Package ‘PWFSLSmoke’

October 12, 2022

Type Package

Version 1.2.117

Title Utilities for Working with Air Quality Monitoring Data

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

Depends R (>= 3.5.0), dplyr (>= 0.8.3), maps, MazamaCoreUtils (>= 0.4.3), MazamaSpatialUtils (>= 0.6.3)

Imports curl, cluster, dygraphs (>= 1.1.1.4), geosphere, ggmap, glue, httr, jsonlite, leaflet (>= 1.0.0), lubridate, magrittr, mapproj, maptools, methods, openair, png, purrr, raster, RColorBrewer, rgdal, RgoogleMaps, readr, reshape2, rlang, sf, slippymath, sp, stringr, tibble, tidyr, xts

Suggests knitr, lintr, rmarkdown, testthat (>= 2.1.0)

Description Utilities for working with air quality monitoring data with a focus on small particulates (PM2.5) generated by wildfire smoke. Functions are provided for downloading available data from the United States ‘EPA’ <https://www.epa.gov/outdoor-air-quality-data> and its ‘AirNow’ air quality site <https://www.airnow.gov>. Additional sources of PM2.5 data made accessible by the package include: 'AIRSIS' (aka "Oceaneering", not public) and 'WRCC' <https://wrcc.dri.edu/cgi-bin/smoke.pl>. Data compilations are hosted by the USFS ‘AirFire’ research team <https://www.airfire.org>.

License GPL-3

VignetteBuilder knitr

Repository CRAN

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

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

Encoding UTF-8

LazyData true

RoxygenNote 7.1.2
R topics documented:

- addAQILegend .................................................. 5
- addAQILines .................................................... 6
- addAQIStackedBar ............................................. 6
- addBullseye .................................................... 7
- addIcon .......................................................... 7
- addMarker ....................................................... 8
- addPolygon ...................................................... 9
- addShadedBackground ....................................... 10
- addShadedNight ............................................... 11
- addWindBarbs .................................................. 11
- airnow_createDataDataframes ................................. 12
- airnow_createMetaDataframes ................................. 14
- airnow_createMonitorObjects ................................. 16
- airnow_downloadHourlyData ................................. 18
- airnow_downloadParseData ................................... 19
- airnow_downloadSites ........................................ 20
- airnow_load ..................................................... 21
- airnow_loadAnnual ............................................ 22
- airnow_loadDaily .............................................. 23
- airnow_loadLatest ............................................ 25
- airnow_qualityControl ....................................... 26
- AIRSIS .......................................................... 27
- airsis_availableUnits ........................................ 28
- airsis_BAM1020QualityControl .............................. 29
- airsis_createDataDataFrame .................................. 30
- airsis_createMetaDataframe .................................. 30
- airsis_createMonitorObject .................................. 31
- airsis_createRawDataFrame .................................. 34
- airsis_downloadData .......................................... 35
- airsis_EBAMQualityControl ................................. 36
- airsis_EBAM_MULTI2QualityControl ......................... 38
- airsis_EBAM_MULTI2_BQualityControl ....................... 39

NeedsCompilation no

Author Jonathan Callahan [aut, cre],
        Hans Martin [ctb],
        Spencer Pease [ctb],
        Helen Miller [ctb],
        Zach Dingels [ctb],
        Rohan Aras [ctb],
        Jon Hagg [ctb],
        Jimin Kim [ctb],
        Rex Thompson [ctb],
        Alice Yang [ctb]

Date/Publication 2021-11-23 09:40:02 UTC
topics documented:

airsis_EBAM_PLUS_MULTIQualityControl ........................................ 40
airsis_ESAMQualityControl .............................................................. 41
airsis_ESAM_MULTIQualityControl ...................................................... 43
airsis_identifyMonitorType ............................................................... 44
airsis_load ....................................................................................... 45
airsis_loadAnnual .......................................................................... 45
airsis_loadDaily .............................................................................. 47
airsis_loadLatest ............................................................................ 48
airsis_parseData ............................................................................. 49
airsis_qualityControl ....................................................................... 50
AQI ................................................................................................. 51
aqiColors ......................................................................................... 52
aqiPalette ......................................................................................... 53
AQI_en .............................................................................................. 54
AQI_es ............................................................................................... 54
Camp_Fire ......................................................................................... 55
Carmel_Valley ................................................................................... 56
CONUS ............................................................................................ 56
distance .......................................................................................... 57
epa_createDataDataframe ................................................................. 57
epa_createMetaDataframe ................................................................. 58
epa_createMonitorObject ................................................................. 59
epa_downloadData ............................................................................. 60
epa_load ........................................................................................... 62
epa_loadAnnual ............................................................................... 62
epa_parseData ................................................................................... 63
esriToken ........................................................................................ 65
generic_downloadData ................................................................. 65
generic_parseData ......................................................................... 66
getEsriToken .................................................................................... 67
getGoogleApiKey ............................................................................. 68
googleApiKey .................................................................................... 68
initializeMazamaSpatialUtils ......................................................... 69
loadDaily ........................................................................................ 69
loadLatest ....................................................................................... 70
monitor_aqi ..................................................................................... 71
monitor_asDataframe ................................................................. 72
monitor-collapse ......................................................................... 73
monitor_combine ....................................................................... 75
monitor_dailyBarplot ................................................................. 76
monitor_dailyStatistic ............................................................... 77
monitor_dailyStatisticList .......................................................... 78
monitor_dailyThreshold ............................................................. 80
monitor_distance ....................................................................... 81
monitor_downloadAnnual ............................................................ 82
monitor_downloadDaily ............................................................... 83
monitor_downloadLatest .............................................................. 84
monitor_dygaph ............................................................................. 85
monitor_extractDataFrame ........................................ 87
monitor_getCurrentStatus ...................................... 88
monitor_getDailyMean ........................................... 90
monitor_hourlyBarplot .......................................... 91
monitor_isEmpty .................................................. 93
monitor_isMonitor .............................................. 93
monitor_isolate ................................................... 94
monitor_isTidy .................................................... 95
monitor_join ....................................................... 96
monitor_load ....................................................... 98
monitor_loadAnnual ............................................ 100
monitor_loadDaily ............................................... 101
monitor_loadLatest ............................................. 102
monitor_map ....................................................... 104
monitor_nowcast .................................................. 105
monitor_performance ............................................ 106
monitor_performanceMap .......................................... 108
monitor_print ..................................................... 110
monitor_reorder .................................................. 111
monitor_replaceData ............................................ 111
monitor_rollingMean ............................................ 112
monitor_rollingMeanPlot ......................................... 113
monitor_scaleData ............................................... 115
monitor_stamenmap .............................................. 115
monitor_staticmap ............................................... 117
monitor_subset ................................................... 119
monitor_subsetBy ............................................... 120
monitor_subsetByDistance ...................................... 121
monitor_subsetData ............................................. 122
monitor_subsetMeta ............................................. 123
monitor_timeAverage ........................................... 124
monitor_timeInfo ................................................ 125
monitor_timeseriesPlot ......................................... 126
monitor_toTidy ................................................... 128
monitor_trim ...................................................... 129
monitor_writeCSV ............................................... 130
monitor_writeCurrentStatusGeoJSON ........................... 131
Northwest_Megafires ............................................ 132
rawPlot_pollutionRose .......................................... 133
rawPlot_timeOfDaySpaghetti .................................. 134
rawPlot_timeseries ............................................. 135
rawPlot_windRose ................................................ 136
raw_enhance ...................................................... 137
raw_getHighlightDates ......................................... 138
setEsriToken ..................................................... 139
setGoogleApiKey ................................................ 139
skill_confusionMatrix ......................................... 140
addAQILegend

Add an AQI Legend to a Map

Description

This function is a convenience wrapper around graphics::legend(). It will show the AQI colors and names by default if col and legend are not specified.

Usage

addAQILegend(
  x = "topright",
  y = NULL,
  col = rev(AQI$colors),
  legend = rev(AQI$names),
  pch = 16,
  title = "Air Quality Index",
  ... )
addAQIStackedBar

Arguments

- **x**: x coordinate passed on to the `legend()` command
- **y**: y coordinate passed on to the `legend()` command
- **col**: the color for points/lines in the legend
- **legend**: a character vector to be shown in the legend
- **pch**: plotting symbols in the legend
- **title**: title for the legend
- **...**: additional arguments to be passed to `legend()`

addAQILines

*Add AQI Lines to a Plot*

Description

This function is a convenience for:

```r
abline(h = AQI$breaks_24, col = AQI$colors)
```

Usage

```r
addAQILines(...)```

Arguments

- **...**: additional arguments to be passed to `abline()`

addAQIStackedBar

*Create Stacked AQI Bar*

Description

Draws a stacked bar indicating AQI levels on one side of a plot

Usage

```r
addAQIStackedBar(width = 0.01, height = 1, pos = "left")```

Arguments

- **width**: width of the bar as a fraction of the width of the plot area (default = .02)
- **height**: height of the bar as a fraction of the height of the plot area (default = 1)
- **pos**: position of the stacked bar. Either 'left' or 'right'

Value

Stacked AQI Bar
**addBullseye**

Add a Bullseyes to a Map or RgoogleMap Plot

**Description**

Draws a bullseye with concentric circles of black and white at the specified location.

**Usage**

```r
addBullseye(longitude, latitude, map = NULL, cex = 2, lwd = 2)
```

**Arguments**

- `longitude`: vector of longitudes
- `latitude`: vector of latitudes
- `map`: optional RgoogleMaps map object
- `cex`: character expansion
- `lwd`: line width of individual circles

**Examples**

```r
wa <- monitor_subset(Northwest_Megafires, stateCodes='WA', tlim=c(20150821,20150828))
monitor_map(wa, cex=4)
addBullseye(wa$meta$longitude, wa$meta$latitude)
```

**addIcon**

Add Icons to a Map or RgoogleMap Plot

**Description**

Adds an icon to `map` – an RgoogleMaps map object. The following icons are available:

- `orangeFlame` – yellow-orange flame
- `redFlame` – orange-red flame

You can use other .png files as icons by passing an absolute path as the `icon` argument.

**Usage**

```r
addIcon(icon, longitude, latitude, map = NULL, expansion = 0.1, pos = 0)
```
addMarker

Add Icons to a Map or RgoogleMap Plot

Description

Adds a marker to a plot or map – an RgoogleMaps map object or Raster* object.

Usage

addMarker(longitude, latitude, color = "red", map = NULL, expansion = 1, ...)

Arguments

longitude vector of longitudes
latitude vector of latitudes
color marker color: ‘red’, ‘green’, ‘yellow’, ‘orange’, or ‘blue’. Also includes AQI category colors, specified ‘AQI[number]’ eg. ‘AQI1’

Notes

For RgoogleMaps, the expansion will be ~ 0.1 while for basic plots it may need to be much smaller, perhaps ~ 0.001.

Examples

## Not run:
library(PWFSLSmoke)
monitor_map(Camp_Fire)

addIcon("orangeFlame",
longitude = -121.437222,
latitude = 39.810278,
expansion = 0.003
)

## End(Not run)
addPolygon

Add a Colored Polygon to a Plot

Description

Add a multi-sided polygon to a plot.

Usage

addPolygon(
  x = 0,
  y = 0,
  sides = 72,
  radius = 1,
  rotation = 0,
  border = NULL,
  col = NA,
  ... NA,
)

Arguments

x          x location of center
y          y location of center
sides      number of sides
radius     radius
rotation   amount to rotate the polygon in radians

Examples

## Not run:
library(PWFSLSmoke)

monitor_map(Camp_Fire)

addMarker(
  longitude = -121.437222,
  latitude = 39.810278,
  color = "red",
  expansion = 1
)

## End(Not run)
addShadedBackground

Description

Adds vertical lines to an existing plot using any variable that shares the same length as the time axis of the current plot. Line widths corresponds to magnitude of values.

Usage

addShadedBackground(
  param,
  timeAxis,
  breaks = stats::quantile(param, na.rm = TRUE),
  col = "blue",
  maxOpacity = 0.2,
  lwd = 1
)

Arguments

  param         vector of data to be represented
  timeAxis      vector of times of the same length as param
  breaks        set of breaks used to assign colors
  col           color for vertical lines
  maxOpacity    maximum opacity
  lwd           line width
addShadedNight  Add Nighttime Shading to a Plot

Description

Draw shading rectangles on a plot to indicate nighttime hours.

Usage

addShadedNight(timeInfo, col = adjustcolor("black", 0.1))

Arguments

timeInfo  dataframe with local time, sunrise, and sunset
col  color used to shade nights – defaults to adjustcolor('black', 0.2)

See Also

timeInfo

addWindBarbs  Add wind barbs to a map

Description

Add a multi-sided polygon to a plot.

Usage

addWindBarbs(
  x,
  y,
  speed,
  dir,
  circleSize = 1,
  circleFill = "transparent",
  lineCol = 1,
  extraBarbLength = 0,
  barbSize = 1,
  ...
)
Arguments

- **x**: vector of longitudes
- **y**: vector of latitudes
- **speed**: vector of wind speeds in knots
- **dir**: wind directions in degrees clockwise from north
- **circleSize**: size of the circle
- **circleFill**: circle fill color
- **lineCol**: line color (currently not supported)
- **extraBarbLength**: add length to barbs
- **barbSize**: size of the barb
- **...**: additional arguments to be passed to `lines`

References

https://commons.wikimedia.org/wiki/Wind_speed

Examples

```r
maps::map('state', "washington")
x <- c(-121, -122)
y <- c(47.676057, 47)
addWindBarbs(x, y, speed = c(45,65), dir = c(45, 67),
  circleSize = 1.8, circleFill = c('orange', 'blue'))
```

Description

This function uses the `airnow_downloadParseData` function to download monthly dataframes of AirNow data and restructures that data into a format that is compatible with the PWFSLSmoke package `ws_monitor` data model. The output is meant to be used as the `df` argument in other PWFSLSmoke functions (e.g. `airnow_qualityControl`).

AirNow data parameters include at least the following list:

1. BARPR
2. BC
3. CO
4. NO
5. NO2
6. NO2Y
airnow_createDataDataframes

7. NO2X
8. NOX
9. NOOY
10. OC
11. OZONE
12. PM10
13. PM2.5
14. PRECIP
15. RHUM
16. SO2
17. SRAD
18. TEMP
19. UV-AETH
20. WD
21. WS

Setting parameters=NULL will generate a separate dataframe for each of the above parameters.

Usage

```r
airnow_createDataDataframes(
  parameters = NULL,
  startdate = strftime(lubridate::now(tzone = "UTC"), "%Y%m%d%00", tz = "UTC"),
  hours = 24
)
```

Arguments

- **parameters**: Vector of names of desired pollutants or NULL for all pollutants.
- **startdate**: Desired start date (integer or character representing YYYYMMDD[HH]).
- **hours**: Desired number of hours of data to assemble.

Value

List of dataframes where each dataframe contains all data for a unique parameter (e.g: "PM2.5", "NOX").

Note

As of 2016-12-27, it appears that hourly data are available only for 2016 and not for earlier years.

See Also

- `airnow_downloadParseData`
- `airnow_qualityControl`
Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({

airnowList <- airnow_createDataDataframes("PM2.5", 2019062500)

}, silent = FALSE)

## End(Not run)
```

---

**airnow_createMetaDataframes**

Create dataframes of AirNow site location metadata

**Description**

The `airnow_createMetaDataframes()` function uses the `airnow_downloadSites()` function to download site metadata from AirNow and restructures that data into a format that is compatible with the PWFSLSmoke package `ws_monitor` data model.

The meta dataframe in the `ws_monitor` data model has metadata associated with monitoring site locations for a specific parameter and must contain at least the following columns:

- monitorID – per deployment unique ID
- longitude – decimal degrees E
- latitude – decimal degrees N
- elevation – height above sea level in meters
- timezone – olson timezone
- countryCode – ISO 3166-1 alpha-2

The meta dataframe will have rownames matching `monitorID`.

This function takes a dataframe obtained from AirNowTech's `monitoring_site_locations.dat` file, splits it up into separate dataframes, one for each parameter, and performs the following cleanup:

- convert incorrect values to NA e.g. longitude=0 & latitude=0
- add timezone information

Parameters included in AirNow data include at least the following list:

1. BARPR
2. BC
3. CO
4. NO
airnow_createMetaDataframes

5. NO2
6. NO2Y
7. NO2X
8. NOX
9. NOOY
10. OC
11. OZONE
12. PM10
13. PM2.5
14. PRECIP
15. RHUM
16. SO2
17. SRAD
18. TEMP
19. UV-AETH
20. WD
21. WS

Setting parameters=NULL will generate a separate dataframe for each of the above parameters.

Usage

airnow_createMetaDataframes(
  parameters = NULL,
  pwfs1DataIngestSource = "AIRNOW",
  addGoogleMeta = TRUE
)

Arguments

parameters vector of names of desired pollutants or NULL for all pollutants
pwfs1DataIngestSource identifier for the source of monitoring data, e.g. 'AIRNOW'
addGoogleMeta logical specifying wheter to use Google elevation and reverse geocoding services

Value

List of `meta` dataframes with site metadata for unique parameters (e.g: "PM2.5", "NOX").

See Also

airnow_downloadSites
Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
    metaList <- airnow_createMetaDataframes(parameters = "PM2.5")
}, silent = FALSE)
## End(Not run)
```

### Description

This function uses the `airnow_downloadParseData` function to download monthly dataframes of AirNow data and restructures that data into a format that is compatible with the PWFSLSmoke package `ws_monitor` data model.

AirNow data parameters include at least the following list:

1. BARPR
2. BC
3. CO
4. NO
5. NO2
6. NO2Y
7. NO2X
8. NOX
9. NOOY
10. OC
11. OZONE
12. PM10
13. PM2.5
14. PRECIP
15. RHUM
16. SO2
17. SRAD
18. TEMP
19. UV-AETH
20. WD
21. WS

Setting parameters=NULL will generate a separate ws_monitor object for each of the above parameters.

Usage

```r
airnow_createMonitorObjects(
  parameters = NULL,
  startdate = strftime(lubridate::now(tzone = "UTC"), "%Y%m%d", tz = "UTC"),
  hours = 24,
  zeroMinimum = TRUE,
  addGoogleMeta = TRUE
)
```

Arguments

- `parameters` vector of names of desired pollutants or NULL for all pollutants
- `startdate` desired start date (integer or character representing YYYYMMDD[HH])
- `hours` desired number of hours of data to assemble
- `zeroMinimum` logical specifying whether to convert negative values to zero
- `addGoogleMeta` logical specifying whether to use Google elevation and reverse geocoding services

Value

List where each element contains a ws_monitor object for a unique parameter (e.g: "PM2.5", "NOX").

Note

As of 2017-12-17, it appears that hourly data are available only for 2016 and not for earlier years.

See Also

- `airnow_createDataDataframes`
- `airnow_createMetaDataframes`

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try{

monList <- airnow_createMonitorObjects(c("PM2.5"), 20190625)
pm25 <- monList$PM2.5
o3 <- monList$O3
```
Description

The https://airnowtech.org site provides both air pollution monitoring data as well as monitoring site location metadata. This function retrieves a single, hourly data file and returns it as a dataframe which includes a monitor’s site name and parameters monitored.

Usage

```r
airnow_downloadHourlyData(
  datestamp = strftime(lubridate::now(tzone = "UTC"), "%Y%m%d00", tz = "UTC"),
  baseUrl = "https://files.airnowtech.org/airnow"
)
```

Arguments

- `datestamp` Integer or character representing YYYYMMDDHH.
- `baseUrl` Base URL for archived hourly data.

Value

Dataframe of AirNow hourly data.

Note

As of 2016-12-27, it appears that hourly data are available only for 2016 and not for earlier years. Data from locations whose timezones have a fractional offset from UTC are removed as the PWF-SLSmoke data model only supports data reported on hour boundaries. As of 2019-06-26, this only applies to US Department of State monitors in Myanmar, Sri Lanka, India and Nepal.

See Also

- `airnow_createDataDataframes`
- `airnow_downloadParseData`
Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  df <- airnow_downloadHourlyData(2018070112)
}, silent = FALSE)
## End(Not run)
```

---

**airnow_downloadParseData**

*Download and aggregate multiple hourly data files from AirNow*

---

**Description**

This function makes repeated calls to `airnow_downloadHourlyData` to obtain data from AirNow. All data obtained are then combined into a single tibble and returned.

Parameters included in AirNow data include at least the following list:

1. BARPR
2. BC
3. CO
4. NO
5. NO2
6. NO2Y
7. NO2X
8. NOX
9. NOOY
10. OC
11. OZONE
12. PM10
13. PM2.5
14. PRECIP
15. RHUM
16. SO2
17. SRAD
18. TEMP
19. UV-AETH
20. WD
21. WS

Passing a vector of one or more of the above names as the `parameters` argument will cause the resulting tibble to be filtered to contain only records for those parameters.
Usage

```r
airnow_downloadParseData(
  parameters = NULL,
  startdate = strftime(lubridate::now(tzone = "UTC"), "%Y%m%d00", tz = "UTC"),
  hours = 24
)
```

Arguments

- `parameters` vector of names of desired pollutants or NULL for all pollutants
- `startdate` desired start date (integer or character representing YYYYMMDD[HH])
- `hours` desired number of hours of data to assemble

Value

Tibble of aggregated AirNow data.

Note

As of 2016-12-27, it appears that hourly data are available only for 2016 and not for earlier years.

See Also

- `airnow_createDataDataframes`
- `airnow_downloadHourlyData`

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  tbl <- airnow_downloadParseData("PM2.5", 2016070112, hours = 24)
}, silent = FALSE)
## End(Not run)
```

Description

The [https://airnowtech.org](https://airnowtech.org) site provides both air pollution monitoring data as well as monitoring site location metadata. This function retrieves the most recent version of the site location metadata file and returns it as a dataframe.

A description of the data format is publicly available at the Monitoring Site Fact Sheet.
Usage

```r
airnow_downloadSites(
  baseUrl = "https://files.airnowtech.org/airnow/today/",
  file = "monitoring_site_locations.dat"
)
```

Arguments

- `baseUrl`: location of the AirNow monitoring site locations file
- `file`: name of the AirNow monitoring site locations file

Value

Tibble of site location metadata.

Note

As of December, 2016, the `monitoring_site_locations.dat` file has an encoding of "CP437" (aka "Non-ISO extended-ASCII" or "IBMPC 437") and will be converted to "UTF-8" so that French and Spanish language place names are properly encoded in the returned dataframe.

See Also

- `airnow_createMetaDataframes`

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  sites <- airnow_downloadSites()
}, silent = FALSE)
## End(Not run)
```

---

**airnow_load**

*Load Processed AirNow Monitoring Data*

Description

Please use `airnow_loadAnnual` instead of this function. It will soon be deprecated.
Usage

```r
airnow_load(
    year = 2017,
    month = NULL,
    parameter = "PM2.5",
    baseUrl = "https://haze.airfire.org/monitoring/AirNow/RData/
)
```

Arguments

- **year**: desired year (integer or character representing YYYY)
- **month**: desired month (integer or character representing MM)
- **parameter**: parameter of interest
- **baseUrl**: base URL for AirNow meta and data files

Value

A `ws_monitor` object with AirNow data.

Description

Loads pre-generated .RData files containing annual AirNow data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The annual files loaded by this function are updated on the 15’th of each month and cover the period from the beginning of the year to the end of the last month.

For data during the last 45 days, use `airnow_loadDaily()`.

For the most recent data, use `airnow_loadLatest()`.

AirNow parameters include the following:

1. PM2.5

Available AirNow RData and associated log files can be seen at: [https://haze.airfire.org/monitoring/AirNow/RData/](https://haze.airfire.org/monitoring/AirNow/RData/)

Usage

```r
airnow_loadAnnual(
    year = NULL,
    parameter = "PM2.5",
    baseUrl = "https://haze.airfire.org/monitoring",
    dataDir = NULL
)
```
**airnow_loadDaily**

### Description

Load recent AirNow monitoring data.

If **dataDir** is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The daily files loaded by this function are updated once a day, shortly after midnight and contain data for the previous 45 days.

For the most recent data, use `airnow_loadLatest()`.

For data extended more than 45 days into the past, use `airnow_loadAnnual()`.

AirNow parameters include the following:

---

**Arguments**

- **year**: Desired year (integer or character representing YYYY).
- **parameter**: Parameter of interest.
- **baseUrl**: Base URL for 'annual' AirNow data files.
- **dataDir**: Local directory containing 'annual' data files.

**Value**

A `ws_monitor` object with AirNow data.

**See Also**

- `airnow_loadDaily`
- `airnow_loadLatest`

**Examples**

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  airnow_loadAnnual(2017) %>%
  monitor_subset(stateCodes='MT', tlim=c(20170701,20170930)) %>%
  monitor_dailyStatistic() %>%
  monitor_timeseriesPlot(style = 'gnats', ylim=c(0,300), xpd=NA)
  addAQIStackedBar()
  addAQILines()
  title("Montana 2017 -- AirNow Daily Average PM2.5")
}, silent = FALSE)
## End(Not run)
```

---

**airnow_loadDaily**  
_Load recent AirNow monitoring data_
1. PM2.5

Available AirNow RData and associated log files can be seen at: https://haze.airfire.org/monitoring/AirNow/RData/latest/

Usage

```r
airnow_loadDaily(
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL
)
```

Arguments

- `parameter`: Parameter of interest.
- `baseUrl`: Base URL for 'daily' AirNow data files.
- `dataDir`: Local directory containing 'daily' data files.

Value

A `ws_monitor` object with AirNow data.

See Also

- `airnow_loadAnnual`
- `airnow_loadLatest`

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  airnow_loadDaily() %>%
    monitor_subset(stateCodes=CONUS) %>%
    monitor_map()
}, silent = FALSE)

## End(Not run)
```
Description

Loads pre-generated .RData files containing the most recent AirNow data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The files loaded by this function are updated multiple times an hour and contain data for the previous 10 days.

