`, as `pas_load()` uses the spatial data loaded by `initializeMazamaSpatialUtils()` to enhance raw synoptic data via `pas_enhanceData()`.

If file logging is desired, these commands should be run individually with output log files specified as arguments to `logger.setup()` from the **MazamaCoreUtils** package.

Usage

```
initializeMazamaSpatialUtils(
  spatialDataDir = "~/Data/Spatial",
  stateCodeDataset = "NaturalEarthAdm1",
  logLevel = WARN
)
```

Arguments

`spatialDataDir`	Directory where spatial datasets are created.
`stateCodeDataset`	MazamaSpatialUtils dataset returning ISO 3166-2 . alpha-2 stateCodes
`logLevel`	Logging level used if logging has not already been initialized.

`multi_ggplot` *Display multiple plots on one page*

Description

A plotting function that uses ggplot2 to display multiple ggplot objects in a single pane.

Usage

```
multi_ggplot(..., plotList = NULL, cols = 1)
```

Arguments

`...`	any number of ggobjects to be plotted
`plotList`	a list() of any number of ggplot objects to plot on a single pane
`cols`	Number of columns in the plot layout

Note

Additional documentation of the multiplot algorithm is available at cookbook-r.com.

`pas_addAirDistrict` *Add an air district to PurpleAir Synoptic Data*

Description

Adds an air district (if any) to a pa_synoptic object via the `MazamaSpatialUtils` Package using PurpleAir location coordinates to determine the air basin the sensor is in.

Usage

```
pas_addAirDistrict(pas = NULL)
```

Arguments

`pas`	PurpleAir Synoptic *pas* object.

Value

A pa_synoptic dataframe

Note

As of 2020-04-14, only California air basins is supported.

See Also

pas_enhanceData

Examples

```
library(AirSensor)

initializeMazamaSpatialUtils()

pas_enhanced <-
  example_pas_raw %>%
  pas_addSpatialMetadata() %>%
  pas_addAirDistrict()
```

---

pas_addCommunityRegion

Add an air district to PurpleAir Synoptic Data

Description

Adds a community region (if any) to a pa_synoptic object via the pa_synotpic object via pre-defined labeling shcema.

Usage

```
pas_addCommunityRegion(pas = NULL)
```

Arguments

pas	PurpleAir Synoptic *pas* object.

Value

A pa_synoptic dataframe

Note

As of 2020-04-14, only California air basins is supported.

See Also

pas_enhanceData

Examples

```
library(AirSensor)

initializeMazamaSpatialUtils()

pas_enhanced <-
  example_pas_raw %>%
  pas_addSpatialMetadata() %>%
  pas_addCommunityRegion()
```

---

pas_addSpatialMetadata

Add Spatial Metadata to PurpleAIr Synoptic Data

Description

Adds spatial metadata to a pa_synoptic object via the `MazamaSpatialUtils` Package using PurpleAir location coordinates to determine country, state, and timezone.

Usage

```
pas_addSpatialMetadata(pas = NULL, countryCodes = NULL)
```

Arguments

`pas`	PurpleAir Synoptic *pas* object.
`countryCodes`	(optional) ISO country codes used to subset the data.

Value

A pa_synoptic dataframe

See Also

pas_enhanceData

Examples

```
library(AirSensor)

initializeMazamaSpatialUtils()

pas_enhanced <-
  example_pas_raw %>%
  pas_addSpatialMetadata()
```

`pas_addUniqueIDs` *Add Unique Identifiers to PurpleAir Synoptic Data*

Description

Generates and adds a unique identification vector to PurpleAir sensors using the `MazamaLocationUtils` package, which creates a unique ID based upon coordinate location and device id.

Adds the following vectors:

- deviceID –PurpleAir ID
- locationID – MazamaLocationUtils generated location ID
- deviceDeploymentID – A combination of device and location IDs

Usage

```
pas_addUniqueIDs(pas = NULL)
```

Arguments

`pas`	a pa_synoptic dataframe

Value

A dataframe with generated unique ID columns added.

See Also

pas_addSpatialMetadata

Examples

```
library(AirSensor)

initializeMazamaSpatialUtils()

pas_enhanced <-
  example_pas_raw %>%
  pas_addSpatialMetadata() %>%
  pas_addUniqueIDs()
```

---

pas_createNew *Load latest PurpleAir synoptic data*

Description

Download, parse and enhance synoptic data from PurpleAir and return the results as a useful tibble with class `pa_synoptic`.

Steps include:

1) Download and parse synoptic data

2) Replace variable with more consistent, more human readable names.

3) Add spatial metadata for each sensor including:

- timezone – olson timezone
- countryCode – ISO 3166-1 alpha-2
- stateCode – ISO 3166-2 alpha-2

4) Convert data types from character to `POSIXct` and `numeric`.

5) Add distance and monitorID for the closest PWFSL monitor

Filtering by country may be performed by specifying the `countryCodes` argument.

Usage

```
pas_createNew(
  countryCodes = NULL,
  includePWFSL = TRUE,
  lookbackDays = 1,
  baseUrl = "https://www.purpleair.com/json"
)
```

Arguments

countryCodes	ISO country codes used to subset the data.
includePWFSL	Logical specifying whether to calculate distances from PWFSL monitors.
lookbackDays	Number of days to "look back" for valid data. Data are filtered to only include sensors with data more recent than lookbackDays ago.
baseUrl	Base URL for synoptic data.

Value

A PurpleAir Synoptic *pas* object.

See Also

pas_load

pas_downloadParseRawData

Examples

```
library(AirSensor)

initializeMazamaSpatialUtils()

pas <- pas_createNew("US")

if ( interactive() ) {
  pas %>%
    pas_filter(stateCode == "CA") %>%
    pas_leaflet()
}
```

---

pas_downloadParseRawData

Download synoptic data from PurpleAir

Description

Download and parse synoptic data from the Purple Air network of particulate sensors.

The synoptic data provides a view of the entire Purple Air network and includes both metadata and recent PM2.5 averages for each deployed sensor.

Usage

```
pas_downloadParseRawData(baseUrl = "https://www.purpleair.com/json")
```

Arguments

`baseUrl`	base URL for synoptic data

Value

Dataframe of synoptic PurpleAir data.

References

json formatted PurpleAir data

See Also

pas_enhanceData

Examples

```
library(AirSensor)

initializeMazamaSpatialUtils()

pas_raw <- pas_downloadParseRawData()

if ( interactive() ) {
  View(pas_raw[1:100,])
}
```

`pas_enhanceData` *Enhance synoptic data from PurpleAir*

Description

Enhance raw synoptic data from PurpleAir to create a generally useful dataframe.

Steps include:

1) Replace variable with more consistent, more human readable names.

2) Add spatial metadata for each sensor including:

- timezone – olson timezone
- countryCode – ISO 3166-1 alpha-2
- stateCode – ISO 3166-2 alpha-2
- airDistrict – CARB air districts

3) Convert data types from character to `POSIXct` and `numeric`.

4) Add distance and monitorID for the two closest PWFSL monitors

5) Add additional metadata items:

- sensorManufacturer = "Purple Air"
- targetPollutant = "PM"
- technologyType = "consumer-grade"
- communityRegion – (where known)

Filtering by country can speed up the process of enhancement and may be performed by providing a vector ISO country codes to the `countryCodes` argument. By default, no subsetting is performed.

Setting `outsideOnly = TRUE` will return only those records marked as 'outside'.

Usage

```
pas_enhanceData(pas_raw = NULL, countryCodes = NULL, includePWFSL = TRUE)
```

Arguments

`pas_raw`	Dataframe returned by `pas_downloadParseRawData()`.
`countryCodes`	ISO country codes used to subset the data.
`includePWFSL`	Logical specifying whether to calculate distances from PWFSL monitors.

Value

Enhanced Dataframe of synoptic PurpleAir data.

Note

For data obtained on July 28, 2018 this will result in removal of all 'B' channels, even those whose parent 'A' channel is marked as 'outside'. This is useful if you want a quick, synoptic view of the network, e.g. for a map.

See Also

pas_downloadParseRawData

Examples

```
library(AirSensor)

initializeMazamaSpatialUtils()

pas <- pas_enhanceData(example_pas_raw, 'US')

setdiff(names(pas), names(example_pas_raw))
setdiff(names(example_pas_raw), names(pas))

if ( interactive() ) {
  View(pas[1:100,])
}
```

`pas_filter` *General purpose filtering for PurpleAir Synoptic objects*

Description

A generalized data filter for *pas* objects to choose rows/cases where conditions are true. Rows where the condition evaluates to NA are dropped.

Usage

```
pas_filter(pas, ...)
```

Arguments

`pas`	PurpleAir Synoptic *pas* object.
`...`	Logical predicates defined in terms of the variables in the `pas`. Multiple conditions are combined with & or seperated by a comma. Only rows where the condition evaluates to TRUE are kept.

Value

A subset of the given *pas* object.

See Also

pas_filterArea, pas_filterNear

Examples

```
library(AirSensor)

nrow(example_pas)

# California
ca <- pas_filter(example_pas, stateCode == "CA")
nrow(ca)

# Seal Beach
scsb <-
  ca %>%
  pas_filter(stringr::str_detect(label, "^SCSB_"))
nrow(scsb)

if ( interactive() ) {
  pas_leaflet(ca)

  pas_leaflet(scsb, maptype = "satellite")
}
```

pas_filterArea *Rectangle area filtering for PurpleAir Synoptic objects*

Description

Filters *pas* object sensors based on a bounding box.

Usage

```
pas_filterArea(pas = NULL, w = NULL, e = NULL, s = NULL, n = NULL)
```

Arguments

pas	PurpleAir Synoptic *pas* object.
w	West edge of area bounding box (deg E).
e	East edge of area bounding box (deg E).
s	South edge of area bounding box (deg N).
n	North edge of area bounding box (deg N).

Value

A subset of the given *pas* object.

See Also

pas_filter, pas_filterNear

Examples

```
library(AirSensor)

pas <- example_pas
range(pas$longitude)
range(pas$latitude)
scsb <-
  pas %>%
  pas_filterArea(
    w = -118.10,
    e = -118.07,
    s = 33.75,
    n = 33.78
  )
range(scsb$longitude)
range(scsb$latitude)

if ( interactive() ) {
  pas_leaflet(scsb)
}
```

pas_filterNear — *Find PurpleAir sensors within radial distance*

Description

Filter for PurpleAir sensors within a specified distance from specified target coordinates.

Usage

```
pas_filterNear(pas = NULL, longitude = NULL, latitude = NULL, radius = "1 km")
```

Arguments

`pas`	PurpleAir *pas* object.
`longitude`	a Target longitude.
`latitude`	a Target latitude.
`radius`	Distance from target with unit (i.e "15 km").

Details

`radius` Should be a numeric string with a metric unit separated by a space, such as `"250 m"`.

Value

A subset of the given *pas* object.

See Also

pas_filter
pas_filterArea

Examples

```
library(AirSensor)

# Near Diamond Bar, CA
pas <- example_pas
diamond_bar <-
  pas %>%
  pas_filterNear(
    longitude = -117.820833,
    latitude = 34.001667,
    radius = "20 km"
  )

if ( interactive() ) {
  pas_leaflet(diamond_bar)
}
```

`pas_getColumn` *Return column of data from filtered PurpleAir Synoptic objects*

Description

The incoming pas object is first filtered based on the values of `states`, `pattern`, `isOutside` and `isParent`. The values associated with the `name` column are then returned.

This function is useful for returning values associated with specific *devices*, which are represented by records with `isParent = TRUE`.

Usage

```
pas_getColumn(
  pas = NULL,
  name = NULL,
  pattern = ".*",
  idPattern = ".*",
  isOutside = TRUE,
  isParent = TRUE
)
```

Arguments

pas	PurpleAir Synoptic *pas* object.
name	Name of the column to return.
pattern	Text pattern used to filter sensor labels.
idPattern	Text pattern used to filter `deviceDeploymentID`.
isOutside	Logical, is the sensor located outside?
isParent	Logical, is the record associated with a the A channel?

Value

Vector of values.

See Also

pas_getIDs, pas_getLabels

Examples

```
library(AirSensor)

example_pas %>%
  pas_getColumn(name = "latitude") %>%
  head(10)
```

`pas_getDeviceDeploymentIDs`
Return timeseries identifiers from filtered PurpleAir Synoptic objects

Description

The incoming `pas` object is first filtered based on the values of `stateCodes`, `pattern`, `isOutside` and `isParent`. The values associated with the `"deviceDeploymentID"` column are then returned.

This function is useful for returning a vector of unique time series identifiers. These are used in the names of pre-generated *pat* files found in data archives.

Usage

```
pas_getDeviceDeploymentIDs(
  pas = NULL,
  pattern = ".*",
  idPattern = ".*",
  isOutside = TRUE,
  isParent = TRUE
)
```

Arguments

`pas`	PurpleAir Synoptic *pas* object.
`pattern`	Text pattern used to filter station labels.
`idPattern`	Text pattern used to filter `deviceDeploymentID`.
`isOutside`	Logical, is the sensor located outside?
`isParent`	Logical, is the record associated with a the A channel?

Value

Vector of values.

See Also

`pas_getColumn`, `pas_getLabels`

pas_getIDs *Return IDs from filtered PurpleAir Synoptic objects*

Description

The incoming pas object is first filtered based on the values of stateCodes, patter, isOutside and isParent. The values associated with the "ID" column are then returned.

This function is useful for returning values associated with specific *devices*, which are represented by records with isParent = TRUE.

Usage

```
pas_getIDs(
  pas = NULL,
  pattern = ".*",
  idPattern = ".*",
  isOutside = TRUE,
  isParent = TRUE
)
```

Arguments

pas	PurpleAir Synoptic *pas* object.
pattern	Text pattern used to filter station labels.
idPattern	Text pattern used to filter deviceDeploymentID.
isOutside	Logical, is the sensor located outside?
isParent	Logical, is the record associated with a the A channel?

Value

Vector of values.

See Also

pas_getColumn, pas_getLabels

pas_getLabels *Return labels from filtered PurpleAir Synoptic objects*

Description

The incoming pas object is first filtered based on the values of `stateCodes`, `pattern`, `isOutside` and `isParent`. The values associated with the `"label"` column are then returned.

