Package: photobiologySunCalc (via r-universe)

October 24, 2024

Type Package

Title Sun and Atmosphere Calculations

Version 0.1.0

Date 2024-08-04

Description Compute the position of the sun, day and night length, local solar time using Meeus' very accurate formulae. Estimate air mass (AM) from solar elevation, reference evapotranspiration, and interconvert air water content expressed as different physical quantities.

License GPL (>= 2)

Depends R (>= 4.0.0)

Imports stats, tibble (>= 3.1.6), lubridate (>= 1.9.0), dplyr (>= 1.0.9)

Suggests knitr (>= 1.41), rmarkdown (>= 2.18), testthat (>= 3.1.4), roxygen2 (>= 7.2.0), lutz (>= 0.3.1), covr

LazyLoad yes

ByteCompile true

URL https://docs.r4photobiology.info/photobiologySunCalc/, https://github.com/aphalo/photobiologySunCalc

BugReports https://github.com/aphalo/photobiologySunCalc/issues

Encoding UTF-8

RoxygenNote 7.3.2

VignetteBuilder knitr

Repository https://aphalo.r-universe.dev

RemoteUrl https://github.com/aphalo/photobiologySunCalc

RemoteRef HEAD

RemoteSha 45aab755cf3800fdf5ea6015ad814180a08478dc

Contents

`photobiologySunCalc-package`
photobiologySunCalc: Sun and Atmosphere Calculations

Description

Compute the position of the sun, day and night length, local solar time using Meeus' very accurate formulae. Estimate air mass (AM) from solar elevation, reference evapotranspiration, and interconvert air water content expressed as different physical quantities.

Details

Please see the vignette *0: The R for Photobiology Suite* for a description of the suite.

Author(s)

Maintainer: Pedro J. Aphalo `<pedro.aphalo@helsinki.fi>` (ORCID)

References

Aphalo, Pedro J. (2015) The r4photobiology suite. *UV4Plants Bulletin*, 2015:1, 21-29. doi:10.19232/uv4pb.2015.1.14.

See Also

Useful links:

- https://docs.r4photobiology.info/xx/
- https://github.com/aphalo/xx
- Report bugs at https://github.com/aphalo/xx/issues

Examples

```
# daylength
sunrise_time(lubridate::today(tzone = "EET"), tz = "EET",
             geocode = data.frame(lat = 60, lon = 25),
             unit.out = "hour")
day_length(lubridate::today(tzone = "EET"), tz = "EET",
           geocode = data.frame(lat = 60, lon = 25),
           unit.out = "hour")
sun_angles(lubridate::now(tzone = "EET"), tz = "EET",
           geocode = data.frame(lat = 60, lon = 25))

water_vp_sat(23) # 23 C -> vapour pressure in Pa
```

---

as.solar_date — *Convert a solar_time object into solar_date object*

Description

Convert a solar_time object into solar_date object

Usage

```
as.solar_date(x, time)
```

Arguments

x	solar_time object.
time	an R date time object

Value

For method `as.solar_date()` a date-time object with the class attr set to "solar.time". This is needed only for unambiguous formatting and printing.

See Also

Other Local solar time functions: `is.solar_time()`, `print.solar_time()`, `solar_time()`

as_tod *Convert datetime to time-of-day*

Description

Convert a datetime into a time of day expressed in hours, minutes or seconds from midnight in local time for a time zone. This conversion is useful when time-series data for different days needs to be compared or plotted based on the local time-of-day.

Usage

```
as_tod(x, unit.out = "hours", tz = NULL)
```

Arguments

x	a datetime object accepted by lubridate functions
unit.out	character string, One of "tod_time", "hours", "minutes", or "seconds".
tz	character string indicating time zone to be used in output.

Value

A numeric vector of the same length as x. If unit.out = "tod_time" an object of class "tod_time" which the same as for unit.out = "hours" but with the class attribute set, which dispatches to special format() nad print() methods.

See Also

solar_time

Other Time of day functions: format.tod_time(), print.tod_time()

Examples

```
library(lubridate)
my_instants <- ymd_hms("2020-05-17 12:05:03") + days(c(0, 30))
my_instants
as_tod(my_instants)
as_tod(my_instants, unit.out = "tod_time")
```

day_night   *Times for sun positions*

Description

Functions for calculating the timing of solar positions, given geographical coordinates and dates. They can be also used to find the time for an arbitrary solar elevation between 90 and -90 degrees by supplying "twilight" angle(s) as argument.