For daily updates covering the most recent 45 days, use `airnow_loadDaily()`.

For data extended more than 45 days into the past, use `airnow_loadAnnual()`.

AirNow parameters include the following:

1. PM2.5

Available RData and associated log files can be seen at: https://haze.airfire.org/monitoring/AirNow/RData/latest/

Usage

```r
airnow_loadLatest(
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL
)
```

Arguments

- `parameter` Parameter of interest.
- `baseUrl` Base URL for 'daily' AirNow data files.
- `dataDir` Local directory containing 'daily' data files.

Value

A `ws_monitor` object with AirNow data.

See Also

- `airnow_loadAnnual`
- `airnow_loadDaily`
Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  airnow_loadLatest() %>%
    monitor_subset(stateCodes=CONUS) %>%
    monitor_map()
}, silent = FALSE)
## End(Not run)
```

---

`airnow_qualityControl`  
Apply Quality Control to AirNow dataframe

Description

Perform range validation on AirNow data. This function also replaces values of -999 with `NA`.

Usage

```r
airnow_qualityControl(df, limits = c(-Inf, Inf))
```

Arguments

- `df`: multi-site restructured dataframe created within `airnow_createDataDataframe()`
- `limits`: low and high range of valid values

Value

Cleaned up dataframe of AIRSIS monitor data.

See Also

`airnow_createDataDataframes`
AIRSIS

AIRSIS Unit Types

Description
AIRSIS provides access to data by unit type at URLs like: http://usfs.airsis.com/vision/common/CSVExport.aspx?utid=38&StartDate=2017-11-06&EndDate=2017-11-07

The AIRSIS object is a list of lists. The element named `unitTypes` is itself a list of named unit types:

Unit types include:

- DATARAM 21 = Dataram
- BAM1020 24 = Bam 1020
- EBAM_NEW 30 = eBam-New
- EBAM 38 = Iridium - Ebam
- ESAM 39 = Iridium - Esam
- AUTOMET 43 = Automet

Usage
AIRSIS

Format
A list of lists

Details
AIRSIS monitor types and codes

Note
This list of monitor types was created on Feb 09, 2017.
airsis_availableUnits  Get AIRSIS available unit identifiers

Description

Returns a list of unitIDs with data during a particular time period.

Usage

```r
airsis_availableUnits(
  startdate = strftime(lubridate::now(tzone = "UTC"), "%Y010100", tz = "UTC"),
  enddate = strftime(lubridate::now(tzone = "UTC"), "%Y%m%d23", tz = "UTC"),
  provider = "USFS",
  unitTypes = c("BAM1020", "EBAM", "ESAM"),
  baseUrl = "http://xxxx.airsis.com/vision/common/CSVExport.aspx?"
)
```

Arguments

- **startdate**  
  desired start date (integer or character representing YYYYMMDD[HH])
- **enddate**  
  desired end date (integer or character representing YYYYMMDD[HH])
- **provider**  
  identifier used to modify baseURL ['APCD' | 'USFS']
- **unitTypes**  
  vector of unit types
- **baseUrl**  
  base URL for data queries

Value

Vector of AIRSIS unitIDs.

References

Interagency Real Time Smoke Monitoring

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  unitIDs <- airsis_availableUnits(20150701, 20151231,
    provider = "USFS",
    unitTypes = c("EBAM", "ESAM"))
}, silent = FALSE)
```

## End(Not run)
Apply Quality Control to raw AIRSIS BAM1020 dataframe

Description

Perform various QC measures on AIRSIS BAM1020 data.
A POSIXct datetime column (UTC) is also added based on DateTime.

Usage

```r
airsis_BAM1020QualityControl(
  tbl,
  valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90),
  remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE,
  valid_Flow = c(0.834 * 0.95, 0.834 * 1.05),
  valid_AT = c(-Inf, 45),
  valid_RHi = c(-Inf, 45),
  valid_Conc = c(-Inf, 5000),
  flagAndKeep = FALSE
)
```

Arguments

- `tbl`: single site tibble created by `airsis_parseData()`
- `valid_Longitude`: range of valid Longitude values
- `valid_Latitude`: range of valid Latitude values
- `remove_Lon_zero`: flag to remove rows where Longitude == 0
- `remove_Lat_zero`: flag to remove rows where Latitude == 0
- `valid_Flow`: range of valid Flow values
- `valid_AT`: range of valid AT values
- `valid_RHi`: range of valid RHi values
- `valid_Conc`: range of valid ConcHr values
- `flagAndKeep`: flag, rather than remove, bad data during the QC process

Value

Cleaned up tibble of AIRSIS monitor data.
airsis_createDataDataframe

Create AIRSIS data dataframe

Description

After quality control has been applied to an AIRSIS tibble, we can extract the PM2.5 values and store them in a data dataframe organized as time-by-deployment (aka time-by-site).

The first column of the returned dataframe is named 'datetime' and contains a POSIXct time in UTC. Additional columns contain data for each separate deployment of a monitor.

Usage

airsis_createDataDataframe(tbl, meta)

Arguments

tbl single site AIRSIS tibble created by airsis_clustering()
meta AIRSIS meta dataframe created by airsis_createMetaDataframe()

Value

A data dataframe for use in a ws_monitor object.

airsis_createMetaDataframe

Create AIRSIS site location metadata dataframe

Description

After an AIRSIS tibble has been enhanced with additional columns generated by addClustering we are ready to pull out site information associated with unique deployments.

These will be rearranged into a dataframe organized as deployment-by-property with one row for each monitor deployment.

This site information found in tbl is augmented so that we end up with a uniform set of properties associated with each monitor deployment. The list of columns in the returned meta dataframe is:

> names(p$meta)

[1] "monitorID"       "longitude"     "latitude"
[4] "elevation"       "timezone"      "countryCode"
[7] "stateCode"       "siteName"      "agencyName"
[10] "countyName"      "msaName"      "monitorType"
[13] "monitorInstrument" "aqsID"       "pwfs1ID"
[16] "pwfs1DataIngestSource" "telemetryAggregator" "telemetryUnitID"
Usage

```r
airsis_createMetaDataframe(
  tbl,
  provider = as.character(NA),
  unitID = as.character(NA),
  pwfs1DataIngestSource = "AIRSIS",
  existingMeta = NULL,
  addGoogleMeta = FALSE,
  addEsriMeta = FALSE
)
```

Arguments

- **tbl**: single site AIRSIS tibble after metadata enhancement
- **provider**: identifier used to modify baseURL ['APCD' | 'USFS']
- **unitID**: character or numeric AIRSIS unit identifier
- **pwfs1DataIngestSource**: identifier for the source of monitoring data, e.g. 'AIRSIS'
- **existingMeta**: existing 'meta' dataframe from which to obtain metadata for known monitor deployments
- **addGoogleMeta**: logical specifying wheter to use Google elevation and reverse geocoding services
- **addEsriMeta**: logical specifying wheter to use ESRI elevation and reverse geocoding services

Value

A `meta` dataframe for use in a `ws_monitor` object.

See Also

- `airsis_createMonitorObject`
- `addMazamaMetadata`

Description

Obtains monitor data from an AIRSIS webservice and converts it into a quality controlled, metadata enhanced `ws_monitor` object ready for use with all `monitor_~` functions.

Steps involved include:

1. download CSV text
2. parse CSV text
3. apply quality control
4. apply clustering to determine unique deployments
5. enhance metadata to include: elevation, timezone, state, country, site name
6. reshape AIRSIS data into deployment-by-property meta and and time-by-deployment data dataframes

QC parameters that can be passed in the ... include the following valid data ranges as taken from airsis_EBAMQualityControl():

- valid_Longitude=c(-180,180)
- valid_Latitude=c(-90,90)
- remove_Lon_zero = TRUE
- remove_Lat_zero = TRUE
- valid_Flow = c(16.7*0.95,16.7*1.05)
- valid_AT = c(-Inf,45)
- valid_RHi = c(-Inf,50)
- valid_Conc = c(-Inf,5.000)

Note that appropriate values for QC thresholds will depend on the type of monitor.

Usage

```r
airsis_createMonitorObject(
  startdate = strftime(lubridate::now(tzone = "UTC"), "%Y010100", tz = "UTC"),
  enddate = strftime(lubridate::now(tzone = "UTC"), "%Y%m%d23", tz = "UTC"),
  provider = NULL,
  unitID = NULL,
  clusterDiameter = 1000,
  zeroMinimum = TRUE,
  baseUrl = "http://xxxx.airsis.com/vision/common/CSVExport.aspx?",
  saveFile = NULL,
  existingMeta = NULL,
  addGoogleMeta = FALSE,
  addEsriMeta = FALSE,
  ...
)
```

Arguments

- `startdate` desired start date (integer or character representing YYYYMMDD[HH])
- `enddate` desired end date (integer or character representing YYYYMMDD[HH])
- `provider` identifier used to modify baseURL ['APCD' | 'USFS']
- `unitID` character or numeric AIRSIS unit identifier
- `clusterDiameter` diameter in meters used to determine the number of clusters (see addClustering())
zeroMinimum  logical specifying whether to convert negative values to zero
baseUrl     base URL for data queries
saveFile    optional filename where raw CSV will be written
existingMeta existing 'meta' dataframe from which to obtain metadata for known monitor deployments
addGoogleMeta logical specifying whether to use Google elevation and reverse geocoding services
addEsriMeta logical specifying whether to use ESRI elevation and reverse geocoding services
... additional parameters are passed to type-specific QC functions

Value

A `ws_monitor` object with AIRSIS data.

Note

The downloaded CSV may be saved to a local file by providing an argument to the `saveFile` parameter.

See Also

`airsis_downloadData`
`airsis_parseData`
`airsis_qualityControl`
`addClustering`
`airsis_createMetaDataframe`
`airsis_createDataDataframe`

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({

  library(PWFSLSmoke)
  initializeMazamaSpatialUtils()

  usfs_1072 <- airsis_createMonitorObject(20200601, 20200620, 'USFS', unitID='1072')
  monitor_timeseriesPlot(usfs_1072)
}, silent = FALSE)
## End(Not run)
```
airsis_createRawDataframe

Obtain AIRSIS data and parse into a raw tibble

Description

Obtains monitor data from an AIRSIS webservice and converts it into a quality controlled, metadata enhanced "raw" tibble ready for use with all raw_~ functions.

Steps involved include:

1. download CSV text
2. parse CSV text
3. apply quality control
4. apply clustering to determine unique deployments
5. enhance metadata to include: elevation, timezone, state, country, site name

Usage

airsis_createRawDataframe(
  startdate = strftime(lubridate::now(tzone = "UTC"), "%Y010100", tz = "UTC"),
  enddate = strftime(lubridate::now(tzone = "UTC"), "%Y%m%d23", tz = "UTC"),
  provider = NULL,
  unitID = NULL,
  clusterDiameter = 1000,
  baseUrl = "http://xxxx.airsis.com/vision/common/CSVExport.aspx?",
  saveFile = NULL,
  flagAndKeep = FALSE
)

Arguments

startdate Desired start date (integer or character representing YYYYMMDD[HH]).
enddate Desired end date (integer or character representing YYYYMMDD[HH]).
provider Identifier used to modify baseURL ['APCD'|'USFS'].
unitID Character or numeric AIRSIS unit identifier.
clusterDiameter Diameter in meters used to determine the number of clusters (see addClustering).
baseUrl Base URL for data queries.
saveFile Optional filename where raw CSV will be written.
flagAndKeep Flag, rather then remove, bad data during the QC process.

Value

Raw tibble of AIRSIS data.
airsis_downloadData

Download AIRIS data

Description

Request data from a particular station for the desired time period. Data are returned as a single character string containing the AIRIS output.

Usage

```r
airsis_downloadData(
  startdate = strftime(lubridate::now(tzone = "UTC"), "%Y0101", tz = "UTC"),
  enddate = strftime(lubridate::now(tzone = "UTC"), "%Y%m%d", tz = "UTC"),
  provider = "USFS",
  unitID = NULL,
  baseUrl = "http://xxxx.airsis.com/vision/common/CSVExport.aspx?"
)
```

Note

The downloaded CSV may be saved to a local file by providing an argument to the `saveFile` parameter.

See Also

airsis_downloadData
airsis_parseData
airsis_qualityControl
addClustering

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  library(PWFSLSmoke)
  raw <- airsis_createRawDataframe(startdate = 20160901,
                                      provider = "USFS",
                                      unitID = '1033')
  raw <- raw_enhance(raw)
  rawPlot_timeseries(raw, tlim = c(20160908,20160917))
}, silent = FALSE)
## End(Not run)
```
**Arguments**

- **startdate**
  desired start date (integer or character representing YYYYMMDD[HH])

- **enddate**
  desired end date (integer or character representing YYYYMMDD[HH])

- **provider**
  identifier used to modify baseURL ['APCD' | 'USFS']

- **unitID**
  unit identifier

- **baseUrl**
  base URL for data queries

**Value**

String containing AIRSIS output.

**References**

Interagency Real Time Smoke Monitoring

**Examples**

```r
## Not run:
# Fail gracefully if any resources are not available
try({

fileString <- airsis_downloadData( 20150701, 20151231, provider='USFS', unitID='1026')
df <- airsis_parseData(fileString)
}, silent = FALSE)
## End(Not run)
```

**Description**

Perform various QC measures on AIRSIS EBAM data.

The following columns of data are tested against valid ranges:

- Flow
- AT
- RHi
- ConcHr

A POSIXct datetime column (UTC) is also added based on Date.Time.GMT.
Usage

```r
airsis_EBAMQualityControl(
  tbl,
  valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90),
  remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE,
  valid_Flow = c(16.7 * 0.95, 16.7 * 1.05),
  valid_AT = c(-Inf, 45),
  valid_RHi = c(-Inf, 50),
  valid_Conc = c(-Inf, 5),
  flagAndKeep = FALSE
)
```

Arguments

- `tbl`: single site tibble created by `airsis_parseData()`
- `valid_Longitude`: range of valid Longitude values
- `valid_Latitude`: range of valid Latitude values
- `remove_Lon_zero`: flag to remove rows where Longitude == 0
- `remove_Lat_zero`: flag to remove rows where Latitude == 0
- `valid_Flow`: range of valid Flow values
- `valid_AT`: range of valid AT values
- `valid_RHi`: range of valid RHi values
- `valid_Conc`: range of valid ConcHr values
- `flagAndKeep`: flag, rather than remove, bad data during the QC process

Value

Cleaned up tibble of AIRSIS monitor data.

See Also

- `airsis_qualityControl`
airsis\_EBAM\_MULTI2QualityControl

Apply Quality Control to raw AIRSIS EBAM MULTI2 tibble

Description

Perform various QC measures on AIRSIS EBAM MULTI2 data. This data format began appearing in December, 2019 and is associated with data available at https://arb3.airsis.com.

The following columns of data are tested against valid ranges:

- Flow
- AT
- RHi
- ConcHr

A POSIXct datetime column (UTC) is also added based on Date.Time.GMT.

Usage

```r
airsis\_EBAM\_MULTI2QualityControl(
  tbl,
  valid\_Longitude = c(-180, 180),
  valid\_Latitude = c(-90, 90),
  remove\_Lon\_zero = TRUE,
  remove\_Lat\_zero = TRUE,
  valid\_Flow = c(16.7 * 0.95, 16.7 * 1.05),
  valid\_AT = c(-Inf, 45),
  valid\_RHi = c(-Inf, 50),
  valid\_Conc = c(-Inf, 5),
  flagAndKeep = FALSE
)
```

Arguments

- `tbl` single site tibble created by `airsis\_parseData()`
- `valid\_Longitude` range of valid Longitude values
- `valid\_Latitude` range of valid Latitude values
- `remove\_Lon\_zero` flag to remove rows where Longitude == 0
- `remove\_Lat\_zero` flag to remove rows where Latitude == 0
- `valid\_Flow` range of valid Flow values
- `valid\_AT` range of valid AT values
- `valid\_RHi` range of valid RHi values
- `valid\_Conc` range of valid ConcHr values
- `flagAndKeep` flag, rather than remove, bad data during the QC process
Value

Cleaned up tibble of AIRSIS monitor data.

See Also

airsis_qualityControl

airsis_EBAM_MULTI2_BQualityControl

Apply Quality Control to raw AIRSIS EBAM MULTI2_B tibble

Description

Perform various QC measures on AIRSIS EBAM MULT2 data. This data format began appearing in December, 2019 and is associated with data available at https://arb3.airsis.com.

The following columns of data are tested against valid ranges:

- Flow
- AT
- RHi
- ConcHr

A POSIXct datetime column (UTC) is also added based on Date.Time.GMT.

Usage

```r
airsis_EBAM_MULTI2_BQualityControl(
  tbl,
  valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90),
  remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE,
  valid_Flow = c(16.7 * 0.95, 16.7 * 1.05),
  valid_AT = c(-Inf, 45),
  valid_RHi = c(-Inf, 50),
  valid_Conc = c(-Inf, 5),
  flagAndKeep = FALSE
)
```

Arguments

- `tbl` single site tibble created by `airsis_parseData()`
- `valid_Longitude` range of valid Longitude values
- `valid_Latitude` range of valid Latitude values
remove_Lon_zero
flag to remove rows where Longitude == 0

remove_Lat_zero
flag to remove rows where Latitude == 0

valid_Flow    range of valid Flow values
valid_AT      range of valid AT values
valid_RHi     range of valid RHi values
valid_Conc    range of valid ConcHr values
flagAndKeep   flag, rather than remove, bad data during the QC process

Value
Cleaned up tibble of AIRSIS monitor data.

See Also
airsis_qualityControl

Description
Perform various QC measures on AIRSIS EBAM PLUS_MULTI data. This data format began appearing in December, 2019 and is associated with data available at https://apcd.airsis.com.

The following columns of data are tested against valid ranges:

- Flow
- AT
- RHi
- ConcHr

A POSIXct datetime column (UTC) is also added based on Date.Time.GMT.

Usage
airsis_EBAM_PLUS_MULTIQualityControl(
tbl,
valid_Longitude = c(-180, 180),
valid_Latitude = c(-90, 90),
remove_Lon_zero = TRUE,
remove_Lat_zero = TRUE,
valid_Flow = c(16.7 * 0.95, 16.7 * 1.05),
valid_AT = c(-Inf, 45),
valid_RHi = c(-Inf, 50),
valid_Conc = c(-Inf, 5),
flagAndKeep = FALSE
)

Arguments

tbl single site tibble created by airsis_parseData()
valid_Longitude range of valid Longitude values
valid_Latitude range of valid Latitude values
remove_Lon_zero flag to remove rows where Longitude == 0
remove_Lat_zero flag to remove rows where Latitude == 0
valid_Flow range of valid Flow values
valid_AT range of valid AT values
valid_RHi range of valid RHi values
valid_Conc range of valid ConcHr values
flagAndKeep flag, rather than remove, bad data during the QC process

Value

Cleaned up tibble of AIRSIS monitor data.

See Also

airsis_qualityControl

Apply Quality Control to raw AIRSIS E-Sampler dataframe

Description

Perform various QC measures on AIRSIS E-Sampler data.
The following columns of data are tested against valid ranges:

- Flow
- AT
- RHi
- ConcHr

A POSIXct datetime column (UTC) is also added based on TimeStamp.
Usage

```r
airsis_ESAMQualityControl(
  tbl,
  valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90),
  remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE,
  valid_Flow = c(1.999, 2.001),
  valid_AT = c(-Inf, 150),
  valid_RHi = c(-Inf, 55),
  valid_Conc = c(-Inf, 5000),
  flagAndKeep = FALSE
)
```

Arguments

- `tbl` single site tibble created by `airsis_downloadData()
- `valid_Longitude` range of valid Longitude values
- `valid_Latitude` range of valid Latitude values
- `remove_Lon_zero` flag to remove rows where Longitude == 0
- `remove_Lat_zero` flag to remove rows where Latitude == 0
- `valid_Flow` range of valid Flow.l.m values
- `valid_AT` range of valid AT.C. values
- `valid_RHi` range of valid RHi... values
- `valid_Conc` range of valid Conc.mg.m3. values
- `flagAndKeep` flag, rather then remove, bad data during the QC process

Value

Cleaned up tibble of AIRSIS monitor data.

See Also

- `airsis_qualityControl`
airsis_ESAM_MULTIQualityControl

Apply Quality Control to raw AIRSIS E-Sampler dataframe

Description

Perform various QC measures on AIRSIS E-Sampler data.

The following columns of data are tested against valid ranges:

- Flow
- AT
- RHi
- ConcHr

A POSIXct datetime column (UTC) is also added based on TimeStamp.

Usage

```r
airsis_ESAM_MULTIQualityControl(
  tbl,
  valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90),
  remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE,
  valid_Flow = c(1.999, 2.001),
  valid_AT = c(-Inf, 150),
  valid_RHi = c(-Inf, 55),
  valid_Conc = c(-Inf, 5),
  flagAndKeep = FALSE
)
```

Arguments

- `tbl`: single site tibble created by `airsis_downloadData()`
- `valid_Longitude`: range of valid Longitude values
- `valid_Latitude`: range of valid Latitude values
- `remove_Lon_zero`: flag to remove rows where Longitude == 0
- `remove_Lat_zero`: flag to remove rows where Latitude == 0
- `valid_Flow`: range of valid Flow.l.m values
- `valid_AT`: range of valid AT.C. values
- `valid_RHi`: range of valid RHi... values
- `valid_Conc`: range of valid Conc.mg.m3. values
- `flagAndKeep`: flag, rather then remove, bad data during the QC process
Description

Examine the column names of the incoming dataframe (or first line of raw text) to identify different types of monitor data provided by AIRSIS.

The return is a list includes everything needed to identify and parse the raw data using `readr::read_csv()`:

- `monitorType` – identification string
- `rawNames` – column names from the data (including special characters)
- `columnName` – assigned column names (special characters replaced with ‘.’)
- `columnType` – column type string for use with `readr::read_csv()`

The `monitorType` will be one of:

- "BAM1020" – BAM1020 (e.g. USFS #49 in 2010)
- "EBAM" – EBAM (e.g. USFS #1026 in 2010)
- "ESAM" – E-Sampler (e.g. USFS #1002 in 2010)
- "UNKOWN" – ???

Usage

```r
airsis_identifyMonitorType(df)
```

Arguments

- `df` dataframe or raw character string containing AIRSIS data

Value

List including `monitorType`, `rawNames`, `columnName` and `columnType`.

References

Interagency Real Time Smoke Monitoring
Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  fileString <- airsis_downloadData(20150701, 20151231, provider='USFS', unitID='1026')
  monitorTypeList <- airsis_identifyMonitorType(fileString)
}, silent = FALSE)

## End(Not run)
```

---

**airsis_load**  
*Load Processed AIRSIS Monitoring Data*

**Description**

Please use `airsis_loadAnnual` instead of this function. It will soon be deprecated.

**Usage**

```r
airsis_load(
  year = 2017,
  baseUrl = "https://haze.airfire.org/monitoring/AIRSIS/RData/
)
```

**Arguments**

- **year**  
  desired year (integer or character representing YYYY)

- **baseUrl**  
  base URL for AIRSIS meta and data files

**Value**

A `ws_monitor` object with AIRSIS data.

---

**airsis_loadAnnual**  
*Load annual AIRSIS monitoring data*
Description

Loads pre-generated .RData files containing annual AIRSIS data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The annual files loaded by this function are updated on the 15'th of each month and cover the period from the beginning of the year to the end of the last month.

For data during the last 45 days, use `airsis_loadDaily()`.

For the most recent data, use `airsis_loadLatest()`.

AIRSIS parameters include the following:

1. PM2.5

Available AIRSIS RData and associated log files can be seen at: https://haze.airfire.org/monitoring/AIRSIS/RData/

Usage

```r
airsis_loadAnnual(
  year = NULL,
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring",
  dataDir = NULL
)
```

Arguments

- `year`: Desired year (integer or character representing YYYY).
- `parameter`: Parameter of interest.
- `baseUrl`: Base URL for 'annual' AIRSIS data files.
- `dataDir`: Local directory containing 'annual' data files.

Value

A `ws_monitor` object with AIRSIS data.

See Also

- `airsis_loadDaily`
- `airsis_loadLatest`

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  airsis_loadAnnual(2017) %>%
  monitor_subset(stateCodes='MT', tlim=c(20170701,20170930)) %>%
```
airsis_loadDaily

Load recent AIRSIS monitoring data

Description

Loads pre-generated .RData files containing recent AIRSIS data.

If dataDir is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The daily files loaded by this function are updated once a day, shortly after midnight and contain data for the previous 45 days.

For the most recent data, use airsis_loadLatest().

For data extended more than 45 days into the past, use airsis_loadAnnual().

AIRSIS parameters include the following:

1. PM2.5

Available AIRSIS RData and associated log files can be seen at: https://haze.airfire.org/monitoring/AIRSIS/RData/latest/

Usage

airsis_loadDaily(
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL
)

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter</td>
<td>Parameter of interest.</td>
</tr>
<tr>
<td>baseUrl</td>
<td>Base URL for <code>daily</code> AirNow data files.</td>
</tr>
<tr>
<td>dataDir</td>
<td>Local directory containing <code>daily</code> data files.</td>
</tr>
</tbody>
</table>

Value

A `ws_monitor` object with AIRSIS data.
See Also

airsis_loadAnnual
airsis_loadLatest

Examples

## Not run:
# Fail gracefully if any resources are not available
try({

  airsis_loadDaily() %>%
    monitor_subset(stateCodes=CONUS) %>%
    monitor_map()
}, silent = FALSE)

## End(Not run)

---

**airsis_loadLatest**  
*Load most recent AIRSIS monitoring data*

Description

Loads pre-generated .RData files containing the most recent AIRSIS data. If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The files loaded by this function are updated multiple times an hour and contain data for the previous 10 days.

