Package 'PWFSLSmoke'

January 17, 2018

Type Package

Version 1.0.10

Title Utilities for Working with Air Quality Monitoring Data

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

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

Depends R (>= 3.1.0), dplyr, magrittr, maps, MazamaSpatialUtils (>= 0.4.4)

Imports cluster, dygraphs (>= 1.1.1.4), futile.logger, ggmap, httr, jsonlite, leaflet (>= 1.0.0), lubridate, maptools, mapproj, openair, png, raster, RColorBrewer, rgdal, RgoogleMaps, readr, reshape2, sp, stringr, xts, zoo

Suggests knitr, rmarkdown

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 it's 'AirNow' air quality site <https://www.airnow.gov>. Additional sources of PM2.5 data made accessible by the package include: 'AIRSIS' (password protected) <https://www.oceaneering.com/data-management/> and 'WRCC' <https://wrcc.dri.edu/cgi-bin/smoke.pl>. Pre-generated data compilations are provided by 'PWFSL' <https://www.fs.fed.us/pnw/pwfsl/>.

License GPL-3

VignetteBuilder knitr

Repository CRAN

LazyData true

RoxygenNote 6.0.1

NeedsCompilation no

Date/Publication 2018-01-17 18:36:19 UTC

R topics documented:

`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", ...)
```

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:

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

Usage

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

Arguments

...	additional arguments to be passed to abline()

---

addBullseye *Add a Bullseyes to a Map or RgoogleMap Plot*

Description

Draws a bullseye with concentric circles of black and white.

Usage

```
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

```
wa <- monitor_subset(Northwest_Megafires, stateCodes='WA', tlim=c(20150821,20150828))
monitorMap(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

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

Arguments

icon	object to be plotted
longitude	vector of longitudes
latitude	vector of latitudes
map	optional RgoogleMaps map object
expansion	icon expansion factor
pos	position of icon relative to location (0=center, 1=bottom, 2=left, 3=top,4=right)

Note

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:
ca <- airnow_load(20160801, 20160831, stateCodes='ca')
# Google map
map <- monitorGoogleMap(ca)
addIcon("orangeFlame", ca$meta$longitude, ca$meta$latitude, map=map, expansion=0.1)
# line map
monitorMap(ca)
addIcon("orangeFlame", ca$meta$longitude, ca$meta$latitude, expansion=0.002)

## End(Not run)
```

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'
map	optional RgoogleMaps map object or Raster* object
expansion	icon expansion factor. Ignored if width and height are specified.
...	arguments passed on to rasterImage

Examples

```
## Not run:
ca <- airnow_load(20160801, 20160831, stateCodes='ca')
# Google map
map <- monitorGoogleMap(ca)
addMarker(ca$meta$longitude, ca$meta$latitude, map=map)
# line map
monitorMap(ca)
addMarker(ca$meta$longitude, ca$meta$latitude, color = "blue", expansion = 1)

## End(Not run)
```

addShadedBackground *Add Shaded Background to a Plot*

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

`airnow_createDataDataframes` *Return Reshaped, Dataframes of AirNow Data*

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 dataframe for each of the above parameters.

Usage

```
airnow_createDataDataframes(parameters = NULL, startdate = "", 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

```
## Not run:
airnow_data <- airnow_createDataDataframes("PM2.5", 20160701)

## 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
- stateCode – ISO 3166-2 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
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,
  pwfslDataIngestSource = "AIRNOW", addGoogleMeta = TRUE)
```

Arguments

parameters	vector of names of desired pollutants or NULL for all pollutants
pwfslDataIngestSource	identifier for the source of monitoring data, e.g. 'AIRNOW'
addGoogleMeta	logicial 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

```
## Not run:
metaList <- airnow_createMetaDataframes(parameters="PM2.5")

## End(Not run)
```

`airnow_createMonitorObjects` *Obain AirNow Data and Create ws_monitor Objects*

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

```
airnow_createMonitorObjects(parameters = NULL,
  startdate = strftime(lubridate::now(), "%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`	logicial specifying wheter 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

```
## Not run:
monList <- airnow_createMonitorObjects(c("O3","PM2.5"), 20160701)
pm25 <- monList$PM2.5
o3 <- monList$O3

## End(Not run)
```

`airnow_downloadHourlyData` *Download Hourly Data from AirNow*

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.

Usage

```
airnow_downloadHourlyData(datestamp = strftime(lubridate::now(),
  "%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.

See Also

airnow_createDataDataframes

airnow_downloadParseData

Examples

```
## Not run:
df <- airnow_downloadHourlyData(2016070112)

## 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 ore 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

```
airnow_downloadParseData(parameters = NULL,
  startdate = strftime(lubridate::now(), "%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

```
## Not run:
tbl <- airnow_downloadParseData("PM2.5", 2016070112, hours=24)

## End(Not run)
```

---

airnow_downloadSites *Download AirNow Site Location Metadata*

Description

The 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

```
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

```
## Not run:
sites <- airnow_downloadSites()

## End(Not run)
```

`airnow_load` *Load Processed AirNow Monitoring Data*

Description

Loads pre-generated .RData files containing AirNow data. This function loads annual or monthly fiels from the data archive.