Usage

```
day_night(
  date = lubridate::now(tzone = "UTC"),
  tz = ifelse(lubridate::is.Date(date), "UTC", lubridate::tz(date)),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  twilight = "none",
  unit.out = "hours"
)

day_night_fast(date, tz, geocode, twilight, unit.out)

is_daytime(
  date = lubridate::now(tzone = "UTC"),
  tz = ifelse(lubridate::is.Date(date), "UTC", lubridate::tz(date)),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  twilight = "none",
  unit.out = "hours"
)

noon_time(
  date = lubridate::now(tzone = "UTC"),
  tz = lubridate::tz(date),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  twilight = "none",
  unit.out = "datetime"
)

sunrise_time(
  date = lubridate::now(tzone = "UTC"),
  tz = lubridate::tz(date),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  twilight = "sunlight",
  unit.out = "datetime"
)

sunset_time(
  date = lubridate::now(tzone = "UTC"),
```

```
  tz = lubridate::tz(date),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  twilight = "sunlight",
  unit.out = "datetime"
)

day_length(
  date = lubridate::now(tzone = "UTC"),
  tz = "UTC",
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  twilight = "sunlight",
  unit.out = "hours"
)

night_length(
  date = lubridate::now(tzone = "UTC"),
  tz = "UTC",
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  twilight = "sunlight",
  unit.out = "hours"
)
```

Arguments

`date`	"vector" of `POSIXct` times or`Date` objects, any valid TZ is allowed, default is current date at Greenwich matching the default for `geocode`.
`tz`	character vector indicating time zone to be used in output and to interpret `Date` values passed as argument to `date`.
`geocode`	data frame with one or more rows and variables lon and lat as numeric values (degrees). If present, address will be copied to the output.
`twilight`	character string, one of "none", "rim", "refraction", "sunlight", "civil", "nautical", "astronomical", or a `numeric` vector of length one, or two, giving solar elevation angle(s) in degrees (negative if below the horizon).
`unit.out`	character string, One of "datetime", "day", "hour", "minute", or "second".

Details

Twilight names are interpreted as follows. "none": solar elevation = 0 degrees. "rim": upper rim of solar disk at the horizon or solar elevation = -0.53 / 2. "refraction": solar elevation = 0 degrees + refraction correction. "sunlight": upper rim of solar disk corrected for refraction, which is close to the value used by the online NOAA Solar Calculator. "civil": -6 degrees, "naval": -12 degrees, and "astronomical": -18 degrees. Unit names for output are as follows: "day", "hours", "minutes" and "seconds" times for sunrise and sunset are returned as times-of-day since midnight expressed in the chosen unit. "date" or "datetime" return the same times as datetime objects with TZ set (this is much slower than "hours"). Day length and night length are returned as numeric values expressed in hours when ‘"datetime"’ is passed as argument to `unit.out`. If twilight is a numeric vector of length two, the element with index 1 is used for sunrise and that with index 2 for sunset.

is_daytime() supports twilight specifications by name, a test like sun_elevation() > 0 may be used directly for a numeric angle.

Value

A tibble with variables day, tz, twilight.rise, twilight.set, longitude, latitude, address, sunrise, noon, sunset, daylength, nightlength or the corresponding individual vectors.

is_daytime() returns a logical vector, with TRUE for day time and FALSE for night time.

noon_time, sunrise_time and sunset_time return a vector of POSIXct times

day_length and night_length return numeric a vector giving the length in hours

Warning

Be aware that R's Date class does not save time zone metadata. This can lead to ambiguities in the current implementation based on time instants. The argument passed to date should be of class POSIXct, in other words an instant in time, from which the correct date will be computed based on the tz argument.

The time zone in which times passed to date as argument are expressed does not need to be the local one or match the geocode, however, the returned values will be in the same time zone as the input.

Note

Function day_night() is an implementation of Meeus equations as used in NOAAs on-line web calculator, which are very precise and valid for a very broad range of dates. For sunrise and sunset the times are affected by refraction in the atmosphere, which does in turn depend on weather conditions. The effect of refraction on the apparent position of the sun is only an estimate based on "typical" conditions. The more tangential to the horizon is the path of the sun, the larger the effect of refraction is on the times of visual occlusion of the sun behind the horizon—i.e. the largest timing errors occur at high latitudes. The computation is not defined for latitudes 90 and -90 degrees, i.e. at the poles.

There exists a different R implementation of the same algorithms called "AstroCalcPureR" available as function astrocalc4r in package 'fishmethods'. Although the equations used are almost all the same, the function signatures and which values are returned differ. In particular, the implementation in 'photobiology' splits the calculation into two separate functions, one returning angles at given instants in time, and a separate one returning the timing of events for given dates. In 'fishmethods' (= 1.11-0) there is a bug in function astrocalc4r() that affects sunrise and sunset times. The times returned by the functions in package 'photobiology' have been validated against the NOAA base implementation.

In the current implementation functions sunrise_time, noon_time, sunset_time, day_length, night_length and is_daytime are all wrappers on day_night, so if more than one quantity is needed it is preferable to directly call day_night and extract the different components from the returned list.

night_length returns the length of night-time conditions in one day (00:00:00 to 23:59:59), rather than the length of the night between two consecutive days.