For daily updates covering the most recent 45 days, use `airsis_loadDaily()`.

For data extended more than 45 days into the past, use `airsis_loadAnnual()`.

AIRSIS parameters include the following:

1. PM2.5

Available RData and associated log files can be seen at: [https://haze.airfire.org/monitoring/AIRSIS/RData/latest/](https://haze.airfire.org/monitoring/AIRSIS/RData/latest/)

Usage

```r
airsis_loadLatest(
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL
)
```
### airsis_parseData

**Parse AIRSIS data string**

Raw character data from AIRSIS are parsed into a tibble. The incoming fileString can be read in directly from AIRSIS using `airsis_downloadData()` or from a local file using `readr::read_file()`.

The type of monitor represented by this fileString is inferred from the column names using `airsis_identifyMonitorType()` and appropriate column types are assigned. The character data are then read into a tibble and augmented in the following ways:

1. Longitude, Latitude and any System Voltage values, which are only present in GPS timestamp rows, are propagated forward using a last-observation-carry-forward algorithm.
2. Longitude, Latitude and any System Voltage values, which are only present in GPS timestamp rows, are propagated backwards using a first-observation-carry-backward algorithm.
3. GPS timestamp rows are removed.

### Arguments

- **parameter** Parameter of interest.
- **baseUrl** Base URL for 'daily' AirNow data files.
- **dataDir** Local directory containing 'daily' data files.

### Value

A `ws_monitor` object with AIRSIS data.

### See Also

- `airsis_loadAnnual`
- `airsis_loadDaily`

### Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
    airsis_loadLatest() %>%
    monitor_subset(stateCodes=CONUS) %>%
    monitor_map()
}, silent = FALSE)
## End(Not run)
```

---

**Description**

Raw character data from AIRSIS are parsed into a tibble. The incoming fileString can be read in directly from AIRSIS using `airsis_downloadData()` or from a local file using `readr::read_file()`.

The type of monitor represented by this fileString is inferred from the column names using `airsis_identifyMonitorType()` and appropriate column types are assigned. The character data are then read into a tibble and augmented in the following ways:

1. Longitude, Latitude and any System Voltage values, which are only present in GPS timestamp rows, are propagated forward using a last-observation-carry-forward algorithm.
2. Longitude, Latitude and any System Voltage values, which are only present in GPS timestamp rows, are propagated backwards using a first-observation-carry-backward algorithm.
3. GPS timestamp rows are removed.
Usage

airsis_parseData(fileString)

Arguments

fileString character string containing AIRSIS data as a csv

Value

Dataframe of AIRSIS raw monitor data.

References

Interagency Real Time Smoke Monitoring

Examples

## Not run:
# Fail gracefully if any resources are not available
try({

library(PWFSLSmoke)
fileString <- airsis_downloadData(20150701, 20151231, provider='USFS', unitID='1026')
tbl <- airsis_parseData(fileString)
summary(tbl)

}, silent = FALSE)

## End(Not run)

airsis_qualityControl  Apply Quality Control to raw AIRSIS dataframe

Description

Various QC steps are taken to clean up the incoming raw tibble including:

1. Ensure GPS location data are included in each measurement record.
2. Remove GPS location records.
3. Remove measurement records with values outside of valid ranges.

See the individual airsis_{~QualityControl()} functions for details.

QC parameters that can be passed in the ... include the following valid data ranges as taken from airsis_EBAMQualityControl():

- valid_Longitude=c(-180,180)
- valid_Latitude=c(-90,90)
• remove_Lon_zero = TRUE
• remove_Lat_zero = TRUE
• valid_Flow = c(16.7*0.95, 16.7*1.05)
• valid_AT = c(-Inf, 45)
• valid_RHi = c(-Inf, 50)
• valid_Conc = c(-Inf, 5.000)

Note that appropriate values for QC thresholds will depend on the type of monitor.

Usage
airsis_qualityControl(tbl, ...)

Arguments
tbl single site tibble created by airsis_downloadData()
... additional parameters are passed to type-specific QC functions

Value
Cleaned up tibble of AIRSIS monitor data.

See Also
airsis_EBAMQualityControl
airsis_ESAMQualityControl

---

AQI

Official Air Quality Index Levels, Names and Colors

Description
Official AQI levels, names and colors are provided in a list for easy coloring and labeling.

Usage
AQI

Format
A list with named elements

Details
AQI breaks and associated names and colors
The AQI object contains english language text.
AQI breaks and colors are defined at https://docs.airnowapi.org/aq101
Note

The low end of each break category is used as the breakpoint.

See Also

AQI_en AQI_es

---

**aqiColors**

*Generate AQI Colors*

**Description**

This function uses the `leaflet::colorBin()` function to return a vector or matrix of colors derived from PM2.5 values.

**Usage**

```r
aqiColors(
  x,
  palette = AQI$colors,
  domain = c(0, 1e+06),
  bins = AQI$breaks_24,
  na.color = NA
)
```

**Arguments**

- `x`: vector or matrix of PM2.5 values or a `ws_monitor` object
- `palette`: color palette (see `leaflet::colorBin()`)
- `domain`: range of valid data (see `leaflet::colorBin()`)
- `bins`: color bins (see `leaflet::colorBin()`)
- `na.color`: missing value color (see `leaflet::colorBin()`)

**Value**

A vector or matrix of AQI colors to be used in maps and plots.

**Examples**

```r
wa <- monitor_subset(Northwest_Megafires, stateCodes='WA', tlim=c(20150821,20150828))
colorMatrix <- aqiColors(wa)
time <- wa$data$datetime
pm25 <- wa$data[,,-1]
plot(time, pm25[,1], col=colorMatrix[,1],
    ylim=range(pm25, na.rm=TRUE),
    xlab="2015", ylab="PM 2.5 (ug/m3)", main="Washington State Smoke")
for ( i in seq_along(pm25) ) {
```

Description

Creates a *leaflet* color palette function that can be used to convert monitoring data into vectors of colors.

Usage

```r
aqiPalette(style = "aqi", reverse = FALSE)
```

Arguments

- `style` 
  Palette style, one of ‘aqi’.
- `reverse` 
  Logical specifying whether the colors (or color function) in palette should be used in reverse order.

Value

A function that takes a single parameter x; when called with a vector of numbers, #RRGGBB color strings are returned.

See Also

- ‘leaflet::colorBin()’

Examples

```r
pm25 <- PWFSLSmoke::Carmel_Valley$data[,2]
binned_colors <- aqiPalette("aqi")(pm25)
plot(pm25, col=binned_colors, pch=15, main='Binned Colors')
```
Official Air Quality Index Levels, Names and Colors

Description

Official AQI levels, names and colors are provided in a list for easy coloring and labeling.

Usage

AQI_en

Format

A list with named elements

Details

AQI breaks and associated names and colors (english language)
The AQI_es object contains english language text. It is equivalent to the AQI object and provided for consistency with other language versions.

Note

The low end of each break category is used as the breakpoint.

See Also

AQI AQI_es

Official Air Quality Index Levels, Names and Colors

Description

Official AQI levels, names and colors are provided in a list for easy coloring and labeling.

Usage

AQI_es

Format

A list with named elements
Details

AQI breaks and associated names and colors (Spanish language)
The AQI_es object contains Spanish language text.
AQI breaks and colors are defined at https://docs.airnowapi.org/aq101

Note

The low end of each break category is used as the breakpoint.

See Also

AQI_en AQI

Description

The Camp_Fire dataset provides a quickly loadable version of a ws_monitor object for practicing and code examples.

This dataset was generated on 2021-11-18 by running:

```r
library(PWFSLSmoke)
Camp_Fire <-
  monitor_loadAnnual(2018)
  monitor_subset(stateCodes = 'CA')
  monitor_subset(tlim = c(20181108, 20181123))
save(Camp_Fire, file = "data/Camp_Fire.RData")
```

Usage

Camp_Fire

Format

A ws_monitor object with "meta" and "data" dataframes.
Carmel_Valley  

*Carmel Valley example dataset*

**Description**

In August of 2016, the Soberanes fire in California burned along the Big Sur coast. It was at the time the most expensive wildfire in US history. This dataset contains PM2.5 monitoring data for the monitor in Carmel Valley which shows heavy smoke as well as strong diurnal cycles associated with sea breezes.

The Camp_Fire dataset provides a quickly loadable version of a *ws_monitor* object for practicing and code examples.

**Usage**

Carmel_Valley

**Format**

A *ws_monitor* object with "meta" and "data" dataframes.

---

CONUS  

*CONUS State Codes*

**Description**

State codes for the 48 contiguous states +DC that make up the CONtinental US

**Usage**

CONUS

**Format**

A vector with 49 elements

**Details**

CONUS state codes
**distance**

*Calculate distances between points*

**Description**

This function uses the Haversine formula for calculating great circle distances between points. This formula is purported to work better than the spherical law of cosines for very short distances.

**Usage**

\[
\text{distance(targetLon, targetLat, longitude, latitude)}
\]

**Arguments**

- **targetLon**: longitude (decimal degrees) of the point from which distances are calculated
- **targetLat**: latitude (decimal degrees) of the point from which distances are calculated
- **longitude**: vector of longitudes for which a distance is calculated
- **latitude**: vector of latitudes for which a distance is calculated

**Value**

Vector of distances in km.

**Examples**

# Seattle to Portland airports
SEA_lon <- -122.3088
SEA_lat <- 47.4502
PDX_lon <- -122.5951
PDX_lat <- 45.5898
distance(SEA_lon, SEA_lat, PDX_lon, PDX_lat)

---

**epa_createDataDataFrame**

*Create EPA data dataframe*

**Description**

After additional columns (i.e., `datetime`, and `monitorID`) have been applied to an EPA dataframe, we are ready to extract the PM2.5 values and store them in a data dataframe organized as time-by-monitor.

The first column of the returned dataframe is named `datetime` and contains a POSIXct time in UTC. Additional columns contain data for each separate `monitorID`. 
epa_createMetaDataframe

Create dataframe of EPA site location metadata

Usage

epa_createMetaDataframe(tbl)

Arguments

tbl an EPA raw tibble after metadata enhancement

Value

A data dataframe for use in a ws_monitor object.

Description

After additional columns (i.e. datetime, and monitorID) have been applied to an EPA dataframe, we are ready to pull out site information associated with unique monitorID.

These will be rearranged into a dataframe organized as deployment-by-property with one row for each monitorID.

This site information found in tbl is augmented so that we end up with a uniform set of properties associated with each monitorID. The list of columns in the returned meta dataframe is:

```r
> names(p$meta)
[1] "monitorID"       "longitude"      "latitude"
[4] "elevation"       "timezone"       "countryCode"
[7] "stateCode"       "siteName"       "agencyName"
[10] "countyName"      "msaName"        "monitorType"
[13] "monitorInstrument" "aqsID"          "pwfs1ID"
[16] "pwfs1DataIngestSource" "telemetryAggregator" "telemetryUnitID"
```

Usage

epa_createMetaDataframe(
  tbl,
  pwfs1DataIngestSource = "EPA",
  existingMeta = NULL,
  addGoogleMeta = TRUE
)
epa_createMonitorObject

**Arguments**

- **tbl**: an EPA raw tibble after metadata enhancement
- **pws1DataIngestSource**: identifier for the source of monitoring data, e.g. 'EPA_hourly_88101_2016.zip'
- **existingMeta**: existing 'meta' dataframe from which to obtain metadata for known monitor deployments
- **addGoogleMeta**: logical specifying whether to use Google elevation and reverse geocoding services

**Value**

A meta dataframe for use in a `ws_monitor` object.

**References**

- EPA AirData Pre-Generated Data Files
  - file format description

---

### epa_createMonitorObject

*Download and convert hourly EPA air quality data*

**Description**

Convert EPA data into a `ws_monitor` object, ready for use with all `monitor_~` functions.

**Usage**

```r
epa_createMonitorObject(
  zipFile = NULL,
  zeroMinimum = TRUE,
  addGoogleMeta = TRUE
)
```

**Arguments**

- **zipFile**: absolute path to monitoring data .zip file
- **zeroMinimum**: logical specifying whether to convert negative values to zero
- **addGoogleMeta**: logical specifying whether to use Google elevation and reverse geocoding services

**Value**

A `ws_monitor` object with EPA data.
Note

Before running this function you must first enable spatial data capabilities as in the example.

References

EPA AirData Pre-Generated Data Files
file format description

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
 initializeMazamaSpatialUtils()
 zipFile <- epa_downloadData(2016, "88101", downloadDir = '~/Data/EPA')
 mon <- epa_createMonitorObject(zipFile, addGoogleMeta = FALSE)
}, silent = FALSE)

## End(Not run)
```

description

epa_downloadData  Download EPA air quality data

Description

This function downloads air quality data from the EPA and saves it to a directory.
Available parameter codes include:

1. 44201 – Ozone
2. 42401 – SO2
3. 42101 – CO
4. 42602 – NO2
5. 88101 – PM2.5
6. 88502 – PM2.5
7. 81102 – PM10
8. SPEC – PM2.5
9. WIND – Wind
10. TEMP – Temperature
11. PRESS – Barometric Pressure
12. RH_DP – RH and dewpoint
13. HAPS – HAPs
14. VOCS – VOCs
15. NONOxNOy
epa_downloadData

Usage

epa_downloadData(
    year = NULL,
    parameterCode = "88101",
    downloadDir = tempdir(),
    baseUrl = "https://aqs.epa.gov/aqsweb/airdata/"
)

Arguments

year year  
parameterCode pollutant code  
downloadDir directory where monitoring data .zip file will be saved  
baseUrl base URL for archived daily data

Value

Filepath of the downloaded zip file.

Note

Unzipped CSV files are almost 100X larger than the compressed .zip files.

References

EPA AirData Pre-Generated Data Files

Examples

## Not run:
# Fail gracefully if any resources are not available
try({

zipFile <- epa_downloadData(2016, "88101", '~/Data/EPA')
tbl <- epa_parseData(zipFile, "PM2.5")

}, silent = FALSE)

## End(Not run)
Description

Please use `airsis_loadAnnual` instead of this function. It will soon be deprecated.

Usage

```r
epa_load(
  year = strftime(lubridate::now(tzone = "UTC"), "%Y", tz = "UTC"),
  parameterCode = "88101",
  baseUrl = "https://haze.airfire.org/monitoring/EPA/RData/
)
```

Arguments

- `year`: desired year (integer or character representing YYYY)
- `parameterCode`: pollutant code
- `baseUrl`: base URL for EPA .RData files

Value

A `ws_monitor` object with EPA data for an entire year.

Description

Loads pre-generated .RData files containing annual EPA data.

EPA parameter codes include:

1. 88101 – PM2.5 FRM/FEM Mass (begins in 2008)
2. 88502 – PM2.5 non FRM/FEM Mass (begins in 1998)

Available RData and associated log files can be seen at: https://haze.airfire.org/monitoring/EPA/RData/

Usage

```r
epa_loadAnnual(
  year = NULL,
  parameterCode = NULL,
  baseUrl = "https://haze.airfire.org/monitoring",
  dataDir = NULL
)
```
## epa_parseData

### Description

This function uncompress previously downloaded air quality .zip files from the EPA and reads it into a tibble.

Available parameters include:

1. Ozone
2. SO2
3. CO
4. NO2
5. PM2.5
6. PM10
7. Wind
8. Temperature

### Arguments

- **year**: Desired year (integer or character representing YYYY).
- **parameterCode**: Pollutant code.
- **baseUrl**: Base URL for ‘annual’ EPA data files.
- **dataDir**: Local directory containing ‘annual’ data files.

### Value

A `ws_monitor` object with EPA data.

### References

- [EPA AirData Pre-Generated Data Files](https://www.epa.gov/air-data/air-quality-data-annual-monitoring-data)

### Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  epa_loadAnnual(2000, "88502") %>%
    monitor_subset(stateCodes = 'WA', tlim=c(20000701,20000801)) %>%
    monitor_map()
}, silent = FALSE)
## End(Not run)
```
9. Barometric_Pressure
10. RH_and_Dewpoint
11. HAPs
12. VOCs
13. NONOxNOy

Associated parameter codes include:

1. 44201 – Ozone
2. 42401 – SO2
3. 42101 – CO
4. 42602 – NO2
5. 88101 – PM2.5
6. 88502 – PM2.5
7. 81102 – PM10
8. SPEC – PM2.5
9. WIND – Wind
10. TEMP – Temperature
11. PRESS – Barometric Pressure
12. RH_DP – RH and dewpoint
13. HAPS – HAPs
14. VOCS – VOCs
15. NONOxNOy

Usage

epa_parseData(zipFile = NULL)

Arguments

zipFile absolute path to monitoring data .zip file

Value

Tibble of EPA data.

Note

Unzipped CSV files are almost 100X larger than the compressed .zip files. CSV files are removed after data are read into a dataframe.

References

EPA AirData Pre-Generated Data Files
file format description
esriToken

Examples

## Not run:
# Fail gracefully if any resources are not available
try({

  zipFile <- epa_downloadData(2016, "88101", '~/Data/EPA'
  tbl <- epa_parseData(zipFile, "PM2.5")
}
}, silent = FALSE)
## End(Not run)

esriToken

Token used for ESRI Geocoding Requests

Description

All package functions that interact with ESRI location services will use the token whenever a request is made.

Format

Character string.

See Also

addEsriAddress

generic_downloadData

Download generic data

Description

This function takes a location to a delimited file, gets the file, and returns a string containing the file data.

Usage

generic_downloadData(filePath)

Arguments

filePath

Either a path to a file, or a connection (http(s)://, ftp(s)://).

Details

This function is essentially a wrapper for read_file.
generic_parseData

Value

A character vector of length 1, containing data from the file located at filePath.

Examples

## Not run:
# Fail gracefully if any resources are not available
try({

# make current directory PWFSLSmoke package directory
filePath <- "./localData/airsis_ebam_example-clean.csv"

fileString <- generic_downloadData(filePath)

}, silent = FALSE)
## End(Not run)

generic_parseData Parse generic air quality files

Description

Given a string of delimited file data, this function will parse the file as a table of data and apply some transformations and augmentations as specified by a given configuration list.

Usage

generic_parseData(fileString = NULL, configList = NULL)

Arguments

fileString  Character string of delimited data to parse.
configList  A R list or JSON file containing key-value pairs which affect how the parsing of fileString is handled. If configList is in JSON format, it can be passed in as a file, string, or URL.

Value

A tibble of the data contained in fileString parsed according to parameters in configList. The data is coerced into a format that is more easily convertible into a ws_monitor object at a later point.

Parsing data

Internally, this function uses read_delim to convert fileString into a tibble. If any lines of data cannot be properly parsed, an error will be thrown and the problem lines will be printed.
getEsriToken

Creating a configList

For more information on how to build a configList, see the Rmarkdown document "Working with Generic Data" in the localNotebooks directory.

Examples

```r
filePath <- system.file(
  "extdata", "generic_data_example.csv",
  package = "PWFSLSmoke",
  mustWork = TRUE
)

configPath <- system.file(
  "extdata", "generic_configList_example.json",
  package = "PWFSLSmoke",
  mustWork = TRUE
)

configList <- jsonlite::fromJSON(configPath)
fileString <- generic_downloadData(filePath)
parsedData <- generic_parseData(fileString, configList)
```

getesriToken  Get ESRI Token

Description

Returns the current esriToken.

Usage

getesriToken()  

Value

String.

See Also

addEsriAddress
esriToken
setEsriToken
getGoogleApiKey  

Get Google API Key

Description

Returns the current Google API key.

Usage

getGoogleApiKey()

Value

String.

See Also

addGoogleAddress
addGoogleElevation
googleApiKey
setGoogleApiKey

googleApiKey  

API Key used for Google Geocoding Requests

Description

All package functions that interact with Google location services will use API key whenever a request is made.

Format

Character string.

See Also

addGoogleAddress
addGoogleElevation
initializeMazamaSpatialUtils

Initialize Mazama Spatial Utils

Description

Convenience function that wraps:

```r
logger.setup()
logger.setLevel(WARN)
setSpatialDataDir('~/Data/Spatial')
loadSpatialData('NaturalEarthAdm1')
```

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

Usage

```r
initializeMazamaSpatialUtils(
  spatialDataDir = '~/Data/Spatial',
  stateCodeDataset = "NaturalEarthAdm1"
)
```

Arguments

- `spatialDataDir` directory where spatial datasets are created
- `stateCodeDataset` MazamaSpatialUtils dataset returning ISO 3166-2 alpha-2 stateCodes

See Also

- `{link[logger.setup]`

loadDaily

Load Recent PM2.5 Monitoring Data

Description

Wrapper function to load and combine recent data from AirNow, AIRSIS and WRCC:

```r
airnow <- airnow_loadDaily()
airsis <- airsis_loadDaily()
wrcc <- wrcc_loadDaily()
ws_monitor <- monitor_combine(list(airnow, airsis, wrcc))
```
The daily files are generated once a day, shortly after midnight and contain data for the previous 45 days.
For the most recent data, use `loadLatest()`.
Available RData and associated log files can be seen at: https://haze.airfire.org/monitoring/latest/RData/

Usage

`loadDaily()`

Value

A `ws_monitor` object with PM2.5 monitoring data.

See Also

`loadLatest`

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  ca <- loadDaily() %>% monitor_subset(stateCodes='CA')
}, silent = FALSE)
## End(Not run)
```

---

### loadLatest

**Load Recent PM2.5 Monitoring Data**

Wrapper function to load and combine the most recent data from AirNow, AIRSIS and WRCC:

```r
airnow <- airnow_loadLatest()
airsis <- airsis_loadLatest()
wrcc <- wrcc_loadLatest()
ws_monitor <- monitor_combine(list(airnow, airsis, wrcc))
```

Available RData and associated log files can be seen at: https://haze.airfire.org/monitoring/latest/RData/

Usage

`loadLatest()`
monitor_aqi

Value

A ws_monitor object with PM2.5 monitoring data.

See Also

airsis_loadDaily

Examples

## Not run:
# Fail gracefully if any resources are not available
try({
  ca <- loadLatest() %>% monitor_subset(stateCodes='CA')
}, silent = FALSE)
## End(Not run)

---

monitor_aqi Calculate hourly NowCast-based AQI values

Description

Nowcast and AQI algorithms are applied to the data in the ws_monitor object.

Usage

monitor_aqi(
  ws_monitor,
  aqiParameter = "pm25",
  nowcastVersion = "pm",
  includeShortTerm = FALSE
)

Arguments

ws_monitor ws_monitor object

aqiParameter parameter type; used to define reference breakpointsTable

nowcastVersion character identity specifying the type of nowcast algorithm to be used. See ?monitor_nowcast for more information.

includeShortTerm calcluate preliminary values starting with the 2nd hour

References

https://docs.airnowapi.org/aq101
## Examples
```r
## Not run:
library(PWFSLSmoke)

ws_monitor <- monitor_subset(Northwest_Megafires, tlim=c(20150815,20150831))
aqi <- monitor_aqi(ws_monitor)
monitor_timeseriesPlot(aqi, monitorID=aqi$meta$monitorID[1], ylab="PM25 AQI")

## End(Not run)
```

### monitor_asDataframe

**Return Monitor Data in a Single Dataframe**

**Description**

Creates a dataframe with data from a `ws_monitor` object, essentially flattening the object. This is especially useful when monitoring data will be shared with non-R users working with spreadsheets. The returned dataframe will contain data from the monitor specified with `monitorID`.

The number of data columns in the returned dataframe can include all metadata as well as additional calculated values.

By default, the following, core columns are included in the dataframe:

- `utcTime` UTC datetime
- `localTime` local datetime
- `pm25` PM2.5 values in ug/m3

Any column from `ws_monitor$meta` may be included in the vector of `metaColumns`.

The following additional columns of data may be included by adding one of the following to the vector of `extraColumns`:

- `aqi` hourly AQI values as calculated with `monitor_aqi()`
- `nowcast` hourly Nowcast values as calculatd with `monitor_nowcast()`
- `dailyAvg` daily average PM2.5 values as calculated with `monitor_dailyStatistic()`

**Usage**

```r
monitor_asDataframe(
  ws_monitor, 
  monitorID = NULL,
  extraColumns = NULL,
  metaColumns = NULL,
  tlim = NULL
)
```
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws_monitor</td>
<td>ws_monitor object</td>
</tr>
<tr>
<td>monitorID</td>
<td>monitor ID of interest (not needed if ws_monitor contains only one monitor)</td>
</tr>
<tr>
<td>extraColumns</td>
<td>optional vector of additional data columns to generate</td>
</tr>
<tr>
<td>metaColumns</td>
<td>optional vector of column names from ws_monitor$meta</td>
</tr>
<tr>
<td>tlim</td>
<td>optional vector with start and end times (integer or character representing YYYY-MM-DD[HH] or POSIXct)</td>
</tr>
</tbody>
</table>

Value

A dataframe version of a ws_monitor object.

Note

The tlim argument is interpreted as localtime, not UTC.