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

AirNow parameters include the following:

1. PM2.5

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

Usage

```
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.

See Also

airnow_loadDaily

airnow_loadLatest

Examples

```
## Not run:
airnow <- airnow_load(2017, 09)
airnow_conus <- monitor_subset(airnow, stateCodes=CONUS)
monitorLeaflet(airnow_conus)

## End(Not run)
```

---

airnow_loadDaily *Load Recent Processed AirNow Monitoring Data*

Description

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

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 `airnow_loadLatest()`.

AirNow parameters include the following:

1. PM2.5

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

Usage

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

Arguments

`parameter`	parameter of interest
`baseUrl`	base URL for AirNow data

Value

A *ws_monitor* object with AirNow data.

See Also

airnow_load

airnow_loadLatest

Examples

```
## Not run:
airnow <- airnow_loadDaily()
airnow %>% monitor_subset(stateCodes=CONUS) %>% monitorMap()

## End(Not run)
```

---

airnow_loadLatest *Load Most Recent Processed AirNow Monitoring Data*

Description

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

AirNow parameters include the following:

1. PM2.5

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

Usage

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

Arguments

parameter	parameter of interest
baseUrl	base URL for AirNow data

Value

A *ws_monitor* object with AirNow data.

See Also

airnow_load

airnow_loadDaily

Examples

```
## Not run:
airnow <- airnow_loadLatest()
ca_mean <- monitor_subset(airnow, stateCodes='CA') %>%
           monitor_collapse()
monitorPlot_timeseries(ca_mean, shadedNight=TRUE)

## 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

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

Arguments

`df`	multi-site restructured dataframe created within `airnow_createDataDataframe()`
`limits`	lo and hi 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&S
11-06&EndDate=2017-11-07

The `AIRSIS` objectis 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 Available Unit Identifiers from AIRSIS*

Description

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

Usage

```
airsis_availableUnits(startdate = strftime(lubridate::now(), "%Y010100", tz =
  "UTC"), enddate = strftime(lubridate::now(), "%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

```
## Not run:
unitIDs <- airsis_availableUnits(20150701, 20151231,
                                 provider='USFS',
                                 unitTypes=c('EBAM','ESAM'))

## End(Not run)
```

---

airsis_BAM1020QualityControl

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

```
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.

See Also

airsis_qualityControl

---

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"                 "pwfslID"
[16] "pwfslDataIngestSource" "telemetryAggregator"   "telemetryUnitID"
```

Usage

```
airsis_createMetaDataframe(tbl, provider = as.character(NA),
  unitID = as.character(NA), pwfslDataIngestSource = "AIRSIS",
  existingMeta = NULL, addGoogleMeta = TRUE)
```

Arguments

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

Value

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

See Also

`addGoogleMetadata`

`addMazamaMetadata`

`airsis_createMonitorObject` *Obain AIRSIS Data and Create ws_monitor Object*

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,45)`
- `valid_Conc = c(-Inf,5.000)`

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

Usage

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

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
`...`	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

```
## Not run:
initializeMazamaSpatialUtils()
usfs_1013 <- airsis_createMonitorObject(20150301, 20150831, 'USFS', unitID='1013')
monitorLeaflet(usfs_1013)

## End(Not run)
```

`airsis_createRawDataframe` *Obain AIRSIS Data and Create 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(), "%Y010100",
  tz = "UTC"), enddate = strftime(lubridate::now(), "%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 tibblee of 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

Examples

```
## Not run:
raw <- airsis_createRawDataframe(startdate=20160901, provider='USFS', unitID='1033')
raw <- raw_enhance(raw)
rawPlot_timeseries(raw,tlim=c(20160908,20160917))

## End(Not run)
```

---

airsis_downloadData *Download Data from AIRSIS*

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

```
airsis_downloadData(startdate = strftime(lubridate::now(), "%Y0101", tz =
  "UTC"), enddate = strftime(lubridate::now(), "%Y%m%d", tz = "UTC"),
  provider = "USFS", unitID = NULL,
  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']
unitID	unit identifier
baseUrl	base URL for data queries

Value

String containing AIRSIS output.

References

Interagency Real Time Smoke Monitoring

Examples

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

## End(Not run)
```

`airsis_EBAMQualityControl`
Apply Quality Control to Raw AIRSIS EBAM Tibble

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

```
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, 45), 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_ESAMQualityControl` *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

```
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_identifyMonitorType` *Identify AIRSIS Monitor Type*

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)
- `columnNames` – assigned column names (special characters repaced with '.')
- `columnTypes` – 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

```
airsis_identifyMonitorType(df)
```

Arguments

`df`	dataframe or raw character string containing AIRSIS data

Value

List including `monitorType`, `rawNames`, `columnNames` and `columnTypes`.

References

Interagency Real Time Smoke Monitoring

Examples

```
## Not run:
fileString <- airsis_downloadData( 20150701, 20151231, provider='USFS', unitID='1026')
monitorTypeList <- airsis_identifyMonitorType(fileString)

## End(Not run)
```

`airsis_load` *Load Processed AIRSIS Monitoring Data*

Description

Loads pre-generated .RData files containing AIRSIS PM2.5 data.

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

Usage

```
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.

See Also

`airsis_loadDaily`

`airsis_loadLatest`

Examples

```
## Not run:
airsis <- airsis_load(2017)
airsis_conus <- monitor_subset(airsis, stateCodes=CONUS)
monitorLeaflet(airsis_conus)

## End(Not run)
```

`airsis_loadDaily` *Load Recent AIRSIS Monitoring Data*

Description

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

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 `airsis_loadLatest()`.

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

Usage

```
airsis_loadDaily(baseUrl = "https://haze.airfire.org/monitoring/AIRSIS/RData/")
```

Arguments

`baseUrl`	location of the AIRSIS latest data file

Value

A *ws_monitor* object with AIRSIS data.