References

The primary source for the algorithm used is the book: Meeus, J. (1998) Astronomical Algorithms, 2 ed., Willmann-Bell, Richmond, VA, USA. ISBN 978-0943396613.

A different implementation is available at https://github.com/NEFSC/READ-PDB-AstroCalc4R/ and in R paclage 'fishmethods'. In 'fishmethods' (= 1.11-0) there is a bug in function astrocalc4r() that affects sunrise and sunset times.

An interactive web page using the same algorithms is available at https://gml.noaa.gov/grad/solcalc/. There are small differences in the returned times compared to our function that seem to be related to the estimation of atmospheric refraction (about 0.1 degrees).

See Also

sun_angles.

Other astronomy related functions: format.solar_time(), sun_angles()

Examples

```
library(lubridate)

my.geocode <- data.frame(lon = 24.93838,
                         lat = 60.16986,
                         address = "Helsinki, Finland")

day_night(ymd("2015-05-30", tz = "EET"),
          geocode = my.geocode)
day_night(ymd("2015-05-30", tz = "EET") + days(1:10),
          geocode = my.geocode,
          twilight = "civil")
sunrise_time(ymd("2015-05-30", tz = "EET"),
             geocode = my.geocode)
noon_time(ymd("2015-05-30", tz = "EET"),
          geocode = my.geocode)
sunset_time(ymd("2015-05-30", tz = "EET"),
            geocode = my.geocode)
day_length(ymd("2015-05-30", tz = "EET"),
           geocode = my.geocode)
day_length(ymd("2015-05-30", tz = "EET"),
           geocode = my.geocode,
           unit.out = "day")
is_daytime(ymd("2015-05-30", tz = "EET") + hours(c(0, 6, 12, 18, 24)),
           geocode = my.geocode)
is_daytime(ymd_hms("2015-05-30 03:00:00", tz = "EET"),
           geocode = my.geocode)
is_daytime(ymd_hms("2015-05-30 00:00:00", tz = "UTC"),
           geocode = my.geocode)
is_daytime(ymd_hms("2015-05-30 03:00:00", tz = "EET"),
           geocode = my.geocode,
           twilight = "civil")
is_daytime(ymd_hms("2015-05-30 00:00:00", tz = "UTC"),
           geocode = my.geocode,
```

```
twilight = "civil")
```

ET_ref *Evapotranspiration*

Description

Compute an estimate of reference (= potential) evapotranspiration from meteorologial data. Evapotranspiration from vegetation includes transpiraction by plants plus evaporation from the soil or other wet surfaces. ET_0 is the reference value assuming no limitation to transpiration due to soil water, similar to potential evapotranspiration (PET). An actual evapotranpiration value ET can be estimated only if additional information on the plants and soil is available.

Usage

```
ET_ref(
  temperature,
  water.vp,
  wind.speed,
  net.irradiance,
  nighttime = FALSE,
  atmospheric.pressure = 10.13,
  soil.heat.flux = 0,
  method = "FAO.PM",
  check.range = TRUE
)

ET_ref_day(
  temperature,
  water.vp,
  wind.speed,
  net.radiation,
  atmospheric.pressure = 10.13,
  soil.heat.flux = 0,
  method = "FAO.PM",
  check.range = TRUE
)
```

Arguments

`temperature`	numeric vector of air temperatures (C) at 2 m height.
`water.vp`	numeric vector of water vapour pressure in air (Pa).
`wind.speed`	numeric Wind speed (m/s) at 2 m height.
`net.irradiance`	numeric Long wave and short wave balance (W/m2).

`nighttime`	logical Used only for methods that distinguish between daytime- and nighttime canopy conductances.
`atmospheric.pressure`	numeric Atmospheric pressure (Pa).
`soil.heat.flux`	numeric Soil heat flux (W/m2), positive if soil temperature is increasing.
`method`	character The name of an estimation method.
`check.range`	logical Flag indicating whether to check or not that arguments for temperature are within range of method. Passed to function calls to `water_vp_sat()` and `water_vp_sat_slope()`.
`net.radiation`	numeric Long wave and short wave balance (J/m2/day).

Details

Currently three methods, based on the Penmann-Monteith equation formulated as recommended by FAO56 (Allen et al., 1998) as well as modified in 2005 for tall and short vegetation according to ASCE-EWRI are implemented in function `ET_ref()`. The computations rely on data measured according WHO standards at 2 m above ground level to estimate reference evapotranspiration (ET_0). The formulations are those for ET expressed in mm/h, but modified to use as input flux rates in W/m2 and pressures expressed in Pa.

Value

A numeric vector of reference evapotranspiration estimates expressed in mm/h for `ET_ref()` and `ET_PM()` and in mm/d for `ET_ref_day()`.