See Also

- monitor_aqi
- monitor_nowcast
- monitor_dailyStatistic

Examples

```r
library(PWFSLSmoke)

wa <- monitor_subset(Northwest_Megafires, stateCodes='WA')

Omak_df <- monitor_asDataframe(wa, monitorID='530470013_01',
                                extraColumns=c('nowcast','dailyAvg'),
                                metaColumns=c('aqsID','siteName','timezone'),
                                tlim=c(20150801,20150901))

dplyr::glimpse(Omak_df)
```

Description

Collapses data from all the monitors in ws_monitor into a single-monitor ws_monitor object using the function provided in the FUN argument. The single-monitor result will be located at the mean longitude and latitude unless longitude and latitude parameters are specified. Any columns of meta that are common to all monitors will be retained in the returned ws_monitor meta.
monitor_collapse

Usage

```r
monitor_collapse(
    ws_monitor,
    longitude = NULL,
    latitude = NULL,
    monitorID = "generated_id",
    FUN = mean,
    na.rm = TRUE,
    ...
)
```

Arguments

- `ws_monitor`: `ws_monitor` object.
- `longitude`: Longitude of the collapsed monitoring station.
- `latitude`: Latitude of the collapsed monitoring station.
- `monitorID`: Monitor ID assigned to the collapsed monitoring station.
- `FUN`: Function to be applied to all the monitors at a single time index.
- `na.rm`: Logical specifying whether NA values should be ignored when FUN is applied.
- `...`: Additional arguments to be passed on to the `apply()` function.

Value

A `ws_monitor` object with meta and data that for the the collapsed single monitor

Note

After `FUN` is applied, values of `+Inf` and `-Inf` are converted to `NA`. This is a convenience for the common case where `FUN=min` or `FUN=max` and some of the timesteps have all missing values. See the R documentation for `min` for an explanation.

Examples

```r
library(PWFSLSmoke)
N_M <- Northwest_Megafires
# monitor_leaflet(N_M) # to identify Spokane monitorIDs

Spokane <- monitor_subsetBy(N_M, stringr::str_detect(N_M$meta$monitorID, '^53063'))
Spokane_min <- monitor_collapse(Spokane, monitorID='Spokane_min', FUN=min)
Spokane_max <- monitor_collapse(Spokane, monitorID='Spokane_max', FUN=max)

monitor_timeseriesPlot(Spokane, tlim=c(20150619,20150626),
    style='gnats', shadedNight=TRUE)
monitor_timeseriesPlot(Spokane_max, col='red', type='s', add=TRUE)
monitor_timeseriesPlot(Spokane_min, col='blue', type='s', add=TRUE)
title('Spokane Range of PM2.5 Values, June 2015')
```
**monitor_combine**  

Combine List of ws_monitor Objects into Single ws_monitor Object

---

**Description**

Combines a list of one or more `ws_monitor` objects into a single `ws_monitor` object by merging the `meta` and `data` dataframes from each object in `monitorList`. When `monitorList` contains only two `ws_monitor` objects the `monitor_combine()` function can be used to extend time ranges for `monitorIDs` that are found in both `ws_monitor` objects. This can be used to 'grow' a `ws_monitor` object by appending subsequent months or years. (Note, however, that this can be CPU intensive process.)

**Usage**

```r
monitor_combine(monitorList)
```

**Arguments**

- `monitorList`  
  list containing one or more `ws_monitor` objects

**Value**

A `ws_monitor` object combining all monitoring data from `monitorList`.

**Examples**

```r
# Fail gracefully if any resources are not available
try{

library(PWFSLSmoke)
initializeMazamaSpatialUtils()

monitorList <- list()
monitorList[[1]] <- airsis_createMonitorObject(20160701, 20161231, 'USFS', '1031')
monitorList[[2]] <- airsis_createMonitorObject(20160701, 20161231, 'USFS', '1032')
monitorList[[3]] <- airsis_createMonitorObject(20160701, 20161231, 'USFS', '1033')
monitorList[[4]] <- airsis_createMonitorObject(20160701, 20161231, 'USFS', '1034')
ws_monitor <- monitor_combine(monitorList)

if ( interactive() ) {
  monitor_leaflet(ws_monitor)
}
}
```

}, silent = FALSE)
Description

Creates a bar plot showing daily average PM 2.5 values for a specific monitor in a `ws_monitor` object. Each bar is colored according to its AQI category.

This function is a wrapper around `base::barplot` and any arguments to that function may be used.

Each 'day' is the midnight-to-midnight period in the monitor local timezone. When `tlim` is used, it is converted to the monitor local timezone.

Usage

```r
monitor_dailyBarplot(
  ws_monitor,
  monitorID = NULL,
  tlim = NULL,
  minHours = 18,
  gridPos = "",
  gridCol = "black",
  gridLwd = 0.5,
  gridLty = "solid",
  labels_x_nudge = 0,
  labels_y_nudge = 0,
  ...
)
```

Arguments

- `ws_monitor`: `ws_monitor` object
- `monitorID`: monitor ID for a specific monitor in `ws_monitor` (optional if `ws_monitor` only has one monitor)
- `tlim`: optional vector with start and end times (integer or character representing YYYY-MM-DD[HH] or POSIXct)
- `minHours`: minimum number of valid data hours required to calculate each daily average
- `gridPos`: position of grid lines either 'over', 'under' ('' for no grid lines)
- `gridCol`: color of grid lines (see graphical parameter 'col')
- `gridLwd`: line width of grid lines (see graphical parameter 'lwd')
- `gridLty`: type of grid lines (see graphical parameter 'lty')
- `labels_x_nudge`: nudge x labels to the left
- `labels_y_nudge`: nudge y labels down
- `...`: additional arguments to be passed to `barplot()`
Details

The `labels_x_nudge` and `labels_y_nudge` can be used to tweak the date labeling. Units used are the same as those in the plot.

Examples

```r
library(PWFSLSmoke)
N_M <- monitor_subset(Northwest_Megafires, tlim = c(20150715, 20150930))
main <- "Daily Average PM2.5 for Omak, WA"
monitor_dailyBarplot(N_M, monitorID = "530470013_01", main = main,
                     labels_x_nudge = 1)
addAQILegend(fill = rev(AQI$colors), pch = NULL)
```

Description

Calculates daily statistics for each monitor in `ws_monitor`.

Usage

```r
monitor_dailyStatistic(
  ws_monitor,
  FUN = get("mean"),
  dayStart = "midnight",
  na.rm = TRUE,
  minHours = 18
)
```

Arguments

- `ws_monitor`: `ws_monitor` object
- `FUN`: function used to collapse a day’s worth of data into a single number for each monitor in the `ws_monitor` object
- `dayStart`: one of `sunset`, `midnight`, `sunrise`
- `na.rm`: logical value indicating whether NA values should be ignored
- `minHours`: minimum number of valid data hours required to calculate each daily statistic

Details

Sunrise and sunset times are calculated based on the first monitor encountered. This should be accurate enough for all use cases involving co-located monitors. Monitors from different regions should have daily statistics calculated separately.
A ws_monitor object with daily statistics for the local timezone.

Note

Note that the incoming ws_monitor object should have UTC (GMT) times and that this function calculates daily statistics based on local (clock) time. If you choose a date range based on UTC times this may result in an insufficient number of hours in the first and last daily records of the returned ws_monitor object.

The returned ws_monitor object has a daily time axis where each datetime is set to the beginning of each day, 00:00:00, local time.

Examples

N_M <- monitor_subset(Northwest_Megafires, tlim=c(20150801,20150831))
WinthropID <- '530470010_01'
TwispID <- '530470009_01'
MethowValley <- monitor_subset(N_M,
    tlim=c(20150801,20150831),
    monitorIDs=c(WinthropID,TwispID))
MethowValley_dailyMean <- monitor_dailyStatistic(MethowValley,
    FUN=get('mean'),
    dayStart='midnight')

# Get the full Y scale
monitor_timeseriesPlot(MethowValley, style='gnats', col='transparent')
monitor_timeseriesPlot(MethowValley, monitorID=TwispID,
    style='gnats', col='forestgreen', add=TRUE)
monitor_timeseriesPlot(MethowValley, monitorID=WinthropID,
    style='gnats', col='purple', add=TRUE)
monitor_timeseriesPlot(MethowValley_dailyMean, monitorID=TwispID,
    type='s', lwd=2, col='forestgreen', add=TRUE)
monitor_timeseriesPlot(MethowValley_dailyMean, monitorID=WinthropID,
    type='s', lwd=2, col='purple', add=TRUE)
addAQILines()
addAQILegend("topleft", lwd=1, pch=NULL)
title("Winthrop & Twisp, Washington Daily Mean PM2.5, 2015")

Description

Calculates daily statistics for each monitor in ws_monitor.
monitor_dailyStatisticList

Usage

monitor_dailyStatisticList(
  ws_monitor,
  FUN = get("mean"),
  dayStart = "midnight",
  na.rm = TRUE,
  minHours = 18
)

Arguments

ws_monitor   ws_monitor object
FUN           function used to collapse a day's worth of data into a single number for each
              monitor in the ws_monitor object
dayStart      one of sunset|midnight|sunrise
na.rm         logical value indicating whether NA values should be ignored
minHours      minimum number of valid data hours required to calculate each daily statistic

Details

Splits the ws_monitor object by timezone and applies the monitor_dailyStatistic() function separately for each timezone. See monitor_dailyStatistic for more details.

The results are returned as a list of ws_monitor objects with each element of the list named with the associated timezone. Note that each ws_monitor$data$datetime will be in local time. This is desirable as it ensures proper date formatting in tables and plots.

You should not attempt to reassemble a single ws_monitor object from the elements in this list.

Value

A list of ws_monitor objects with daily statistics for each local timezone.

References

monitor_dailyStatistic

Examples

# Fail gracefully if any resources are not available
try({

  library(PWFSLSmoke)

  airnow <- airnow_loadLatest()
  nw <- monitor_subset(airnow, stateCodes = c('WA','OR','ID','MT'))
  dailyList <- monitor_dailyStatisticList(nw)

  monitor_leaflet(dailyList[["America/Los_Angeles"]])
})
monitor_leaflet(dailyList[["America/Boise"]])
monitor_leaflet(dailyList[["America/Denver"]])
}


monitor_dailyThreshold

*Calculate Daily Counts of Values At or Above a Threshold*

**Description**

Calculates the number of hours per day each monitor in `ws_monitor` was at or above a given threshold

**Usage**

```r
monitor_dailyThreshold(
  ws_monitor, 
  threshold = "unhealthy", 
  dayStart = "midnight", 
  minHours = 0, 
  na.rm = TRUE 
)
```

**Arguments**

- `ws_monitor`  
  *ws_monitor* object
- `threshold`  
  AQI level name (e.g. "unhealthy") or numerical threshold at or above which a measurement is counted
- `dayStart`  
  one of "sunset|midnight|sunrise"
- `minHours`  
  minimum number of hourly observations required
- `na.rm`  
  logical value indicating whether NA values should be ignored

**Details**

**NOTE:** The returned counts include values at OR ABOVE the given threshold; this applies to both categories and values. For example, passing a threshold argument = "unhealthy" will return a daily count of values that are unhealthy, very unhealthy, or extreme (i.e. >= 55.5), as will passing a threshold argument = 55.5.

AQI levels for threshold argument = one of "good|moderate|usg|unhealthy|very unhealthy|extreme"

Sunrise and sunset times are calculated based on the first monitor encountered. This should be accurate enough for all use cases involving co-located monitors. Monitors from different regions should have daily statistics calculated separately.

The returned `ws_monitor` object has a daily time axis where each time is set to 00:00, local time.
monitor_distance

Value

A `ws_monitor` object with a daily count of hours at or above threshold.

Examples

```r
library(PWFSLSmoke)

N_M <- monitor_subset(Northwest_Megafires, tlim=c(20150801,20150831))
Twisp <- monitor_subset(N_M, monitorIDs='530470009_01')
Twisp_daily <- monitor_dailyThreshold(Twisp, "unhealthy", dayStart='midnight', minHours=1)
monitor_timeseriesPlot(Twisp_daily, type='h', lwd=6, ylab="Hours")
title("Twisp, Washington Hours per day Above 'Unhealthy', 2015")
```

---

**monitor_distance**  
*Calculate distances from monitors to a location of interest*

Description

This function returns the distances (km) between monitoring sites and a location of interest. These distances can be used to create a mask identifying monitors within a certain radius of the location of interest.

Usage

```r
monitor_distance(ws_monitor, longitude, latitude)
```

Arguments

- `ws_monitor`  
  `ws_monitor` object

- `longitude`  
  longitude of the location of interest

- `latitude`  
  latitude of the location of interest

Value

Vector of of distances (km).

See Also

`distance`
Examples

```r
library(PWFSLSmoke)

N_M <- Northwest_Megafires
# Walla Walla
WW_lon <- -118.330278
WW_lat <- 46.065
distance <- monitor_distance(N_M, WW_lon, WW_lat)
closestIndex <- which(distance == min(distance))
distance[closestIndex]
N_M$meta[closestIndex,]
```

---

`monitor_downloadAnnual`

*Download annual PM2.5 monitoring data*

Description

Downloads 'annual' data files into `dataDir` for later use. Downloaded versions of PWFSL monitoring .RData files allow users to work with the package without access to the internet. Once data are downloaded to `dataDir`, any of the data loading functions can be called with the `dataDir` argument to replace internet downloads with local file access.

The recommended directory for PWFSL monitoring data is "~/data/monitoring/RData".

For data during the last 45 days, use `monitor_downloadDaily()`.

For the most recent data, use `monitor_downloadLatest()`.

Currently supported parameters include the following:

1. `PM2.5`

Available RData files can be seen at: https://haze.airfire.org/monitoring/latest/RData/

Usage

```r
monitor_downloadAnnual(
  year = NULL,
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring",
  dataDir = "~/Data/monitoring/RData",
  ...
)
```

Arguments

- `year`: Desired year (integer or character representing YYYY).
- `parameter`: Parameter of interest.
- `baseUrl`: Base URL for data files.
- `dataDir`: Local directory in which to save the data file.
- `...`: Additional arguments passed to `download.file`.
### Description

Downloads 'daily' data files into `dataDir` for later use. Downloaded versions of PWFSL monitoring `.RData` files allow users to work with the package without access to the internet. Once data are downloaded to `dataDir`, any of the data loading functions can be called with the `dataDir` argument to replace internet downloads with local file access.

The recommended directory for PWFSL monitoring data is `~/data/monitoring/RData`.

For the most recent data, use `monitor_downloadLatest()`.

For data extended more than 45 days into the past, use `monitor_downloadAnnual()`.

Currently supported parameters include the following:

1. **PM2.5**

Available `.RData` files can be seen at: [https://haze.airfire.org/monitoring/latest/RData/](https://haze.airfire.org/monitoring/latest/RData/)

### Usage

```r
monitor_downloadDaily(
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData/",
  dataDir = "~/Data/monitoring/RData",
  ...
)
```

---

**See Also**

- `monitor_loadDaily`

**Examples**

```r
# Fail gracefully if any resources are not available
try({
  library(PWFSLSmoke)
  monitor_loadAnnual(2018) %>%
    monitor_subset(stateCodes = "WA", tlim = c(20180701, 20181001)) %>%
    monitor_timeseriesPlot(style = 'gnats')
}, silent = FALSE)
```
Download recent PM2.5 monitoring data

Description

Downloads 'latest' data files into dataDir for later use. Downloaded versions of PWFSL monitoring .RData files allow users to work with the package without access to the internet. Once data are downloaded to dataDir, any of the data loading functions can be called with the dataDir argument to replace internet downloads with local file access.

The recommended directory for PWFSL monitoring data is "~/data/monitoring/RData".

For daily updates covering the most recent 45 days, use monitor_downloadDaily().

For data extended more than 45 days into the past, use monitor_downloadAnnual().

Currently supported parameters include the following:

1. PM2.5

Available RData files can be seen at: https://haze.airfire.org/monitoring/latest/RData/
Usage

```r
monitor_downloadLatest(
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData/",
  dataDir = "~/Data/monitoring/RData",
  ...
)
```

Arguments

- `parameter`: Parameter of interest.
- `baseUrl`: Base URL for data files.
- `dataDir`: Local directory in which to save the data file.
- `...`: Additional arguments passed to `download.file`.

See Also

- `monitor_loadDaily`

Examples

```r
# Fail gracefully if any resources are not available
try({
  library(PWFSLSmoke)
  monitor_loadLatest() %>%
    monitor_subset(stateCodes=CONUS) %>%
    monitor_map()
}, silent = FALSE)
```

---

**monitor_dygraph**

Create Interactive Time Series Plot

Description

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

Usage

```r
monitor_dygraph(
  ws_monitor,
  title = "title",
  ylab = "PM2.5 Concentration",
```
monitor_dygraph

tlim = NULL,
rollPeriod = 1,
showLegend = TRUE
)

Arguments

ws_monitor  ws_monitor object
title  title text
ylab  title for the y axis
tlim  optional vector with start and end times (integer or character representing YYYY-MM-DD[HH])
rollPeriod  rolling mean to be applied to the data
showLegend  logical to toggle display of the legend

Value

Initiates the interactive dygraph plot in RStudio’s ‘Viewer’ tab.

Examples

# Fail gracefully if any resources are not available
try({
  library(PWFSLSmoke)

  # Napa Fires -- October, 2017
  ca <- airnow_load(2017) %>%
    monitor_subset(tlim=c(20171001,20171101), stateCodes='CA')

  Vallejo <- monitor_subset(ca, monitorIDs='060950004_01')

  Napa_Fires <- monitor_subsettingByDistance(
    ca,
    longitude = Vallejo$meta$longitude,
    latitude = Vallejo$meta$latitude,
    radius = 50
  )

  if ( interactive() ) {
    monitor_dygraph(Napa_Fires, title='Napa Fires in California, Oct. 2017')
  }
}, silent = FALSE)
monitor_extractDataFrame

Extract dataframes from ws_monitor objects

Description

These functions are convenient wrappers for extracting the dataframes that comprise a ws_monitor object. These functions are designed to be useful when manipulating data in a pipe chain.

Below is a table showing equivalent operations for each function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitor_extractData</td>
<td>ws_monitor[[&quot;data&quot;]]</td>
</tr>
<tr>
<td>monitor_extractMeta</td>
<td>ws_monitor[[&quot;meta&quot;]]</td>
</tr>
</tbody>
</table>

Usage

monitor_extractData(ws_monitor)

monitor_extractMeta(ws_monitor)

Arguments

ws_monitor  ws_monitor object to extract dataframe from.

Value

A dataframe from the given ws_monitor object

Examples

library(PWFSLSmoke)

ws_monitor <- Northwest_Megafires

data <- ws_monitor %>%
  monitor_subset(
    stateCodes = "WA",
    tlim = c(20150801, 20150831)
  ) %>%
  monitor_extractData()

meta <- ws_monitor %>%
  monitor_subset(
    stateCodes = "WA",
    tlim = c(20150801, 20150831)
  ) %>%
  monitor_extractMeta()
monitor_getCurrentStatus

*Get current status of monitors*

Description

This function augments the metadata from a `ws_monitor` object with summarized and aggregate data from the `ws_monitor` object.

Usage

```r
monitor_getCurrentStatus(
    ws_monitor,
    endTime = NULL,
    monitorURLBase = "http://tools.airfire.org/monitoring/v4/#!/?monitors="
)
```

Arguments

- `ws_monitor` - `ws_monitor` object.
- `endTime` - Time to which the status of the monitors will be current. By default, it is the most recent time in `ws_monitor`. This time can be given as a POSIXct time, or a string/numeric value in ymd format (e.g., 20190301). This time converted to UTC.
- `monitorURLBase` - A URL prefix pointing to where more information about a monitor can be found. By default, it points to the AirFire monitoring site.

Value

A table containing the current status information for all the monitors in `ws_monitor`.

"Last" and "Previous"

The goal of this function is to provide useful information about what happened recently with each monitor in the provided `ws_monitor` object. Monitors sometimes don’t consistently report data, however, and it’s not useful to have NA’s reported when there is still valid data at other times. To address this, `monitor_getCurrentStatus` uses `last` and `previous` valid times. These are the time when a monitor most recently reported data, and the most recent time of valid data before that, respectively. By reporting on these times, the function ensures that valid data is returned and provides information on how outdated this information is.
Calculating latency

According to https://docs.airnowapi.org/docs/HourlyDataFactSheet.pdf a datum assigned to 2pm represents the average of data between 2pm and 3pm. So, if we check at 3:15pm and see that we have a value for 2pm but not 3pm then the data are completely up-to-date with zero latency.

`monitor_getCurrentStatus()` defines latency as the difference in time between the given time index and the next most recent time index. If there is no more recent time index, then the difference is measured to the given `endTime` parameter. These differences are recorded in hours.

For example, if the recorded values for a monitor are [16.2, 15.8, 16.4, NA, 14.0, 12.5, NA, NA, 13.3, NA], then the last valid time index is 9, and the previous valid time index is 6. The last latency is then 1 (hour), and the previous latency is 3 (hours).

Summary data

The table created by `monitor_getCurrentStatus()` includes summary information for the data part of the given `ws_monitor` object. The summaries included are listed below with a description:

- `yesterday_pm25_24hr` Daily AQI value for the day prior to `endTime`
- `last_nowcast_1hr` Last valid NowCast measurement
- `last_PM2.5_1hr` Last valid raw PM2.5 measurement
- `last_PM2.5_3hr` Mean of the last valid raw PM2.5 measurement with the preceding two measurements
- `previous_nowcast_1hr` Previous valid NowCast measurement
- `previous_PM2.5_1hr` Previous valid raw PM2.5 measurement
- `previous_PM2.5_3hr` Mean of the previous valid raw PM2.5 measurement with the preceding two measurements

It should be noted that all averages are "right-aligned", meaning that the three hour mean of data at time \( n \) will comprise of the data at times \([n-2, n-1, n]\). Data for \( n-2 \) and \( n-1 \) is not guaranteed to exist, so a three hour average may include 1 to 3 data points.

Event flags

The table created by `monitor_getCurrentStatus()` also includes binary flags representing events that may have occurred for a monitor within the bounds of the specified end time and data in the `ws_monitor` object. Each flag is listed below with its corresponding meaning:

- `last_nowcastLevel` NowCast level at the last valid time
- `previous_nowcastLevel` NowCast level at the previous valid time
- `NR6` Monitor not reporting for more than 6 hours
- `NEW6` New monitor reporting in the last 6 hours
- `USG6` NowCast level increased to Unhealthy for Sensitive Groups in the last 6 hours
- `U6` NowCast level increased to Unhealthy in the last 6 hours
- `VU6` NowCast level increased to Very Unhealthy in the last 6 hours
- `HAZ6` NowCast level increased to Hazardous in the last 6 hours
- `MOD6` NowCast level decreased to Moderate or Good in the last 6 hours
- `MAL6` Monitor malfunctioning the last 6 hours (not currently implemented)
Examples

# Fail gracefully if any resources are not available
try({

library(PWFSLSmoke)

ws_monitor <- monitor_loadLatest() %>% monitor_subset(stateCodes = "WA")
statusTbl <- monitor_getCurrentStatus(ws_monitor)

}, silent = FALSE)

Description

Calculates and returns daily means for a monitor. If either startdate or enddate is NULL, a single value is returned for that date.

Usage

monitor_getDailyMean(
  ws_monitor,
  monitorID = NULL,
  startdate = NULL,
  enddate = NULL
)

Arguments

ws_monitor  ws_monitor object
monitorID    monitor ID of interest
startdate    desired start date (integer or character in Ymd format or POSIXct)
enddate      desired end date (integer or character in Ymd format or POSIXct)

Value

A dataframe of daily means.
Examples

```r
library(PWFSLSmoke)

monitor_getDailyMean(
  PWFSLSmoke::Carmel_Valley,
  startdate = "2016-08-01",
  enddate = "2016-08-08"
)
```

---

**monitor_hourlyBarplot**  
*Create Hourly Barplot*

**Description**

Creates a bar plot showing hourly PM 2.5 values for a specific monitor in a *ws_monitor* object. Colors are assigned to one of the following styles:

- **AQI** – hourly values colored with AQI colors using AQI 24-hour breaks
- **brownScaleAQI** – hourly values colored with brownscale colors using AQI 24-hour breaks
- **grayScaleAQI** – hourly values colored grayscale colors using AQI 24-hour breaks

**Usage**

```r
monitor_hourlyBarplot(
  ws_monitor,
  monitorID = NULL,
  tlim = NULL,
  localTime = TRUE,
  style = "AQI",
  shadedNight = TRUE,
  gridPos = "",
  gridCol = "black",
  gridLwd = 0.5,
  gridLty = "solid",
  labels_x_nudge = 0,
  labels_y_nudge = 0,
  dayCol = "black",
  dayLwd = 2,
  dayLty = "solid",
  hourCol = "black",
  hourLwd = 1,
  hourLty = "solid",
  hourInterval = 6,
  ...
)
```
Arguments

ws_monitor  ws_monitor object
monitorID   monitor ID for a specific monitor in ws_monitor (optional if ws_monitor only
            has one monitor)
tlim        optional vector with start and end times (integer or character representing YYYYMM-
            MDD[HH])
localTime   logical specifying whether tlim is in local time or UTC
style       named style specification ("AirFire")
shadedNight add nighttime shading
gridPos     position of grid lines either 'over', 'under' ("" for no grid lines)
gridCol     grid color
gridLwd     grid line width
gridLty     grid line type
labels_x_nudge nudge x labels to the left
labels_y_nudge nudge y labels down
dayCol      day boundary color
dayLwd      day boundary line width (set to 0 to omit day lines)
dayLty      day boundary type
hourCol     hour boundary color
hourLwd     hour boundary line width (set to 0 to omit hour lines)
hourLty     hour boundary type
hourInterval interval for hour boundary lines
...         additional arguments to be passed to barplot()

Details

The labels_x_nudge and labels_y_nudge can be used to tweak the date labeling. Units used are
the same as those in the plot.