See Also

`airsis_load`

`airsis_loadLatest`

Examples

```
## Not run:
airsis <- airsis_loadDaily()

## End(Not run)
```

airsis_loadLatest *Load Recent AIRSIS Monitoring Data*

Description

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

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

Usage

```
airsis_loadLatest(baseUrl = "https://haze.airfire.org/monitoring/AIRSIS/RData/")
```

Arguments

baseUrl	location of the AIRSIS latest data file

Value

A *ws_monitor* object with AIRSIS data.

See Also

airsis_load

airsis_loadDaily

Examples

```
## Not run:
airsis <- airsis_loadLatest()

## End(Not run)
```

airsis_parseData *Parse AIRSIS Data String*

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 foward 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 backwords 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:
fileString <- airsis_downloadData(20150701, 20151231, provider='USFS', unitID='1026')
tbl <- airsis_parseData(fileString)

## 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,45)
- 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 five elements

Details

AQI breaks and associated names and colors

AQI breaks and colors are defined in https://www3.epa.gov/airnow/aqi-technical-assistance-document-may2016.pdf

Note

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

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 wildifre 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. Data are stored as a *ws_monitor* object and are used in some examples in the package documentation.

Format

A list with two elements

Details

Carmel Valley example dataset

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 the Distance Between Points*

Description

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

Usage

```
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.

References

https://www.r-bloggers.com/great-circle-distance-calculations-in-r/

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 addtional 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.

Usage

```
epa_createDataDataframe(tbl)
```

Arguments

`tbl`	an EPA raw tibble after metadata enhancement

Value

A `data` dataframe for use in a *ws_monitor* object.

epa_createMetaDataframe

Create Sites Metadata Dataframe

Description

After addtional 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:

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

Usage

```
epa_createMetaDataframe(tbl, pwfslDataIngestSource = "EPA",
  existingMeta = NULL, addGoogleMeta = TRUE)
```

Arguments

tbl	an EPA raw tibble after metadata enhancement
pwfslDataIngestSource	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	logicial specifying wheter 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

```
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	logicial specifying wheter 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

```
## Not run:
initializeMazamaSpatialUtils()
zipFile <- epa_downloadData(2016, "88101", downloadDir='~/Data/EPA')
mon <- epa_createMonitorObject(zipFile, addGoogleMeta=FALSE)

## End(Not run)
```

---

epa_downloadData *Download Data from EPA*

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

Usage

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

Arguments

year	year
parameterCode	pollutant code
downloadDir	directoroy 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:
zipFile <- epa_downloadData(2016, "88101", '~/Data/EPA')
tbl <- epa_parseData(zipFile, "PM2.5")

## End(Not run)
```

---

epa_load *Load Processed EPA Monitoring Data*

Description

Loads a pre-generated .RData file containing a year's worth of monitoring 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)

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

Usage

```
epa_load(year = strftime(lubridate::now(), "%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.

References

EPA AirData Pre-Generated Data Files

Examples

```
## Not run:
epa_frm <- epa_load(2015, 88101)
epa_frm_conus <- monitor_subset(epa_frm, stateCodes=CONUS)
monitorLeaflet(epa_frm_conus)

## End(Not run)
```

---

epa_parseData *Parse Data from EPA*

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. Temperatue

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

Examples

```
## Not run:
zipFile <- epa_downloadData(2016, "88101", '~/Data/EPA')
tbl <- epa_parseData(zipFile, "PM2.5")

## End(Not run)
```

---

esriMap_getMap *Download a Spatial Raster Object from ESRI*

Description

Downloads a PNG from ESRI and creates a `raster::rasterBrick` object with layers for red, green, and blue. This can then passed as the `mapRaster` object to the `esriMap_plotOnStaticMap()` function for plotting.

Available `maptypes` include:

- natGeo
- worldStreetMap
- worldTopoMap
- satellite
- deLorme

Additional base maps are found at: http://resources.arcgis.com/en/help/arcgis-rest-api/index.html#/Basemaps/02r3000001mt000000/

Usage

```
esriMap_getMap(centerLon = NULL, centerLat = NULL, bboxString = NULL,
  bboxSR = "4326", maptype = "worldStreetMap", zoom = 12, width = 640,
  height = 640, crs = sp::CRS("+init=epsg:4326"), additionalArgs = NULL)
```

Arguments

`centerLon`	map center longitude
`centerLat`	map center latitude
`bboxString`	comma separated string with bounding box (xmin, ymin, xmax, ymax). If not null, centerLon, centerLat, and zoom are ignored.
`bboxSR`	spatial reference of the bounding box
`maptype`	map type
`zoom`	map zoom level; corresponds to googleMaps zoom level
`width`	width of image, in pixels
`height`	height of image, in pixels

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.
additionalArgs	character string with additional arguments to be pasted into the image URL eg. "&rotation=90"

Value

A rasterBrick object which can be plotted with esriMap_plotOnStaticMap() 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 bboxString 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.

References

http://resources.arcgis.com/en/help/arcgis-rest-api/index.html#/Export_Map/02r3000000v7000000/

See Also

esriMap_plotOnStaticMap

Examples

```
## Not run:
map <- esriMap_getMap(-122.3318, 47.668)
esriMap_plotOnStaticMap(map)

## End(Not run)
```

---

esriMap_plotOnStaticMap

Plot a map from a RGB rasterBrick.

Description

The map is plotted using plotRGB from **raster**.