This function is useful for returning values associated with specific *devices*, which are represented by records with `isParent` = `TRUE`.

Usage

```
pas_getLabels(
  pas = NULL,
  pattern = ".*",
  idPattern = ".*",
  isOutside = TRUE,
  isParent = TRUE
)
```

Arguments

`pas`	PurpleAir Synoptic *pas* object.
`pattern`	Text pattern used to filter station labels.
`idPattern`	Text pattern used to filter `deviceDeploymentID`.
`isOutside`	Logical, is the sensor located outside?
`isParent`	Logical, is the record associated with a the A channel?

Value

Vector of values.

See Also

`pas_getColumn`, `pas_getIDs`, `pas_getDeviceDeploymentIDs`

Examples

```
library(AirSensor)

pas <- example_pas

pas_getLabels(pas = pas) %>% head(10)

pas_getLabels(pas = pas, pattern = "back") %>% head(10)
```

pas_hasSpatial — *Test for spatial metadata in* pa_synoptic *object*

Description

Tests for the existence of the following core spatial metadata columns:

- longitude – decimal degrees E
- latitude – decimal degrees N
- timezone – Olson timezone
- countryCode – ISO 3166-1 alpha-2
- stateCode – ISO 3166-2 alpha-2

If these columns are missing, they can be added by with `pas_addSpatialMetadata`.

Usage

```
pas_hasSpatial(pas)
```

Arguments

`pas`	A *pa_synoptic* object.

Value

`TRUE` if `pas` contains core spatial metadata, `FALSE` otherwise.

Examples

```
pas <- example_pas
pas_hasSpatial(pas)
```

pas_isEmpty — *Test for an empty* pa_synoptic *object*

Description

Convenience function for `nrow(pas) == 0`. This makes for more readable code in functions that need to test for this.

Usage

```
pas_isEmpty(pas)
```

Arguments

`pas`	A *pa_synoptic* object.

Value

`TRUE` if no data exist in `pas`, `FALSE` otherwise.

Examples

```
pas <- example_pas
pas_isEmpty(pas)
pas <- pas %>% pas_filter(ID < 0)
pas_isEmpty(pas)
```

`pas_isPas` *Test for correct structure in a* pa_synoptic *object*

Description

The `pas` is checked for the "pas" class name and presence of core metadata columns:

- ID – Purple Air ID
- label – location label
- sensorType – PurpleAir sensor type
- longitude – decimal degrees E
- latitude – decimal degrees N
- timezone – Olson timezone
- countryCode – ISO 3166-1 alpha-2
- stateCode – ISO 3166-2 alpha-2
- pm25_1hr – hourly PM2.5
- pm25_1day – daily PM2.5
- temperature – deg F
- humidity – %
- pressure – mb
- deviceID – unique device identifier
- locationID – unique location identifier
- deviceDeploymentID – unique time series identifier

Usage

```
pas_isPas(pas = NULL)
```

Arguments

`pas`	A *pa_synoptic* object.

Value

`TRUE` if `pas` has the correct structure, `FALSE` otherwise.

See Also

pas_enhanceData

Examples

```
pas_isPas(example_pas)
pas_isPas(1:10)
```

`pas_leaflet` *Leaflet interactive map of PurpleAir sensors*

Description

This function creates interactive maps that will be displayed in RStudio's 'Viewer' tab.

Typical usage would be to use the `parameter` argument to display pm25 values from one of:

- "pm25_current"
- "pm25_10min"
- "pm25_30min"
- "pm25_1hr"
- "pm25_6hr"
- "pm25_1day"
- "pm25_1week"

Auxiliary `parameter` arguments can be used to display various Purple Air sensor data. Currently supported `parameter` arguments include:

- "humidity"
- "pressure"
- "temperature"
- "pwfsl_closestDistance"

Usage

```
pas_leaflet(
  pas = NULL,
  parameter = "pm25_1hr",
  paletteName = NULL,
  radius = 10,
  opacity = 0.8,
  maptype = "terrain",
  outsideOnly = TRUE
)
```

Arguments

pas	PurpleAir Synoptic *pas* object.
parameter	Value to plot, e.g. `pm25_1hr`.
paletteName	Predefined color palette name. Can be of the following: • "AQI" • "humidity • "temperature • "distance"
radius	Radius (pixels) of monitor circles.
opacity	Opacity of monitor circles.
maptype	Optional name of leaflet ProviderTiles to use, e.g. `terrain`.
outsideOnly	Logical specifying subsetting for monitors marked as 'outside'.

Details

The `maptype` argument is mapped onto leaflet "ProviderTile" names. Current mappings include:

1. "roadmap" – "OpenStreetMap"
2. "satellite" – "Esri.WorldImagery"
3. "terrain" – "Esri.WorldTopoMap"
4. "toner" – "Stamen.Toner"

If a character string not listed above is provided, it will be used as the underlying map tile if available. See https://leaflet-extras.github.io/leaflet-providers/ for a list of "provider tiles" to use as the background map.

Value

A leaflet "plot" object which, if not assigned, is rendered in Rstudio's 'Viewer' tab.

Note

The `paletteName` parameter can take the name of an RColorBrewer paeltte, *e.g.* `"BuPu"` or `"Greens"`.

Examples

```
library(AirSensor)
setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

# California
ca <-
  pas_load() %>%
  pas_filter(stateCode == 'CA')

if ( interactive() ) {
  pas_leaflet(ca, parameter = "pm25_1hr")

  pas_leaflet(ca, parameter = "temperature")

  pas_leaflet(ca, parameter = "humidity")

  pas_leaflet(ca, parameter = "pwfsl_closestDistance", maptype = "satellite")
}
```

`pas_load` *Load PurpleAir synoptic data*

Description

A pre-generated *pa_synoptic* object will be loaded for the given date. These files are generated each day and provide a record of all currently installed PurpleAir sensors for the day of interest. With default arguments, this function will always load data associated with the most recent pre-generated file – typically less than one hour old.

The `datestamp` can be anything that is understood by `lubrdiate::ymd()` including either of the following recommended formats:

- `"YYYYmmdd"`
- `"YYYY-mm-dd"`

By default, the host computer's date is used.

The *pas* object for a specific hour may be loaded by specifying `datestamp = "YYYYmmddHH"`.

Usage

```
pas_load(
  datestamp = NULL,
  retries = 30,
  timezone = "America/Los_Angeles",
  archival = FALSE,
  verbose = TRUE
)
```

Arguments

`datestamp`	Local date string in valid YYYY-mm-dd format. See description.
`retries`	Max number of days to go back and try to load if requested date cannot be retrieved.
`timezone`	Timezone used to interpret `datestamp`.
`archival`	Logical specifying whether a version should be loaded that includes sensors that have stopped reporting.
`verbose`	Logical controlling the generation of warning and error messages.

Value

A PurpleAir Synoptic *pas* object.

See Also

pas_createNew

Examples

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

pas <- pas_load()

if ( interactive() ) {
  pas %>%
    pas_filter(stateCode == "CA") %>%
    pas_leaflet()
}
```

`pas_palette` *Color palettes for PurpleAir*

Description

Generates color palettes for PurpleAir synoptic data with the intention of having a reproducible functional color generator.

Usage

```
pas_palette(pas = NULL, paletteName = "AQI", parameter = "pm25_1hr", ...)
```

Arguments

`pas`	Enhanced data frame of PurpleAir synoptic data.
`paletteName`	A predefined color palette name. Can be of the following: • "AQI" • "humidity • "temperature • "distance"
`parameter`	Value to generate colors for, e.g. `pm25_1hr`.
`...`	Additional arguments passed on to `leaflet::color~` functions.

Value

An object that consists of a label and color dataframe, and calculated color values from PurpleAir sensors

Note

The `paletteName` parameter can take the name of an RColorBrewer paeltte, *e.g.* `"BuPu"` or `"Greens"`.

`pas_staticMap` *Static map of PurpleAir sensors*

Description

Creates a static map of a *pas* object

Users can create a map using any numeric data column within the *pas* object:

`"pm25" "temperature" "humidity" "pressure" "pm25_current" "pm25_10min" "pm25_30min" "pm25_1hr" "pm25_6hr" "pm25_1day" "pm25_1week" "pwfsl_closestDistance"`

Available `paletteName` options include an `"AQI"` color palette, as well as a suite of sequential and diverging palettes from the **RColorBrewer** R package.

The sequential palette names are

`"Blues" "BuGn" "BuPu" "GnBu" "Greens" "Greys" "Oranges" "OrRd" "PuBu" "PuBuGn" "PuRd" "Purples" "RdPu" "Reds" "YlGn" "YlGnBu" "YlOrBr" "YlOrRd"`

The diverging palette names are

`"BrBG" "PiYG" "PRGn" "PuOr" "RdBu" "RdGy" "RdYlBu" "RdYlGn" "Spectral"`

Additional map tile info found at: http://maps.stamen.com/

Usage

```
pas_staticMap(
  pas = NULL,
  parameter = "pm25_1hr",
  paletteName = "Purples",
  mapTheme = "terrain",
  mapShape = "sq",
  direction = 1,
  minScale = 0,
  maxScale = 150,
  shape = 15,
  size = 2,
  alpha = 0.8,
  bbuff = 0.5,
  zoomAdjust = 0,
  ...
)
```

Arguments

`pas`	PurpleAir Synoptic *pas* object.
`parameter`	Value to plot, e.g. `pm25_1hr`.
`paletteName`	Base color or palette name to be used.
`mapTheme`	Default is "terrain", see description for additional options.
`mapShape`	Default is "square", can also be "natural".
`direction`	Legend color direction.
`minScale`	Minimum value to set scale for color gradient. Default is 0.
`maxScale`	Maximum value to set scale for color gradient. Default is 150.
`shape`	Symbol to use for points.
`size`	Size of points.
`alpha`	Opacity of points.
`bbuff`	Bounding box buffer. Default is 0.1.
`zoomAdjust`	Adjustment to map zoom level (-1:3).
`...`	Additional options: the legend can disabled `guide = FALSE`, and renamed with `name= "Example name"`.

Value

A ggplot object.

Examples

```
library(AirSensor)

LA_basin <-
```

```
  example_pas %>%
  pas_filterArea(-118.5, -117.5, 33.5, 34.5)
pas_staticMap(LA_basin, paletteName = "AQI", zoomAdjust = 1)
```

`pas_upgrade` *Upgrade* pa_synoptic *object format*

Description

The pas is checked for the latest pa_synoptic format and presence of core metadata columns:

- ID – Purple Air ID
- label – location label
- DEVICE_LOCATIONTYPE – location descriptor
- THINGSPEAK_PRIMARY_ID – Thingspeak API access ID
- THINGSPEAK_PRIMARY_ID_READ_KEY – Thingspeak API access key
- THINGSPEAK_SECONDARY_ID – Thingspeak API access ID
- THINGSPEAK_SECONDARY_ID_READ_KEY – Thingspeak API access key
- longitude – decimal degrees E
- latitude – decimal degrees N
- pm25 – latest PM25
- lasteSeenDate – last update datetime
- sensorType – PurpleAir sensor type
- flag_hidden – hidden flag
- isOwner – owner logical
- humidity – %
- temperature – deg F
- pressure – mb
- age – sensor age
- parentID – device parent ID
- timezone – Olson timezone
- flag_highValue – out of spec flag
- flag_attenutation_hardware – hardware failure flag
- Ozone1 – latest ozone data
- pm25_current – current PM2.5 data
- pm25_10min – 10-minute average PM2.5 data
- pm25_30min – 30-minute average PM2.5 data
- pm25_1hr – 1-hour average PM2.5 data
- pm25_6hr – 6-hour average PM2.5 data

- pm25_1day – 1-day PM2.5 average data
- pm25_1week – 1-week PM2.5 average data
- statsLastModifiedDate – last modified date
- statsLastModifiedInterval – interval between modified date
- deviceID – unique device identifier
- locationID – generated location ID
- deviveDeploymentID – generated unique ID
- countryCode – ISO 3166-1 alpha-2
- stateCode – ISO 3166-2 alpha-2
- timezone – location timezone
- airDistrict – Air district, if any
- pwfsl_closestDistance – nearest regulatory monitor distance, meters
- pwfsl_closestMonitorID – nearest regularoty monitor ID
- sensorManufacturer – hardware manufacturer
- targetPollutant – target pollutant data
- technologyType – type of sensor technology
- communityRegion – defined regional community.

Usage

```
pas_upgrade(pas = NULL, verbose = TRUE)
```

Arguments

pas	A *pa_synoptic* object.
verbose	(logical) Display upgrade messages.

Value

`TRUE` if `pas` has the correct structure, `FALSE` otherwise.

Examples

```
library(AirSensor)

# Initialize the required spatial utilities
initializeMazamaSpatialUtils()

# Use outdated pa_synoptic database
setArchiveBaseUrl('http://smoke.mazamascience.com/data/PurpleAir')

pas <-
  pas_load() %>%
  pas_upgrade()
```

patData_aggregate *Aggregate PurpleAir Timeseries Data*

Description

Aggregate a dataframe into temporal bins and apply a function. Temporal aggregation involves splitting a dataframe into separate bins along its `datetime` axis. `FUN` is mapped to the `df` dataframe records in each bin which are then recombined into an aggregated dataframe.

Usage

```
patData_aggregate(
  df,
  FUN = function(df) {     mean(df$pm25_A + df$pm25_B, na.rm = TRUE) },
  unit = "minutes",
  count = 60
)
```

Arguments

`df`	Timeseries *pat* data, or timeseries data.frame with valid *datetime* column.
`FUN`	The function to be applied to each vector of numeric `df`.
`unit`	Character string specifying temporal units for binning.
`count`	Number of units per bin.

Details

This function is intended for advanced users who wish to have more flexibility than the standard *pat_aggregate()* while aggregating timeseries data. `FUN` can operate and access all numeric vectors within the data frame `df` and must return a matrix or tibble of numeric values. Any errors generated during application of `FUN` on subsets of `df` must be handled as in the example.

Value

Returns an aggregated *data.frame* object.

Examples

```
library(AirSensor)

# Single day subset
pat <-
  example_pat %>%
  pat_filterDate(20180813, 20180814)

# Two Sample Student T-Test (advanced users only - see details.)
FUN_ttest <- function(x) {
```

```
  result <- try({
    hourly_ttest <- stats::t.test(x$pm25_A, x$pm25_B, paired = FALSE)
    tbl <- dplyr::tibble(
      t_score = as.numeric(hourly_ttest$statistic),
      p_value = as.numeric(hourly_ttest$p.value),
      df_value = as.numeric(hourly_ttest$parameter)
    )
  }, silent = TRUE)
  if ( "try-error" %in% class(result) ) {
    tbl <- dplyr::tibble(
      t_score = as.numeric(NA),
      p_value = as.numeric(NA),
      df_value = as.numeric(NA)
    )
  }
  return(tbl)
}

t.testStats <-
  pat %>%
  pat_extractData() %>% # Note: Extract the timeseries data.frame
  patData_aggregate(FUN_ttest)

head(t.testStats)
```

`pat_aggregate` *Aggregate PurpleAir Timeseries Object*

Description

Aggregate PurpleAir timeseries (*pat*) object along its datetime axis. Temporal aggregation involves splitting a *pat* object into separate bins along its datetime axis. `FUN` is mapped to the *pat* numeric variables in each bin, which are then recombined into an aggregated *pat* object containing the same metadata as the incoming `pat`.

Usage

```
pat_aggregate(
  pat,
  FUN = function(x) {     mean(x, na.rm = TRUE) },
  unit = "minutes",
  count = 60
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`FUN`	The function to be applied to each vector of numeric `pat` data.
`unit`	Character string specifying temporal units for binning.
`count`	Number of units per bin.