Examples

library(PWFLSSmoke)

C_V <- monitor_subset(Carmel_Valley, tlim = c(2016080800, 2016081023),
                      timezone = "America/Los_Angeles")

monitor_hourlyBarplot(C_V, main = "1-Hourly Average PM2.5",
                       labels_x_nudge = 1, labels_y_nudge = 0)
monitor_isEmpty  

Test for an Empty ws_monitor Object

Description

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

Usage

```r
monitor_isEmpty(ws_monitor)
```

Arguments

- `ws_monitor` : `ws_monitor` object

Value

- `TRUE` if no monitors exist in `ws_monitor`, `FALSE` otherwise.

Examples

```r
monitor_isEmpty(Carmel_Valley)
```

monitor_isMonitor  

Test for a correct structure of ws_monitor Object

Description

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

- monitorID – per deployment unique ID
- longitude – decimal degrees E
- latitude – decimal degrees N
- elevation – height above sea level in meters
- timezone – olson timezone
- countryCode – ISO 3166-1 alpha-2

Usage

```r
monitor_isMonitor(ws_monitor)
```
monitor_isolate

Arguments

ws_monitor  ws_monitor object

Value

TRUE if ws_monitor has the correct structure, FALSE otherwise.

Examples

monitor_isEmpty(Carmel_Valley)

monitor_isolate  Isolate Individual Monitors

Description

Filters ws_monitor according to the parameters passed in. If any parameter is not specified, that parameter will not be used in the filtering.

After filtering, each monitorID found in ws_monitor is extracted and its data dataframe is restricted to the times from when that monitor first datapoint until its last datapoint.

This function is useful when ws_monitor objects are created for mobile monitors that are deployed to different locations in different years.

Usage

monitor_isolate(
  ws_monitor,
  xlim = NULL,
  ylim = NULL,
  tlim = NULL,
  monitorIDs = NULL,
  stateCodes = NULL,
  timezone = "UTC"
)

Arguments

ws_monitor  ws_monitor object
xlim  optional vector with low and high longitude limits
ylim  optional vector with low and high latitude limits
tlim  optional vector with start and end times (integer or character representing YYYY-MM-DD[HH] or POSIXct)
monitorIDs  optional vector of monitorIDs
stateCodes  optional vector of stateCodes
timezone  Olson timezone passed to parse_datetime when parsing numeric tlim
monitor_isTidy

Value
A list of isolated ws_monitor objects.

See Also
monitor_subset

Examples

```r
N_M <- Northwest_Megafires
# monitor_leaflet(N_M) # to identify Spokane monitorIDs
Spokane <- monitor_subsetBy(N_M, stringr::str_detect(N_M$meta$monitorID, '^53063'))
Spokane$meta$monitorID
monitorList <- monitor_isolate(Spokane)
names(monitorList)
```

monitor_isTidy

Check if data is tidy-formatted ws_monitor data

Description
Verifies that the given data can be treated as tidy-formatted "ws_monitor" data. This is done by verifying that the data is a tibble data.frame object with columns for information in all 'ws_monitor' objects.

Usage

```r
monitor_isTidy(data = NULL)
```

Arguments
data Data to validate.

Value
True if the data is in a recognized 'Tidy' format, otherwise False.

Examples

```r
ws_monitor <- monitor_subset(
    Northwest_Megafires,
    monitorIDs = c('530470009_01', '530470010_01')
)

ws_monTidy <- monitor_toTidy(ws_monitor)
monitor_isTidy(ws_monTidy)

## Not run:
monitor_isTidy(ws_monitor)
```
## monitor_join

**Merge Data for Monitors with Shared monitorIDs**

### Description

For each monitor in `monitorIDs`, an attempt is made to merge the associated data from `ws_monitor1` and `ws_monitor2` and.

This is useful when the same `monitorID` appears in different `ws_monitor` objects representing different time periods. The returned `ws_monitor` object will cover both time periods.

### Usage

```r
monitor_join(ws_monitor1 = NULL, ws_monitor2 = NULL, monitorIDs = NULL)
```

### Arguments

- `ws_monitor1`: `ws_monitor` object
- `ws_monitor2`: `ws_monitor` object
- `monitorIDs`: vector of shared monitorIDs that are to be joined together. Defaults to all shared monitorIDs.

### Value

A `ws_monitor` object with merged timeseries.

### Examples

```r
## Not run:
Jul <- monitor_subset(Northwest_Megafires, tlim=c(2015070100,2015073123), timezone='America/Los_Angeles')
Aug <- monitor_subset(Northwest_Megafires, tlim=c(2015080100,2015083123), timezone='America/Los_Angeles')
Methow_Valley <- monitor_join(Jul, Aug, monitorIDs=c('530470010_01','530470009_01'))
## End(Not run)
```
monitor_leaflet

Leaflet interactive map of monitoring stations

Description

This function creates interactive maps that will be displayed in RStudio's 'Viewer' tab. The `slice` argument is used to collapse a `ws_monitor` timeseries into a single value. If `slice` is an integer, that row index will be selected from the `ws_monitor$data` dataframe. If `slice` is a function (unquoted), that function will be applied to the timeseries with the argument `na.rm=TRUE` (e.g. `max(..., na.rm=TRUE)`).

If `slice` is a user defined function it will be used with argument `na.rm=TRUE` to collapse the time dimension. Thus, user defined functions must accept `na.rm` as an argument.

Usage

```r
monitor_leaflet(
  ws_monitor,
  slice = get("max"),
  breaks = AQI$breaks_24,
  colors = AQI$colors,
  labels = AQI$names,
  legendTitle = "Max AQI Level",
  radius = 10,
  opacity = 0.7,
  maptype = "terrain",
  popupInfo = c("siteName", "monitorID", "elevation")
)
```

Arguments

- `ws_monitor`: `ws_monitor` object
- `slice`: either a time index or a function used to collapse the time axis – defaults to `get("max")`
- `breaks`: set of breaks used to assign colors
- `colors`: a set of colors for different levels of air quality data determined by `breaks`
- `labels`: a set of text labels, one for each color
- `legendTitle`: legend title
- `radius`: radius of monitor circles
- `opacity`: opacity of monitor circles
- `maptype`: optional name of leaflet ProviderTiles to use, e.g. "terrain"
- `popupInfo`: a vector of column names from `ws_monitor$meta` to be shown in a popup window
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

Invisibly returns a leaflet map of class "leaflet".

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try(

# Napa Fires -- October, 2017
car <- airnow_load(2017) %>%
  monitor_subset(tlim = c(20171001,20171101), stateCodes = 'CA')
  v_low <- AQI$breaks_24[5]
CA_very_unhealthy_monitors <- monitor_subset(car, vlim = c(v_low, Inf))
monitor_leaflet(CA_very_unhealthy_monitors,
  legendTitle = "October, 2017",
  maptype = "toner")

), silent = FALSE)
## End(Not run)
```

---

**monitor_load**  
*Load PM2.5 monitoring data*

**Description**

Loads monitoring data for a given time range. Data from AirNow, AIRSIS and WRCC are combined into a single *ws_monitor* object.

Archival datasets are joined with 'daily' and 'latest' datasets as needed to satisfy the requested date range.
Usage

monitor_load(
    startdate = NULL,
    enddate = NULL,
    monitorIDs = NULL,
    parameter = "PM2.5",
    baseUrl = "https://haze.airfire.org/monitoring",
    dataDir = NULL,
    aqsPreference = "airnow"
)

Arguments

startdate  Desired start date (integer or character in ymd[hms] format or POSIXct).
enddate  Desired end date (integer or character in ymd[hms] format or POSIXct).
monitorIDs  Optional vector of monitorIDs.
parameter  Parameter of interest.
baseUrl  Base URL for data files.
dataDir  Local directory containing monitoring data files.
aqsPreference  Preferred data source for AQS data when annual data files are available from both 'epa' and 'airnow'.

Value

A ws_monitor object with PM2.5 monitoring data.

Note

Joining datasets is a computationally expensive task when many monitors are involved. It is highly recommend that monitorIDs be specified when loading recent data with this function.

See Also

loadDaily
loadLatest

Examples

## Not run:
# Fail gracefully if any resources are not available
try({
    ca <- monitor_load(20170601,20171001) %>% monitor_subset(stateCodes='CA')
}, silent = FALSE)
## End(Not run)
Description

Wrapper function to load and combine annual data from AirNow, AIRSIS and WRCC.
If dataDir is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.
The annual files loaded by this function are updated on the 15’th of each month and cover the period from the beginning of the year to the end of the last month.
For data during the last 45 days, use monitor_loadDaily().
For the most recent data, use monitor_loadLatest().
Currently supported parameters include the following:

1. PM2.5

Available RData files can be seen at: https://haze.airfire.org/monitoring/

Usage

```r
monitor_loadAnnual(
  year = NULL,
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring",
  dataDir = NULL,
  aqsPreference = "airnow"
)
```

Arguments

- **year**: Desired year (integer or character representing YYYY).
- **parameter**: Parameter of interest.
- **baseUrl**: Base URL for data files.
- **dataDir**: Local directory containing 'daily' data files.
- **aqsPreference**: Preferred data source for AQS data when annual data files are available from both ‘epa’ and ‘airnow’.

Value

A `ws_monitor` object with PM2.5 monitoring data.

See Also

- `monitor_loadDaily`
- `monitor_loadLatest`
monitor_loadDaily

Examples

## Not run:
# Fail gracefully if any resources are not available
try({
    monitor_loadAnnual(2014) %>%
        monitor_subset(stateCodes = 'MT', tlim = c(20140801, 20140901)) %>%
        monitor_map()
}, silent = FALSE)

## End(Not run)

monitor_loadDaily  Load recent PM2.5 monitoring data

Description

Wrapper function to load and combine recent data from AirNow, AIRSIS and WRCC:

```r
airnow <- airnow_loadDaily()
airsis <- airsis_loadDaily()
wrcc <- wrcc_loadDaily()
ws_monitor <- monitor_combine(list(airnow, airsis, wrcc))
```

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The daily files loaded by this function are updated once a day, shortly after midnight and contain data for the previous 45 days.

For the most recent data, use `monitor_loadLatest()`.

For data extended more than 45 days into the past, use `monitor_load()`.

Currently supported parameters include the following:

1. PM2.5

Available RData files can be seen at: https://haze.airfire.org/monitoring/latest/RData/

Usage

```r
monitor_loadDaily(
    parameter = "PM2.5",
    baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
    dataDir = NULL
)
```
monitor_loadLatest

Arguments

- parameter: Parameter of interest.
- baseUrl: Base URL for 'daily' AirNow data files.
- dataDir: Local directory containing 'daily' data files.

Value

A ws_monitor object with PM2.5 monitoring data.

See Also

- monitor_load
- monitor_loadDaily
- monitor_loadAnnual

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({

  monitor_loadDaily() %>%
    monitor_subset(stateCodes=CONUS) %>%
    monitor_map()
}, silent = FALSE)

## End(Not run)
```

Description

Wrapper function to load and combine recent data from AirNow, AIRSIS and WRCC:

```r
airnow <- airnow_loadLatest()
airsis <- airsis_loadLatest()
wrcc <- wrcc_loadLatest()
ws_monitor <- monitor_combine(list(airnow, airsis, wrcc))
```

If dataDir is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The files loaded by this function are updated multiple times an hour and contain data for the previous 10 days.

For daily updates covering the most recent 45 days, use `monitor_loadDaily()`. 
For data extended more than 45 days into the past, use `monitor_load()`.
Currently supported parameters include the following:

1. PM2.5

Available RData files can be seen at: https://haze.airfire.org/monitoring/latest/RData/

Usage

```r
monitor_loadLatest(
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData/",
  dataDir = NULL
)
```

Arguments

- **parameter**: Parameter of interest.
- **baseUrl**: Base URL for 'daily' AirNow data files.
- **dataDir**: Local directory containing 'daily' data files.

Value

A `ws_monitor` object with PM2.5 monitoring data.

See Also

- `monitor_load`
- `monitor_loadAnnual`
- `monitor_loadDaily`

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  monitor_loadLatest() %>%
    monitor_subset(stateCodes=CONUS) %>%
    monitor_map()
}, silent = FALSE)

## End(Not run)
```
monitor_map

Static map of monitoring stations

Description

Creates a map of monitoring stations in a given ws_monitor object. Individual monitor timeseries are reduced to a single value by applying the function passed in as slice to the entire timeseries of each monitor with na.rm=TRUE. These values are then plotted over a map of the United States. Any additional arguments specified in ‘...’ are passed on to the points() function.

If slice is an integer, it will be used as an index to pull out a single timestep.

If slice is a function (not a function name) it will be used with argument na.rm=TRUE to collapse the time dimension. Thus, any user defined functions passed in as slice must accept na.rm as a parameter.

Usage

```r
monitor_map(
  ws_monitor,
  slice = get("max"),
  breaks = AQI$breaks_24,
  colors = AQI$colors,
  pch = par("pch"),
  cex = par("cex"),
  stateCol = "grey60",
  stateLwd = 2,
  countyCol = "grey70",
  countyLwd = 1,
  add = FALSE,
  ...
)
```

Arguments

- **ws_monitor**: `ws_monitor` object
- **slice**: either a time index or a function used to collapse the time axis
- **breaks**: set of breaks used to assign colors
- **colors**: set of colors must be one less than the number of breaks
- **pch**: Plot symbols used to draw points on the map.
- **cex**: the amount that the points will be magnified on the map
- **stateCol**: color for state outlines on the map
- **stateLwd**: width for state outlines
- **countyCol**: color for county outline on the map
- **countyLwd**: width for county outlines
- **add**: logical specifying whether to add to the current plot
- **...**: additional arguments passed to `maps::map()` such as 'projection' or 'parameters'
Details

Using a single number for the breaks argument will result in the use of quantiles to determine a set of breaks appropriate for the number of colors.

Examples

```r
library(PWFSLSmoke)
N_M <- monitor_subset(Northwest_Megafires, tlim = c(20150821,20150828))
monitor_map(N_M, cex = 2)
addAQILegend()
```

monitor_nowcast  Apply Nowcast Algorithm to ws_monitor Object

Description

A Nowcast algorithm is applied to the data in the ws_monitor object. The version argument specifies the minimum weight factor and number of hours to be considered in the calculation.

Available versions include:

1. pm: hours=12, weight=0.5
2. pmAsian: hours=3, weight=0.1
3. ozone: hours=8, weight=NA

The default, version='pm', is appropriate for typical usage.

Usage

```r
monitor_nowcast(ws_monitor, version = "pm", includeShortTerm = FALSE)
```

Arguments

- `ws_monitor`: `ws_monitor` object
- `version`: character identity specifying the type of nowcast algorithm to be used
- `includeShortTerm`: calculte preliminary NowCast values starting with the 2nd hour

Details

This function calculates the current hour’s NowCast value based on the value for the given hour and the previous N-1 hours, where N is the number of hours corresponding to the version argument (see Description above). For example, if version=pm, then the NowCast value for Hour 12 is based on the data from Hours 1-12.

The function requires valid data for at least two of the three latest hours; NA’s are returned for hours where this condition is not met.
By default, the function will not return a valid value until the Nth hour. If `includeShortTerm=TRUE`, the function will return a valid value after only the 2nd hour (provided, of course, that both hours are valid).

Calculated Nowcast values are truncated to the nearest .1 ug/m³ for 'pm' and nearest .001 ppm for 'ozone' regardless of the precision of the data in the incoming `ws_monitor` object.

**Value**

A `ws_monitor` object with data that have been processed by the Nowcast algorithm.

**References**

https://en.wikipedia.org/wiki/Nowcast_(Air_Quality_Index)
https://forum.airnowtech.org/t/the-nowcast-for-ozone-and-pm/172
https://forum.airnowtech.org/t/the-aqi-equation/169
https://forum.airnowtech.org/t/how-does-airnow-handle-negative-hourly-concentrations/143

**Examples**

```r
library(PWFSLSmoke)

N_M <- monitor_subset(Northwest_Megafires, tlim=c(20150815,20150831))
Omak <- monitor_subset(N_M, monitorIDs=c('530470013_01'))
Omak_nowcast <- monitor_nowcast(Omak, includeShortTerm=TRUE)
monitor_timeseriesPlot(Omak, type='l', lwd=2)
monitor_timeseriesPlot(Omak_nowcast, add=TRUE, type='l', col='purple', lwd=2)
addAQILines()
addAQILegend(lwd=1, pch=NULL)
legend("topleft", lwd=2, col=c('black','purple'), legend=c('hourly','nowcast'))
title("Omak, Washington Hourly and Nowcast PM2.5 Values in August, 2015")
# Zooming in to check on handling of missing values
monitor_timeseriesPlot(Omak, tlim=c(20150823,20150825))
monitor_timeseriesPlot(Omak_nowcast, tlim=c(20150823,20150825), pch=16,col='red',type='b', add=TRUE)
abline(v=Omak$data[is.na(Omak$data[,2]),1])
title("Missing values")
```

**Description**

This function uses confusion matrix analysis to calculate different measures of predictive performance for every timeseries found in predicted with respect to the observed values found in the single timeseries found in observed.

The requested metric is returned in a dataframe organized with one row per monitor, all available metrics are returned.
Usage

```r
monitor_performance(
  predicted, 
  observed, 
  t1, 
  t2, 
  metric = NULL, 
  FPCost = 1, 
  FNCost = 1
)
```

Arguments

- `predicted`: `ws_monitor` object with predicted data
- `observed`: `ws_monitor` object with observed data
- `t1`: value used to classify `predicted` measurements
- `t2`: threshold used to classify `observed` measurements
- `metric`: confusion matrix metric to be used
- `FPCost`: cost associated with false positives (type II error)
- `FNCost`: cost associated with false negatives (type I error)

Value

Dataframe of monitors vs named measure of performance.

See Also

- `monitor_performanceMap`
- `skill_confusionMatrix`

Examples

```r
# Fail gracefully if any resources are not available
try({

  library(PWFSLSmoke)

  # If daily avg data were the prediction and Spokane were
  # the observed, which WA State monitors had skill?

  wa <- airnow_loadAnnual(2017) %>% monitor_subset(stateCodes='WA')
  wa_dailyAvg <- monitor_dailyStatistic(wa, mean)
  Spokane_dailyAvg <- monitor_subset(wa_dailyAvg, monitorIDs='530630021_01')

  threshold <- AQI$breaks_24[4] # Unhealthy
  performanceMetrics <- monitor_performance(wa_dailyAvg, 
                                           Spokane_dailyAvg, 
                                           ...)
})
```
monitor_performanceMap

threshold, threshold)

monitorIDs <- rownames(performanceMetrics)
mask <- performanceMetrics$heidkeSkill &
  !is.na(performanceMetrics$heidkeSkill)

skillfulIDs <- monitorIDs[mask]
skillful <- monitor_subset(wa_dailyAvg, monitorIDs=skillfulIDs)

monitor_leaflet(skillful)
}), silent = FALSE)

---

monitor_performanceMap

*Create map of monitor prediction performance*

Description

This function uses *confusion matrix* analysis to calculate different measures of predictive performance for every timeseries found in *predicted* with respect to the observed values found in the single timeseries found in *observed*.

Using a single number for the *breaks* argument will cause the algorithm to use quantiles to determine breaks.

Usage

```r
monitor_performanceMap(
  predicted,
  observed,
  threshold = AQI$breaks_24[3],
  cex = par("cex"),
  sizeBy = NULL,
  colorBy = "heidikeSkill",
  breaks = c(-Inf, 0.5, 0.6, 0.7, 0.8, Inf),
  paletteFunc = grDevices::colorRampPalette(RColorBrewer::brewer.pal(length(breaks),
                                        "Purples")[-1]),
  showLegend = TRUE,
  legendPos = "topright",
  stateCol = "grey60",
  stateLwd = 2,
  countyCol = "grey70",
  countyLwd = 1,
  add = FALSE,
  ...
)
```

monitor_performanceMap

Arguments

- **predicted**: `ws_monitor` object with predicted values
- **observed**: `ws_monitor` object with observed values
- **threshold**: value used to classify predicted and observed measurements
- **cex**: the amount that the points will be magnified on the map
- **sizeBy**: name of the metric used to create relative sizing
- **colorBy**: name of the metric used to create relative colors
- **breaks**: set of breaks used to assign colors or a single integer used to provide quantile based breaks - Must also specify the `colorBy` parameter
- **paletteFunc**: a palette generating function as returned by `colorRampPalette`
- **showLegend**: logical specifying whether to add a legend (default: `TRUE`)
- **legendPos**: legend position passed to `legend()`
- **stateCol**: color for state outlines on the map
- **stateLwd**: width for state outlines
- **countyCol**: color for county outline on the map
- **countyLwd**: width for county outlines
- **add**: logical specifying whether to add to the current plot
- **...**: additional arguments to be passed to the `maps::map()` function such as graphical parameters (see `code?par`)

Details

Setting either `sizeBy` or `colorBy` to NULL will cause the size/colors to remain constant.

See Also

- `monitor_performance`

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try(

try{

library(PWFSLSmoke)

# Napa Fires -- October, 2017
ca <- airnow_load(2017) %>%
  monitor_subset(tlim=c(20171001,20171101), stateCodes='CA')
Vallejo <- monitor_subset(ca, monitorIDs='060950004_01')
Napa_Fires <- monitor_subsetByDistance(ca,
  longitude = Vallejo$meta$longitude,
  latitude = Vallejo$meta$latitude,
  radius = 50)
monitor_performanceMap(ca, Vallejo, cex = 2)
```

title('Heidke Skill of monitors predicting another monitor.')
}

## End(Not run)

---

**monitor_print**

*Print monitor data as CSV*

### Description

Prints out the contents of the `ws_monitor` object as CSV. By default, the output is a text string with "human readable" CSV that includes both meta and data. When saved as a file, this format is useful for point-and-click spreadsheet users who want to have everything on a single sheet.

To obtain machine parseable CSV strings you can use `metaOnly` or `dataOnly` which are mutually exclusive but which return CSV strings that can be automatically ingested.

By default, the CSV formatted text is printed to the console as well as returned invisibly but not saved to a file unless `saveFile` is specified.

### Usage

```r
monitor_print(
  ws_monitor,
  saveFile = NULL,
  metaOnly = FALSE,
  dataOnly = FALSE,
  quietly = FALSE
)
```

### Arguments

- **ws_monitor**: `ws_monitor` object
- **saveFile**: optional filename where CSV will be written
- **metaOnly**: flag specifying whether to return `ws_monitor$meta` only as a machine parseable CSV
- **dataOnly**: flag specifying whether to return `ws_monitor$data` only as a machine parseable CSV
- **quietly**: do not print to console, just return the string representation of the CSV

### Note

The `monitor_writeCSV` function is an alias for this function but defaults to `quietly = TRUE`. 
**monitor_reorder**

Reorder a `ws_monitor` object

Description

This function is a convenience function that merely wraps the `monitor_subset` function which reorders as well as subsets.

Usage

```r
monitor_reorder(ws_monitor, monitorIDs = NULL, dropMonitors = FALSE)
```

Arguments

- `ws_monitor`: `ws_monitor` object
- `monitorIDs`: Optional vector of monitor IDs used to reorder the meta and data dataframes.
- `dropMonitors`: Logical specifying whether to remove monitors with no data.

Value

A `ws_monitor` object reordered to match `monitorIDs`.

**monitor_replaceData**

Replace `ws_monitor` Data with Another Value

Description

Use an R expression to identify values for replacement.

The R expression given in `filter` is used to identify elements in `ws_monitor$data` that should be replaced. Typical usage would include

1. replacing negative values with 0
2. replacing unreasonably high values with NA

Expressions should use `data` for the left hand side of the comparison.
Usage

    monitor_replaceData(ws_monitor, filter, value)

Arguments

    ws_monitor  ws_monitor object
    filter      an R expression used to identify values for replacement
    value       replacement value

Examples

library(PWFSLSmoke)

wa <- monitor_subset(Northwest_Megafires, stateCodes = 'WA')
wa_zero <- monitor_replaceData(wa, data < 0, 0)

---

monitor_rollingMean  Calculate Rolling Means

Description

    Calculates rolling means for each monitor in ws_monitor using the openair::rollingMean() function

Usage

    monitor_rollingMean(ws_monitor, width = 8, data.thresh = 75, align = "center")

Arguments

    ws_monitor  ws_monitor object
    width       number of periods to average (e.g. for hourly data, width = 24 calculates 24-hour rolling means)
    data.thresh minimum number of valid observations required as a percent of width; NA is returned if insufficient valid data to calculate mean
    align       alignment of averaging window relative to point being calculated; one of "left|center|right"

Details

    • align = 'left': Forward roll, using hour of interest and the (width-1) subsequent hours (e.g. 3-hr left-aligned roll for Hr 5 will consist of average of Hrs 5, 6 and 7)
    • align = 'right': Backwards roll, using hour of interest and the (width-1) prior hours (e.g. 3-hr right-aligned roll for Hr 5 will consist of average of Hrs 3, 4 and 5)
    • align = 'center' for odd width: Average of hour of interest and (width-1)/2 on either side (e.g. 3-hr center-aligned roll for Hr 5 will consist of average of Hrs 4, 5 and 6)
    • align = 'center' for even width: Average of hour of interest and (width/2)-1 hours prior and width/2 hours after (e.g. 4-hr center-aligned roll for Hr 5 will consist of average of Hrs 4, 5, 6 and 7)
Value

A `ws_monitor` object with data that have been processed by a rolling mean algorithm.