Usage

```
esriMap_plotOnStaticMap(mapRaster, grayscale = FALSE, ...)
```

Arguments

mapRaster	a RGB rasterBrick object. It is assumed that layer 1 represents red, layer 2 represents gree, and layer 3 represents blue.
grayscale	logical, if TRUE one layer is plotted with grayscale values. If FALSE, a color map is plotted from red, green, and blue colors.
...	arguments passed on to `plot` (for grayscale = TRUE) or `plotRGB` (for grayscale = FALSE)

Value

An plot of the map

See Also

esriMap_getMap

Examples

```
## Not run:
map <- esriMap_getMap(-122.3318, 47.668)
esriMap_plotOnStaticMap(map)
esriMap_plotOnStaticMap(map, grayscale = TRUE)

## End(Not run)
```

---

initializeMazamaSpatialUtils

Initialize Mazama Spatial Utils

Description

Convenience function that wraps:

```
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

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

Arguments

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

See Also

{link{logger.setup}

---

logger.debug *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `DEBUG` level log statements.

Usage

```
logger.debug(msg, ...)
```

Arguments

msg	message with format strings applied to additional arguments
...	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

---

logger.error *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `ERROR` level log statements.

Usage

```
logger.error(msg, ...)
```

Arguments

`msg`	message with format strings applied to additional arguments
`...`	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

`logger.setup`

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

`logger.fatal` *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `FATAL` level log statements.

Usage

```
logger.fatal(msg, ...)
```

Arguments

`msg`	message with format strings applied to additional arguments
`...`	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

`logger.setup`

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

---

logger.info *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `INFO` level log statements.

Usage

```
logger.info(msg, ...)
```

Arguments

`msg`	message with format strings applied to additional arguments
`...`	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

`logger.setup`

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

---

logger.setLevel *Set Console Log Level*

Description

By default, the looger threshold is set to FATAL so that the console will typically receive no log messages. By setting the level to one of the other log levels: TRACE, DEBUG, INFO, WARN, ERROR users can see logging messages while running commands at the command line.

Usage

```
logger.setLevel(level)
```

Arguments

level	threshold level

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Set up console logging only
logger.setup()
logger.setLevel(DEBUG)

## End(Not run)
```

`logger.setup` *Set Up Python-Style Logging*

Description

Good logging allows package developers and users to create log files at different levels to track and debug lengthy or complex calculations. "Python-style" logging is intended to suggest that users should set up multiple log files for different log severities so that the `errorLog` will contain only log messages at or above the `ERROR` level while a `debugLog` will contain log messages at the `DEBUG` level as well as all higher levels.

Python-style log files are set up with `logger.setup()`. Logs can be set up for any combination of log levels. Accepting the default `NULL` setting for any log file simply means that log file will not be created.

Python-style logging requires the use of `logger.debug()` style logging statements as seen in the example below.

Usage

```
logger.setup(traceLog = NULL, debugLog = NULL, infoLog = NULL,
  warnLog = NULL, errorLog = NULL, fatalLog = NULL)
```

Arguments

`traceLog`	file name or full path where `logger.trace()` messages will be sent
`debugLog`	file name or full path where `logger.debug()` messages will be sent
`infoLog`	file name or full path where `logger.info()` messages will be sent
`warnLog`	file name or full path where `logger.warn()` messages will be sent
`errorLog`	file name or full path where `logger.error()` messages will be sent
`fatalLog`	file name or full path where `logger.fatal()` messages will be sent

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.trace logger.debug logger.info logger.warn logger.error logger.fatal

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow lot statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

---

logger.trace *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate TRACE level log statements.

Usage

```
logger.trace(msg, ...)
```

Arguments

msg	message with format strings applied to additional arguments
...	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

---

logger.warn *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `WARN` level log statements.

Usage

```
logger.warn(msg, ...)
```

Arguments

`msg`	message with format strings applied to additional arguments
`...`	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

`logger.setup`

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

---

logLevels *Log Levels*

Description

Log levels matching those found in **futile.logger**. Available levels include:

```
FATAL ERROR WARN INFO DEBUG TRACE
```

Usage

```
FATAL
```

Format

An object of class `integer` of length 1.

---

monitorDygraph *Create Interactive Time Series Plot*

Description

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

Usage

```
monitorDygraph(ws_monitor, title = "title", ylab = "PM2.5 Concentration",
  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 YYYYMMDD[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

```
## Not run:
airnow <- airnow_load(20140913, 20141010)
King_Fire <- monitor_subsetByDistance(airnow,
                                      longitude=-120.604,
                                      latitude=38.782,
                                      radius=50)
monitorDygraph(King_Fire, title='KingFire/California/2014', rollPeriod=3)

## End(Not run)
```

---

monitorEsriMap *Create an ESRI Map of ws_monitor Object*

Description

Creates a Google map of a *ws_monitor* object using the **RgoogleMaps** package.

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

Usage

```
monitorEsriMap(ws_monitor, slice = get("max"), breaks = AQI$breaks_24,
  colors = AQI$colors, width = 640, height = 640, centerLon = NULL,
  centerLat = NULL, zoom = NULL, maptype = "worldStreetMap",
  grayscale = FALSE, mapRaster = 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 – defautls 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
mapRaster	optional RGB Raster* object returned from esriMap_getMap)
cex	character expansion for points
pch	plotting character for points
...	arguments passed on to esriMap_plotOnStaticMap (*e.g.* destfile, cex, pch, etc.)

Value

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

Examples

```
## Not run:
N_M <- Northwest_Megafires
# monitorLeaflet(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))
monitorEsriMap(Spokane)

## End(Not run)
```

---

monitorGoogleMap *Create a Google Map of ws_monitor Object*

Description

Creates a Google map of a *ws_monitor* object using the **RgoogleMaps** package.