References

Allen R G, Pereira L S, Raes D, Smith M. 1998. Crop evapotranspiration: Guidelines for computing crop water requirements. Rome: FAO. Allen R G, Pruitt W O, Wright J L, Howell T A, Ventura F, Snyder R, Itenfisu D, Steduto P, Berengena J, Yrisarry J, et al. 2006. A recommendation on standardized surface resistance for hourly calculation of reference ETo by the FAO56 Penman-Monteith method. Agricultural Water Management 81.

See Also

Other Evapotranspiration and energy balance related functions.: `net_irradiance()`

Examples

```
# instantaneous
ET_ref(temperature = 20,
       water.vp = water_RH2vp(relative.humidity = 70,
                             temperature = 20),
       wind.speed = 0,
       net.irradiance = 10)

ET_ref(temperature = c(5, 20, 35),
       water.vp = water_RH2vp(70, c(5, 20, 35)),
       wind.speed = 0,
```

```
        net.irradiance = 10)

# Hot and dry air
ET_ref(temperature = 35,
       water.vp = water_RH2vp(10, 35),
       wind.speed = 5,
       net.irradiance = 400)

ET_ref(temperature = 35,
       water.vp = water_RH2vp(10, 35),
       wind.speed = 5,
       net.irradiance = 400,
       method = "FAO.PM")

ET_ref(temperature = 35,
       water.vp = water_RH2vp(10, 35),
       wind.speed = 5,
       net.irradiance = 400,
       method = "ASCE.PM.short")

ET_ref(temperature = 35,
       water.vp = water_RH2vp(10, 35),
       wind.speed = 5,
       net.irradiance = 400,
       method = "ASCE.PM.tall")

# Low temperature and high humidity
ET_ref(temperature = 5,
       water.vp = water_RH2vp(95, 5),
       wind.speed = 0.5,
       net.irradiance = -10,
       nighttime = TRUE,
       method = "ASCE.PM.short")

ET_ref_day(temperature = 35,
           water.vp = water_RH2vp(10, 35),
           wind.speed = 5,
           net.radiation = 35e6) # 35 MJ / d / m2
```

---

format.solar_time *Encode in a Common Format*

Description

Format a solar_time object for pretty printing

Usage

```
## S3 method for class 'solar_time'
format(x, ..., sep = ":")
```

Arguments

x	an R object
...	ignored
sep	character used as separator

See Also

Other astronomy related functions: day_night(), sun_angles()

---

format.tod_time *Encode in a Common Format*

Description

Format a tod_time object for pretty printing

Usage

```
## S3 method for class 'tod_time'
format(x, ..., sep = ":")
```

Arguments

x	an R object
...	ignored
sep	character used as separator

See Also

Other Time of day functions: as_tod(), print.tod_time()

---

irrad_extraterrestrial *Extraterrestrial irradiance*

Description

Estimate of down-welling solar (short wave) irradiance at the top of the atmosphere above a location on Earth, computed based on angles, Sun-Earth distance and the solar constant. Astronomical computations are done with function sun_angles().

Usage

```
irrad_extraterrestrial(
  time = lubridate::now(tzone = "UTC"),
  tz = lubridate::tz(time),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  solar.constant = "NASA"
)
```

Arguments

`time`	A "vector" of POSIXct Time, with any valid time zone (TZ) is allowed, default is current time.
`tz`	character string indicating time zone to be used in output.
`geocode`	data frame with variables lon and lat as numeric values (degrees), nrow > 1, allowed.
`solar.constant`	numeric or character If character, "WMO" or "NASA", if numeric, an irradiance value in the same units as the value to be returned.

Value

Numeric vector of extraterrestrial irradiance (in W / m2 if solar constant is a character value).

See Also

Function `sun_angles`.

Examples

```
library(lubridate)

irrad_extraterrestrial(ymd_hm("2021-06-21 12:00", tz = "UTC"))

irrad_extraterrestrial(ymd_hm("2021-12-21 20:00", tz = "UTC"))

irrad_extraterrestrial(ymd_hm("2021-06-21 00:00", tz = "UTC") + hours(1:23))
```

`is.solar_time` *Query class*

Description

Query class

Usage

```
is.solar_time(x)

is.solar_date(x)
```

Arguments

`x`	an R object.

See Also

Other Local solar time functions: `as.solar_date()`, `print.solar_time()`, `solar_time()`

`net_irradiance` *Net radiation flux*

Description

Estimate net radiation balance expressed as a flux in W/m2. If `lw.down.irradiance` is passed a value in W / m2 the difference is computed directly and if not an approximate value is estimated, using `R_rel = 0.75` which corresponds to clear sky, i.e., uncorrected for cloudiness. This is the approach to estimation is that recommended by FAO for hourly estimates while here we use it for instantaneous or mean flux rates.