Details

FUN must operate on univariate numeric vectors and return a scalar value. Besides the data variable, no additional arguments will be provided to this function. This means that functions like mean and max will need to be wrapped in a function that specifies na.rm = TRUE. See the examples below.

Value

Returns an aggregated *pat* object.

Examples

```
library(AirSensor)

# Single day subset
pat <-
  example_pat %>%
  pat_filterDate(20180813, 20180814)

# Create aggregation functions
FUN_mean <- function(x) mean(x, na.rm = TRUE)
FUN_max <- function(x) max(x, na.rm = TRUE)
FUN_count <- function(x) length(na.omit(x))

# Hourly means
pat %>%
  pat_aggregate(FUN_mean) %>%
  pat_extractData() %>%
  dplyr::select(1:9)

# Hourly maxes
pat %>%
  pat_aggregate(FUN_max) %>%
  pat_extractData() %>%
  dplyr::select(1:9)

# Hourly counts
pat %>%
  pat_aggregate(FUN_count) %>%
  pat_extractData() %>%
  dplyr::select(1:9)

# Alternative 10 minute aggregation (advanced users only - see details.)
pat %>%
  pat_aggregate(FUN_max, unit = "minutes", count = 10) %>%
  pat_extractData() %>%
  dplyr::select(1:9) %>%
  dplyr::slice(1:6)
```

`pat_aggregateOutlierCounts`
Aggregate data with count of outliers in each bin

Description

Aggregate data with count of outliers in each bin

Usage

```
pat_aggregateOutlierCounts(
  pat = NULL,
  unit = "minutes",
  count = 60,
  windowSize = 23,
  thresholdMin = 8
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`unit`	Character string specifying temporal units for binning.
`count`	Number of units per bin.
`windowSize`	the size of the rolling window. Must satisfy windowSize <= count.
`thresholdMin`	the minimum threshold value to detect outliers via hampel filter

Value

`data.frame` A data.frame with flag counts per bin.

See Also

pat_aggregateData

Examples

```
library(AirSensor)
library(ggplot2)

df <-
  pat_aggregateOutlierCounts(example_pat_failure_A)

# Plot the counts
multi_ggplot(
  # A Channel
  ggplot(df, aes(x = datetime, y = pm25_A_outlierCount)) + geom_point(),
```

```
  # B Channel
  ggplot(df, aes(x = datetime, y = pm25_B_outlierCount)) + geom_point(),
  # Humidity
  ggplot(df, aes(x = datetime, y = humidity_outlierCount)) + geom_point(),
  # Temperature
  ggplot(df, aes(x = datetime, y = temperature_outlierCount)) + geom_point()
)
```

---

pat_createAirSensor *Create an Air Sensor object*

Description

Converts data from a *pat* object with an irregular time axis to an *airsensor* object where the numeric data has been aggregated along a standardized hourly time axis, as well as adding additional required metadata for compatibility with the *PWFSLSmoke* package.

Usage

```
pat_createAirSensor(
  pat = NULL,
  parameter = "pm25",
  FUN = PurpleAirQC_hourly_AB_01,
  ...
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`parameter`	Parameter for which to create an univariate *airsensor* object. See details.
`FUN`	Algorithm applied to *pat* object for hourly aggregation and quality control. See details.
`...`	(optional) Additional parameters passed into `FUN`.

Details

`FUN` allows users to provide custom aggregation and quality-control functions that are used to create an *airsensor* object. The `FUN` must accept a *pat* object as the first argument and return a dataframe with a regular hourly datetime axis. `FUN` can access and utilize any component of a standard *pat* object (e.g pm25_A, temperature, etc.) as well as define new variables in the *pat* data. See examples.

`parameter` allows user to select which variable to use for the univariate *airsensor* object (e.g 'pm25_A', 'humidity', etc.). Furthermore the `parameter` can be a new variable created via `FUN` evaluation. See examples.

Additional named parameters can be be passed to `FUN` through `...`.

Value

An "airsensor" object of aggregated PurpleAir Timeseries data.

See Also

PurpleAirQC_hourly_AB_01

pat_aggregate

Examples

```
library(AirSensor)

# Default FUN = PurpleAirQC_hourly_AB_00
sensor <- pat_createAirSensor(example_pat)

PWFSLSmoke::monitor_timeseriesPlot(sensor, shadedNight = TRUE)

# Try out other package QC functions
example_pat %>%
  pat_createAirSensor(FUN = PurpleAirQC_hourly_AB_01) %>%
  PWFSLSmoke::monitor_timeseriesPlot(shadedNight = TRUE)

example_pat %>%
  pat_createAirSensor(FUN = PurpleAirQC_hourly_AB_01) %>%
  PWFSLSmoke::monitor_timeseriesPlot(shadedNight = TRUE)

# Custom FUN
humidity_correction <- function(pat, z = 0) {

  # Default hourly aggregation
  hourlyData <-
    pat %>%
    pat_aggregate() %>%
    pat_extractData()

  # Create custom_pm variable
  pm25 <- (hourlyData$pm25_A + hourlyData$pm25_B) / 2
  hum <- hourlyData$humidity
  temp <- hourlyData$temperature
  hourlyData$custom_pm <- pm25 - (pm25 * hum * z)

  return(hourlyData)

}

# Evaluate custom FUN
sensor <- pat_createAirSensor(
  example_pat,
  parameter = "custom_pm",
  FUN = humidity_correction,
  z = .005
```

```
)

PWFSLSmoke::monitor_timeseriesPlot(sensor, shadedNight = TRUE)
```

`pat_createNew` *Load latest PurpleAir time series data*

Description

Retrieve and parse timeseries data from the Thingspeak API for specific PurpleAir sensors.

Dates can be anything that is understood by `MazamaCoreUtils::parseDatetime()` including any of the following recommended formats:

- `"YYYYmmdd"`
- `"YYYY-mm-dd"`
- `"YYYY-mm-dd HH:MM:SS"`

Usage

```
pat_createNew(
  id = NULL,
  label = NULL,
  pas = NULL,
  startdate = NULL,
  enddate = NULL,
  timezone = NULL,
  baseUrl = "https://api.thingspeak.com/channels/",
  verbose = FALSE
)
```

Arguments

`id`	PurpleAir sensor 'deviceDeploymentID'.
`label`	PurpleAir sensor 'label'.
`pas`	PurpleAir Synoptic *pas* object.
`startdate`	Desired UTC start time (ISO 8601) or `POSIXct`.
`enddate`	Desired UTC end time (ISO 8601) or `POSIXct`.
`timezone`	Timezone used to interpret start and end dates.
`baseUrl`	Base URL for Thingspeak API.
`verbose`	Logical controlling the generation of warning and error messages.

Value

A PurpleAir Timeseries *pat* object.

Note

When `timezone = NULL`, the default, dates are interpreted to be in the local timezone for the sensor of interest.

Starting with **AirSensor** version 0.6, archive file names are generated with a unique "device-deployment" identifier by combining a unique location ID with a unique device ID. These "device-deployment" identifiers guarantee that movement of a sensor will result in the creation of a new time series.

Users may request a *pat* object in one of two ways:

1) Pass in `id` with a valid a `deviceDeploymentID`

2) Pass in both `label` and `pas` so that the `deviceDeploymentID` can be looked up.

See Also

pat_downloadParseRawData

Examples

```
library(AirSensor)

pat <- pat_createNew(
  label = "Seattle",
  pas = example_pas,
  startdate = 20180701,
  enddate = 20180901
)
pat_multiPlot(pat)
```

---

pat_createPATimeseriesObject

Combine PurpleAir raw dataframes

Description

The `pat_downloadParseRawData()` function returns four dataframes of data from ThingSpeak. These must be combined into the single `data` dataframe found in a 'pat' object. This process involves selecting data columns to use and bringing all data onto a unified time axis.

Two sets of data values exist in the raw data, one for each of two algorithms that convert particle counts into aerosol density.

PurpleAir has the following description:

The CF_ATM and CF_1 values are calculated from the particle count data with a proprietary algorithm developed by the PMS5003 laser counter manufacturer, PlanTower. The specifics of the calculation are not available to the public (or us for that matter). However, to convert the particle count data (um/dl) to a mass concentration (ug/m3) they must use an average particle density. They *do provide 2 different mass concentration conversion options; CF_1 uses the "average particle density" for indoor particulate matter and CF_ATM uses the "average particle density" for outdoor particulate matter.*

The **AirSensor** package and all associated archive data use PlanTower algorithm `CF_ATM`.

Usage

```
pat_createPATimeseriesObject(pat_rawList = NULL)
```

Arguments

`pat_rawList`	List of dataframes as returned by `pat_downloadParseRawData()`.

Value

A PurpleAir Timeseries *pat* object.

References

https://www2.purpleair.com/community/faq#!hc-what-is-the-difference-between-cf-1-and-cf-atm

Examples

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

pas <- pas_load()

pat_rawList <- pat_downloadParseRawData(
  id = "78df3c292c8448f7_21257",
  pas = pas
)

pat <- pat_createPATimeseriesObject(pat_rawList)

pat_multiPlot(pat)
```

`pat_dailySoH` *Daily state of health*

Description

This function combines the output of the State of Health (SoH) function arguments into a single tibble.

Usage

```
pat_dailySoH(
  pat = NULL,
  SoH_functions = c("PurpleAirSoH_dailyPctDC", "PurpleAirSoH_dailyPctReporting",
    "PurpleAirSoH_dailyPctValid", "PurpleAirSoH_dailyOtherFit",
    "PurpleAirSoH_dailyABFit", "PurpleAirSoH_dailyABtTest")
)
```

Arguments

pat	PurpleAir Timeseries pat object.
SoH_functions	Vector of function names. All the passed in functions must output tibbles with a daily datetime variable and must cover the same period of time.

See Also

pat_dailySoHPlot

Examples

```
library(AirSensor)

SoH <-
  example_pat_failure_B %>%
  pat_dailySoH()

timeseriesTbl_multiPlot(SoH, ncol = 4)
```

---

pat_dailySoHIndexPlot *Daily State of Health metric plot*

Description

This function plots a subset of the most useful State of Health metrics calculated with SoHIndex_FUN.

Both minPctReporting and breaks are passed to SoHIndex_FUN.

Usage

```
pat_dailySoHIndexPlot(
  pat = NULL,
  minPctReporting = 50,
  breaks = c(0, 0.2, 0.8, 1),
  SoHIndex_FUN = pat_dailySoHIndex_00
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`minPctReporting`	Percent reporting threshold for A and B channels.
`breaks`	Breaks used to convert index values into index bins.
`SoHIndex_FUN`	Function used to create `SoHIndex` tibble. (Not quoted.)

See Also

pat_dailySoHIndex_00

Examples

```
library(AirSensor)

gg_A <- pat_dailySoHIndexPlot(example_pat_failure_A)
gg_B <- pat_dailySoHIndexPlot(example_pat_failure_B)

multi_ggplot(gg_A, gg_B)
```

`pat_dailySoHIndex_00` *State of Health index plot*

Description

This function calculates the `pat_dailySoH` function and returns a tibble containing a state of health index for each day of the `pat` provided. The returned tibble contains columns: `datetime`, `index`, and `index_bin`.

The `index` column contains a value normalized between 0 and 1 where 0 represents low confidence in the sensor data and 1 represents high confidence. The `index_bin` is one of 1, 2, or 3 and represents poor, fair, and good data respectively.

The `index` is calculated in the following manner:

1. If the A or B channel percent reporting is < `minPctReporting`, `index = 0`
2. Otherwise, `index = pm25_A_pm25_B_rsquared`

The `breaks` are used to convert `index` into the `index_bin` poor-fair-good values.

Usage

```
pat_dailySoHIndex_00(
  pat = NULL,
  minPctReporting = 50,
  breaks = c(0, 0.2, 0.8, 1)
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`minPctReporting`	Percent reporting threshold for A and B channels.
`breaks`	Breaks used to convert index values into index bins.

Examples

```
library(AirSensor)

tbl <-
  example_pat_failure_A %>%
  pat_dailySoHIndex_00()

head(tbl)
```

`pat_dailySoHPlot` *Daily State of Health metric plot*

Description

This function plots a subset of the most useful State of Health metrics calculated by the `pat_dailySoH` function. The function runs `pat_dailySoH` internally and uses the output to create the plot.

Usage

```
pat_dailySoHPlot(pat = NULL, ncol = 2)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`ncol`	Number of columns in the faceted plot.

See Also

pat_dailySoH

Examples

```
library(AirSensor)

pat_dailySoHPlot(example_pat_failure_B)
```

`pat_distinct` *Retain only distinct data records in pat$data*

Description

Performs two passes to guarantee that the `datetime` axis contains no repeated values:

1. remove any duplicate records
2. guarantee that rows are in `datetime` order
3. average together fields for any remaining records that share the same `datetime`

Usage

```
pat_distinct(pat)
```

Arguments

`pat`	*pat* object

Value

A *pat* object with no duplicated data records.