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

Usage

```
monitorGoogleMap(ws_monitor, slice = get("max"), breaks = AQI$breaks_24,
  colors = AQI$colors, width = 640, height = 640, centerLon = NULL,
  centerLat = NULL, zoom = NULL, maptype = "roadmap", grayscale = FALSE,
  map = NULL, ...)
```

Arguments

ws_monitor	*ws_monitor* object
slice	either a time index or a function used to collapse the time axis – defautls 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
map	optional map object returned from monitorGoogleMap()
...	arguments passed on to RgoogleMaps::PlotOnStaticMap() (*e.g.* destfile, cex, pch, etc.)

Value

A *MyMap* RgoogleMaps map object object that can serve as a base plot.

Examples

```
## Not run:
N_M <- Northwest_Megafires
# monitorLeaflet(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))
monitorGoogleMap(Spokane)

## End(Not run)
```

`monitorLeaflet` *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 timeseires 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

```
monitorLeaflet(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 – defautls 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

Initiates the interactive leaflet plot in Rstudio's 'Viewer' tab.

Examples

```
## Not run:
airnow <- airnow_load(20140913, 20141010)
v_low <- AQI$breaks_24[4]
CA_unhealthy_monitors <- monitor_subset(airnow, stateCodes='CA', vlim=c(v_low, Inf))
monitorLeaflet(CA_unhealthy_monitors, maptype="toner")

## End(Not run)
```

---

monitorMap *Create 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

```
monitorMap(ws_monitor, slice = get("max"), breaks = AQI$breaks_24,
  colors = AQI$colors, 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
`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

```
N_M <- monitor_subset(Northwest_Megafires, tlim=c(20150821,20150828))
monitorMap(N_M, cex=2)
addAQILegend()
```

---

monitorMap_performance

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

```
monitorMap_performance(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, ...)
```

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 paramater
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() funciton 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

```
## Not run:
# Spokane summer of 2015
wa <- airnow_load(20150701, 20150930, stateCodes='WA')
wa <- monitor_rollingMean(wa, width=3)
MonroeSt <- monitor_subset(wa, monitorIDs="530630047_01")
monitorMap_performance(wa, MonroeSt, cex=2)
title('Heidike Skill of monitors predicting another monitor.')

## End(Not run)
```

---

monitorPlot_dailyBarplot

Create Daily Barplot

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

```
monitorPlot_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 YYYYMMDD[HH])
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

```
## Not run:
N_M <- monitor_subset(Northwest_Megafires, tlim=c(20150715,20150930))
main <- "Daily Average PM2.5 for Omak, WA"
monitorPlot_dailyBarplot(N_M, monitorID="530470013", main=main,
                         labels_x_nudge=1)
addAQILegend(fill=rev(AQI$colors), pch=NULL)

## End(Not run)
```

monitorPlot_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

```
monitorPlot_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 YYYYMMDD[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

```
C_V <- monitor_subset(Carmel_Valley, tlim=c(2016080800,2016081023),
                      timezone='America/Los_Angeles')
monitorPlot_hourlyBarplot(C_V, main='1-Hourly Average PM2.5',
                          labels_x_nudge=1, labels_y_nudge=0)
```

---

monitorPlot_rollingMean

Create Rolling Mean Plot

Description

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

Usage

```
monitorPlot_rollingMean(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)
```

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 insufficicnet valid data to calculate mean
tlim	optional vector with start and end times (integer or character representing YYYYMMDD[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

```
N_M <- Northwest_Megafires
Roseburg <- monitor_subset(N_M, tlim=c(20150821, 20150831),
                           monitorIDs=c('410190002_01'))
monitorPlot_rollingMean(Roseburg, shadedNight=TRUE)
```

---

monitorPlot_timeOfDaySpaghetti

Create Time of Day Spaghetti Plot

Description

Creates a spaghetti plot of PM2.5 levels by hour for one or more days. The average by hour over the period is also calculated and plotted as a thick red line.

Usage

```
monitorPlot_timeOfDaySpaghetti(ws_monitor, monitorID = NULL, tlim = NULL,
  ylim = NULL, aqiLines = TRUE, shadedNight = TRUE, title = NULL, ...)
```

Arguments

ws_monitor	*ws_monitor* object
monitorID	id for a specific monitor in the ws_monitor object
tlim	optional vector with start and end times (integer or character representing YYYYMMDD[HH])
ylim	y limits for the plot
aqiLines	horizontal lines indicating AQI levels
shadedNight	add nighttime shading based on of middle day in selected period
title	plot title
...	additional arguments to pass to `lines()`

Examples

```
monitorPlot_timeOfDaySpaghetti(Carmel_Valley, tlim=c(20160801,20160809))
```

---

monitorPlot_timeseries

Create Timeseries Plot

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

```
monitorPlot_timeseries(ws_monitor, monitorID = NULL, tlim = NULL,
  localTime = TRUE, style = NULL, shadedNight = FALSE, add = FALSE,
  gridPos = "", gridCol = "black", 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 YYYYMMDD[HH])

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

```
N_M <- Northwest_Megafires
# monitorLeaflet(N_M) # to identify Spokane monitorIDs
Spokane <- monitor_subsetBy(N_M, stringr::str_detect(N_M$meta$monitorID,'^53063'))
monitorPlot_timeseries(Spokane, style='gnats')
title('Spokane PM2.5 values, 2015')
monitorPlot_timeseries(Spokane, tlim=c(20150801,20150831), style='aqidots', pch=16)
addAQILegend()
title('Spokane PM2.5 values, August 2015')
monitorPlot_timeseries(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_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://www3.epa.gov/airnow/aqi-technical-assistance-document-may2016.pdf

https://www.ecfr.gov/cgi-bin/retrieveECFR?n=40y6.0.1.1.6#ap40.6.58_161.g

Examples

```
## Not run:
ws_monitor <- monitor_subset(Northwest_Megafires, tlim=c(20150815,20150831))
aqi <- monitor_aqi(ws_monitor)
monitorPlot_timeseries(aqi, monitorID=aqi$meta$monitorID[1], ylab="PM25 AQI")

## End(Not run)
```

---

monitor_collapse *Collapse a ws_monitor Object into a ws_monitor Object with a Single Monitor*

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.