Examples

```r
library(PWFSLSmoke)
N_M <- Northwest_Megafires
wa_smoky <- monitor_subset(N_M, stateCodes='WA', tlim=c(20150801, 20150808), vlim=c(100,Inf))
wa_smoky_3hr <- monitor_rollingMean(wa_smoky, width=3, align="center")
wa_smoky_24hr <- monitor_rollingMean(wa_smoky, width=24, align="right")

monitor_timeseriesPlot(wa_smoky, type='l', shadedNight=TRUE)
monitor_timeseriesPlot(wa_smoky_3hr, type='l', col='red', add=TRUE)
monitor_timeseriesPlot(wa_smoky_24hr, type='l', col='blue', lwd=2, add=TRUE)
legend('topright', c("hourly","3-hourly","24-hourly"),
      col=c('black','red','blue'), lwd=c(1,1,2))
title('Smoky Monitors in Washington -- August, 2015')
```

Description

Creates a plot of individual (e.g. hourly) and rolling mean PM2.5 values for a specific monitor.

Usage

```r
monitor_rollingMeanPlot(
  ws_monitor,
  monitorID = NULL,
  width = 3,
  align = "center",
  data.thresh = 75,
  tlim = NULL,
  ylim = NULL,
  localTime = TRUE,
  shadedNight = FALSE,
  aqiLines = TRUE,
  gridHorizontal = FALSE,
  grid24hr = FALSE,
  grid3hr = FALSE,
  showLegend = TRUE
)
```
monitor_rollingMeanPlot

Arguments

ws_monitor  
ws_monitor object

monitorID  
Monitor ID for a specific monitor in the ws_monitor object (optional if only one monitor in the ws_monitor object).

width  
Number of periods to average (e.g. for hourly data, width = 24 plots 24-hour rolling means).

align  
Alignment of averaging window relative to point being calculated; one of "left|center|right".

data.thresh  
Minimum number of valid observations required as a percent of width; NA is returned if insufficient valid data to calculate. mean

tlim  
Optional vector with start and end times (integer or character representing YYYY-MM-DD[HH]).

ylim  
y limits for the plot.

localTime  
Logical specifying whether tlim is in local time or UTC.

shadedNight  
Add nighttime shading.

aqiLines  
Horizontal lines indicating AQI levels.

gridHorizontal  
Add dashed horizontal grid lines.

grid24hr  
Add dashed grid lines at day boundaries.

grid3hr  
Add dashed grid lines every 3 hours.

showLegend  
Include legend in top left.

Details

- align = "left": Forward roll, using hour of interest and the (width-1) subsequent hours (e.g. 3-hr left-aligned roll for Hr 5 will consist of average of Hrs 5, 6 and 7)
- align = "right": Backwards roll, using hour of interest and the (width-1) prior hours (e.g. 3-hr right-aligned roll for Hr 5 will consist of average of Hrs 3, 4 and 5)
- align = "center" for odd width: Average of hour of interest and (width-1)/2 on either side (e.g. 3-hr center-aligned roll for Hr 5 will consist of average of Hrs 4, 5 and 6)
- align = "center" for even width: Average of hour of interest and (width/2)-1 hours prior and width/2 hours after (e.g. 4-hr center-aligned roll for Hr 5 will consist of average of Hrs 4, 5, 6 and 7)

Note

This function attempts to provide a 'publication ready' rolling mean plot.

Examples

library(PWFSLSmoke)

N_M <- Northwest_Megafires
Roseburg <- monitor_subset(N_M, tlim = c(20150821, 20150831),
                            monitorIDs = c("410190002_01"))
monitor_rollingMeanPlot(Roseburg, shadedNight = TRUE)
**monitor_scaleData**  
*Scale ws_monitor Data*

**Description**

Scale the data in a *ws_monitor* object by multiplying it with *factor*.

**Usage**

```r
monitor_scaleData(ws_monitor, factor)
```

**Arguments**

- `ws_monitor`: *ws_monitor* object  
- `factor`: numeric used to scale the data

**Value**

A *ws_monitor* object with scaled data.

**Examples**

```r
library(PWFSLSmoke)

wa <- monitor_subset(Northwest_Megafires, stateCodes='WA')
w_zero <- monitor_scaleData(wa, 3.4)
```

---

**monitor_stamenmap**  
*Create a static map of ws_monitor object*

**Description**

Plots a map showing *ws_monitor* locations and values.

# Available maptypes include:

- terrain
- toner
- watercolor

See `staticmap_getStamenmapBrick` for details.

If `centerLon`, `centerMap` or `zoom` are not specified, appropriate values will be calculated using data from the *ws_monitor*$meta dataframe.
Usage

monitor_stamenmap(
  ws_monitor,
  slice = get("max"),
  breaks = AQI$breaks_24,
  colors = AQI$colors,
  width = 640,
  height = 640,
  centerLon = NULL,
  centerLat = NULL,
  zoom = NULL,
  maptype = "terrain",
  grayscale = FALSE,
  rasterBrick = NULL,
  cex = par("cex") * 2,
  pch = 16,
  ...
)

Arguments

ws_monitor  ws_monitor object
slice    either a time index or a function used to collapse the time axis – defaults to
get('max')
breaks    set of breaks used to assign colors
colors    a set of colors for different levels of air quality data determined by breaks
width    width of image, in pixels
height    height of image, in pixels
centerLon    map center longitude
centerLat    map center latitude
zoom    map zoom level
maptype    map type
grayscale    logical, if TRUE the colored map tile is rendered into a black & white image
rasterBrick    optional RGB rasterBrick object returned from staticmap_get~Brick)
cex    character expansion for points
pch    plotting character for points
... arguments passed on to staticmap_plotRasterBrick() (e.g. destfile, cex, pch, etc.)

Value

Plots a map loaded from arcGIS REST with points for each monitor.
**monitor_staticmap**

Create a static map of *ws_monitor* object

### Description

Plots a map showing *ws_monitor* locations and values.

See [staticmap_getRasterBrick](#) for a list of available maptype options.

If `centerLon`, `centerMap` or `zoom` are not specified, appropriate values will be calculated using data from the `ws_monitor$meta` dataframe.

### Usage

```r
monitor_staticmap(
  ws_monitor,
  slice = get("max"),
  breaks = AQI$breaks_24,
  colors = AQI$colors,
  width = 640,
  height = 640,
  centerLon = NULL,
  centerLat = NULL,
  zoom = NULL,
  maptype = "terrain",
  grayscale = FALSE,
  rasterBrick = NULL,
)```

### Examples

```r
# Fail gracefully if any resources are not available
try({
  library(PWFSLSmoke)
  N_M <- Northwest_Megafires
  # monitor_leaflet(N_M) # to identify Spokane monitorIDs
  Spokane <- monitor_subsetBy(N_M, stringr::str_detect(N_M$meta$monitorID, "^53063"))
  Spokane <- monitor_subset(Spokane, tlim=c(20150815, 20150831))
  monitor_stamenmap(Spokane)
}, silent = FALSE)
```
cex = par("cex") * 2,
pch = 16,
...
)

Arguments

- **ws_monitor**: `ws_monitor` object
- **slice**: either a time index or a function used to collapse the time axis – defaults to get('max')
- **breaks**: set of breaks used to assign colors
- **colors**: a set of colors for different levels of air quality data determined by `breaks`
- **width**: width of image, in pixels
- **height**: height of image, in pixels
- **centerLon**: map center longitude
- **centerLat**: map center latitude
- **zoom**: map zoom level
- **maptype**: map type
- **grayscale**: logical, if TRUE the colored map tile is rendered into a black & white image
- **rasterBrick**: optional RGB rasterBrick object returned from `staticmap_get~Brick`
- **cex**: character expansion for points
- **pch**: plotting character for points
- **...**: arguments passed on to `staticmap_plotRasterBrick()` (e.g. `destfile`, `cex`, `pch`, etc.)

Value

A plot with a basemap and colored dots for each monitor.

See Also

- `staticmap_getStamenmapBrick`
- `staticmap_plotRasterBrick`

Examples

```r
# Fail gracefully if any resources are not available
try({
  library(PWFLSsmoke)
  
  N_M <- Northwest_Megafires
  # monitor_leaflet(N_M) # to identify Spokane monitorIDs
  Spokane <- monitor_subsetBy(N_M, stringr::str_detect(N_M$meta$monitorID,'^53063'))
  Spokane <- monitor_subset(Spokane, tlim=c(20150815, 20150831))
})
```
monitor_subset

monitor_staticmap(Spokane)

}, silent = FALSE)

---

**monitor_subset**  
*Subset ws_monitor Object*

**Description**

Creates a subset *ws_monitor* based on one or more optional input parameters. If any input parameter is not specified, that parameter will not be used to subset *ws_monitor*.

**Usage**

```r
monitor_subset(
  ws_monitor,
  xlim = NULL,
  ylim = NULL,
  tlim = NULL,
  vlim = NULL,
  monitorIDs = NULL,
  stateCodes = NULL,
  countryCodes = NULL,
  dropMonitors = TRUE,
  timezone = "UTC"
)
```

**Arguments**

- `ws_monitor`  
  *ws_monitor* object

- `xlim`  
  optional vector with low and high longitude limits

- `ylim`  
  optional vector with low and high latitude limits

- `tlim`  
  optional vector with start and end times (integer or character representing YYYYMMDD[HH] or POSIXct)

- `vlim`  
  optional vector with low and high data value limits

- `monitorIDs`  
  optional vector of monitor IDs used to filter the data

- `stateCodes`  
  optional vector of state codes used to filter the data

- `countryCodes`  
  optional vector of country codes used to filter the data

- `dropMonitors`  
  flag specifying whether to remove monitors with no data

- `timezone`  
  Olson timezone passed to `parseDatetime` when parsing numeric `tlim`
monitor_subsetBy

Details

By default, this function will return a ws_monitor object whose data dataframe has the same number of columns as the incoming dataframe, unless any of the columns consist of all NAs, in which case such columns will be removed (e.g. if there are no valid data for a specific monitor after subsetting by tlim or vlim). If dropMonitors=FALSE, columns that consist of all NAs will be retained.

Value

A ws_monitor object with a subset of ws_monitor.

Examples

```r
library(PWFSLSmoke)

N_M <- monitor_subset(Northwest_Megafires, tlim=c(20150701,20150731))
xlim <- c(-124.73, -122.80)
ylim <- c(47.20, 48.40)
Olympic_Peninsula <- monitor_subset(N_M, xlim, ylim)

monitor_map(Olympic_Peninsula, cex=2)
rect(xlim[1], ylim[1], xlim[2], ylim[2], col=adjustcolor('black',0.1))
```

Description

The incoming ws_monitor object is filtered according to filter. Either meta data or actual data can be filtered.

Usage

```r
monitor_subsetBy(ws_monitor, filter)
```

Arguments

- `ws_monitor` : ws_monitor object
- `filter` : a filter to use on the ws_monitor object

Value

A ws_monitor object with a subset of the input ws_monitor object.
Examples

library(PWFSLSmoke)

N_M <- Northwest_Megafires
boise_tz <- monitor_subsetBy(N_M, timezone == 'America/Boise')
boise_tz_very_unhealthy <- monitor_subsetBy(boise_tz, data > AQI$breaks_24[5])
boise_tz_very_unhealthy$meta$siteName

monitor_subsetByDistance

Subset ws_monitor Object by Distance from Target Location

Description

Subsets ws_monitor to include only those monitors (or grid cells) within a certain radius of a target location. If no monitors (or grid cells) fall within the specified radius, ws_monitor$data and ws_monitor$meta are set to NULL.

When count is used, a ws_monitor object is created containing up to count monitors, ordered by increasing distance from the target location. Thus, note that the number of monitors (or grid cells) returned may be less than the specified count value if fewer than count monitors (or grid cells) are found within the specified radius of the target location.

Usage

monitor_subsetByDistance(
  ws_monitor,
  longitude = NULL,
  latitude = NULL,
  radius = 50,
  count = NULL
)

Arguments

ws_monitor  ws_monitor object
longitude    target longitude from which the radius will be calculated
latitude     target latitude from which the radius will be calculated
radius       distance (km) of radius from target location – default=300
count        number of grid cells to return

Value

A ws_monitor object with monitors near a location.

See Also

monitorDistance
Examples

```r
# Fail gracefully if any resources are not available
try({

library(PWFSLSmoke)

# Napa Fires -- October, 2017
cal <- airnow_loadAnnual(2017) %>%
  monitor_subset(tlim=c(20171001,20171101), stateCodes='CA')
Vallejo <- monitor_subset(cal, monitorIDs='060950004_01')
Napa_Fires <- monitor_subsetByDistance(cal,
  longitude = Vallejo$meta$longitude,
  latitude = Vallejo$meta$latitude,
  radius = 50)

if ( interactive() ) {
  monitor_leaflet(Napa_Fires)
}
}, silent = FALSE)
```

---

### `monitor_subsetData`

**Subset ws_monitor Object 'data' Dataframe**

**Description**

Subsets a `ws_monitor` object’s data dataframe by removing any monitors that lie outside the specified ranges of time and values and that are not mentioned in the list of monitorIDs.

If `tlim` or `vlim` is not specified, it will not be used in the subsetting.

Intended for use by the `monitor_subset` function.

**Usage**

```r
monitor_subsetData(
  data,
  tlim = NULL,
  vlim = NULL,
  monitorIDs = NULL,
  dropMonitors = FALSE,
  timezone = "UTC"
)
```
monitor_subsetMeta

Arguments

- **data**: `ws_monitor` object data dataframe
- **tlim**: optional vector with start and end times (integer or character representing YYYY-MM-DD[HH] or POSIXct)
- **vlim**: optional vector with low and high data value limits
- **monitorIDs**: optional vector of monitorIDs
- **dropMonitors**: flag specifying whether to remove columns – defaults to FALSE
- **timezone**: Olson timezone passed to `parseDateTime` when parsing numeric tlim

Details

By default, filtering by tlim or vlim will always return a dataframe with the same number of columns as the incoming dataframe. If `dropMonitors=TRUE`, columns will be removed if there are not valid data for a specific monitor after subsetting.

Filtering by vlim is open on the left and closed on the right, i.e.

\[ x > vlim[1] \& x \leq vlim[2] \]

Value

A `ws_monitor` object data dataframe, or NULL if filtering removes all monitors.

---

monitor_subsetMeta  Subset `ws_monitor` Object 'meta' Dataframe

Description

Subsets the `ws_monitor$data` dataframe by removing any monitors that lie outside the geographical ranges specified (i.e. outside of the given longitudes and latitudes and/or states) and that are not mentioned in the list of monitorIDs.

If any parameter is not specified, that parameter will not be used in the subsetting.

Intended for use by the monitor_subset function.

Usage

```r
monitor_subsetMeta(
  meta,
  xlim = NULL,
  ylim = NULL,
  stateCodes = NULL,
  countryCodes = NULL,
  monitorIDs = NULL
)
```
monitor_timeAverage

Arguments

- **meta**: `ws_monitor` object meta dataframe
- **xlim**: optional vector with low and high longitude limits
- **ylim**: optional vector with low and high latitude limits
- **stateCodes**: optional vector of stateCodes
- **countryCodes**: optional vector of countryCodes
- **monitorIDs**: optional vector of monitorIDs

Details

Longitudes must be specified in the domain \([-180,180]\).

Value

A `ws_monitor` object meta dataframe, or `NULL` if filtering removes all monitors.

Description

This function extracts the data dataframe from `ws_monitor` object and renames the 'datetime' column so that it can be processed by the `openair` package's `timeAverage()` function. (See that function for details.)

Usage

`monitor_timeAverage(ws_monitor, ...)`

Arguments

- **ws_monitor**: `ws_monitor` object
- **...**: additional arguments to be passed to `openair::timeAverage()`

Value

A `ws_monitor` object with data that have been processed by `openair::timeAverage()`.

Examples

```r
library(PWFSLSmoke)

C_V <- monitor_subset(Carmel_Valley, tlim=c(2016080800,2016081023),
                       timezone='America/Los_Angeles')
C_V_3hourly <- monitor_timeAverage(C_V, avg.time="3 hour")
head(C_V$data, n=15)
head(C_V_3hourly$data, n=5)
```
monitor_timeInfo  

Get time related information for a monitor

Description

Calculate the local time for the monitor, as well as sunrise, sunset and solar noon times, and create several temporal masks.

The returned dataframe will have as many rows as the length of the incoming UTC time vector and will contain the following columns:

- **localStdTime_UTC** – UTC representation of local standard time
- **daylightSavings** – logical mask = TRUE if daylight savings is in effect
- **localTime** – local clock time
- **sunrise** – time of sunrise on each localTime day
- **sunset** – time of sunset on each localTime day
- **solarnoon** – time of solar noon on each localTime day
- **day** – logical mask = TRUE between sunrise and sunset
- **morning** – logical mask = TRUE between sunrise and solarnoon
- **afternoon** – logical mask = TRUE between solarnoon and sunset
- **night** – logical mask = opposite of day

Usage

```r
monitor_timeInfo(ws_monitor = NULL, monitorID = NULL)
```

Arguments

- **ws_monitor** 
  - `ws_monitor` object.
- **monitorID** 
  - Monitor ID for a specific monitor in `ws_monitor` – optional if `ws_monitor` only has one monitor.

Details

While the **lubridate** package makes it easy to work in local timezones, there is no easy way in R to work in "Local Standard Time" (LST) as is often required when working with air quality data. EPA regulations mandate that daily averages be calculated based on LST.

The **localStdTime_UTC** is primarily for use internally and provides an important tool for creating LST daily averages and LST axis labeling.

Value

A dataframe with times and masks.
Examples

library(PWFSLSmoke)

carmel <- monitor_subset(Carmel_Valley, tlim = c(20160801,20160810))

# Create timeInfo object for this monitor
ti <- monitor_timeInfo(carmel)

# Subset the data based on day/night masks
data_day <- carmel$data[ti$day,]
data_night <- carmel$data[ti$night,]

# Build two monitor objects
carmel_day <- list(meta = carmel$meta, data = data_day)
carmel_night <- list(meta = carmel$meta, data = data_night)

# Plot them
monitor_timeseriesPlot(carmel_day, shadedNight = TRUE, pch = 8, col = "goldenrod")
monitor_timeseriesPlot(carmel_night, pch = 16, col = "darkblue", add = TRUE)

Description

Creates a time series plot of PM2.5 data from a `ws_monitor` object (see note below). Optional arguments color code by AQI index, add shading to indicate nighttime, and adjust the time display (local vs. UTC).

When a named style is used, some graphical parameters will be overridden. Available styles include:

- `aqidots`—hourly values are individually colored by 24-hr AQI levels
- `gnats`—semi-transparent dots like a cloud of gnats

Usage

`monitor_timeseriesPlot`

    `monitor_timeseriesPlot(`
      `ws_monitor,`
      `monitorID = NULL,`
      `tlim = NULL,`
      `localTime = TRUE,`
      `style = NULL,`
      `shadedNight = FALSE,`
      `add = FALSE,`
      `gridPos = "",`
      `gridCol = "black",`
monitor_timeseriesPlot

gridLwd = 1,
gridLty = "solid",
dayLwd = 0,
hourLwd = 0,
hourInterval = 6,
...
)

Arguments

ws_monitor  ws_monitor object.
monitorID    Monitor ID for one or more monitor in the ws_monitor object.
tlim         Optional vector with start and end times (integer or character representing YYYYM-DDDHH).
localTime    Logical specifying whether tlim is in local time or UTC.
style        Custom styling, one of "aqidots".
shadedNight  Add nighttime shading.
add          Logical specifying whether to add to the current plot.
gridPos      Position of grid lines either "over", "under" ("" for no grid lines).
gridCol      Grid line color.
gridLwd      Grid line width.
gridLty      Grid line type.
dayLwd       Day marker line width.
hourLwd      Hour marker line width.
hourInterval Interval for grid (max = 12).
...          Additional arguments to be passed to points().

Note

Remember that a ws_monitor object can contain data from more than one monitor, and thus, this function may produce a time series of data from multiple monitors. To plot a time series of an individual monitor’s data, specify a single monitorID.

Examples

library(PWFSLSmoke)

N_M <- Northwest_Megafires
# monitor_leaflet(N_M) # to identify Spokane monitorIDs
Spokane <- monitor_subsetBy(
  N_M,
  stringr::str_detect(N_M$meta$monitorID, "^53063")
)

monitor_timeseriesPlot(Spokane, style = "gnats")
title("Spokane PM2.5 values, 2015")
monitor_timeseriesPlot(
    Spokane,
    tlim = c(20150801, 20150831),
    style = "aqidots",
    pch = 16
)
}
addAQILegend()
title("Spokane PM2.5 values, August 2015")
monitor_timeseriesPlot(
    Spokane,
    tlim = c(20150821, 20150828),
    shadedNight = TRUE,
    style = "gnats"
)
abline(h = AQI$breaks_24, col = AQI$colors, lwd = 2)
addAQILegend()
title("Spokane PM2.5 values, August 2015")

---

monitor_toTidy  
**Convert 'ws_monitor' data to a tidy format**

**Description**
Changes write-optomized 'ws_monitor' formatted data into a read-optomized 'tidy' format that is useful for 'tidyverse' functions. If the given data is already in a tidy format, it is returned as is.

**Usage**

```r
monitor_toTidy(data = NULL)
```

**Arguments**

data  
Data to potentially convert.

**Value**
'Tidy' formatted 'ws_monitor' data.

**Examples**

```r
library(PWFSLSmoke)

ws_monitor <- monitor_subset(
    Northwest_Megafires,
    monitorIDs = c("530470009_01", "530470010_01")
)

ws_monTidy <- monitor_toTidy(ws_monitor)

## Not run:
```
monitor_trim

ws_monTidy2 <- monitor_toTidy(ws_monTidy)
## End(Not run)

---

**monitor_trim**

*Trim ws_monitor Time Axis to Remove NA Periods From Beginning and End*

**Description**

Trims the time axis of a *ws_monitor* object to exclude timestamps prior to the first and after the last valid datapoint for any monitor.

**Usage**

```
monitor_trim(ws_monitor)
```

**Arguments**

- `ws_monitor`: *ws_monitor* object

**Value**

A *ws_monitor* object with missing data trimmed.

**Examples**

## Not run:

```r
# Fail gracefully if any resources are not available
try({

library(PWFSLSmoke)
library(MazamaSpatialUtils)

sm13 <- wrcc_createMonitorObject(20150101, 20151231, unitID = 'sm13')
sm13$meta[,c('stateCode','countyName','siteName','monitorID')]
Deschutes <- monitor_subset(sm13, monitorIDs='lon_.121.453_lat_43.878_wrcc.sm13')
Deschutes <- monitor_trim(Deschutes)
monitor_dailyBarplot(Deschutes)
}, silent = FALSE)
## End(Not run)
```
monitor_writeCSV  Write monitor data as CSV

Description
Prints out the contents of the ws_monitor object as CSV. By default, the output is a text string with "human readable" CSV that includes both meta and data. When saved as a file, this format is useful for point-and-click spreadsheet users who want to have everything on a single sheet.
To obtain machine parseable CSV strings you can use metaOnly or dataOnly which are mutually exclusive but which return CSV strings that can be automatically ingested.
By default, the CSV formatted text is returned invisibly but not saved to a file unless saveFile is specified.

Usage
monitor_writeCSV(
  ws_monitor,
  saveFile = NULL,
  metaOnly = FALSE,
  dataOnly = FALSE,
  quietly = TRUE
)

Arguments

ws_monitor       ws_monitor object
saveFile         optional filename where CSV will be written
metaOnly         flag specifying whether to return ws_monitor$meta only as a machine parseable CSV
dataOnly          flag specifying whether to return ws_monitor$data only as a machine parseable CSV
quietly          do not print to console, just return the string representation of the CSV

Note
This function wraps the monitor_print function but defaults to quietly = FALSE.

Examples
library(PWFSLSmoke)
data("Carmel_Valley")
Carmel_Valley <- monitor_subset(Carmel_Valley, tlim = c(20160802, 20160803))
monitor_print(Carmel_Valley)
monitor_print(Carmel_Valley, metaOnly = TRUE)
monitor_print(Carmel_Valley, dataOnly = TRUE)
monitor_writeCurrentStatusGeoJSON

Write current monitor data to geojson file

Description

Writes a geoJSON file containing current monitor data. For details on what is included, see `monitor_getCurrentStatus`.

Usage

```r
monitor_writeCurrentStatusGeoJSON(
  ws_monitor,
  filename,
  datetime = lubridate::now(tzone = "UTC"),
  properties = NULL,
  propertyNames = NULL,
  metadataList = list()
)
```

Arguments

- `ws_monitor`: `ws_monitor` object.
- `filename`: Filename where geojson file will be saved.
- `datetime`: Time to which data will be ‘current’ (integer or character representing YYYY-MM-DDHH or POSIXct. If not POSIXct, interpreted as UTC time). So if `datetime` is 3 hours ago, a dataframe with the most current data from 3 hours ago will be returned.
- `properties`: Optional character vector of properties to include for each monitor in geoJSON. If `NULL` all are included. May include any `ws_monitor` metadata and additional columns generated in `monitor_getCurrentStatus`.
- `propertyNames`: Optional character vector supplying custom names for properties in geoJSON. If `NULL` or different length than `properties` defaults will be used.
- `metadataList`: List of top-level foreign members to include. May include nested lists as long as they can be converted into JSON using `jsonlite::toJSON()`. For more information on what can be included see https://tools.ietf.org/html/rfc7946#section-6.1.

Value

Invisibly returns geoJSON string.
Examples