`pat_downloadParseRawData` *Download PurpleAir timeseries data*

Description

Downloads timeseries data for a specific PurpleAir sensor from the ThingSpeak API and parses the content into individual dataframes. This function will always return dataframes with the appropriate columns even if no data are returned from ThingSpeak.

The returned list contains the following dataframes:

- `meta` – `pas` records for the specified sensor
- `A_PRIMARY` – channel A primary dataset
- `A_SECONDARY` – channel A secondary dataset
- `B_PRIMARY` – channel B primary dataset
- `B_SECONDARY` – channel B secondary dataset

These dataframes contain **ALL** data available from ThingSpeak for the specified sensor and time period.

See the references.

Usage

```
pat_downloadParseRawData(
  id = NULL,
  label = NULL,
  pas = NULL,
  startdate = NULL,
  enddate = NULL,
  timezone = NULL,
  baseUrl = "https://api.thingspeak.com/channels/"
)
```

Arguments

id	PurpleAir sensor 'deviceDeploymentID'.
label	PurpleAir sensor 'label'.
pas	PurpleAir Synoptic *pas* object.
startdate	Desired start time (ISO 8601).
enddate	Desired end time (ISO 8601).
timezone	Timezone used to interpret start and end dates.
baseUrl	Base URL for Thingspeak API.

Value

List containing multiple timeseries dataframes.

References

https://www2.purpleair.com/community/faq#!hc-sd-card-csv-file-header

Examples

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

pas <- pas_load()

pat_rawList <-
  pat_downloadParseRawData(
    id = "78df3c292c8448f7_21257",
    pas = pas
  )

lapply(pat_rawList, head)
```

`pat_dygraph` *Interactive time series plot*

Description

This function creates interactive graphs that will be displayed in RStudio's 'Viewer' tab.

The list of available parameters include:

- `pm25` – A and B channel PM2.5 (ug/m3)
- `temperature` – temperature (F)
- `humidity` – humidity (%)
- `pressure` – pressure (hPa)

Usage

```
pat_dygraph(
  pat = NULL,
  parameter = "pm25",
  sampleSize = 5000,
  title = NULL,
  xlab = NULL,
  ylab = NULL,
  tlim = NULL,
  rollPeriod = 1,
  showLegend = TRUE,
  colors = NULL,
  timezone = NULL
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object from `pat_createNew()`
`parameter`	Data to display: "pm25", "humidity", "temperature"
`sampleSize`	Either an integer or fraction to determine sample size
`title`	title text
`xlab`	optional title for the x axis
`ylab`	optional title for the y axis
`tlim`	optional vector with start and end times (integer or character representing YYYYMMDD[HH])
`rollPeriod`	rolling mean to be applied to the data
`showLegend`	logical to toggle display of the legend
`colors`	string vector of colors to be used for plotting
`timezone`	Olson timezone used to interpret `tlim`. (Defaults to `pat` local time.)

Value

Initiates the interactive dygraph plot in RStudio's 'Viewer' tab.

Examples

```
library(AirSensor)

North_Bend_Weather <-
  pat_createNew(
    label = "North Bend Weather",
    pas = example_pas,
    startdate = 20180801,
    enddate = 20180901,
    verbose = TRUE
  )

if ( interactive() ) {
  North_Bend_Weather %>%
    pat_sample(sampleSize = 1000, setSeed = 1) %>%
    pat_dygraph(xlab = "2018", rollPeriod = 6)
}
```

---

pat_externalFit *Linear model fitting of PurpleAir and federal PWFSL time series data*

Description

Produces a linear model between data from PurpleAir and data from the closest PWFSL monitor. A diagnostic plot is produced if 'showPlot = TRUE'.

Usage

```
pat_externalFit(
  pat = NULL,
  showPlot = TRUE,
  size = 1,
  pa_color = "purple",
  pwfsl_color = "black",
  alpha = 0.5,
  lr_shape = 15,
  lr_color = "black",
  lr_lwd = 1.5,
  lr_lcolor = "tomato",
  lr_lalpha = 0.45,
  ts_shape = 1,
```

```
  xylim = NULL,
  channel = "ab",
  replaceOutliers = TRUE,
  qc_algorithm = "hourly_AB_01",
  min_count = 20
)
```

Arguments

pat	PurpleAir Timeseries *pat* object.
showPlot	Logical specifying whether to generate a model fit plot.
size	Size of points.
pa_color	Color of hourly points.
pwfsl_color	Color of hourly points.
alpha	Opacity of points.
lr_shape	Symbol to use for linear model points.
lr_color	Color of linear model plot points.
lr_lwd	Width of linear regression line.
lr_lcolor	Color of linear regression line.
lr_lalpha	Opacity of linear regression line.
ts_shape	Symbol to use for time series points.
xylim	Vector of (lo,hi) limits used as limits on the correlation plot axes – useful for zooming in.
channel	Data channel to use for PM2.5 – one of "a", "b or "ab".
replaceOutliers	Logical specifying whether or not to replace outliers.
qc_algorithm	Named QC algorithm to apply to hourly aggregation stats.
min_count	Aggregation bins with fewer than ‘min_count‘ measurements will be marked as ‘NA‘.

Value

A linear model, fitting the ‘pat‘ PurpleAir readings to the closest PWFSL monitor readings.

Examples

```
library(AirSensor)

pat_externalFit(example_pat)
```

pat_extractDataFrame *Extract dataframes from* pat *objects*

Description

These functions are convenient wrappers for extracting the dataframes that comprise a *pat* object. These functions are designed to be useful when manipulating data in a pipeline chain using `%>%`.

Below is a table showing equivalent operations for each function.

Function	**Equivalent Operation**
`pat_extractData(pat)`	`pat[["data"]]`
`pat_extractMeta(pat)`	`pat[["meta"]]`

Usage

```
pat_extractData(pat)

pat_extractMeta(pat)
```

Arguments

`pat`	*pat* object to extract dataframe from.

Value

A dataframe from the given *pat* object

pat_filter *General purpose data filtering for PurpleAir Timeseries objects*

Description

A generalized data filter for *pat* objects to choose rows/cases where conditions are true. Multiple conditions are combined with `&` or separated by a comma. Only rows where the condition evaluates to TRUE are kept.Rows where the condition evaluates to `NA` are dropped.

Usage

```
pat_filter(pat, ...)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`...`	Logical predicates defined in terms of the variables in the `pat$data`.

Value

A subset of the incoming `pat`.

See Also

pat_filterDate

pat_filterDatetime

Examples

```
library(AirSensor)

unhealthy <- pat_filter(example_pat, pm25_A > 55.5, pm25_B > 55.5)
head(unhealthy$data)
```

`pat_filterDate` *Date filtering for PurpleAir Timeseries objects*

Description

Subsets a PurpleAir Timeseries object by date. This function always filters to day-boundaries. For sub-day filtering, use `pat_filterDatetime()`.

Dates can be anything that is understood by `lubridate::ymd()` including either of the following recommended formats:

- `"YYYYmmdd"`
- `"YYYY-mm-dd"`

Usage

```
pat_filterDate(
  pat = NULL,
  startdate = NULL,
  enddate = NULL,
  days = NULL,
  weeks = NULL,
  timezone = NULL
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`startdate`	Desired start datetime (ISO 8601).
`enddate`	Desired end datetime (ISO 8601).
`days`	Number of days to include in the filterDate interval.
`weeks`	Number of weeks to include in the filterDate interval.
`timezone`	Olson timezone used to interpret dates.

Value

A subset of the given *pat* object.

Note

The returned data will run from the beginning of `startdate` until the **beginning** of `enddate` – *i.e.* no values associated with `enddate` will be returned. The exception being when `enddate` is less than 24 hours after `startdate`. In that case, a single day is returned.

See Also

pat_filter

pat_filterDatetime

Examples

```
library(AirSensor)

example_pat %>%
  pat_filterDate(startdate = 20180808, enddate = 20180815) %>%
  pat_multiPlot()
```

`pat_filterDatetime` *Datetime filtering for PurpleAir Timeseries objects*

Description

Subsets a PurpleAir Timeseries object by datetime. This function allows for sub-day filtering as opposed to `pat_filterDate()` which always filters to day-boundaries.

Datetimes can be anything that is understood by `MazamaCoreUtils::parseDatetime()`. For non-`POSIXct` values, the recommended format is `"YYYY-mm-dd HH:MM:SS"`.

Timezone determination precedence assumes that if you are passing in POSIXct times then you know what you are doing.

1. get timezone from `startdate` if it is `POSIXct`
2. use passed in `timezone`
3. get timezone from `pat`

Usage

```
pat_filterDatetime(
  pat = NULL,
  startdate = NULL,
  enddate = NULL,
  timezone = NULL
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`startdate`	Desired start datetime (ISO 8601) or `POSIXct`.
`enddate`	Desired end datetime (ISO 8601) or `POSIXct`.
`timezone`	Olson timezone used to interpret dates.

Value

A subset of the given *pat* object.

See Also

pat_filter

pat_filterDate

Examples

```
library(AirSensor)

example_pat %>%
  pat_filterDatetime(
    startdate = "2018-08-08 06:00:00",
    enddate = "2018-08-14 18:00:00"
  ) %>%
  pat_multiPlot()
```

`pat_internalFit` *Linear model fitting of channel A and B time series data*

Description

Uses a linear model to fit data from channel B to data from channel A.

A diagnostic plot is produced if `showPlot = TRUE`.

Usage

```
pat_internalFit(
  pat = NULL,
  showPlot = TRUE,
  size = 1,
  a_color = "red",
  b_color = "blue",
  alpha = 0.25,
  lr_shape = 15,
  lr_color = "black",
  lr_lwd = 1.5,
  lr_lcolor = "tomato",
  lr_lalpha = 0.45,
  ts_shape = 1,
  xylim = NULL
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`showPlot`	Logical specifying whether to generate a model fit plot.
`size`	Size of points.
`a_color`	Color of time series channel A points.
`b_color`	Color of time series channel B points.
`alpha`	Opacity of points.
`lr_shape`	Symbol to use for linear regression points.
`lr_color`	Color of linear regression points.
`lr_lwd`	Width of linear regression line.
`lr_lcolor`	Color of linear regression line.
`lr_lalpha`	Opacity of linear regression line.
`ts_shape`	Symbol to use for time series points.
`xylim`	Vector of (lo,hi) limits used as limits on the correlation plot axes – useful for zooming in.

Value

A linear model, fitting the `pat` B channel readings to A channel readings.

Examples

```
library(AirSensor)

example_pat %>%
  pat_internalFit()
```

pat_isEmpty *Test for an empty* pat *object*

Description

Convenience function for `nrow(pat$data) == 0`. This makes for more readable code in functions that need to test for this.

Usage

```
pat_isEmpty(pat)
```

Arguments

`pat`	*pat* object

Value

`TRUE` if no data exist in `pat`, `FALSE` otherwise.

Examples

```
pat_isEmpty(example_pat)
```

pat_isPat *Test for correct structure in a* pat *object*

Description

The `pat` is checked for the 'pat' class name and presence of core `meta` and `data` columns.

Core `meta` columns include:

- `ID` – Purple Air ID
- `label` – location label
- `sensorType` – PurpleAir sensor type
- `longitude` – decimal degrees E
- `latitude` – decimal degrees N
- `timezone` – Olson timezone
- `countryCode` – ISO 3166-1 alpha-2
- `stateCode` – ISO 3166-2 alpha-2
- `pwfsl_closestDistance` – distance in meters from an official monitor
- `pwfsl_closestMonitorID` – identifer for the nearest official monitor

The "pwfsl", official, monitors are obtained from the USFS AirFire site using the **PWFSLSmoke** R package.

Core `data` columns include:

- `datetime` – measurement time (UTC)
- `pm25_A` – A channel PM 2.5 concentration (ug/m3)
- `pm25_B` – B channel PM 2.5 concentration (ug/m3)
- `temperature` – temperature (F)
- `humidity` – relative humidity (%)

The "pwfsl", official, monitors are obtained from the USFS AirFire site using the **PWFSLSmoke** R package.

Usage

```
pat_isPat(pat = NULL)
```

Arguments

`pat`	*pat* object

Value

`TRUE` if `pat` has the correct structure, `FALSE` otherwise.

Examples

```
pat_isPat(example_pat)
```

`pat_join` *Join PurpleAir time series data for a single sensor*

Description

Create a merged timeseries using of any number of *pat* objects for a single sensor. If *pat* objects are non-contiguous, the resulting *pat* will have gaps.

Usage

```
pat_join(...)
```

Arguments

`...`	Any number of valid PurpleAir Time series *pat* objects.

Value

A PurpleAir Time series *pat* object.

Note

An error is generated if the incoming *pat* objects have non-identical metadata.

Examples

```
library(AirSensor)

aug01_08 <-
  example_pat %>%
  pat_filterDate(20180801, 20180808)

aug15_22 <-
  example_pat %>%
  pat_filterDate(20180815, 20180822)

pat_join(aug01_08, aug15_22) %>%
  pat_multiPlot(plottype = "pm25")
```

---

pat_load — *Load PurpleAir time series data for a time period*

Description

A pre-generated PurpleAir Timeseries *pat* object will be loaded for the given time interval if available. Data are loaded from the archive set with either `setArchiveBaseUrl()` or `setArchiveBaseDir()` for locally archived files.

Dates can be anything that is understood by `MazamaCoreUtils::parseDatetime()` including any of the following recommended formats:

- `"YYYYmmdd"`
- `"YYYY-mm-dd"`
- `"YYYY-mm-dd HH:MM:SS"`

Usage

```
pat_load(
  id = NULL,
  label = NULL,
  pas = NULL,
  startdate = NULL,
  enddate = NULL,
  timezone = "America/Los_Angeles"
)
```

Arguments

`id`	PurpleAir sensor 'deviceDeploymentID'.
`label`	PurpleAir sensor 'label'.
`pas`	PurpleAir Synoptic *pas* object.
`startdate`	Desired start time (ISO 8601) or `POSIXct`.
`enddate`	Desired end time (ISO 8601) or `POSIXct`.
`timezone`	Timezone used to interpret start and end dates.