Usage

```
net_irradiance(
  temperature,
  sw.down.irradiance,
  lw.down.irradiance = NULL,
  sw.albedo = 0.23,
  lw.emissivity = 0.98,
  water.vp = 0,
  R_rel = 1
)
```

Arguments

`temperature`	numeric vector of air temperatures (C) at 2 m height.
`sw.down.irradiance`, `lw.down.irradiance`	numeric Down-welling short wave and long wave radiation radiation (W/m2).
`sw.albedo`	numeric Albedo as a fraction of one (/1).
`lw.emissivity`	numeric Emissivity of the surface (ground or vegetation) for long wave radiation.
`water.vp`	numeric vector of water vapour pressure in air (Pa), ignored if `lw.down.irradiance` is available.
`R_rel`	numeric The ratio of actual and clear sky short wave irradiance (/1).

Value

A numeric vector of evapotranspiration estimates expressed as W / m-2.

See Also

Other Evapotranspiration and energy balance related functions.: `ET_ref()`

`print.solar_time` *Print solar time and solar date objects*

Description

Print solar time and solar date objects

Usage

```
## S3 method for class 'solar_time'
print(x, ...)

## S3 method for class 'solar_date'
print(x, ...)
```

Arguments

`x`	an R object
`...`	passed to `format` method

Note

Default is to print the underlying POSIXct as a solar time.

See Also

Other Local solar time functions: `as.solar_date()`, `is.solar_time()`, `solar_time()`

print.tod_time *Print time-of-day objects*

Description

Print time-of-day objects

Usage

```
## S3 method for class 'tod_time'
print(x, ...)
```

Arguments

`x`	an R object
`...`	passed to `format` method

Note

Default is to print the underlying `numeric` vector as a solar time.

See Also

Other Time of day functions: `as_tod()`, `format.tod_time()`

relative_AM *Relative Air Mass (AM)*

Description

Approximate relative air mass (AM) from sun elevation or sun zenith angle.

Usage

```
relative_AM(elevation.angle = NULL, zenith.angle = NULL, occluded.value = NA)
```

Arguments

`elevation.angle, zenith.angle`	numeric vector Angle in degrees for the sun position. An argument should be passed to one and only one of `elevation_angle` or `zenith_angle`.
`occluded.value`	numeric Value to return when elevation angle is negative (sun below the horizon).

Details

This is an implementation of equation (3) in Kasten and Young (1989). This equation is only an approximation to the tabulated values in the same paper. Returned values are rounded to three significant digits.

Note

Although relative air mass is not defined when the sun is not visible, returning a value different from the default NA might be useful in some cases.

References

F. Kasten, A. T. Young (1989) Revised optical air mass tables and approximation formula. Applied Optics, 28, 4735-. doi:10.1364/ao.28.004735.

Examples

```
relative_AM(c(90, 60, 30, 1, -10))
relative_AM(c(90, 60, 30, 1, -10), occluded.value = Inf)
relative_AM(zenith.angle = 0)
```

---

solar_time *Local solar time*

Description

solar_time() computes the time of day expressed in seconds since the astronomical midnight using and instant in time and a geocode as input. Solar time is useful when we want to plot data according to the local solar time rather than the local time in use at a time zone. How the returned instant in time is expressed depends on the argument passed to unit.out.

Usage

```
solar_time(
  time = lubridate::now(),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  unit.out = "time"
)
```

Arguments

time	POSIXct Time, any valid time zone (TZ) is allowed, default is current time
geocode	data frame with variables lon and lat as numeric values (degrees).
unit.out	character string, One of "datetime", "time", "hour", "minute", or "second".

Details

Solar time is determined by the position of the sun in the sky and it almost always differs from the time expressed in the local time coordinates in use. The differences can vary from a few minutes up to a couple of hours depending on the exact location within the time zone and the use or not of daylight saving time.

Value

In all cases solar time is expressed as time since local astronomical midnight and, thus, lacks date information. If `unit.out = "time"`, a numeric value in seconds with an additional class attribute "solar_time"; if `unit.out = "datetime"`, a "POSIXct" value in seconds from midnight but with an additional class attribute "solar_date"; if `unit.out = "hour"` or `unit.out = "minute"` or `unit.out = "second"`, a numeric value.

Warning!

Returned values are computed based on the time zone of the argument for parameter time. In the case of solar time, this timezone does not affect the result. However, in the case of solar dates the date part may be off by one day, if the time zone does not match the coordinates of the geocode value provided as argument.

Note

The algorithm is approximate, it calculates the difference between local solar noon and noon in the time zone of `time` and uses this value for the whole day when converting times into solar time. Days are not exactly 24 h long. Between successive days the shift is only a few seconds, and this leads to a small jump at midnight.

See Also

`as_tod`

Other Local solar time functions: `as.solar_date()`, `is.solar_time()`, `print.solar_time()`

Examples

```
BA.geocode <-
  data.frame(lon = -58.38156, lat = -34.60368, address = "Buenos Aires, Argentina")
sol_t <- solar_time(lubridate::dmy_hms("21/06/2016 10:00:00", tz = "UTC"),
                    BA.geocode)
sol_t
class(sol_t)

sol_d <- solar_time(lubridate::dmy_hms("21/06/2016 10:00:00", tz = "UTC"),
                    BA.geocode,
                    unit.out = "datetime")
sol_d
class(sol_d)
```

sun_angles *Solar angles*

Description

Function sun_angles() returns the solar angles and Sun to Earth relative distance for given times and locations using a very precise algorithm. Convenience functions sun_azimuth(), sun_elevation(), sun_zenith_angle() and distance_to_sun() are wrappers on sun_angles() that return individual vectors.