```r
# Fail gracefully if any resources are not available
try({

library(PWFSLSmoke)

wa <-
  monitor_loadLatest() %>%
  monitor_subset(stateCodes = "WA")

geojson_file <- tempfile(fileext = ".geojson")
wa_current_geojson <- monitor_writeCurrentStatusGeoJSON(wa, geojson_file)
wa_current_list <- jsonlite::fromJSON(wa_current_geojson)
wa_spdf <- rgdal::readOGR(dsn = geojson_file)
map("state", "washington")
points(wa_spdf)
}, silent = FALSE)
```

Northwest_Megafires

Northwest_Megafires example dataset

Description

In the summer of 2015 Washington state had several catastrophic wildfires that led to many days of heavy smoke in eastern Washington, Oregon and northern Idaho. The Northwest_Megafires dataset contains AirNow ambient monitoring data for the Pacific Northwest from May 31 through November 01, 2015 (UTC).

The Northwest_Megafires dataset provides a quickly loadable version of a `ws_monitor` object for practicing and code examples.

Usage

Northwest_Megafires

Format

A `ws_monitor` object with "meta" and "data" dataframes.
Create Pollution Rose Plot from a Raw Dataframe

Description
Create pollution rose plot from an enhanced raw data frame. This function is based on `openair::pollutionRose()`. If normalized, black line indicates frequency by direction.

Usage
```
rawPlot_pollutionRose(
  df, 
  parameter = "pm25", 
  tlim = NULL, 
  localTime = TRUE, 
  normalize = FALSE, 
  ...
)
```

Arguments
- **df**: enhanced, raw dataframe as created by the `raw_enhance()` function
- **parameter**: parameter to plot
- **tlim**: optional vector with start and end times (integer or character representing YYYY-MM-DD[HH])
- **localTime**: logical specifying whether `tlim` is in local time or UTC
- **normalize**: normalize slices to fill entire area, allowing for easier comparison of counts of magnitudes by direction
- **...**: additional arguments to pass on to `openair::pollutionRose()`

Note
If more than one timezone is found, `localTime` is ignored and UTC is used.

Examples
```r
## Not run:
# Fail gracefully if any resources are not available
try({
  raw <- airsis_createRawDataframe(20160901, 20161015, 'USFS', 1012)
  raw <- raw_enhance(raw)
  rawPlot_pollutionRose(raw)
}, silent = FALSE)
## End(Not run)
```
Create Time of Day Spaghetti Plot from a Raw Dataframe

Description
Spaghetti Plot that shows data by hour-of-day.

Usage
rawPlot_timeOfDaySpaghetti(
  df,
  parameter = "pm25",
  tlim = NULL,
  shadedNight = TRUE,
  meanCol = "black",
  meanLwd = 4,
  meanLty = 1,
  highlightDates = c(),
  highlightCol = "dodgerblue",
  ...
)

Arguments
- df: enhanced, raw dataframe as created by the raw_enhance() function
- parameter: variable to be plotted
- tlim: optional vector with start and end times (integer or character representing YYYYMMDD[HH])
- shadedNight: add nighttime shading
- meanCol: color used for the mean line (use NA to omit the mean)
- meanLwd: line width used for the mean line
- meanLty: line type used for the mean line
- highlightDates: dates to be highlighted in YYYYMMDD format
- highlightCol: color used for highlighted days
- ... additional graphical parameters are passed to the lines() function for day lines

Examples
# Not run:
# Fail gracefully if any resources are not available
try{
  raw <- airsis_createRawDataframe(20160901, 20161015, 'USFS', 1012)
  raw <- raw_enhance(raw)
rawPlot_timeseries

rawPlot_timeOfDaySpaghetti(raw, parameter = "temperature")

}, silent = FALSE)

## End(Not run)

---

**rawPlot_timeseries**  
Create Timeseries Plot from a Raw Dataframe

**Description**

Creates a plot of raw monitoring data as generated using raw_enhance().

Other options for parameter include "temperature", "humidity", "windSpeed", "windDir", "pressure" or any of the other raw parameters (try names(df) to see list of options)

**Usage**

```r
rawPlot_timeseries(
  df,
  parameter = "pm25",
  tlim = NULL,
  localTime = TRUE,
  shadedNight = TRUE,
  shadedBackground = NULL,
  sbLwd = 1,
  add = FALSE,
  gridPos = "",
  gridCol = "black",
  gridLwd = 1,
  gridLty = "solid",
  dayLwd = 0,
  hourLwd = 0,
  hourInterval = 6,
  ...
)
```

**Arguments**

- `df`  
  enhanced, raw dataframe as created by the raw_enhance() function
- `parameter`  
  raw parameter to plot
- `tlim`  
  optional vector with start and end times (integer or character representing YYYY-MM-DD[HH])
- `localTime`  
  logical specifying whether `tlim` is in local time or UTC
- `shadedNight`  
  add nighttime shading
- `shadedBackground`  
  add vertical lines for a second parameter
rawPlot_windRose

Create Wind Rose Plot from a Raw Dataframe

Description

Create wind rose plot from raw_enhance object. Based on openair::windRose().

Usage

rawPlot_windRose(df, tlim = NULL, localTime = TRUE, ...)

Arguments

df enhanced, raw dataframe as created by the raw_enhance() function
tlim optional vector with start and end times (integer or character representing YYYY-MM-DD[HH])
localTime logical specifying whether tlim is in local time or UTC
... additional arguments to pass on to openair::windRose()

Note

If more than one timezone is found, localTime is ignored and UTC is used.
Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  raw <- airsis_createRawDataframe(20160901, 20161015, provider='USFS', unitID=1012)
  raw <- raw_enhance(raw)
  rawPlot_windRose(raw)
}, silent = FALSE)
## End(Not run)
```

---

**raw_enhance**  
*Process Raw Monitoring Data to Create raw_enhance Object*

### Description
Processes raw monitor data to add a uniform time axis and consistent data columns that can be handled by various `raw~` functions. All original raw data is retained, and the following additional columns are added:

- `dataSource`
- `longitude`
- `latitude`
- `temperature`
- `humidity`
- `windSpeed`
- `windDir`
- `pressure`
- `pm25`

The `datetime` column in the incoming dataframe may have missing hours. This time axis is expanded to a uniform, hourly axes with missing data fields added for data columns.

### Usage

```r
raw_enhance(df)
```

### Arguments

- `df`: raw monitor data, as created by `airsis_createRawDataframe` or `wrcc_createRawDataframe`

### Value
Dataframe with original raw data, plus new columns with raw naming scheme for downstream use.
Examples

## Not run:
# Fail gracefully if any resources are not available
try(

library(PWFSLSmoke)
df <- airsis_createRawDataframe(startdate = 20160901, enddate = 20161015, provider = 'USFS', unitID = 1012)
df <- raw_enhance(df)
rawPlot_timeseries(df, tlim = c(20160908, 20160917))
}, silent = FALSE)

## End(Not run)

raw_getHighlightDates
Return Day Stamps for Values Above a Threshold

Description

Return a list of dates in YYYYMMDD format where the dataVar is within highlightRange.

Usage

raw_getHighlightDates(
  df,
dataVar,
tzone = NULL,
highlightRange = c(1e+12, Inf)
)

Arguments

df          dataframe with datetime column in UTC
dataVar      variable to be evaluated
tzone        timezone where data were collected
highlightRange range of values of to be highlighted

Examples

## Not run:
# Fail gracefully if any resources are not available
try(

raw <- airsis_createRawDataframe(startdate = 20160901, provider = 'USFS', unitID = '1033')
raw <- raw_enhance(raw)
highlightRange <- c(50, Inf)
dataVar <- 'pm25'
tzone <- "America/Los_Angeles"

)}}
highlightDates <- raw_getHighlightDates(raw, dataVar, tzone, highlightRange)
rawPlot_timeOfDaySpaghetti(df=raw, highlightDates = highlightDates)
}

## End(Not run)

---

**setEsriToken**

*Set ESRI Token*

**Description**

Sets the current esriToken.

**Usage**

```
setEsriToken(token)
```

**Arguments**

- `token`  
  ESRI token used when interacting with ESRI location services

**Value**

Silently returns previous value of esriToken.

**See Also**

- addEsriAddress
- getEsriToken
- esriToken

---

**setGoogleApiKey**

*Set Google API Key*

**Description**

Sets the current Google API key.

**Usage**

```
setGoogleApiKey(key)
```

**Arguments**

- `key`  
  Google API key used when interacting with Google location services
Value

Silently returns previous value of googleApiKey.

See Also

addGoogleAddress
addGoogleElevation
getGoogleApiKey
googleApiKey

Description

Measurements of categorical forecast accuracy have a long history in weather forecasting. The standard approach involves making binary classifications (detected/not-detected) of predicted and observed data and combining them in a binary contingency table known as a confusion matrix.

This function creates a confusion matrix from predicted and observed values and calculates a wide range of common statistics including:

- TP (true positive)
- FP (false positive) (type I error)
- FN (false negative) (type II error)
- TN (true negative)
- TPRate (true positive rate) = sensitivity = recall = TP / (TP + FN)
- FPRate (false positive rate) = FP / (FP + TN)
- FNRate (false negative rate) = FN / (TP + FN)
- TNRate (true negative rate) = specificity = TN / (FP + TN)
- accuracy = proportionCorrect = (TP + TN) / total
- errorRate = 1 - accuracy = (FP + FN) / total
- falseAlarmRatio = PPV (positive predictive value) = precision = TP / (TP + FP)
- FDR (false discovery rate) = FP / (TP + FP)
- NPV (negative predictive value) = TN / (TN + FN)
- FOR (false omission rate) = FN / (TN + FN)
- f1_score = (2 * TP) / (2 * TP + FP + FN)
- detectionRate = TP / total
- baseRate = detectionPrevalence = (TP + FN) / total
- probForecastOccurance = prevalence = (TP + FP) / total
- balancedAccuracy = (TPRate + TNRate) / 2
\[ \text{expectedAccuracy} = \frac{((TP + FP) \times (TP + FN) / \text{total}) + ((FP + TN) \times \text{sum}(FN + TN) / \text{total})}{\text{total}} \]

\[ \text{heidkeSkill} = \kappa = \frac{\text{accuracy} - \text{expectedAccuracy}}{1 - \text{expectedAccuracy}} \]

\[ \text{bias} = \frac{(TP + FP)}{(TP + FN)} \]

\[ \text{hitRate} = \frac{TP}{(TP + FN)} \]

\[ \text{falseAlarmRate} = \frac{FP}{(FP + TN)} \]

\[ \text{pierceSkill} = \frac{((TP \times TN) - (FP \times FN))}{((FP + TN) \times (TP + FN))} \]

\[ \text{criticalSuccess} = \frac{TP}{(TP + FP + FN)} \]

\[ \text{oddsRatioSkill} = yulesQ = \frac{((TP \times TN) - (FP \times FN))}{((TP + TN) + (FP + FN))} \]

**Usage**

```r
skill_confusionMatrix(
  predicted, observed, FPCost = 1, FNCost = 1, lightweight = FALSE
)
```

**Arguments**

- `predicted`: logical vector of predicted values
- `observed`: logical vector of observed values
- `FPCost`: cost associated with false positives (type I error)
- `FNCost`: cost associated with false negatives (type II error)
- `lightweight`: flag specifying creation of a return list without derived metrics

**Value**

List containing a table of confusion matrix values and a suite of derived metrics.

**References**

*Simple Guide to Confusion Matrix Terminology*

**See Also**

- `skill_ROC`
- `skill_ROCPlot`

**Examples**

```r
predicted <- sample(c(TRUE,FALSE), 1000, replace=TRUE, prob=c(0.3,0.7))
observed <- sample(c(TRUE,FALSE), 1000, replace=TRUE, prob=c(0.3,0.7))
cm <- skill_confusionMatrix(predicted, observed)
print(cm)
```
skill_ROC

ROC Curve

Description
This function calculates an ROC dataframe of TPR, FPR, and Cost for a range of thresholds as well as the area under the ROC curve.

Usage

skill_ROC(predicted, observed, t1Range = NULL, t2 = NULL, n = 101)

Arguments

- **predicted**: vector of predicted values (or a `ws_monitor` object with a single location)
- **observed**: vector of observed values (or a `ws_monitor` object with a single location)
- **t1Range**: lo and high values used to generate test thresholds for classifying predicted data
- **t2**: used to classify observed data
- **n**: number of test thresholds in ROC curve

Value
List containing an `roc` matrix and the `auc` area under the ROC curve.

References

Receiver Operating Characteristic

See Also

- `skill_confusionMatrix`
- `skill_ROCPlot`

Examples

```r
## Not run:
try({
  # Napa Fires -- October, 2017
  ca <- airnow_loadAnnual(2017) %>%
    monitor_subset(tlim = c(20171001, 20171101), stateCodes = 'CA')
  Vallejo <- monitor_subset(ca, monitorIDs = '060950004_01')
  Napa <- monitor_subset(ca, monitorIDs = '060550003_01')
  t2 <- AQI$breaks_24[4] # 'Unhealthy'
  rocList <- skill_ROC(Vallejo, Napa, t1Range = c(0,100), t2 = t2)
})
```
roc <- rocList$roc
auc <- rocList$auc
plot(roc$TPR ~ roc$FPR, type = 'S')
title(paste0('Area Under Curve = ', format(auc, digits = 3)))
}
## End(Not run)

---

**skill_ROCPlot**

**ROC Plot**

**Description**

This function plots ROC curves for a variety of observed classification thresholds.

**Usage**

```r
skill_ROCPlot(
  predicted,  
  observed,   
  t1Range = c(0, 100),  
  t2s = seq(10, 100, 10),  
  n = 101,  
  colors = grDevices::rainbow(length(t2s))
)
```

**Arguments**

- **predicted**: vector of predicted values (or a `ws_monitor` object with a single location)
- **observed**: vector of observed values (or a `ws_monitor` object with a single location)
- **t1Range**: lo and high values used to generate test thresholds for classifying predicted data
- **t2s**: vector of thresholds used to classify observed data
- **n**: number of test thresholds in ROC curve
- **colors**: vector of colors used when plotting curves

**References**

- Receiver Operating Characteristic

**See Also**

- `skill_confusionMatrix`
- `skill_ROC`
Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  # Napa Fires -- October, 2017
  ca <- airnow_loadAnnual(2017) %>%
    monitor_subset(tlim = c(20171001,20171101), stateCodes = 'CA')
  Vallejo <- monitor_subset(ca, monitorIDs = '060950004_01')
  Napa <- monitor_subset(ca, monitorIDs = '060550003_01')
  skill_ROCPlot(Vallejo, Napa)
}, silent = FALSE)
## End(Not run)
```

---

### staticmap_getEsrimapBrick

Create a rasterBrick from an Esri tiled image server

#### Description

Uses the input coordinates to fetch and composite a raster from the tile server. Returns a `raster::rasterBrick` object. This can then passed as the `rasterBrick` object to the `staticmap_plotRasterBrick()` function for plotting.

As of July 2019, this list is a handy reference to the freely available tile servers which can be previewed at the following URL:

[https://leaflet-extras.github.io/leaflet-providers/preview/](https://leaflet-extras.github.io/leaflet-providers/preview/)

#### Usage

```r
staticmap_getEsrimapBrick(
  centerLon = NULL,
  centerLat = NULL,
  maptype = "world_topo",
  zoom = 12,
  width = 640,
  height = 640,
  bbox = NULL,
  maxTiles = 20,
  crs = sp::CRS("+init=epsg:4326"),
  tileCacheDir = tempdir()
)
```
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>centerLon</td>
<td>Map center longitude.</td>
</tr>
<tr>
<td>centerLat</td>
<td>Map center latitude.</td>
</tr>
<tr>
<td>maptype</td>
<td>Selects the appropriate Esri tile server. Options include:</td>
</tr>
<tr>
<td></td>
<td>• &quot;world_topo&quot;</td>
</tr>
<tr>
<td></td>
<td>• &quot;world_imagery&quot;</td>
</tr>
<tr>
<td></td>
<td>• &quot;world_terrain&quot;</td>
</tr>
<tr>
<td></td>
<td>• &quot;de_Lorne&quot;</td>
</tr>
<tr>
<td></td>
<td>• &quot;world_grey&quot;</td>
</tr>
<tr>
<td></td>
<td>• &quot;world_streets&quot;</td>
</tr>
<tr>
<td>zoom</td>
<td>Map Zoom level.</td>
</tr>
<tr>
<td>width</td>
<td>Width of image, in pixels.</td>
</tr>
<tr>
<td>height</td>
<td>Height of image, in pixels.</td>
</tr>
<tr>
<td>bbox</td>
<td>Bounding box vector (lonLo, latLo, lonHi, latHi). If not null, centerLon, centerLat, and zoom are ignored.</td>
</tr>
<tr>
<td>maxTiles</td>
<td>Maximum number of tiles to be returned. The greater the number, the slower the performance – arbitrarily set to 20 by default.</td>
</tr>
<tr>
<td>crs</td>
<td>Object of class CRS. The Coordinate Reference System (CRS) for the returned map. If the CRS of the downloaded map does not match, it will be projected to the specified CRS using raster::projectRaster.</td>
</tr>
<tr>
<td>tileCacheDir</td>
<td>Optional location for cached tiles.</td>
</tr>
</tbody>
</table>

Value

A rasterBrick object which can be plotted with staticmap_plotRasterBrick() or raster::plotRGB() and serve as a base plot.

Note

The spatial reference of the image when it is downloaded is 3857. If the crs argument is different, projecting may cause the size and extent of the image to differ very slightly from the input, on a scale of 1-2 pixels or 10^-3 degrees.

If bbox is specified and the bbox aspect ratio does not match the width/height aspect ratio the extent is resized to prevent the map image from appearing stretched, so the map extent may not match the bbox argument exactly.

If both zoom and maxTiles are specified, maxTiles takes precedence. To get a specified zoom level, set maxTiles = NULL.

See Also

staticmap_getRasterBrick
staticmap_plotRasterBrick
**staticmap_getRasterBrick**

Create a rasterBrick from a tiled image server

**Description**

Uses the input coordinates to select an appropriate method to build a raster::rasterBrick object. It will either use the staticmap_getStamenmapBrick() function or the staticmap_getEsrimapBrick() function. This can then be passed as the rasterBrick object to the staticmap_plotRasterBrick() function for plotting.

**Usage**

```r
centerLon = NULL,
centerLat = NULL,
maptype = "world_topo",
zoom = 12,
width = 640,
height = 640,
bbox = NULL,
maxTiles = 40,
crs = sp::CRS("+init=epsg:4326"),
tileCacheDir = tempdir()
```

**Arguments**

- **centerLon**: Map center longitude.
- **centerLat**: Map center latitude.
- **maptype**: Defaults to **Esri Topographic** Available to select between Stamen basemaps or Esri basemaps.
  - **Stamen**
    - terrain

**Examples**

```r
try({
  rasterBrick <- staticmap_getEsrimapBrick(-122.3318, 47.668)
  staticmap_plotRasterBrick(rasterBrick)
}, silent = FALSE)
```
staticmap_getRasterBrick

- terrain-background
- terrain-labels
- terrain-lines
- toner
- toner-background
- toner-hybrid
- toner-labels
- toner-labels
- toner-lines
- toner-lite
- watercolor

**Esri**

- "world_topo"
- "world_imagery"
- "world_terrain"
- "de_Lorme"
- "world_grey"
- "world_streets"

**zoom**  
Map zoom level.

**width**  
Width of image, in pixels.

**height**  
Height of image, in pixels.

**bbox**  
If you are using the Esri maps, then the bbox parameter must be an st_bbox object as specified in the sf package documentation [https://www.rdocumentation.org/packages/sfversions/0.7-4/topics/st_bbox]. If using Stamen Maps, use a vector organized as (lonLo, latLo, lonHi, latHi) If not null, centerLon, centerLat, and zoom are ignored.

**maxTiles**  
Only utilized if selecting an esri basemap, specifies the maximum number of tiles to be returned. The greater the number, the slower the performance – arbitrarily set to 20 by default.

**crs**  
Object of class CRS. The Coordinate Reference System (CRS) for the returned map. If the CRS of the downloaded map does not match, it will be projected to the specified CRS using raster::projectRaster.

**tileCacheDir**  
Optional location for cached tiles.

**Value**

A rasterBrick object which can be plotted with staticmap_plotRasterBrick() or raster::plotRGB() and serve as a base plot.
staticmap_getStamenmapBrick

Create a rasterBrick from stamenmap tiles

Description

Downloads a PNG from the stamenmap tile server and creates a raster::rasterBrick object with layers for red, green, and blue. This can then passed as the rasterBrick object to the staticmap_plotRasterBrick() function for plotting.

Stamen maps tiles are freely available (April, 2019) and are described at the following URL:

http://maps.stamen.com/#terrain/12/37.7706/-122.3782

Note

The spatial reference of the image when it is downloaded is 3857. If the crs argument is different, projecting may cause the size and extent of the image to differ very slightly from the input, on a scale of 1-2 pixels or 10^-3 degrees.

If bbox is specified and the bbox aspect ratio does not match the width/height aspect ratio the extent is resized to prevent the map image from appearing stretched, so the map extent may not match the bbox argument exactly.

See Also

staticmap_getStamenmapBrick
staticmap_getEsrimapBrick
staticmap_plotRasterBrick

Examples

## Not run:
# Fail gracefully if any resources are not available
try({
  rasterBrick <- staticmap_getRasterBrick(-122.3318, 47.668)
  staticmap_plotRasterBrick(rasterBrick)

  rasterBrick <- staticmap_getRasterBrick(-122.3318, 47.668, "world_streets", 12)
  staticmap_plotRasterBrick(rasterBrick)

  rasterBrick <- staticmap_getRasterBrick(-122.3318, 47.668, "watercolor", 12)
  staticmap_plotRasterBrick(rasterBrick)
}, silent = FALSE)
## End(Not run)
"These tiles are made available as part of the CityTracking project, funded by the Knight Foundation, in which Stamen is building web services and open source tools to display public data in easy-to-understand, highly visual ways."

**Usage**

```r
staticmap_getStamenmapBrick(
  centerLon = NULL,
  centerLat = NULL,
  maptype = "terrain",
  zoom = 12,
  width = 640,
  height = 640,
  bbox = NULL,
  crs = sp::CRS("+init=epsg:4326"),
  tileCacheDir = tempdir()
)
```

**Arguments**

- `centerLon`: map center longitude
- `centerLat`: map center latitude
- `maptype`: map type
- `zoom`: map zoom level; corresponds to `ggmap::get_map()` zoom level
- `width`: width of image, in pixels
- `height`: height of image, in pixels
- `bbox`: bounding box vector (lonLo, latLo, lonHi, latHi). If not null, `centerLon, centerLat, and zoom` are ignored.
- `crs`: object of class CRS. The Coordinate Reference System (CRS) for the returned map. If the CRS of the downloaded map does not match, it will be projected to the specified CRS using `raster::projectRaster`
- `tileCacheDir`: Optional location for cached tiles.

**Value**

A `rasterBrick` object which can be plotted with `staticmap_plotRasterBrick()` or `raster::plotRGB()` and serve as a base plot.

**Note**

The spatial reference of the image when it is downloaded is 3857. If the `crs` argument is different, projecting may cause the size and extent of the image to differ very slightly from the input, on a scale of 1-2 pixels or $10^{-3}$ degrees.

If `bbox` is specified and the `bbox` aspect ratio does not match the width/height aspect ratio the extent is resized to prevent the map image from appearing stretched, so the map extent may not match the `bbox` argument exactly.
staticmap_plotRasterBrick

## Plot an RGB rasterBrick

### Description

Plots the incoming rasterBrick.

### Usage

```r
staticmap_plotRasterBrick(rasterBrick = NULL, grayscale = FALSE, ...)
```

### Arguments

- `rasterBrick` an RGB rasterBrick object. It is assumed that layer 1 represents red, layer 2 represents green, and layer 3 represents blue.
- `grayscale` logical specifying conversion to grayscale
- `...` arguments passed on to `raster::plot()` (for grayscale = TRUE) or `raster::plotRGB()` (for grayscale = FALSE)

### See Also

- `staticmap_getStamenmapBrick`
Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  rasterBrick <- staticmap_getStamenmapBrick(-122.3318, 47.668)
  staticmap_plotRasterBrick(rasterBrick)
  staticmap_plotRasterBrick(rasterBrick, grayscale = TRUE)
}, silent = FALSE)

## End(Not run)
```

### tidy_toMonitor

**Convert `ws_tidy` data to a `ws_monitor` object**

**Description**

Changes read-optomized `tidy` formatted monitor data into a write-optomized `ws_monitor` format. If the given data is already a `ws_monitor` object, it is returned as is. This function is the inverse of `monitor_toTidy`.

**Usage**

```r
tidy_toMonitor(data = NULL)
```

**Arguments**

- **data** Data to potentially convert.

**Value**

`'ws_monitor'` object

**Examples**

```r
ws_monitor <- monitor_subset(
  Northwest_Megafires,
  monitorIDs = c('530470009_01', '530470010_01')
)

ws_monTidy <- monitor_toTidy(ws_monitor)
ws_monMon <- tidy_toMonitor(ws_monTidy)
head(ws_monMon$data)
head(ws_monitor$data)
```
Get time related information

timeInfo

Description

Calculate the local time at the target location, as well as sunrise, sunset and solar noon times, and create several temporal masks.

If the timezone is provided it will be used. Otherwise, the MazamaSpatialUtils package will be used to determine the timezone from longitude and latitude.

The returned dataframe will have as many rows as the length of the incoming UTC time vector and will contain the following columns:

- `localStdTime_UTC` – UTC representation of local standard time
- `daylightSavings` – logical mask = TRUE if daylight savings is in effect
- `localTime` – local clock time
- `sunrise` – time of sunrise on each localTime day
- `sunset` – time of sunset on each localTime day
- `solarnoon` – time of solar noon on each localTime day
- `day` – logical mask = TRUE between sunrise and sunset
- `morning` – logical mask = TRUE between sunrise and solarnoon
- `afternoon` – logical mask = TRUE between solarnoon and sunset
- `night` – logical mask = opposite of day

Usage

```
timeInfo(time = NULL, longitude = NULL, latitude = NULL, timezone = NULL)
```

Arguments

- `time` POSIXct vector with specified timezone,
- `longitude` Longitude of the location of interest.
- `latitude` Latitude of the location of interest.
- `timezone` Olson timezone at the location of interest.