Value

A PurpleAir Timeseries *pat* object.

Note

Starting with **AirSensor** version 0.6, archive file names are generated with a unique "device-deployment" identifier by combining a unique location ID with a unique device ID. These "device-deployment" identifiers guarantee that movement of a sensor will result in the creation of a new time series.

Users may request a *pat* object in one of two ways:

1) Pass in `id` with a valid a `deviceDeploymentID`

2) Pass in both `label` and `pas` so that the `deviceDeploymentID` can be looked up.

See Also

pat_loadLatest

pat_loadMonth

pat_createNew

Examples

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

# Reference an older 'pas' before this sensor was dropped
pas <- pas_load(20190901, archival = TRUE)

pat <- pat_load(
  label = "SCNP_20",
  pas = pas,
  startdate = 20190411,
  enddate = 20190521
)

pat_multiPlot(pat)
```

pat_loadLatest *Load PurpleAir time series data for a week*

Description

A pre-generated PurpleAir Timeseries *pat* object will be loaded containing data for the most recent 7- or 45-day interval. Data are loaded from the archive set with either `setArchiveBaseUrl()` or `setArchiveBaseDir()` for locally archived files.

Usage

```
pat_loadLatest(id = NULL, label = NULL, pas = NULL, days = 7)
```

Arguments

`id`	PurpleAir sensor 'deviceDeploymentID'.
`label`	PurpleAir sensor 'label'.
`pas`	PurpleAir Synoptic *pas* object.
`days`	Number of days of data to include (7 or 45).

Value

A PurpleAir Timeseries *pat* object.

Note

Starting with **AirSensor** version 0.6, archive file names are generated with a unique "device-deployment" identifier by combining a unique location ID with a unique device ID. These `deviceDeploymentID` identifiers guarantee that movement of a sensor will result in the creation of a new time series.

Users may request a *pat* object in one of two ways:

1) Pass in `id` with a valid a `deviceDeploymentID`

2) Pass in both `label` and `pas` so that the `deviceDeploymentID` can be looked up.

See Also

pat_load

pat_loadMonth

pat_createNew

Examples

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

pas <- pas_load()
pat <- pat_loadLatest(label = "SCSB_07", pas = pas)
pat_multiPlot(pat)
```

`pat_loadMonth` *Load PurpleAir time series data for a month*

Description

A pre-generated PurpleAir Timeseries *pat* object will be loaded for the month requested with `datestamp` if available. Data are loaded from the archive set with either `setArchiveBaseUrl()` or `setArchiveBaseDir()` for locally archived files.

The `datestamp` must be in the following format:

- `"YYYYmm"`

By default, the current month is loaded.

Usage

```
pat_loadMonth(
  id = NULL,
  label = NULL,
  pas = NULL,
  datestamp = NULL,
  timezone = "America/Los_Angeles"
)
```

Arguments

`id`	PurpleAir sensor 'deviceDeploymentID'.
`label`	PurpleAir sensor 'label'.
`pas`	PurpleAir Synoptic *pas* object.
`datestamp`	Date string in ymd order.
`timezone`	Timezone used to interpret `datestamp`.

Value

A PurpleAir Timeseries *pat* object.

Note

Starting with **AirSensor** version 0.6, archive file names are generated with a unique "device-deployment" identifier by combining a unique location ID with a unique device ID. These "device-deployment" identifiers guarantee that movement of a sensor will result in the creation of a new time series.

Users may request a *pat* object in one of two ways:

1) Pass in `id` with a valid a `deviceDeploymentID`

2) Pass in both `label` and `pas` so that the `deviceDeploymentID` can be looked up.

See Also

pat_load

pat_loadLatest

pat_createNew

Examples

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

# Reference an older 'pas' before this sensor was dropped
pas <- pas_load(20190901, archival = TRUE)

may <- pat_loadMonth(label = "SCNP_20", pas = pas, datestamp = 201905)
pat_multiPlot(may)
```

---

pat_monitorComparison *Comparison of Purple Air and federal monitoring data*

Description

Creates and returns a ggplot object that plots raw *pat* data, hourly aggregated *pat* data and hourly data from the nearest federal monitor from the PWFSL database.

Usage

```
pat_monitorComparison(
  pat = NULL,
  FUN = AirSensor::PurpleAirQC_hourly_AB_01,
  distanceCutoff = 20,
  ylim = NULL,
  replaceOutliers = TRUE,
  timezone = NULL
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`FUN`	Algorithm applied to *pat* object for hourly aggregation and quality control. See `pat_createAirSensor()` for more details.
`distanceCutoff`	Numeric distance (km) cutoff for nearest PWFSL monitor.
`ylim`	Vector of (lo,hi) y-axis limits.
`replaceOutliers`	Logical specifying whether replace outliers in the *pat* object.
`timezone`	Olson timezone used for the time axis. (Defaults to `pat` local time.)

Value

A ggplot object.

Examples

```
library(AirSensor)

pat_monitorComparison(example_pat)
```

`pat_multiPlot` *Display multiple plots on one page*

Description

A plotting function that uses ggplot2 to display multiple ggplot objects in a single pane. Can either be passed individual ggplot objects OR a pat object and a plot type. Typical usage would be to supply `pat` and use the `plottype` argument to quickly display preformatted plots.

Available `plottype` options include:

- `"all"` – pm25_A, pm25_B, temperature, humidity
- `"pm25_a"` – PM2.5 from channel A only
- `"pm25_b"` – PM2.5 from channel B only
- `"pm25"` – PM2.5 from channels A and B in separate plots
- `"pm25_over"` – PM2.5 from channels A and B in the same plot
- `"aux"` – auxiliary data (temperature, humidity)

Usage

```
pat_multiPlot(
  pat = NULL,
  plottype = "all",
  sampleSize = 5000,
  columns = NULL,
  ylim = NULL,
  a_size = 1,
  a_shape = 15,
  a_color = rgb(0.9, 0.25, 0.2),
  b_size = 1,
  b_shape = 15,
  b_color = rgb(0.2, 0.25, 0.9),
  t_size = 1,
  t_shape = 15,
  t_color = "black",
  h_size = 1,
  h_shape = 15,
  h_color = "black",
  alpha = 0.5,
  timezone = NULL
)

pat_multiplot(
  pat = NULL,
  plottype = "all",
  sampleSize = 5000,
  columns = NULL,
  ylim = NULL,
  a_size = 1,
  a_shape = 15,
  a_color = rgb(0.9, 0.25, 0.2),
  b_size = 1,
  b_shape = 15,
  b_color = rgb(0.2, 0.25, 0.9),
  t_size = 1,
  t_shape = 15,
  t_color = "black",
  h_size = 1,
  h_shape = 15,
  h_color = "black",
  alpha = 0.5,
  timezone = NULL
)
```

Arguments

pat	PurpleAir Timeseries *pat* object.

plottype	Quick-reference plot types: "all", "aux", "pm25".
sampleSize	Either an integer or fraction to determine sample size.
columns	Number of columns in the plot layout. Use NULL for defaults.
ylim	Vector of (lo,hi) y-axis limits.
a_size	Size of pm25_A points.
a_shape	Symbol to use for pm25_A points.
a_color	Color of pm25_A points.
b_size	Size of pm25_B points.
b_shape	Symbol to use for pm25_B points.
b_color	Color of pm25_B points.
t_size	Size of temperature points.
t_shape	Symbol to use for temperature points.
t_color	Color of temperature points.
h_size	Size of humidity points.
h_shape	Symbol to use for humidity points.
h_color	Color of humidity points.
alpha	Opacity of points.
timezone	Olson timezone used for the time axis. (Defaults to pat local time.)

Value

A ggplot object.

Note

Additional documentation of the multiplot algorithm is available at cookbook-r.com.

Examples

```
library(AirSensor)

example_pat %>%
  pat_multiPlot(plottype = "pm25", alpha = 0.5)
```

`pat_outliers` *Detect and replace time series outliers*

Description

Outlier detection using a Median Average Deviation "Hampel" filter. This function applies a rolling Hampel filter to find those points that are very far out in the tails of the distribution of values within the window.

The `thresholdMin` level is similar to a sigma value for normally distributed data. The default threshold setting `thresholdMin = 8` identifies points that are extremely unlikely to be part of a normal distribution and therefore very likely to be an outlier. By choosing a relatively large value for ‘thresholdMin“ we make it less likely that we will generate false positives.

The default setting of the window size `windowSize = 15` means that 15 samples from a single channel are used to determine the distribution of values for which a median is calculated. Each PurpleAir channel makes a measurement approximately every 120 seconds so the temporal window is 15 * 120 sec or approximately 30 minutes. This seems like a reasonable period of time over which to evaluate PM2.5 measurements.

Specifying `replace = TRUE` allows you to perform smoothing by replacing outliers with the window median value. Using this technique, you can create an highly smoothed, artificial dataset by setting `thresholdMin = 1` or lower (but always above zero).

Usage

```
pat_outliers(
  pat = NULL,
  windowSize = 15,
  thresholdMin = 8,
  replace = FALSE,
  showPlot = TRUE,
  data_shape = 18,
  data_size = 1,
  data_color = "black",
  data_alpha = 0.5,
  outlier_shape = 8,
  outlier_size = 1,
  outlier_color = "red",
  outlier_alpha = 1
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`windowSize`	Integer window size for outlier detection.
`thresholdMin`	Threshold value for outlier detection.
`replace`	Logical specifying whether replace outliers with the window median value.

`showPlot`	Logical specifying whether to generate outlier detection plots.
`data_shape`	Symbol to use for data points.
`data_size`	Size of data points.
`data_color`	Color of data points.
`data_alpha`	Opacity of data points.
`outlier_shape`	Symbol to use for outlier points.
`outlier_size`	Size of outlier points.
`outlier_color`	Color of outlier points.
`outlier_alpha`	Opacity of outlier points.

Value

A *pat* object with outliers replaced by median values.

Note

Additional documentation on the algorithm is available in `seismicRoll::findOutliers()`.

Examples

```
library(AirSensor)

example_pat %>%
  pat_filterDate(20180801, 20180815) %>%
  pat_outliers(replace = TRUE, showPlot = TRUE)
```

---

pat_qc — *Apply quality control on PurpleAir Timeseries object*

Description

Optionally applies QC thresholds to a *pat* object based on the documented specs of the PurpleAir sensor.

The `pat_load()` function returns raw "engineering" data for a PurpleAir Sensor. The very first level of QC that should always be applied is the removal of out-of-spec values that should never be generated by the sensor components. Out-of-spec values imply an electrical or software problem and can never be considered valid measurements.

Setting a `max_humidity` threshold is less fundamental. There are many cases where PM2.5 readings during periods of high humidity should be called into question which is why this QC option is provided. However, this type of filtering is dependent upon a properly functioning humidity sensor. Humidity filtering is disabled by default because it can result in the invalidation of many potentially valid PM2.5 measurements.

Usage

```
pat_qc(pat = NULL, removeOutOfSpec = TRUE, max_humidity = NULL)
```

Arguments

- `pat` — PurpleAir Timeseries *pat* object
- `removeOutOfSpec` — Logical determining whether measurements that are out of instrument specs should be invalidated.
- `max_humidity` — Maximum humidity threshold above which pm25 measurements are invalidated. Disabled unless explicitly set.

Details

Out of spec thresholds are set so that anything outside of these the given range should represent a value that is not physically possible in an ambient setting on planet Earth.

- `humidity` – [0:100]
- `temperature` – [-40:185]
- `pm25` – [0:2000]

Value

A cleaned up *pat* object.

References

PA-II specs

Examples

```
library(AirSensor)

# Use a sensor with problems
pat <- example_pat_failure_A

# Basic plot shows out-of-spec values for humidity
pat %>% pat_multiPlot(sampleSize = NULL)

# Applying QC removes these records
pat %>% pat_qc() %>% pat_multiPlot(sampleSize = NULL)

# We can also remove PM2.5 data at high humidities
pat %>% pat_qc(max_humidity = 80) %>% pat_multiPlot(sampleSize = NULL)
```

pat_sample *Sample PurpleAir time series data*

Description

A sampling function that accepts PurpleAir timeseries dataframes and reduces them by randomly selecting distinct rows of the users chosen size.

If both `sampleSize` and `sampleFraction` are unspecified, `sampleSize = 5000` will be used.

Usage

```
pat_sample(
  pat = NULL,
  sampleSize = NULL,
  sampleFraction = NULL,
  setSeed = NULL,
  keepOutliers = FALSE
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`sampleSize`	Non-negative integer giving the number of rows to choose.
`sampleFraction`	Fraction of rows to choose.
`setSeed`	Integer that sets random number generation. Can be used to reproduce sampling.
`keepOutliers`	logical specifying a graphics focused sampling algorithm (see Details).

Details

When `keepOutliers = FALSE`, random sampling is used to provide a statistically relevant subsample of the data.

When `keepOutliers = TRUE`, a customized sampling algorithm is used that attempts to create subsets for use in plotting that create plots that are visually identical to plots using all data. This is accomplished by preserving outliers and only sampling data in regions where overplotting is expected.

The process is as follows:

1. find outliers using `seismicRoll::findOutliers()`
2. create a subset consisting of only outliers
3. sample the remaining data
4. merge the outliers and sampled data

Value

A subset of the given *pat* object.

Examples

```
library(AirSensor)

example_pat %>%
  pat_extractData() %>%
  dim()

example_pat %>%
  pat_sample(sampleSize = 1000, setSeed = 1) %>%
  pat_extractData() %>%
  dim()
```

---

pat_scatterPlotMatrix *Draw a matrix of PurpleAir Timeseries data scatter plots*

Description

Creates a multi-panel scatterPlot comparing all variables in the *pat* object. If any variables have no valid data, they are omitted from the plot.