Usage

```
sun_angles(
  time = lubridate::now(tzone = "UTC"),
  tz = lubridate::tz(time),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  use.refraction = FALSE
)

sun_angles_fast(time, tz, geocode, use.refraction)

sun_elevation(
  time = lubridate::now(),
  tz = lubridate::tz(time),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  use.refraction = FALSE
)

sun_zenith_angle(
  time = lubridate::now(),
  tz = lubridate::tz(time),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  use.refraction = FALSE
)

sun_azimuth(
  time = lubridate::now(),
  tz = lubridate::tz(time),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  use.refraction = FALSE
)

distance_to_sun(
  time = lubridate::now(),
  tz = lubridate::tz(time),
  geocode = tibble::tibble(lon = 0, lat = 51.5, address = "Greenwich"),
  use.refraction = FALSE
```

```
)
```

Arguments

`time`	A "vector" of POSIXct Time, with any valid time zone (TZ) is allowed, default is current time.
`tz`	character string indicating time zone to be used in output.
`geocode`	data frame with variables lon and lat as numeric values (degrees), nrow > 1, allowed.
`use.refraction`	logical Flag indicating whether to correct for fraction in the atmosphere.

Details

This function is an implementation of Meeus equations as used in NOAAs on-line web calculator, which are precise and valid for a very broad range of dates (years -1000 to 3000 at least). The apparent solar elevations near sunrise and sunset are affected by refraction in the atmosphere, which does in turn depend on weather conditions. The effect of refraction on the apparent position of the sun is only an estimate based on "typical" conditions for the atmosphere. The computation is not defined for latitudes 90 and -90 degrees, i.e. exactly at the poles. The function is vectorized and in particular passing a vector of times for a single geocode enhances performance very much as the equation of time, the most time consuming step, is computed only once.

For improved performance, if more than one angle is needed it is preferable to directly call `sun_angles` instead of the wrapper functions as this avoids the unnecesary recalculation.

Value

A `data.frame` with variables `time` (in same TZ as input), `TZ`, `solartime`, `longitude`, `latitude`, `address`, `azimuth`, `elevation`, `declination`, `eq.of.time`, `hour.angle`, and `distance`. If a data frame with multiple rows is passed to `geocode` and a vector of times longer than one is passed to `time`, sun position for all combinations of locations and times are returned by `sun_angles`. Angles are expressed in degrees, `solartime` is a vector of class `"solar.time"`, `distance` is expressed in relative sun units.

Important!

Given an instant in time and a time zone, the date is computed from these, and may differ by one day to that at the location pointed by `geocode` at the same instant in time, unless the argument passed to `tz` matches the time zone at this location.

Note

There exists a different R implementation of the same algorithms called "AstroCalcPureR" available as function `astrocalc4r` in package 'fishmethods'. Although the equations used are almost all the same, the function signatures and which values are returned differ. In particular, the present implementation splits the calculation into two separate functions, one returning angles at given instants in time, and a separate one returning the timing of events for given dates.

References

The primary source for the algorithm used is the book: Meeus, J. (1998) Astronomical Algorithms, 2 ed., Willmann-Bell, Richmond, VA, USA. ISBN 978-0943396613.

A different implementation is available at https://github.com/NEFSC/READ-PDB-AstroCalc4R/.

An interactive web page using the same algorithms is available at https://gml.noaa.gov/grad/solcalc/. There are small differences in the returned times compared to our function that seem to be related to the estimation of atmospheric refraction (about 0.1 degrees).

See Also

Other astronomy related functions: `day_night()`, `format.solar_time()`

Examples

```
library(lubridate)
sun_angles()
sun_azimuth()
sun_elevation()
sun_zenith_angle()
sun_angles(ymd_hms("2014-09-23 12:00:00"))
sun_angles(ymd_hms("2014-09-23 12:00:00"),
           geocode = data.frame(lat=60, lon=0))
sun_angles(ymd_hms("2014-09-23 12:00:00") + minutes((0:6) * 10))
```

---

tz_time_diff   *Time difference between two time zones*

Description

Returns the difference in local time expressed in hours between two time zones at a given instant in time. The difference due to daylight saving time or Summer and Winter time as well as historical changes in time zones are taken into account.

Usage

```
tz_time_diff(
  when = lubridate::now(),
  tz.target = lubridate::tz(when),
  tz.reference = "UTC"
)
```

Arguments

`when`	datetime A time instant
`tz.target, tz.reference`	character Two time zones using names recognized by functions from package 'lubridate'

Value

A `numeric` value.