Usage

```
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. (Default = mean of the longitudes)
latitude	latitude of the collapsed monitoring station. (Default = mean of the latitudes)
monitorID	monitor ID of the collapsed monitoring station.
FUN	function to be applied to all the monitors at a single time index.
na.rm	logical value indicating 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 corresponds to 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

```
N_M <- Northwest_Megafires
# monitorLeaflet(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)
monitorPlot_timeseries(Spokane, tlim=c(20150619,20150626),
                       style='gnats', shadedNight=TRUE)
monitorPlot_timeseries(Spokane_max, col='red', type='s', add=TRUE)
monitorPlot_timeseries(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

```
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

```
## Not run:
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)
monitorLeaflet(ws_monitor)

## End(Not run)
```

---

monitor_dailyStatistic

Calculate Daily Statistics

Description

Calculates daily statistics for each monitor in `ws_monitor`.

Usage

```
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.

Value

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

```
## Not run:
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
monitorPlot_timeseries(MethowValley, style='gnats', col='transparent')
monitorPlot_timeseries(MethowValley, style='gnats', monitorID=TwispID,
                       col='forestgreen', add=TRUE)
monitorPlot_timeseries(MethowValley, style='gnats', monitorID=WinthropID,
                       col='purple', add=TRUE)
monitorPlot_timeseries(MethowValley_dailyMean, type='s', lwd=2, monitorID=TwispID,
                       col='forestgreen', add=TRUE)
monitorPlot_timeseries(MethowValley_dailyMean, type='s', lwd=2, monitorID=WinthropID,
                       col='purple', add=TRUE)
addAQILines()
addAQILegend("topleft", lwd=1, pch=NULL)
title("Winthrop & Twisp, Washington Daily Mean PM2.5, 2015")

## End(Not run)
```

---

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

```
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.

Value

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

Examples

```
## Not run:
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)
monitorPlot_timeseries(Twisp_daily, type='h', lwd=6, ylab="Hours")
title("Twisp, Washington Hours per day Above 'Unhealthy', 2015")

## End(Not run)
```

---

monitor_distance — *Calculate Distances From ws_monitor Monitors to 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

```
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

```
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_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

```
monitor_isEmpty(ws_monitor)
```

Arguments

ws_monitor	*ws_monitor* object

Value

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

`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 YYYYMMDD[HH] or `POSIXct`)
`monitorIDs`	optional vector of monitorIDs
`stateCodes`	optional vector of stateCodes
`timezone`	Olson timezone passed to `link{parseDatetime}` when parsing numeric `tlim`

Value

A list of isolated ws_monitor objects.

See Also

monitor_subset

Examples

```
N_M <- Northwest_Megafires
# monitorLeaflet(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_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

```
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

Value

A *ws_monitor* object with merged timeseries.

Examples

```
## 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_nowcast` *Apply Nowcast Algorithm to ws_monitor Object*

Description

A Nowcast algorithm is applied to the data in 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

```
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`	calcluate 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 funtion 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://www3.epa.gov/airnow/ani/pm25_aqi_reporting_nowcast_overview.pdf

https://aqicn.org/faq/2015-03-15/air-quality-nowcast-a-beginners-guide/

https://forum.airnowtech.org/t/the-nowcast-for-ozone-and-pm/172

https://forum.airnowtech.org/t/the-aqi-equation/169

https://airnow.zendesk.com/hc/en-us/articles/211625598-How-does-AirNow-make-the-Current-PM-Air-Qua

https://forum.airnowtech.org/t/how-does-airnow-handle-negative-hourly-concentrations/143

Examples

```
## Not run:
N_M <- monitor_subset(Northwest_Megafires, tlim=c(20150815,20150831))
Omak <- monitor_subset(N_M, monitorIDs='530470013_01')
Omak_nowcast <- monitor_nowcast(Omak, includeShortTerm=TRUE)
monitorPlot_timeseries(Omak, type='l', lwd=2)
monitorPlot_timeseries(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
monitorPlot_timeseries(Omak, tlim=c(20150823,20150825))
monitorPlot_timeseries(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")

## End(Not run)
```

---

monitor_performance *Calculate 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`.

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

Usage

```
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

monitorMap_performance

skill_confusionMatrix

Examples

```
## Not run:
airnow <- airnow_load(20150101, 20151231)
airnow_dailyAvg <- monitor_dailyStatistic(airnow, mean)
airnow_dailyAvg <- monitor_subset(airnow_dailyAvg, stateCodes='WA')
airnow_m1 <- airnow_load(20141231, 20151230)
airnow_dailyAvg_m1 <- monitor_dailyStatistic(airnow_m1, mean)
airnow_dailyAvg_m1 <- monitor_subset(airnow_dailyAvg_m1, stateCodes='WA')
threshold <- AQI$breaks_24[3]
performanceMetrices <- monitor_performance(airnow_dailyAvg_m1,
                                           airnow_dailyAve,
                                           threshold, threshold)

## End(Not run)
```

---

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

```
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	flag 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 RR 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 RR expression used to identify values for replacement
value	replacement value

Examples

```
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 thaty have been processed by a rolling mean algorithm.