The list of available parameters include:

- `datetime` – measurement time
- `pm25_A` – A channel PM2.5 (ug/m3)
- `pm25_B` – B channel PM2.5 (ug/m3)
- `temperature` – temperature (F)
- `humidity` – humidity (%)

Usage

```
pat_scatterPlotMatrix(
  pat = NULL,
  parameters = c("datetime", "pm25_A", "pm25_B", "temperature", "humidity"),
  sampleSize = 5000,
  sampleFraction = NULL,
  size = 0.5,
  shape = 15,
  color = "black",
  alpha = 0.25
)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.
`parameters`	Vector of parameters to include.

sampleSize	Integer to determine sample size.
sampleFraction	Fractional sample size.
size	Size of points.
shape	Symbol to use for points.
color	Color of points.
alpha	Opacity of points.

Value

Multi-panel ggplot comparing all parameters.

Examples

```
library(AirSensor)

pat <-
  example_pat %>%
  pat_filterDate(20180811,20180818)

# NOTE:  Warnings are generated when the pat contains NA values
pat_scatterPlotMatrix(pat, sampleSize = 1000)
```

`pat_trimDate` *Trim a PurpleAir Timeseries object to full days*

Description

Trims the date range of a *pat* object to local time date boundaries which are *within* the range of data. This has the effect of removing partial-day data records and is useful when calculating full-day statistics.

Usage

```
pat_trimDate(pat = NULL)
```

Arguments

pat	PurpleAir Timeseries *pat* object.

Value

A subset of the given *pat* object.

Examples

```
library(AirSensor)

UTC_week <- pat_filterDate(
  example_pat,
  startdate = 20180808,
  enddate = 20180815,
  timezone = "UTC"
)

pat_multiPlot(UTC_week)

local_week <- pat_trimDate(UTC_week)
pat_multiPlot(local_week)
```

---

pat_upgrade *Upgrade PurpleAir Timeseries*

Description

The `pat` parameter is checked for the latest `pa_timeseries` format and presence of and/or addition of core data columns:

- datetime – A datetime column
- pm25_A – Channel A PM2.5
- pm25_B – Channel B PM2.5
- temperature – Tmperature in Faarehniet
- humidity – Relative Humidity
- pressure – Pressure in hectopascals (hPa)
- pm1_atm_A – Channel A PM1.0
- pm25_atm_A – Channel A PM2.5
- pm10_atm_A – Channel A PM10.0
- pm1_atm_B – Channel B PM1.0
- pm25_atm_B – Channel B PM2.5
- pm10_atm_B – Channel B PM10.0
- uptime – Sensor uptime in seconds
- rssi – Sensor WiFi signal strength in dBm
- memory – Memory Usage
- adc0 – Voltage
- bsec_iaq – ?
- datetime_A – Record datetime of Channel B
- datetime_B – Record datetime of Channel A

Usage

```
pat_upgrade(pat = NULL, verbose = TRUE)
```

Arguments

pat	PurpleAir Timeseries *pat* object.
verbose	(logical) Display messages.

Value

An upgraded `pa_timeseries` object.

PurpleAirQC_hourly_AB_00

Apply hourly aggregation QC using "AB_OO" algorithm

Description

Creates a `pm25` timeseries by averaging aggregated data from the A and B channels and applying the following QC logic:

1. Create pm25 by averaging the A and B channel aggregation means
2. Invalidate data where: (min_count < 20)
3. No further QC

Usage

```
PurpleAirQC_hourly_AB_00(pat = NULL, min_count = 20, returnAllColumns = FALSE)
```

Arguments

pat	A PurpleAir timeseries object.
min_count	Aggregation bins with fewer than `min_count` measurements will be marked as `NA`.
returnAllColumns	Logical specifying whether to return all columns of statistical data generated for QC algorithm or just the final `pm25` result.

Value

Data frame with columns `datetime` and `pm25`.

Note

Purple Air II sensors reporting after the June, 2019 firmware upgrade report data every 2 minutes or 30 measurements per hour. The default setting of `min_count = 20` is equivalent to a required data recovery rate of 67

Examples

```
library(AirSensor)

df_00 <-
  example_pat %>%
  pat_qc() %>%
  PurpleAirQC_hourly_AB_00()

names(df_00)

plot(df_00, pch = 16, cex = 0.8, col = "red")
```

`PurpleAirQC_hourly_AB_01`   *Apply hourly aggregation QC using "AB_O1" algorithm*

Description

Creates a `pm25` timeseries by averaging aggregated data from the A and B channels and applying the following QC logic:

1. Create pm25 by averaging the A and B channel aggregation means
2. Invalidate data where: (min_count < 20)
3. Invalidate data where: (p-value < 1e-4) & (mean_diff > 10)
4. Invalidate data where: (pm25 < 100) & (mean_diff > 20)

Usage

```
PurpleAirQC_hourly_AB_01(pat = NULL, min_count = 20, returnAllColumns = FALSE)
```

Arguments

`pat`	A PurpleAir timeseries object.
`min_count`	Aggregation bins with fewer than `min_count` measurements will be marked as `NA`.
`returnAllColumns`	Logical specifying whether to return all columns of statistical data generated for QC algorithm or just the final `pm25` result.

Value

Data frame with columns `datetime` and `pm25`.

Note

Purple Air II sensors reporting after the June, 2019 firmware upgrade report data every 2 minutes or 30 measurements per hour. The default setting of `min_count = 20` is equivalent to a required data recovery rate of 67

Examples

```
library(AirSensor)

df_00 <-
  example_pat_failure_A %>%
  pat_qc() %>%
  PurpleAirQC_hourly_AB_00()

df_01 <-
  example_pat_failure_A %>%
  pat_qc() %>%
  PurpleAirQC_hourly_AB_01()

df_02 <-
  example_pat_failure_A %>%
  pat_qc() %>%
  PurpleAirQC_hourly_AB_02()

layout(matrix(seq(2)))

plot(df_00, pch = 16, cex = 0.8, col = "red")
points(df_01, pch = 16, cex = 0.8, col = "black")
title("example_pat_failure_A -- PurpleAirQC_hourly_AB_01")

plot(df_00, pch = 16, cex = 0.8, col = "red")
points(df_02, pch = 16, cex = 0.8, col = "black")
title("example_pat_failure_A -- PurpleAirQC_hourly_AB_02")

layout(1)
```

PurpleAirQC_hourly_AB_02 *Apply hourly aggregation QC using "AB_O2" algorithm*

Description

Creates a pm25 timeseries by averaging aggregated data from the A and B channels and applying the following QC logic:

1. Create pm25 by averaging the A and B channel aggregation means
2. Invalidate data where: (min_count < 20)

3. Invalidate data where: (A/B hourly MAD > 3)
4. Invalidate data where: (A/B hourly pct_diff > 0.5)

MAD = "Median Absolute Deviation"

Usage

```
PurpleAirQC_hourly_AB_02(pat = NULL, min_count = 20, returnAllColumns = FALSE)
```

Arguments

pat	A PurpleAir timeseries object.
min_count	Aggregation bins with fewer than min_count measurements will be marked as NA.
returnAllColumns	Logical specifying whether to return all columns of statistical data generated for QC algorithm or just the final pm25 result.

Value

Data frame with columns datetime and pm25.

Note

Purple Air II sensors reporting after the June, 2019 firmware upgrade report data every 2 minutes or 30 measurements per hour. The default setting of min_count = 20 is equivalent to a required data recovery rate of 67%.

Examples

```
library(AirSensor)

df_00 <-
  example_pat_failure_A %>%
  pat_qc() %>%
  PurpleAirQC_hourly_AB_00()

df_01 <-
  example_pat_failure_A %>%
  pat_qc() %>%
  PurpleAirQC_hourly_AB_01()

df_02 <-
  example_pat_failure_A %>%
  pat_qc() %>%
  PurpleAirQC_hourly_AB_02()

layout(matrix(seq(2)))

plot(df_00, pch = 16, cex = 0.8, col = "red")
```

```
points(df_01, pch = 16, cex = 0.8, col = "black")
title("example_pat_failure_A -- PurpleAirQC_hourly_AB_01")

plot(df_00, pch = 16, cex = 0.8, col = "red")
points(df_02, pch = 16, cex = 0.8, col = "black")
title("example_pat_failure_A -- PurpleAirQC_hourly_AB_02")

layout(1)
```

`PurpleAirQC_hourly_AB_03` *Apply hourly aggregation QC using "AB_O4" algorithm*

Description

Creates a *pm25* timeseries by averaging aggregated data from the A and B channels and applying the following QC logic:

1. Create pm25 by averaging the A and B channel aggregation means
2. Invalidate data where: (min_count < 20)
3. Invalidate data where: (A/B hourly difference > 5 AND A/B hourly percent difference > 70%)
4. Invalidate data where: (A/B hourly data recovery < 90%)

Usage

```
PurpleAirQC_hourly_AB_03(pat = NULL, min_count = 20, returnAllColumns = FALSE)
```

Arguments

`pat`	A PurpleAir timeseries object.
`min_count`	Aggregation bins with fewer than `min_count` measurements will be marked as NA.
`returnAllColumns`	Logical specifying whether to return all columns of statistical data generated for QC algorithm or just the final `pm25` result.

Value

Data frame with columns `datetime` and `pm25`.

Note

Purple Air II sensors reporting after the June, 2019 firmware upgrade report data every 2 minutes or 30 measurements per hour. The default setting of `min_count = 20` is equivalent to a required data recovery rate of 67%.

Examples

```
library(AirSensor)

df_00 <-
  example_pat_failure_A %>%
  pat_qc() %>%
  PurpleAirQC_hourly_AB_00()

df_01 <-
  example_pat_failure_A %>%
  pat_qc() %>%
  PurpleAirQC_hourly_AB_01()

df_03 <-
  example_pat_failure_A %>%
  pat_qc() %>%
  PurpleAirQC_hourly_AB_03()

layout(matrix(seq(2)))

plot(df_00, pch = 16, cex = 0.8, col = "red")
points(df_01, pch = 16, cex = 0.8, col = "black")
title("example_pat_failure_A -- PurpleAirQC_hourly_AB_01")

plot(df_00, pch = 16, cex = 0.8, col = "red")
points(df_03, pch = 16, cex = 0.8, col = "black")
title("example_pat_failure_A -- PurpleAirQC_hourly_AB_03")

layout(1)
```

`PurpleAirSoH_dailyABFit` *Daily linear model fit values*

Description

This function calculates daily linear model values between the `pm25_A` and `pm25_B` channels. A daily r-squared value is returned in addition to the coefficients of the linear fit (slope and intercept)

Usage

```
PurpleAirSoH_dailyABFit(pat = NULL)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.

Examples

```
library(AirSensor)

tbl <-
  example_pat_failure_A %>%
  PurpleAirSoH_dailyABFit()

names(tbl)

timeseriesTbl_multiPlot(
  tbl,
  parameters = c("pm25_A_pm25_B_rsquared", "pm25_A_pm25_B_slope"),
  ylim = c(-1,1)
)
```

PurpleAirSoH_dailyABtTest

Daily t-test

Description

This function calculates a t-test between the `pm25_A`, `pm25_B`. A t-statistic and a p-value will be returned for each day. All returned values are expected to hover near 0 for a properly functioning sensor.

Usage

```
PurpleAirSoH_dailyABtTest(pat = NULL)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.

Examples

```
library(AirSensor)

tbl <-
  example_pat_failure_B %>%
  PurpleAirSoH_dailyABtTest()

timeseriesTbl_multiPlot(tbl)
```

PurpleAirSoH_dailyOtherFit

Daily fit values

Description

This function calculates a daily linear model between the `pm25_A`, `pm25_B`, `humidity`, and `temperature` channels. One r-squared value for each channel pair except `pm25_A`, `pm25_B`, and `humidity`, `temperature` will be returned for each day. All returned values are expected to hover near 0 for a properly functioning sensor.

Usage

```
PurpleAirSoH_dailyOtherFit(pat = NULL)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.

Examples

```
library(AirSensor)

tbl <-
  example_pat_failure_A %>%
  PurpleAirSoH_dailyOtherFit()

timeseriesTbl_multiPlot(
  tbl,
  ncol = 2,
  ylim = c(0,1)
)

timeseriesTbl_multiPlot(
  tbl,
  autoRange = TRUE
)
```

PurpleAirSoH_dailyPctDC

Daily DC Signal percentage

Description

This function calculates the daily percentage of DC signal recorded by the `pm25_A`, `pm25_B`, `humidity`, and `temperature` channels. The data are flagged as DC signal when the standard deviation of an hour of data from each channel equals zero. The number of hours with a DC signal are summed over the day and a daily DC percentage for each channel is returned.

Usage

```
PurpleAirSoH_dailyPctDC(pat = NULL)
```

Arguments

`pat` PurpleAir Timeseries *pat* object.

Examples

```
library(AirSensor)

tbl <-
  example_pat_failure_A %>%
  PurpleAirSoH_dailyPctDC()

timeseriesTbl_multiPlot(tbl, ylim = c(0,100))
```

`PurpleAirSoH_dailyPctReporting`
Daily reporting percentage

Description

The number of sensor readings recorded per hour are summed over the course of a calendar day. This is then divided by the number of samples the sensor would record in an ideal day (`24 * 3600 / samplingInterval`) to return a percentage of each day that the sensor is reporting data.

Usage

```
PurpleAirSoH_dailyPctReporting(pat = NULL, samplingInterval = 120)
```

Arguments

`pat` PurpleAir Timeseries *pat* object.

`samplingInterval`
The number of seconds between samples when the sensor is operating optimally.

Note

Purple Air II sensors reporting after the June, 2019 firmware upgrade report data every 120 seconds. Prior to the upgrade, data were reported every 80 seconds.

Examples

```
library(AirSensor)

tbl <-
  example_pat %>%
  PurpleAirSoH_dailyPctReporting(80)

timeseriesTbl_multiPlot(tbl, ylim = c(0,101))
```

`PurpleAirSoH_dailyPctValid` *Daily valid percentage*

Description

The number of valid (*i.e.*, not NA or out-of-spec) sensor measurements are summed over the course of a calendar day, then divided by the total number of measurements the sensor actually recorded during that day (including NA and out-of-spec values) to return a percentage of the total recorded measurements that are considered plausible.

Usage

```
PurpleAirSoH_dailyPctValid(pat = NULL)
```

Arguments

`pat`	PurpleAir Timeseries *pat* object.