Note

This function is implemented using functions from package 'lubridate'. For details on the handling of time zones, please, consult the documentation for `Sys.timezone` about system differences in time zone names and handling.

`validate_geocode` *Validate a geocode*

Description

Test validity of a geocode or ensure that a geocode is valid.

Usage

```
validate_geocode(geocode)

is_valid_geocode(geocode)

length_geocode(geocode)

na_geocode()
```

Arguments

`geocode`	data.frame with geocode data in columns `"lat"`, `"lon"`, and possibly also `"address"`.

Details

`validate_geocode` Converts to tibble, checks data bounds, converts address to character if it is not already a character vector, or add character NAs if the address column is missing.

`is_valid_geocode` Checks if a geocode is valid, returning 0L if not, and the number of row otherwise.

Value

A valid geocode stored in a tibble.

FALSE for invalid, TRUE for valid.

FALSE for invalid, number of rows for valid.

A geo_code tibble with all fields set to suitable NAs.

Examples

```
validate_geocode(NA)
validate_geocode(data.frame(lon = -25, lat = 66))

is_valid_geocode(NA)
is_valid_geocode(1L)
is_valid_geocode(data.frame(lon = -25, lat = 66))

na_geocode()
```

`water_vp_sat` *Water vapour pressure*

Description

Approximate water pressure in air as a function of temperature, and its inverse the calculation of dewpoint.

Usage

```
water_vp_sat(
  temperature,
  over.ice = FALSE,
  method = "tetens",
  check.range = TRUE
)

water_dp(water.vp, over.ice = FALSE, method = "tetens", check.range = TRUE)

water_fp(water.vp, over.ice = TRUE, method = "tetens", check.range = TRUE)

water_vp2mvc(water.vp, temperature)

water_mvc2vp(water.mvc, temperature)

water_vp2RH(
  water.vp,
  temperature,
  over.ice = FALSE,
  method = "tetens",
  pc = TRUE,
  check.range = TRUE
)

water_RH2vp(
  relative.humidity,
```

```
  temperature,
  over.ice = FALSE,
  method = "tetens",
  pc = TRUE,
  check.range = TRUE
)

water_vp_sat_slope(
  temperature,
  over.ice = FALSE,
  method = "tetens",
  check.range = TRUE,
  temperature.step = 0.1
)

psychrometric_constant(atmospheric.pressure = 101325)
```

Arguments

temperature	numeric vector of air temperatures (C).
over.ice	logical vector Is the estimate for equilibrium with liquid water or with ice.
method	character Currently "tetens", modified "magnus", "wexler" and "goff.gratch" equations are supported.
check.range	logical Flag indicating whether to check or not that arguments for temperature are within the range of validity of the `method` used.
water.vp	numeric vector of water vapour pressure in air (Pa).
water.mvc	numeric vector of water vapour concnetration as mass per volume (gm^{-3}).
pc	logical flag for result returned as percent or not.
relative.humidity	numeric Relative humidity as fraction of 1.
temperature.step	numeric Delta or step used to estimate the slope as a finite difference (C).
atmospheric.pressure	numeric Atmospheric pressure (Pa).

Details

Function `water_vp_sat()` provides implementations of several well known equations for the estimation of saturation vapor pressure in air. Functions `water_dp()` and `water_fp()` use the inverse of these equations to compute the dew point or frost point from water vapour pressure in air. The inverse functions are either analytical solutions or fitted approximations. None of these functions are solved numerically by iteration.

Method `"tetens"` implements Tetens' (1930) equation for the cases of equilibrium with a water and an ice surface. Method `"magnus"` implements the modified Magnus equations of Alduchov and Eskridge (1996, eqs. 21 and 23). Method `"wexler"` implements the equations proposed by Wexler (1976, 1977), and their inverse according to Hardy (1998). Method `"goff.gratch"` implements the equations of Groff and Gratch (1946) with the minor updates of Groff (1956).

The equations are approximations, and in spite of their different names, Tetens' and Magnus' equations have the same form with the only difference in the values of the parameters. However, the modified Magnus equation is more accurate as Tetens equation suffers from some bias errors at extreme low temperatures (< -40 C). In contrast Magnus equations with recently fitted values for the parameters are usable for temperatures from -80 C to +50 C over water and -80 C to 0 C over ice. The Groff Gratch equation is more complex and is frequently used as a reference in comparison as it is considered reliable over a broad range of temperatures. Wexler's equations are computationally simpler and fitted to relatively recent data. There is little difference at temperatures in the range -20 C to +50 C, and differences become large at extreme temperatures. Temperatures outside the range where estimations are highly reliable for each equation return `NA`, unless extrapolation is enabled by passing `FALSE` as argument to parameter `check.range`.