Examples

```
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")
monitorPlot_timeseries(wa_smoky, type='l', shadedNight=TRUE)
monitorPlot_timeseries(wa_smoky_3hr, type='l', col='red', add=TRUE)
```

```
monitorPlot_timeseries(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')
```

`monitor_scaleData` *Scale ws_monitor Data*

Description

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

Usage

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

Arguments

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

Examples

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

`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

```
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 link{parseDatetime} when parsing numeric tlim

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

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

---

monitor_subsetBy *Subset ws_monitor Object with a Filter*

Description

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

Usage

```
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

```
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

```
## Not run:
airnow <- airnow_load(20140913, 20141010)
KingFire <- monitor_subsetByDistance(airnow, longitude=-120.604, latitude=38.782, radius=50)
monitorLeaflet(KingFire)

## End(Not run)
```

`monitor_subsetData` *Subset ws_monitor Object 'data' Dataframe*

Description

Subsets a *ws_monitor* object's `data` dataframe by removing any monitors that lie outisde 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

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

Arguments

`data`	*ws_monitor* object `data` dataframe
`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 monitorIDs
`dropMonitors`	flag specifying whether to remove columns – defaults to `FALSE`
`timezone`	Olson timezone passed to `link{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 <= 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 outisde 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

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

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.

monitor_timeAverage *Calculate Time Averages*

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 proccessed by openair::timeAverage().

Examples

```
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_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:
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)
monitorPlot_dailyBarplot(Deschutes)

## End(Not run)
```

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). Data are stored as a *ws_monitor* object and are used in many examples in the package documentation.

Format

A list with two elements

Details

Northwest_Megafires example dataset

---

parseDatetime *Parse Datetime Strings*

Description

8-, 10-, 12- and 14-digit formats are understood, e.g: 20150721 to 20150721000000. Integers will be converted to character before parsing. All incoming dates will be interpreted in the specified `timezone` ("UTC" by default).

If `datetime` is a `POSIXct` it will be returned unmodified.

Usage

```
parseDatetime(datetime, timezone = "UTC")
```

Arguments

datetime	vector of character or integer datetimes in YYYYMMDD[HHMMSS] format (or POSIXct)
timezone	Olson timezone at the location of interest

Value

POSIXct datetimes.

Examples

```
starttime <- parseDatetime(2015080718,timezone = "America/Los_Angeles")
```

---

rawPlot_pollutionRose *Create Pollution Rose Plot from a Raw Dataframe*

Description

Create pollution rose plot from an enhanced raw dataframe. 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 YYYYMMDD[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

```
## Not run:
raw <- airsis_createRawDataframe(20160901, 20161015, 'USFS', 1012)
raw <- raw_enhance(raw)
rawPlot_pollutionRose(raw)

## End(Not run)
```

`rawPlot_timeOfDaySpaghetti` *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:
raw <- airsis_createRawDataframe(20160901, 20161015, 'USFS', 1012)
raw <- raw_enhance(raw)
rawPlot_timeOfDaySpaghetti(raw,parameter="temperature")

## 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

```
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 YYYYMMDD[HH])
`localTime`	logical specifying whether `tlim` is in local time or UTC
`shadedNight`	add nighttime shading
`shadedBackground`	add vertical lines for a second parameter
`sbLwd`	shaded background line width
`add`	logical specifying whether to add to the current plot
`gridPos`	position of grid lines either `'over'`, `'under'` or `''` 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 pass to lines() function

Details

Note that for multiple deployments, `shadedNight` defaults to use the lat/lon for the first deployment, which in theory could be somewhat unrepresentative, such as if deployments have a large range in latitude.

Note

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

`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 YYYYMMDD[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

```
## Not run:
raw <- airsis_createRawDataframe(20160901, 20161015, provider='USFS', unitID=1012)
raw <- raw_enhance(raw)
rawPlot_windRose(raw)

## 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

```
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:
df <- airsis_createRawDataframe(startdate=20160901, enddate=20161015, provider='USFS', unitID=1012)
df <- raw_enhance(df)
rawPlot_timeseries(df, tlim=c(20160908,20160917))

## End(Not run)
```

raw_getHighlightDates *Return Day Stamps for Values Above a Threshold*

Description

Returna 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:
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)
```

skill_confusionMatrix *Confusion Matrix Statistics*

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 postive)
- FP (false postive) (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
- expectedAccuracy = (((TP + FP) * (TP + FN) / total) + ((FP + TN) * sum(FN + TN) / total)) / total
- heidikeSkill = kappa = (accuracy - expectedAccuracy) / (1 - expectedAccuracy)
- bias = (TP + FP) / (TP + FN)
- hitRate = TP / (TP + FN)
- falseAlarmRate = FP / (FP + TN)
- pierceSkill = ((TP * TN) - (FP * FN)) / ((FP + TN) * (TP + FN))
- criticalSuccess = TP / (TP + FP + FN)
- oddsRatioSkill = yulesQ = ((TP * TN) - (FP * FN)) / ((TP * TN) + (FP * FN))

Usage

```
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

```
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

```
## Not run:
# Spokane summer of 2015
airnow <- airnow_load(20150701,20150930)
airnow <- monitor_rollingMean(airnow, width=3)
MonroeSt <- monitor_subset(airnow, monitorIDs="530630047_01")
EBroadway <- monitor_subset(airnow, monitorIDs="530639997_01")
rocList <- skill_ROC(EBroadway, MonroeSt, t1Range=c(0,100), t2=55)
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