Examples

```
library(AirSensor)

tbl <-
  example_pat_failure_B %>%
  PurpleAirSoH_dailyPctValid()

timeseriesTbl_multiPlot(tbl, ylim = c(0,100))
```

`PurpleAirSoH_dailyToIndex_00` *Daily state of health index*

Description

This function calculates a multi-metric index based on the data in `SoH` dataframe passed in. A tibble is returned containing a state of health index for each day. The returned tibble contains columns: `datetime`, `index`, and `index_bin`.

The `index` column contains a value normalized between 0 and 1 where 0 represents low confidence in the sensor data and 1 represents high confidence. The `index_bin` is one of 1, 2, or 3 and represents poor, fair, and good data respectively.

The `index` is calculated in the following manner:

1. If the A or B channel percent reporting is < `minPctReporting`, `index = 0`
2. Otherwise, `index = pm25_A_pm25_B_rsquared`

The `breaks` are used to convert `index` into the `indenx_bin` poor-fair-good values.

Usage

```
PurpleAirSoH_dailyToIndex_00(
  SoH = NULL,
  minPctReporting = 50,
  breaks = c(0, 0.2, 0.8, 1)
)
```

Arguments

`SoH`	PurpleAir daily State-of-Health dataframe.
`minPctReporting`	Percent reporting threshold for A and B channels.
`breaks`	Breaks used to convert index values into index bins.

Examples

```
library(AirSensor)

tbl <-
  example_pat_failure_A %>%
  pat_dailySoH() %>%
  PurpleAirSoH_dailyToIndex_00()

head(tbl)
```

pwfsl_load — *Get PWFSLSmoke monitoring data*

Description

Loads recent PM2.5 monitoring data from the US Forest Service Pacific Wildland Fire Sciences Lab. This function performs the same data loading step as `pwfsl_loadLatest()`, but has a shorter name for consistency with other data loading functions in the `AirSensor` package. By default, this function loads data from all 50 states for the past 10 days.

By default, this function is a wrapper around `PWFSLSmoke::monitor_loadLatest`. But it can also be used as a wrapper around `PWFSLSmoke::monitor_load` by passing in arguments.

If you pass in arguments, *e.g.* `starttime` and `endtime`, `PWFSLSmoke::monitor_load()` will be invoked. Otherwise, `PWFSLSmoke::monitor_loadLatest()` will be invoked.

Usage

```
pwfsl_load(...)
```

Arguments

`...`	Arguments passed on to `PWFSLSmoke::monitor_load()`.

Value

List with `meta` and `data` elements, a *ws_monitor* object.

Examples

```
library(AirSensor)

pwfsl <- pwfsl_load()
dim(pwfsl$meta)
dim(pwfsl$data)
```

pwfsl_loadLatest — *Get PWFSLSmoke monitoring data*

Description

Loads recent PM2.5 monitoring data from the US Forest Service Pacific Wildland Fire Sciences Lab. This function performs the same data loadingstep as `pwfsl_load()`, but has a longer name for consistency with other data loading functions in the `PWFSLSmoke` package. By default, this function loads data from all 50 states for the past 10 days.

This function is a wrapper around `PWFSLSmoke::monitor_loadLatest`.

Data for the most recent 45 days can be downloaded using `PWFSLSmoke::monitor_loadDaily()`. See the `PWFSLSmoke package` for additional data loading functions.

Usage

```
pwfsl_loadLatest(...)
```

Arguments

`...`	Arguments passed on to `PWFSLSmoke::monitor_load()`.

Value

List with `meta` and `data` elements, a *ws_monitor* object.

Examples

```
library(AirSensor)

pwfsl <- pwfsl_loadLatest()
dim(pwfsl$meta)
dim(pwfsl$data)
```

`scatterPlot` *Matrix scatter plot variables in a data frame*

Description

Creates a multi-panel scatterPlot comparing all variables in the data frame object. If any variables have not valid data, they are omitted from the plot.

Usage

```
scatterPlot(
  data,
  parameters = NULL,
  sampleSize = 5000,
  sampleFraction = NULL,
  shape = 18,
```

```
  size = 1.5,
  color = "black",
  alpha = 0.5
)
```

Arguments

`data`	data frame
`parameters`	the columns of the data frame to plot
`sampleSize`	the integer sample number of rows
`sampleFraction`	the fractional sample of rows
`shape`	symbol to use for points
`size`	size of points
`color`	color of points
`alpha`	opacity of points

`sensor_calendarPlot` *Plot time series values in conventional calendar format*

Description

Function for plotting PM2.5 concentration in a calendar format. This function wraps the **openair** `calendarPlot()` function.

Usage

```
sensor_calendarPlot(
  sensor = NULL,
  colors = NULL,
  breaks = NULL,
  labels = NULL,
  limits = c(0, 100),
  title = NULL,
  data.thresh = 50
)
```

Arguments

`sensor`	An 'airsensor' object
`colors`	Colours to be used for plotting. Options include "aqi", "scaqmd", "default", "increment", "heat", "jet" and **RColorBrewer** colours — see the **openair** `openColours` function for more details. For user defined the user can supply a list of colour names recognised by R (type `colours()` to see the full list). An example would be cols = `c("yellow","green","blue")`

`breaks` If a categorical scale is required then these breaks will be used. For example, `breaks = c(0,50,100,1000)`. In this case "good" corresponds to values berween 0 and 50 and so on. Users should set the maximum value of breaks to exceed the maximum data value to ensure it is within the maximum final range e.g. 100–1000 in this case.

`labels` If a categorical scale is required then these labels will be used. Note there is one less label than break. For example, `labels = c("good","bad","very bad")`. breaks must also be supplied if labels are given.

`limits` Use this option to manually set the colour scale limits. This is useful in the case when there is a need for two or more plots and a consistent scale is needed on each. Set the limits to cover the maximimum range of the data for all plots of interest. For example, if one plot had data covering 0–60 and another 0–100, then set `limits = c(0,100)`. Note that data will be ignored if outside the limits range.

`title` Optional title. If `NULL`, a default title will be constructed.

`data.thresh` Data capture threshold passed to `openair::timeAverage()`. For example, `data.thresh = 75` means that at least 75 be available in a day for the value to be calculate, else the data is removed.

Details

Data are trimmed to the local-time year or month boundaries as appropriate.

Two special options are provided to specify a set of `colors`, `breaks` and `labels`.

Using `colors = "aqi"` will use US EPA Air Quality Index colors and breaks defined by `breaks <-c(-Inf,12,35.5,55.5,150.5,250.5,Inf)`.

Using `colors = "scaqmd"` will use a custom set of colors and breaks defined by `breaks <-c(-Inf,12,35,55,75,Inf)`.

Value

A plot and an object of class "openair".

Note

Daily averages are calculated using LST (Local Standard Time) day boundaries as specified by the US EPA. LST assumes that standard time applies all year round and guarantees that every day has 24 hours – no "spring forward" or "fall back". Because of this, LST daily averages calculated during months where daylight savings time is in effect will differ very slightly from daily averages calculated using local "clock time".

References

EPA AQS Data Dictionary

See Also

https://davidcarslaw.github.io/openair/reference/calendarPlot.html

Examples

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

# Monthly plot
sensor <-
  sensor_loadMonth("scaqmd", 202007) %>%
  sensor_filterMeta(label == "SCSC_33")

sensor_calendarPlot(sensor)

# Annual plot
sensor <-
  sensor_loadYear("scaqmd", 2020) %>%
  sensor_filterMeta(label == "SCSC_33")

sensor_calendarPlot(sensor)

# SCAQMD colors
sensor_calendarPlot(sensor, "scaqmd")

# Custom continuous color palette from RColorBrewer
sensor_calendarPlot(
  sensor,
  colors = "BuPu",
  title = "2020 Purple Scale",
  limits = range(sensor$data[,-1], na.rm = TRUE) # don't use data$datetime
)


# Custom categorical colors
sensor_calendarPlot(
  sensor,
  colors = c("springgreen2", "gold", "tomato3"),
  breaks = c(-Inf, 25, 50, Inf),
  labels = c("Good", "Fair", "Poor"),
  title = "2020 -- Air Quality Stoplight"
)
```

---

sensor_extractDataFrame

Extract dataframes from airsensor *objects*

Description

These functions are convenient wrappers for extracting the dataframes that comprise a *airsensor* object. These functions are designed to be useful when manipulating data in a pipeline chain using `%>%`.

Below is a table showing equivalent operations for each function.

Function	**Equivalent Operation**
`sensor_extractData(sensor)`	`sensor[["data"]]`
`sensor_extractMeta(sensor)`	`sensor[["meta"]]`

Usage

```
sensor_extractData(sensor)

sensor_extractMeta(sensor)
```

Arguments

`sensor`	*sensor* object to extract dataframe from.

Value

A dataframe from the given *sensor* object

`sensor_filter` *Data filtering for AirSensor objects*

Description

A generalized data filter for *sensor* objects to choose rows/cases where conditions are true. Multiple conditions are combined with `&` or seperated by a comma. Only rows where the condition evaluates to TRUE are kept.Rows where the condition evaluates to `NA` are dropped.

Usage

```
sensor_filter(sensor = NULL, ...)
```

Arguments

`sensor`	An AirSensor object.
`...`	Logical predicates defined in terms of the variables in `sensor$data`.

Value

A subset of the incoming *sensor*.

Note

Filtering predicates are applied to the `data` dataframe within the *sensor* object.

See Also

sensor_filterDate

sensor_filterMeta

Examples

```
library(AirSensor)

twenties <- sensor_filter(example_sensor,
                          example_sensor$data$`392a12065eb9940d_5192` <  30,
                          example_sensor$data$`392a12065eb9940d_5192` >= 20)
head(twenties$data)
```

`sensor_filterDate` *Date filtering for AirSensor objects*

Description

Subsets an AirSensor object by date.

Dates can be anything that is understood by `lubrdiate::ymd()` including either of the following recommended formats:

- "YYYYmmdd"
- "YYYY-mm-dd"

Usage

```
sensor_filterDate(
  sensor = NULL,
  startdate = NULL,
  enddate = NULL,
  days = NULL,
  weeks = NULL,
  timezone = "America/Los_Angeles"
)
```

Arguments

sensor	An AirSensor object.
startdate	Desired start datetime (ISO 8601).
enddate	Desired end datetime (ISO 8601).
days	Number of days to include in the filterDate interval.
weeks	Number of weeks to include in the filterDate interval.
timezone	Olson timezone used to interpret dates.

Value

A subset of the given *sensor* object.

See Also

sensor_filter

sensor_filterMeta

Examples

```
library(AirSensor)

example_sensor %>%
  sensor_extractData() %>%
  dplyr::pull("datetime") %>%
  range()

example_sensor %>%
  sensor_filterDate(
    startdate = "2018-08-21",
    enddate = "2018-08-28",
    timezone = "UTC"
  ) %>%
  sensor_extractData() %>%
  dplyr::pull("datetime") %>%
  range()
```

---

sensor_filterDatetime *Datetime filtering for AirSensor objects*

Description

Subsets an AirSensor object by datetime. This function allows for sub-day filtering as opposed to `sensor_filterDate()` which always filters to day-boundaries. Filtering will be performed with >= `startdate` and < `enddate` so that the `startdate` timestep will be included in the output but the `enddate` will not.

Datetimes can be anything that is understood by `MazamaCoreUtils::parseDatetime()`. For non-`POSIXct` values, the recommended format is `"YYYY-mm-dd HH:MM:SS"`.

Timezone determination precedence assumes that if you are passing in POSIXct times then you know what you are doing.

1. get timezone from `startdate` if it is `POSIXct`
2. use passed in `timezone`
3. get timezone from `sensor`

Usage

```
sensor_filterDatetime(
  sensor = NULL,
  startdate = NULL,
  enddate = NULL,
  timezone = NULL
)
```

Arguments

`sensor`	An AirSensor object.
`startdate`	Desired start datetime (ISO 8601).
`enddate`	Desired end datetime (ISO 8601).
`timezone`	Olson timezone used to interpret dates.

Value

A subset of the given *sensor* object.

See Also

sensor_filter

sensor_filterDate

Examples

```
library(AirSensor)

example_sensor %>%
  sensor_extractData() %>%
  dplyr::pull("datetime") %>%
  range()
```

```
example_sensor %>%
  sensor_filterDatetime(
    startdate = "2018-08-21 06:00:00",
    enddate = "2018-08-28 18:00:00",
    timezone = "UTC"
  ) %>%
  sensor_extractData() %>%
  dplyr::pull("datetime") %>%
  range()
```

---

sensor_filterMeta *Metadata filtering for AirSensor objects*

Description

A generalized data filter for *sensor* objects to choose rows/cases where conditions are true. Multiple conditions are combined with `&` or seperated by a comma. Only rows where the condition evaluates to TRUE are kept.Rows where the condition evaluates to `NA` are dropped.

Usage

```
sensor_filterMeta(sensor = NULL, ...)
```

Arguments

`sensor`	An AirSensor object.
`...`	Logical predicates defined in terms of the variables in `sensor$meta`.

Value

A subset of the incoming *sensor*.

Note

Filtering predicates are applied to the `meta` dataframe within the *sensor* object.

See Also

sensor_filter

sensor_filterDate

Examples

```
library(AirSensor)

example_sensor_scaqmd %>%
  sensor_extractMeta() %>%
  dplyr::pull("communityRegion") %>%
  sort () %>%
  unique()

example_sensor_scaqmd %>%
  sensor_filterMeta(communityRegion == "Imperial Valley") %>%
  sensor_extractMeta() %>%
  dplyr::pull("communityRegion") %>%
  sort () %>%
  unique()
```

`sensor_isEmpty` *Test for an empty* sensor *object*

Description

Convenience function for `nrow(sensor$meta) == 0`. This makes for more readable code in functions that need to test for this.

Usage

```
sensor_isEmpty(sensor)
```

Arguments

`sensor`	*sensor* object

Value

`TRUE` if no data exist in `sensor`, `FALSE` otherwise.

Examples

```
example_sensor <- pat_createAirSensor(example_pat)
sensor_isEmpty(example_sensor)
```

`sensor_isSensor` *Test for correct structure in a* sensor *object*

Description

The `sensor` is checked for the 'sensor' class name and presence of core metadata columns:

- ID – Purple Air ID
- label – location label
- sensorType – PurpleAir sensor type
- longitude – decimal degrees E
- latitude – decimal degrees N
- timezone – Olson timezone
- countryCode – ISO 3166-1 alpha-2
- stateCode – ISO 3166-2 alpha-2
- pwfsl_closestDistance – distance in meters from an official monitor
- pwfsl_closestMonitorID – identifer for the nearest official monitor

The "pwfsl", official, monitors are obtained from the USFS AirFire site using the **PWFSLSmoke** R package.

Usage