Details

While the lubridate package makes it easy to work in local timezones, there is no easy way in R to work in "Local Standard Time" (LST) as is often required when working with air quality data. EPA regulations mandate that daily averages be calculated based on LST.

The `localStdTime_UTC` column in the returned dataframe is primarily for internal use and provides an important tool for creating LST daily averages and LST axis labeling.
upgradeMeta_v1.0

Value

A dataframe with times and masks.

Examples

carmel <- monitor_subset(Carmel_Valley, tlim = c(20160801,20160810))

# Create timeInfo object for this monitor
ti <- timeInfo(
  carmel$data$datetime,
  carmel$meta$longitude,
  carmel$meta$latitude,
  carmel$meta$timezone
)

# Subset the data based on day/night masks
data_day <- carmel$data[ti$day,]
data_night <- carmel$data[ti$night,]

# Build two monitor objects
carmel_day <- list(meta = carmel$meta, data = data_day)
carmel_night <- list(meta = carmel$meta, data = data_night)

# Plot them
monitor_timeseriesPlot(carmel_day, shadedNight = TRUE, pch = 8, col = 'goldenrod')
monitor_timeseriesPlot(carmel_night, pch = 16, col = 'darkblue', add = TRUE)

upgradeMeta_v1.0

Upgrade ws_monitor Metadata to Version 1.0

Description

Upgrade a ws_monitor object to version 1.0 standards.

Usage

upgradeMeta_v1.0(ws_monitor)

Arguments

ws_monitor  ws_monitor object

Value

A ws_monitor object with version 1.0 metadata.
**US_52  US State Codes**

**Description**
State codes for the 50 states +DC +PR (Puerto Rico)

**Usage**
US_52

**Format**
A vector with 52 elements

**Details**
US state codes

---

**WRCC  WRCC Monitor Names and Unit IDs**

**Description**
The WRCC [https://wrcc.dri.edu/cgi-bin/smoke.pl](https://wrcc.dri.edu/cgi-bin/smoke.pl) Fire Cache Smoke Monitor Archive provides access to a variety of monitors that can be accessed with the wrcc_createMonitorObject function. Use of this function requires a valid unitID. The WRCC object is a list of lists. The element named unitIDs is itself a list of three named vectors, each containing the unitIDs and associated names for one of the categories of monitors available at WRCC:

- cache
- miscellaneous
- usfs_regional

**Format**
A list of lists

**Details**
WRCC monitor names and unitIDs

**Note**
This list of monitor IDs reflects unitIDs found on the WRCC site on June 12, 2019.
wrcc_createDataDataframe

Create WRCC data dataframe

Description

After quality control has been applied to an WRCC tibble, we can extract the PM2.5 values and store them in a data tibble organized as time-by-deployment (aka time-by-site).

The first column of the returned dataframe is named 'datetime' and contains a POSIXct time in UTC. Additional columns contain data for each separate deployment of a monitor.

Usage

wrcc_createDataDataframe(tbl, meta)

Arguments

tbl single site WRCC tibble created by wrcc_clustering()
meta WRCC meta dataframe created by wrcc_createMetaDataframe()

Value

A data dataframe for use in a ws-monitor object.

wrcc_createMetaDataframe

Create WRCC site location metadata dataframe

Description

After a WRCC tibble has been enhanced with additional columns generated by addClustering we are ready to pull out site information associated with unique deployments.

These will be rearranged into a dataframe organized as deployment-by-property with one row for each monitor deployment.

This site information found in tbl is augmented so that we end up with a uniform set of properties associated with each monitor deployment. The list of columns in the returned meta dataframe is:

```r
> names(p$meta)
[1] "monitorID"          "longitude"          "latitude"
[4] "elevation"          "timezone"           "countryCode"
[7] "stateCode"          "siteName"           "agencyName"
[10] "countyName"         "msaName"            "monitorType"
[13] "monitorInstrument" "aqsID"                "pwfs1ID"
[16] "pwfs1DataIngestSource" "telemetryAggregator" "telemetryUnitID"
```
Usage

\[
\text{wrcc\_createMetaDataframe}( \\
\quad \text{tbl,} \\
\quad \text{unitID = as.character(NA),} \\
\quad \text{pwfs1DataIngestSource = "WRCC",} \\
\quad \text{existingMeta = NULL,} \\
\quad \text{addGoogleMeta = FALSE,} \\
\quad \text{addEsriMeta = FALSE} \\
\)
\]

Arguments

- **tbl**: single site WRCC tibble after metadata enhancement
- **unitID**: character or numeric WRCC unit identifier
- **pwfs1DataIngestSource**: identifier for the source of monitoring data, e.g. 'WRCC'
- **existingMeta**: existing 'meta' dataframe from which to obtain metadata for known monitor deployments
- **addGoogleMeta**: logical specifying wheter to use Google elevation and reverse geocoding services
- **addEsriMeta**: logical specifying wheter to use ESRI elevation and reverse geocoding services

Value

A meta dataframe for use in a \textit{ws\_monitor} object.

See Also

- \texttt{addMazamaMetadata}

---

\texttt{wrcc\_createMonitorObject}

\textit{Obtain WRCC data and create \textit{ws\_monitor} object}

Description

Obtains monitor data from an WRCC webservice and converts it into a quality controlled, metadata enhanced \textit{ws\_monitor} object ready for use with all \texttt{monitor\_~} functions.

Steps involved include:

1. download CSV text
2. parse CSV text
3. apply quality control
4. apply clustering to determine unique deployments
5. enhance metadata to include: elevation, timezone, state, country, site name
6. reshape data into deployment-by-property meta and and time-by-deployment data dataframes

QC parameters that can be passed in the ... include the following valid data ranges as taken from 
wrcc_EBAMQualityControl():

- valid_Longitude=c(-180,180)
- valid_Latitude=c(-90,90)
- remove_Lon_zero = TRUE
- remove_Lat_zero = TRUE
- valid_Flow = c(16.7*0.95,16.7*1.05)
- valid_AT = c(-Inf,45)
- valid_RHi = c(-Inf,50)
- valid_Conc = c(-Inf,5000)

Note that appropriate values for QC thresholds will depend on the type of monitor.

Usage

wrcc_createMonitorObject(
  startdate = strftime(lubridate::now(tzone = "UTC"), "%Y010100", tz = "UTC"),
  enddate = strftime(lubridate::now(tzone = "UTC"), "%Y%m%d23", tz = "UTC"),
  unitID = NULL,
  clusterDiameter = 1000,
  zeroMinimum = TRUE,
  baseUrl = "https://wrcc.dri.edu/cgi-bin/wea_list2.pl",
  saveFile = NULL,
  existingMeta = NULL,
  addGoogleMeta = FALSE,
  addEsriMeta = FALSE,
  ...
)

Arguments

startdate desired start date (integer or character representing YYYYMMDD[HH])
enddate   desired end date (integer or character representing YYYYMMDD[HH])
unitID    station identifier (will be upcased)
clusterDiameter  diameter in meters used to determine the number of clusters (see addClustering)
zeroMinimum    logical specifying whether to convert negative values to zero
baseUrl       base URL for data queries
saveFile      optional filename where raw CSV will be written
existingMeta  existing 'meta' dataframe from which to obtain metadata for known monitor deployments
wrcc_createRawDataframe

addGoogleMeta  logical specifying whether to use Google elevation and reverse geocoding services
addEsriMeta    logical specifying whether to use ESRI elevation and reverse geocoding services
...            additional parameters are passed to type-specific QC functions

Value

A ws_monitor object with WRCC data.

Note

The downloaded CSV may be saved to a local file by providing an argument to the saveFile parameter.

See Also

wrcc_downloadData
wrcc_parseData
wrcc_qualityControl
addClustering
wrcc_createMetaDataframe
wrcc_createDataDataframe

Examples

# Fail gracefully if any resources are not available
try({
library(PWFSLSmoke)
initializeMazamaSpatialUtils()

sm13 <- wrcc_createMonitorObject(20150301, 20150831, unitID = 'sm13')
monitor_leaflet(sm13)

}, silent = FALSE)
Description

 Obtains monitor data from a WRCC webservice and converts it into a quality controlled, metadata enhanced "raw" tibble ready for use with all raw_~ functions.

Steps involved include:

1. download CSV text
2. parse CSV text
3. apply quality control
4. apply clustering to determine unique deployments
5. enhance metadata to include: elevation, timezone, state, country, site name

Usage

```r
wrcc_createRawDataframe(
  startdate = strftime(lubridate::now(tzone = "UTC"), "%Y010100", tz = "UTC"),
  enddate = strftime(lubridate::now(tzone = "UTC"), "%Y%m%d23", tz = "UTC"),
  unitID = NULL,
  clusterDiameter = 1000,
  baseUrl = "https://wrcc.dri.edu/cgi-bin/wea_list2.pl",
  saveFile = NULL,
  flagAndKeep = FALSE
)
```

Arguments

- `startdate` Desired start date (integer or character representing YYYYMMDD[HH]).
- `enddate` Desired end date (integer or character representing YYYYMMDD[HH]).
- `unitID` Station identifier (will be upcased).
- `clusterDiameter` Diameter in meters used to determine the number of clusters (see addClustering).
- `baseUrl` Base URL for data queries.
- `saveFile` Optional filename where raw CSV will be written.
- `flagAndKeep` Flag, rather then remove, bad data during the QC process.

Value

Raw tibble of WRCC data.

Note

The downloaded CSV may be saved to a local file by providing an argument to the `saveFile` parameter.

Monitor unitIDs can be found at https://wrcc.dri.edu/cgi-bin/smoke.pl.
References

Fire Cache Smoke Monitoring Archive

See Also

wrcc_downloadData
wrcc_parseData
wrcc_qualityControl
addClustering

Examples

## Not run:
# Fail gracefully if any resources are not available
try({
library(PWFSLSmoke)

tbl <- wrcc_createRawDataframe(20150701, 20150930, unitID = 'SM16')
dplyr::glimpse(tbl)
}, silent = FALSE)
## End(Not run)

wrcc_downloadData  Download WRCC data

Description

Request data from a particular station for the desired time period. Data are returned as a single
character string containing the WRCC output.

Monitor unitIDs can be found at https://wrcc.dri.edu/cgi-bin/smoke.pl.

Usage

wrcc_downloadData(
  startdate = strftime(lubridate::now(tzone = "UTC"), "%Y010101", tz = "UTC"),
  enddate = strftime(lubridate::now(tzone = "UTC"), "%Y%m%d23", tz = "UTC"),
  unitID = NULL,
  baseUrl = "https://wrcc.dri.edu/cgi-bin/wea_list2.pl"
)
Arguments

- **startdate**: desired start date (integer or character representing YYYYMMDD[HH])
- **enddate**: desired end date (integer or character representing YYYYMMDD[HH])
- **unitID**: station identifier (will be upcased)
- **baseUrl**: base URL for data queries

Value

String containing WRCC output.

References

- Fire Cache Smoke Monitoring Archive

Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({

fileString <- wrcc_downloadData(20150701, 20150930, unitID = 'SM16')
df <- wrcc_parseData(fileString)
}), silent = FALSE)
## End(Not run)
```

Description

Perform various QC measures on WRCC EBAM data.

The any numeric values matching the following are converted to NA:

- \( x < -900 \)
- \( x == -9.9899 \)
- \( x == 99999 \)

The following columns of data are tested against valid ranges:

- **Flow**
- **AT**
- **RHi**
- **ConcHr**

A POSIXct datetime column (UTC) is also added based on DateTime.
Usage

wrcc_EBAMQualityControl(
  tbl,
  valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90),
  remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE,
  valid_Flow = c(16.7 * 0.95, 16.7 * 1.05),
  valid_AT = c(-Inf, 45),
  valid_RHi = c(-Inf, 50),
  valid_Conc = c(-Inf, 5000),
  flagAndKeep = FALSE
)

Arguments

tbl single site tibble created by wrcc_parseData()
valid_Longitude range of valid Longitude values
valid_Latitude range of valid Latitude values
remove_Lon_zero flag to remove rows where Longitude == 0
remove_Lat_zero flag to remove rows where Latitude == 0
valid_Flow range of valid Flow values
valid_AT range of valid AT values
valid_RHi range of valid RHi values
valid_Conc range of valid ConcHr values
flagAndKeep flag, rather than remove, bad data during the QC process

Value

Cleaned up tibble of WRCC monitor data.

See Also

wrcc_qualityControl
**wrcc_ESAMQualityControl**

*Apply Quality Control to raw WRCC E-Sampler tibble*

**Description**

Perform various QC measures on WRCC EBAM data.

The any numeric values matching the following are converted to NA

- $x < -900$
- $x == -9.9899$
- $x == 99999$

The following columns of data are tested against valid ranges:

- Flow
- AT
- RHi
- ConcHr

A POSIXct datetime column (UTC) is also added based on DateTime.

**Usage**

```r
wrcc_ESAMQualityControl(
  tbl,
  valid_Longitude = c(-180, 180),
  valid_Latitude = c(-90, 90),
  remove_Lon_zero = TRUE,
  remove_Lat_zero = TRUE,
  valid_Flow = c(1.999, 2.001),
  valid_AT = c(-Inf, 150),
  valid_RHi = c(-Inf, 55),
  valid_Conc = c(-Inf, 5000),
  flagAndKeep = FALSE
)
```

**Arguments**

- **tbl** single site tibble created by wrcc_parseData()
- **valid_Longitude** range of valid Longitude values
- **valid_Latitude** range of valid Latitude values
- **remove_Lon_zero** flag to remove rows where Longitude == 0
remove_Lat_zero  
flag to remove rows where Latitude == 0
valid_Flow       range of valid Flow values
valid_AT         range of valid AT values
valid_RHi        range of valid RHi values
valid_Conc       range of valid ConcHr values
flagAndKeep      flag, rather than remove, bad data during the QC process

Value

Cleaned up tibble of WRCC monitor data.

See Also

wrcc_qualityControl

Description

Examine the column names of the incoming character vector to identify different types of monitor data provided by WRCC.

The return is a list includes everything needed to identify and parse the raw data using readr::read_tsv():

- monitorType – identification string
- rawNames – column names from the data (including special characters)
- columnNames – assigned column names (special characters repaced with ".")
- columnTypes – column type string for use with readr::read_csv()

The monitorType will be one of:

- "WRCC_TYPE1" – ???
- "WRCC_TYPE2" – ???
- "UNKOWN" – ???

Usage

wrcc_identifyMonitorType(fileString)

Arguments

fileString    character string containing WRCC data
**wrcc_load**

**Value**

List including `monitorType`, `rawNames`, `columnNames` and `columnTypes`.

**References**

**WRCC Fire Cache Smoke Monitor Archive**

**Examples**

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  fileString <- wrcc_downloadData(20160701, 20160930, unitID='1307')
  monitorTypeList <- wrcc_identifyMonitorType(fileString)
}, silent = FALSE)
## End(Not run)
```

---

**wrcc_load**  
*Load Processed WRCC Monitoring Data*

**Description**

Please use `wrcc_loadAnnual` instead of this function. It will soon be deprecated.

**Usage**

```r
wrcc_load(
  year = 2017,
  baseUrl = "https://haze.airfire.org/monitoring/WRCC/RData/"
)
```

**Arguments**

- `year` desired year (integer or character representing YYYY)
- `baseUrl` base URL for WRCC meta and data files

**Value**

A `ws_monitor` object with WRCC data.
Load annual WRCC monitoring data

Description

Loads pre-generated .RData files containing annual WRCC data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The annual files loaded by this function are updated on the 15'th of each month and cover the period from the beginning of the year to the end of the last month.

For data during the last 45 days, use `wrcc_loadDaily()`.

For the most recent data, use `wrcc_loadLatest()`.

WRCC parameters include the following:

1. PM2.5

Available WRCC RData and associated log files can be seen at: https://haze.airfire.org/monitoring/WRCC/RData/

Usage

```r
wrcc_loadAnnual(
  year = NULL,
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring",
  dataDir = NULL
)
```

Arguments

- `year` Desired year (integer or character representing YYYY).
- `parameter` Parameter of interest.
- `baseUrl` Base URL for 'annual' WRCC data files.
- `dataDir` Local directory containing 'annual' data files.

Value

A `ws_monitor` object with WRCC data.

See Also

`wrcc_loadDaily`
`wrcc_loadLatest`
Examples

```r
## Not run:
# Fail gracefully if any resources are not available
try({
  wrcc_loadAnnual(2017) %>%
  monitor_subset(stateCodes='MT', tlim=c(20170701,20170930)) %>%
  monitor_dailyStatistic() %>%
  monitor_timeseriesPlot(style = 'gnats', ylim=c(0,300), xpd=NA)
  addAQIStackedBar()
  addAQILines()
  title("Montana 2017 -- WRCC Daily Average PM2.5")
}, silent = FALSE)
## End(Not run)
```

## Load recent WRCC monitoring data

### Description

Loads pre-generated .RData files containing recent WRCC data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL. The daily files loaded by this function are updated once a day, shortly after midnight and contain data for the previous 45 days.

For the most recent data, use `wrcc_loadLatest()`.

For data extended more than 45 days into the past, use `wrcc_loadAnnual()`.

WRCC parameters include the following:

1. PM2.5

Available WRCC RData and associated log files can be seen at: [https://haze.airfire.org/monitoring/WRCC/RData/latest/](https://haze.airfire.org/monitoring/WRCC/RData/latest/)

### Usage

```
wrcc_loadDaily(
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL
)
```

### Arguments

- **parameter**: Parameter of interest.
- **baseUrl**: Base URL for 'daily' AirNow data files.
- **dataDir**: Local directory containing 'daily' data files.
Value

A `ws_monitor` object with WRCC data.

See Also

wrcc_loadAnnual
wrcc_loadLatest

Examples

## Not run:
# Fail gracefully if any resources are not available
try({
  wrcc_loadDaily() %>%
    monitor_subset(stateCodes=CONUS) %>%
    monitor_map()
}, silent = FALSE)
## End(Not run)

wrcc_loadLatest
Load most recent WRCC monitoring data

Description

Loads pre-generated .RData files containing the most recent WRCC data.

If `dataDir` is defined, data will be loaded from this local directory. Otherwise, data will be loaded from the monitoring data repository maintained by PWFSL.

The files loaded by this function are updated multiple times an hour and contain data for the previous 10 days.

For daily updates covering the most recent 45 days, use `wrcc_loadDaily()`.

For data extended more than 45 days into the past, use `wrcc_loadAnnual()`.

WRCC parameters include the following:

1. PM2.5

Available RData and associated log files can be seen at: https://haze.airfire.org/monitoring/WRCC/RData/latest/

Usage

```
wrcc_loadLatest(
  parameter = "PM2.5",
  baseUrl = "https://haze.airfire.org/monitoring/latest/RData",
  dataDir = NULL
)
```
### Description

Raw character data from WRCC are parsed into a tibble. The incoming `fileString` can be read in directly from WRCC using `wrcc_downloadData()` or from a local file using `readr::read_file()`.

The type of monitor represented by this `fileString` is inferred from the column names using `wrcc_identifyMonitorType()` and appropriate column types are assigned. The character data are then processed, read into a tibble and augmented in the following ways:

1. Spaces at the beginning and end of each line are moved.
2. All header lines beginning with `:.` are removed.

### Usage

`wrcc_parseData(fileString)`
Arguments

fileString character string containing WRCC data

Value

Dataframe of WRCC raw monitor data.

References

Fire Cache Smoke Monitoring Archive

Examples

## Not run:
# Fail gracefully if any resources are not available
try({

  fileString <- wrcc_downloadData(20150701, 20150930, unitID = 'SM16')
  tbl <- wrcc_parseData(fileString)

}, silent = FALSE)

## End(Not run)

---

**wrcc_qualityControl**  
*Apply Quality Control to raw WRCC tibble*

Description

Various QC steps are taken to clean up the incoming raw tibble including:

1. Convert numeric missing value flags to `NA`.
2. Remove measurement records with values outside of valid ranges.

See the individual `wrcc_~QualityControl()` functions for details.

Usage

wrcc_qualityControl(tbl, ...)

Arguments

  - **tbl** single site tibble created by `wrcc_downloadData()`
  - ... additional parameters are passed to type-specific QC functions

Value

Cleaned up tibble of WRCC monitor data.
See Also

wrcc_EBAMQualityControl
wrcc_ESAMQualityControl
Index

* AIRSIS
  airsis_availableUnits, 28
  airsis_BAM1020QualityControl, 29
  airsis_createDataDataframe, 30
  airsis_createMetaDataframe, 30
  airsis_createMonitorObject, 31
  airsis_createRawDataframe, 34
  airsis_downloadData, 35
  airsis_EBAM_MULTI2_BQualityControl, 39
  airsis_EBAM_MULTI2QualityControl, 38
  airsis_EBAM_PLUS_MULTIQualityControl, 40
  airsis_EBAMQualityControl, 36
  airsis_ESAM_MULTIQualityControl, 43
  airsis_ESAMQualityControl, 41
  airsis_identifyMonitorType, 44
  airsis_load, 45
  airsis_loadAnnual, 45
  airsis_loadDaily, 47
  airsis_loadLatest, 48
  airsis_parseData, 49
  airsis_qualityControl, 50
  loadDaily, 69
  loadLatest, 70
  monitor_downloadAnnual, 82
  monitor_downloadDaily, 83
  monitor_downloadLatest, 84
  monitor_load, 98
  monitor_loadAnnual, 100
  monitor_loadDaily, 101
  monitor_loadLatest, 102

* AirNow
  airnow_createDataDataframes, 12
  airnow_createMetaDataframes, 14
  airnow_createMonitorObjects, 16
  airnow_downloadHourlyData, 18
  airnow_downloadParseData, 19
  airnow_downloadsites, 20
  airnow_load, 21
  airnow_loadAnnual, 22
  airnow_loadDaily, 23
  airnow_loadLatest, 25
  airnow_qualityControl, 26
  loadDaily, 69
  loadLatest, 70
  monitor_downloadAnnual, 82
  monitor_downloadDaily, 83
  monitor_downloadLatest, 84
  monitor_load, 98
  monitor_loadAnnual, 100
  monitor_loadDaily, 101
  monitor_loadLatest, 102

* EPA
  epa_createDataDataframe, 57
  epa_createMetaDataframe, 58
  epa_createMonitorObject, 59
  epa_downloadData, 60
  epa_load, 62
  epa_loadAnnual, 62
  epa_parseData, 63

* WRCC
  loadDaily, 69
  loadLatest, 70
  monitor_downloadAnnual, 82
  monitor_downloadDaily, 83
  monitor_downloadLatest, 84
  monitor_load, 98
  monitor_loadAnnual, 100
  monitor_loadDaily, 101
  monitor_loadLatest, 102
  wrcc_createDataDataframe, 155
  wrcc_createMetaDataframe, 155
  wrcc_createMonitorObject, 156
  wrcc_createRawDataframe, 158
  wrcc_downloadData, 160
INDEX

wrcc_EBAMQualityControl, 161
wrcc_ESAMQualityControl, 163
wrcc_identifyMonitorType, 164
wrcc_load, 165
wrcc_loadAnnual, 166
wrcc_loadDaily, 167
wrcc_loadLatest, 168
wrcc_parseData, 169
wrcc_qualityControl, 170

* datasets
AIRSIS, 27
AQI, 51
AQI_en, 54
AQI_es, 54
Camp_Fire, 55
Carmel_Valley, 56
CONUS, 56
Northwest_Megafires, 132
US_52, 154
WRCC, 154

* environment
esriToken, 65
getEsriToken, 67
googleApiKey, 68

setEsriToken, 139
setGoogleApiKey, 139

* plotting
addAQILegend, 5
addAQILines, 6
addAQIStackedBar, 6
addBullseye, 7
addMarker, 8
addPolygon, 9
addWindBarbs, 11
aqiColors, 52
staticmap_getEsrimapBrick, 144
staticmap_getRasterBrick, 146
staticmap_getStamenmapBrick, 148
staticmap_plotRasterBrick, 150

* raw
raw_enhance, 137
rawPlot_pollutionRose, 133
rawPlot_timeOfDaySpaghetti, 134
rawPlot_timeseries, 135
rawPlot_windRose, 136

* ws_monitor
aqiColors, 52

monitor_aqi, 71
monitor_asDataframe, 72
monitor-collapse, 73
monitor_combine, 75
monitor_dailyBarplot, 76
monitor_dailyStatistic, 77
monitor_dailyStatisticList, 78
monitor_dailyThreshold, 80
monitor_distance, 81
monitor_dygraph, 85
monitor_hourlyBarplot, 91
monitor_isEmpty, 93
monitor_isMonitor, 93
monitor_isolate, 94
monitor_join, 96
monitor_leaflet, 97
monitor_map, 104
monitor_nowcast, 105
monitor_performance, 106
monitor_performanceMap, 108
monitor_print, 110
monitor_reorder, 111
monitor_replaceData, 111
monitor_rollingMean, 112
monitor_rollingMeanPlot, 113
monitor_scaleData, 115
monitor_stamenmap, 115
monitor_staticmap, 117
monitor_subset, 119
monitor_subsetBy, 120
monitor_subsetByDistance, 121
monitor_subsetData, 122
monitor_subsetMeta, 123
monitor_timeAverage, 124
monitor_timeseriesPlot, 126
monitor_trim, 129
monitor_writeCSV, 130
monitor_writeCurrentStatusGeoJSON, 131

addAQILegend, 5
addAQILines, 6
addAQIStackedBar, 6
addBullseye, 7
addClustering, 33, 35, 158, 160
addIcon, 7
addMarker, 8
addMazamaMetadata, 31, 156
addPolygon, 9