```
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

```
## Not run:
# Spokane summer of 2015
airnow <- airnow_load(20150701,20150930)
airnow <- monitor_rollingMean(airnow, width=3)
MonroeSt <- monitor_subset(airnow, monitorIDs="530630047_01")
EBroadway <- monitor_subset(airnow, monitorIDs="530639997_01")
skill_ROCPlot(EBroadway, MonroeSt)

## End(Not run)
```

---

timeInfo *Get Time Related Information*

Description

Calculate the local time at the target location, 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:

- `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, 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

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
monitorPlot_timeseries(carmel_day, shadedNight=TRUE, pch=8, col='goldenrod')
monitorPlot_timeseries(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 http://www.wrcc.dri.edu/cgi-bin/smoke.plFire Cache Smoke Monitor Archive provides access to a variety of monitors that can be accessed with the wrcc_createMonitorObject function. Use of this funciton 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 types was created on Feb 09, 2017.

`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 datafra,e 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:

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

Usage

```
wrcc_createMetaDataframe(tbl, unitID = as.character(NA),
  pwfslDataIngestSource = "WRCC", existingMeta = NULL)
```

Arguments

tbl	single site WRCC tibble after metadata enhancement
unitID	character or numeric WRCC unit identifier
pwfslDataIngestSource	identifier for the source of monitoring data, e.g. 'WRCC', 'WRCC_DUMPFILE'
existingMeta	existing 'meta' dataframe from which to obtain metadata for known monitor deployments

Value

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

See Also

addGoogleMetadata

addMazamaMetadata

---

wrcc_createMonitorObject

Obtain WRCC Data and Create ws_monitor Object

Description

Obtains monitor data from an WRCC 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 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,45)
- 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(), "%Y010100",
  tz = "UTC"), enddate = strftime(lubridate::now(), "%Y%m%d23", tz =
  "UTC"), unitID = NULL, clusterDiameter = 1000, zeroMinimum = TRUE,
  baseUrl = "https://wrcc.dri.edu/cgi-bin/wea_list2.pl", saveFile = NULL,
  ...)
```

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
...	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

```
## Not run:
initializeMazamaSpatialUtils()
sm13 <- wrcc_createMonitorObject(20150301, 20150831, unitID='sm13')
monitorLeaflet(sm13)

## End(Not run)
```

`wrcc_createRawDataframe` *Obtain WRCC Data and Parse into Tibbler*

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

```
wrcc_createRawDataframe(startdate = strftime(lubridate::now(), "%Y010100", tz
  = "UTC"), enddate = strftime(lubridate::now(), "%Y%m%d23", tz = "UTC"),
  unitID = NULL, clusterDiameter = 1000,
  baseUrl = "http://www.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 http://www.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:
tbl <- wrcc_createRawDataframe(20150701, 20150930, unitID='SM16')

## End(Not run)
```

`wrcc_downloadData` *Download Data from WRCC*

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 http://www.wrcc.dri.edu/cgi-bin/smoke.pl.

Usage

```
wrcc_downloadData(startdate = strftime(lubridate::now(), "%Y010101", tz =
  "UTC"), enddate = strftime(lubridate::now(), "%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

```
## Not run:
fileString <- wrcc_downloadData(20150701, 20150930, unitID='SM16')
df <- wrcc_parseData(fileString)

## End(Not run)
```

---

wrcc_EBAMQualityControl

Apply Quality Control to Raw WRCC EBAM Titbble

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, 45), valid_Conc = c(-Inf,
  5000), flagAndKeep = FALSE)
```

Arguments

tbl	single site titbble 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 titbble 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

```
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

`wrcc_identifyMonitorType` *Identify WRCC Monitor Type*

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

Value

List including `monitorType`, `rawNames`, `columnNames` and `columnTypes`.

References

WRCC Fire Cache Smoke Monitor Archive

Examples

```
## Not run:
fileString <- wrcc_downloadData(20160701, 20160930, unitID='1307')
monitorTypeList <- wrcc_identifyMonitorType(fileString)

## End(Not run)
```

wrcc_load *Load Processed WRCC Monitoring Data*

Description

Loads pre-generated .RData files containing WRCC PM2.5 data.

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

Usage

```
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.

See Also

wrcc_loadDaily

wrcc_loadLatest

Examples

```
## Not run:
wrcc <- wrcc_load(2017)
wrcc_conus <- monitor_subset(wrcc, stateCodes=CONUS)
monitorLeaflet(wrcc_conus)

## End(Not run)
```

wrcc_loadDaily *Load Recent WRCC Monitoring Data*

Description

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

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 `wrcc_loadLatest()`.

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

Usage

```
wrcc_loadDaily(baseUrl = "https://haze.airfire.org/monitoring/RData/")
```

Arguments

baseUrl	location of the WRCC latest data file

Value

A *ws_monitor* object with WRCC data.

See Also

wrcc_load

wrcc_loadLatest

Examples

```
## Not run:
wrcc <- wrcc_loadDaily()

## End(Not run)
```

wrcc_loadLatest *Load Recent WRCC Monitoring Data*

Description

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

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

Usage

```
wrcc_loadLatest(baseUrl = "https://haze.airfire.org/monitoring/RData/")
```

Arguments

baseUrl	location of the WRCC latest data file

Value

A *ws_monitor* object with WRCC data.

See Also

wrcc_load

wrcc_loadDaily

Examples

```
## Not run:
wrcc <- wrcc_loadLatest()

## End(Not run)
```

---

wrcc_parseData *Parse WRCC Data String*

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:
fileString <- wrcc_downloadData(20150701, 20150930, unitID='SM16')
tbl <- wrcc_parseData(fileString)

## 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