```
sensor_isSensor(sensor = NULL)
```

Arguments

`sensor`	*sensor* object

Value

`TRUE` if `sensor` has the correct structure, `FALSE` otherwise.

Examples

```
example_sensor <- pat_createAirSensor(example_pat)
sensor_isSensor(example_sensor)
```

sensor_join *Join airsensor objects from different time periods*

Description

AirSensor objects are "joined end-to-end" so that time ranges are extended for all sensors that appear in either `sensor1` and `sensor2`.

Only two `airsensor` objects at a time may be joined.

Usage

```
sensor_join(sensor1 = NULL, sensor2 = NULL)
```

Arguments

`sensor1`	An AirSensor object.
`sensor2`	An AirSensor object.

Value

An *airsensor* object containing all data from both incoming objects.

Examples

```
library(AirSensor)
setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

jan <- sensor_loadMonth("scaqmd", 202001)
feb <- sensor_loadMonth("scaqmd", 202002)
mar <- sensor_loadMonth("scaqmd", 202003)
apr <- sensor_loadMonth("scaqmd", 202004)

feb_mar <- sensor_join(feb, mar)
PWFSLSmoke::monitor_timeseriesPlot(feb_mar, style = 'gnats')

# Gaps in the time axis are filled with NA
feb_apr <- sensor_join(feb, apr)
PWFSLSmoke::monitor_timeseriesPlot(feb_apr, style = 'gnats')
```

sensor_load   *Load hourly-aggregated PurpleAir data*

Description

A pre-generated `airsensor` object will be loaded for the given time interval. Archived data for SCAQMD sensors go back to January, 2018.

Dates can be anything that is understood by `lubrdiate::parse_date_time()` including either of the following recommended formats:

- `"YYYYmmdd"`
- `"YYYY-mm-dd"`

By default, the current week is loaded.

Usage

```
sensor_load(
  collection = "scaqmd",
  startdate = NULL,
  enddate = NULL,
  days = 7,
  timezone = "America/Los_Angeles"
)
```

Arguments

`collection`	Name associated with the collection.
`startdate`	Desired start datetime (ISO 8601).
`enddate`	Desired end datetime (ISO 8601).
`days`	Number of days of data to include (7 or 45).
`timezone`	Timezone used to interpret start and end dates.

Value

An object of class "airsensor".

See Also

sensor_loadMonth

sensor_loadYear

Examples

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

sensor_load("scaqmd", 20200411, 20200521) %>%
  PWFSLSmoke::monitor_timeseriesPlot(style = 'gnats')
```

---

sensor_loadLatest *Load hourly-aggregated PurpleAir data for a week*

Description

A pre-generated `airsensor` object will be loaded containing data for the most recent 7 or 45-day interval.

Each `airsensor` object contains data from a named collection of PurpleAir sensors.

Usage

```
sensor_loadLatest(collection = "scaqmd", days = 7)
```

Arguments

collection	Name associated with the collection.
days	Number of days of data to include (7 or 45).

Value

An object of class "pa_timeseries".

See Also

sensor_load

sensor_loadMonth

pat_createAirSensor

Examples

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

sensor_loadLatest("scaqmd") %>%
  PWFSLSmoke::monitor_timeseriesPlot(style = 'gnats')
```

sensor_loadMonth — *Load hourly-aggregated PurpleAir data for a month*

Description

A pre-generated `airsensor` object will be loaded for the given month. Archived data for SCAQMD sensors go back to January, 2018.

The `datestamp` can must be in the following format:

- `"YYYYmm"`

By default, the current month is loaded.

Each `airsensor` object contains data from a named collection of PurpleAir sensors.

Usage

```
sensor_loadMonth(
  collection = "scaqmd",
  datestamp = NULL,
  timezone = "America/Los_Angeles"
)
```

Arguments

`collection`	Name associated with the collection.
`datestamp`	A date string in ymd order.
`timezone`	Timezone used to interpret `datestamp`.

Value

An object of class "pa_timeseries".

See Also

pat_createNew

Examples

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

sensor_loadMonth("scaqmd", 202005) %>%
  PWFSLSmoke::monitor_timeseriesPlot(style = 'gnats')
```

sensor_loadYear *Load hourly-aggregated PurpleAir data for a month*

Description

A pre-generated `airsensor` object will be loaded for the given month. Archived data for SCAQMD sensors go back to January, 2018.

The `datestamp` can must be in the following format:

- "YYYYmm"

By default, the current month is loaded.

Each `airsensor` object contains data from a named collection of PurpleAir sensors.

Usage

```
sensor_loadYear(
  collection = "scaqmd",
  datestamp = NULL,
  timezone = "America/Los_Angeles"
)
```

Arguments

`collection`	Name associated with the collection.
`datestamp`	A date string in ymd order.
`timezone`	Timezone used to interpret `datestamp`.

Value

An object of class "pa_timeseries".

See Also

pat_createNew

sensor_polarPlot *Plot bivariate polar plots with guassian smoothing*

Description

Function for plotting PM2.5 concentration in polar coordinates showing concentration by wind speed and direction. This function wraps the **openair** `polarPlot()` function.

Usage

```
sensor_polarPlot(
  sensor = NULL,
  windData = NULL,
  statistic = "mean",
  resolution = "fine",
  colors = "default",
  alpha = 1,
  angleScale = 315,
  normalize = FALSE,
  key = TRUE,
  keyPosition = "right",
  ws_spread = 15,
  wd_spread = 4,
  verbose = TRUE
)
```

Arguments

`sensor`	an 'airsensor' object
`windData`	a dataframe containing columns "date", "ws", and "wd".
`statistic`	The statistic that should be applied to each wind speed/direction bin. Because of the smoothing involved, the colour scale for some of these statistics is only to provide an indication of overall pattern and should not be interpreted in concentration units e.g. for statistic = "weighted.mean" where the bin mean is multiplied by the bin frequency and divided by the total frequency. In many cases using polarFreq will be better. Setting statistic = "weighted.mean" can be useful because it provides an indication of the concentration * frequency of occurrence and will highlight the wind speed/direction conditions that dominate the overall mean. Can be: "mean" (default), "median", "max" (maximum), "frequency". "stdev" (standard deviation), "weighted.mean"
`resolution`	Two plot resolutions can be set: "normal" and "fine" (the default), for a smoother plot. It should be noted that plots with a "fine" resolution can take longer to render.
`colors`	Colours to be used for plotting. Options include "default", "increment", "heat", "jet" and RColorBrewer colours — see the openair openColours function for

more details. For user defined the user can supply a list of color names recognised by R (type colors() to see the full list). An example would be color = c("yellow", "green", "blue"). Can also take the values "viridis", "magma", "inferno", or "plasma" which are the viridis colour maps ported from Python's Matplotlib library.

`alpha`	The alpha transparency to use for the plotting surface (a value between 0 and 1 with zero being fully transparent and 1 fully opaque).
`angleScale`	The wind speed scale is by default shown at a 315 degree angle. Sometimes the placement of the scale may interfere with an interesting feature. The user can therefore set angleScale to another value (between 0 and 360 degrees) to mitigate such problems. For example angle.scale = 45 will draw the scale heading in a NE direction.
`normalize`	If TRUE concentrations are normalised by dividing by their mean value. This is done after fitting the smooth surface. This option is particularly useful if one is interested in the patterns of concentrations of PM2.5.
`key`	Fine control of the scale key via drawOpenKey. See drawOpenKey for further details.
`keyPosition`	Location where the scale key is to plotted. Allowed arguments currently include "top", "right", "bottom" and "left".
`ws_spread`	An integer used for the weighting kernel spread for wind speed when correlation or regression techniques are used. Default is 15.
`wd_spread`	An integer used for the weighting kernel spread for wind direction when correlation or regression techniques are used. Default is 4.
`verbose`	Logical controlling the generation of progress and error messages.

Value

A plot and an object of class "openair".

See Also

https://davidcarslaw.github.io/openair/reference/polarPlot.html

Examples

```
library(AirSensor)

setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

pas <- pas_load(archival = TRUE)
pat <- pat_loadMonth(label = "SCBB_02", pas = pas, datestamp = 202005)
sensor <- pat_createAirSensor(pat)

# Polar plot
sensor_polarPlot(sensor, resolution = "normal")
```

sensor_pollutionRose *Pollution rose plot*

Description

Plots a traditional wind rose plot for wind direction and PM2.5.

Usage

```
sensor_pollutionRose(
  sensor = NULL,
  windData = NULL,
  statistic = "prop.count",
  key = TRUE,
  keyPosition = "right",
  annotate = TRUE,
  angle = 30,
  angleScale = 315,
  gridLine = NULL,
  breaks = 6,
  paddle = FALSE,
  seg = 0.9,
  normalize = FALSE,
  verbose = TRUE
)
```

Arguments

`sensor`	an 'airsensor' object
`windData`	a dataframe containing columns "date", "ws", and "wd".
`statistic`	The statistic to be applied to each data bin in the plot. Options currently include "prop.count", "prop.mean" and "abs.count". The default "prop.count" sizes bins according to the proportion of the frequency of measurements. Similarly, "prop.mean" sizes bins according to their relative contribution to the mean. "abs.count" provides the absolute count of measurements in each bin.
`key`	control of the scale key via drawOpenKey. See drawOpenKey for further details.
`keyPosition`	location where the scale key is to plotted. Allowed arguments currently include "top", "right", "bottom" and "left".
`annotate`	If TRUE then the percentage calm and mean values are printed in each panel together with a description of the statistic below the plot. If " " then only the stastic is below the plot. Custom annotations may be added by setting value to c("annotation 1", "annotation 2").
`angle`	default angle of "spokes" is 30. Other potentially useful angles are 45 and 10. Note: the width of the wind speed interval may need adjusting using width.

angleScale	The wind speed scale is by default shown at a 315 degree angle. Sometimes the placement of the scale may interfere with an interesting feature. The user can therefore set angle.scale to another value (between 0 and 360 degrees) to mitigate such problems. For example angleScale = 45 will draw the scale heading in a NE direction.
gridLine	Grid line interval to use. If NULL, as in default, this is assigned by based on the available data range. However, it can also be forced to a specific value, e.g. gridLine = 10. grid.line can also be a list to control the interval, line type and colour. For example gridLine = list(value = 10, lty = 5, col = "purple").
breaks	the number of break points for wind speed in pollutant
paddle	Either TRUE (default) or FALSE. If TRUE plots rose using 'paddle' style spokes. If FALSE plots rose using 'wedge' style spokes.
seg	determines with width of the segments. For example, seg = 0.5 will produce segments 0.5 * angle.
normalize	if TRUE each wind direction segment is normalized to equal one. This is useful for showing how the concentrations (or other parameters) contribute to each wind sector when the proprtion of time the wind is from that direction is low. A line showing the probability that the wind.
verbose	Logical controlling the generation of progress and error messages.

Value

a plot or a dataframe

See Also

https://davidcarslaw.github.io/openair/reference/windRose.html

Examples

```
library(AirSensor)

# Set default location of pre-generated data files
setArchiveBaseUrl("http://data.mazamascience.com/PurpleAir/v1")

pas <- pas_load(archival = TRUE)
pat <- pat_loadMonth(label = "SCBB_02", pas = pas, datestamp = 202005)
sensor <- pat_createAirSensor(pat)

# Load wind data from NOAA
windData <- worldmet::importNOAA(
  code = "722975-53141",
  year = 2020
)
windData <- dplyr::select(windData, c("date", "wd", "ws"))

# Plot rose using mean binning
sensor_pollutionRose(sensor, windData, statistic = "prop.mean")
```

setArchiveBaseDir *Set data archive base directory*

Description

Sets the package base directory pointing to an archive of pre-generated data files.

Usage

```
setArchiveBaseDir(archiveBaseDir)
```

Arguments

`archiveBaseDir` Base directory pointing to an archive of pre-generated data files.

Value

Silently returns previous value of base directory.

See Also

ArchiveBaseDir
getArchiveBaseDir

setArchiveBaseUrl *Set data archive base URL*

Description

Sets the package base URL pointing to an archive of pre-generated data files.

Known base URLs include:

- http://data.mazamascience.com/PurpleAir/v1

Usage

```
setArchiveBaseUrl(archiveBaseUrl)
```

Arguments

`archiveBaseUrl` Base URL pointing to an archive of pre-generated data files.

Value

Silently returns previous value of base URL.

See Also

ArchiveBaseUrl

getArchiveBaseUrl

---

spatialIsInitialized *Check if MazamaSpatialUtils has been initialized*

Description

Logical convenience function to check if `initializeMazamaSpatialUtils()` has been run.

Usage

```
spatialIsInitialized()
```

Value

Logical.

---

timeseriesTbl_multiPlot
Faceted plot of a timeseries tibble

Description

A plotting function that uses ggplot2 to display a suite of timeseries plots all at once.

Usage

```
timeseriesTbl_multiPlot(
  tbl = NULL,
  pattern = NULL,
  parameters = NULL,
  nrow = NULL,
  ncol = NULL,
  autoRange = TRUE,
  ylim = NULL,
  style = "line"
)
```

Arguments

`tbl`	Tibble with a `datetime`.
`pattern`	Pattern used to match groups of parameters.
`parameters`	Custom vector of aggregation parameters to view.
`nrow`	Number of rows in the faceted plot.
`ncol`	Number of columns in the faceted plot.
`autoRange`	Logical specifying whether to scale the y axis separately for each plot or to use a common y axis.
`ylim`	Vector of (lo,hi) y-axis limits.
`style`	Style of plot: ("point", "line", "area")

Note

Specification of `ylim` will override the choice of `autoRange`.

Examples

```
library(AirSensor)

tbl <- pat_aggregateOutlierCounts(example_pat_failure_A)

timeseriesTbl_multiPlot(
  tbl,
  pattern = c("humidity|temperature"),
  nrow = 2
)
```

Index