The switch between equations for ice or water cannot be based on air temperature, as it depends on the presence or not of a surface of liquid water. It must be set by passing an argument to parameter `over.ice` which defaults to `FALSE`.

Tetens equation is still very frequently used, and is for example the one recommended by FAO for computing potential evapotranspiration. For this reason it is used as default here.

Value

A numeric vector of partial pressures in pascal (Pa) for `water_vp_sat()` and `water_mvc2vp()`, a numeric vector of dew point temperatures (C) for `water_dp()` and numeric vector of mass per volume concentrations (gm^{-3}) for `water_vp2mvc()`. `water_vp_sat()` and `psychrometric_constant()` both return numeric vectors of pressure per degree of temperature (PaC^{-1})

Note

The inverse of the Groff Gratch equation has yet to be implemented.

References

Tetens, O., 1930. Uber einige meteorologische Begriffe. Zeitschrift fur Geophysik, Vol. 6:297.

Goff, J. A., and S. Gratch (1946) Low-pressure properties of water from -160 to 212 F, in Transactions of the American Society of Heating and Ventilating Engineers, pp 95-122, presented at the 52nd annual meeting of the American Society of Heating and Ventilating Engineers, New York, 1946.

Wexler, A. (1976) Vapor Pressure Formulation for Water in Range 0 to 100°C. A Revision, Journal of Research ofthe National Bureau of Standards: A. Physics and Chemistry, September-December 1976, Vol. 80A, Nos.5 and 6, 775-785

Wexler, A., (1977) Vapor Pressure Formulation for Ice, Journal of Research of the National Bureau of Standards - A. Physics and Chemistry, Vol. 81A, No. 1, 5-19

Alduchov, O. A., Eskridge, R. E., 1996. Improved Magnus Form Approximation of Saturation Vapor Pressure. Journal of Applied Meteorology, 35: 601-609 .

Hardy, Bob (1998) ITS-90 formulations for vapor pressure, frostpoint temperature, dewpoint temperature, andenhancement factors in the range -100 TO +100 C. The Proceedings of the Third International Symposium on Humidity & Moisture, Teddington, London, England, April 1998. https://www.decatur.de/javascript/dew/resources/its90formulas.pdf

Monteith, J., Unsworth, M. (2008) Principles of Environmental Physics. Academic Press, Amsterdam.

Allen R G, Pereira L S, Raes D, Smith M. (1998) Crop evapotranspiration: Guidelines for computing crop water requirements. FAO Irrigation and drainage paper 56. Rome: FAO.

[Equations describing the physical properties of moist air](http://www.conservationphysics.org/atmcalc/atmoclc2.pdf)

Examples

```
water_vp_sat(20) # C -> Pa
water_vp_sat(temperature = c(0, 10, 20, 30, 40)) # C -> Pa
water_vp_sat(temperature = -10) # over water!!
water_vp_sat(temperature = -10, over.ice = TRUE)
water_vp_sat(temperature = 20) / 100 # C -> mbar

water_vp_sat(temperature = 20, method = "magnus") # C -> Pa
water_vp_sat(temperature = 20, method = "tetens") # C -> Pa
water_vp_sat(temperature = 20, method = "wexler") # C -> Pa
water_vp_sat(temperature = 20, method = "goff.gratch") # C -> Pa

water_vp_sat(temperature = -20, over.ice = TRUE, method = "magnus") # C -> Pa
water_vp_sat(temperature = -20, over.ice = TRUE, method = "tetens") # C -> Pa
water_vp_sat(temperature = -20, over.ice = TRUE, method = "wexler") # C -> Pa
water_vp_sat(temperature = -20, over.ice = TRUE, method = "goff.gratch") # C -> Pa

water_dp(water.vp = 1000) # Pa -> C
water_dp(water.vp = 1000, method = "magnus") # Pa -> C
water_dp(water.vp = 1000, method = "wexler") # Pa -> C
water_dp(water.vp = 500, over.ice = TRUE) # Pa -> C
water_dp(water.vp = 500, method = "wexler", over.ice = TRUE) # Pa -> C

water_fp(water.vp = 300) # Pa -> C
water_dp(water.vp = 300, over.ice = TRUE) # Pa -> C

water_vp2RH(water.vp = 1500, temperature = 20) # Pa, C -> RH %
water_vp2RH(water.vp = 1500, temperature = c(20, 30)) # Pa, C -> RH %
water_vp2RH(water.vp = c(600, 1500), temperature = 20) # Pa, C -> RH %

water_vp2mvc(water.vp = 1000, temperature = 20) # Pa -> g m-3

water_mvc2vp(water.mvc = 30, temperature = 40) # g m-3 -> Pa

water_dp(water.vp = water_mvc2vp(water.mvc = 10, temperature = 30)) # g m-3 -> C


water_vp_sat_slope(temperature = 20) # C -> Pa / C


psychrometric_constant(atmospheric.pressure = 81.8e3) # Pa -> Pa / C
```

Index