Package ‘shadow’

June 13, 2020

Type Package

Title Geometric Shadow Calculations

Version 0.6.7

Description Functions for calculating: (1) shadow height, (2) logical shadow flag, (3) shadow footprint, (4) Sky View Factor and (5) radiation load. Basic required inputs include a polygonal layer of obstacle outlines along with their heights (i.e. ``extruded polygons''), sun azimuth and sun elevation. The package also provides functions for related preliminary calculations: breaking polygons into line segments, determining azimuth of line segments, shifting segments by azimuth and distance, constructing the footprint of a line-of-sight between an observer and the sun, and creating a 3D grid covering the surface area of extruded polygons.

License MIT + file LICENSE

LazyData TRUE

Imports rgeos (>= 0.3), raster (>= 2.4-15), methods, parallel (>= 3.4.0)

Depends R (>= 3.2.3), sp (>= 1.1.1)

RoxygenNote 7.1.0

Suggests R.rsp, maptools (>= 0.8), testthat, reshape2 (>= 1.4.2), threejs, rgdal, stars

VignetteBuilder R.rsp

URL https://github.com/michaeldorman/shadow/

BugReports https://github.com/michaeldorman/shadow/issues/

Encoding UTF-8

NeedsCompilation no

Author Michael Dorman [aut, cre],
Evyatar Erell [ctb],
Itai Kloog [ctb],
Adi Vulkan [ctb]

Maintainer Michael Dorman <dorman@post.bgu.ac.il>

Repository CRAN

Date/Publication 2020-06-13 08:20:02 UTC

R topics documented:

`beersheva_build` *Polygonal layer of 376 buildings in Beer-Sheva*

Description

A `SpatialPolygonsDataFrame` object representing the outlines of 367 buildings in the Ramot neighborhood, Beer-Sheva. The attribute `height_m` contains building height, in meters.

Usage

```
beersheva_build
```

Format

A `SpatialPolygonsDataFrame` with 10 features and 4 attributes:

build_id Building ID

floors Number of floors for building

apartments Number of apartments

height_m Building height, in meters

elev Elevation above sea level of building base, in meters

`beersheva_elev` *DEM of Ramot neighborhood, Beer-Sheva*

Description

Digital Elevation Model (DEM) of Ramot neighborhood, Beer-Sheva. Raster values represent elevation above sea level, in meters.

Usage

`beersheva_elev`

Format

A `RasterLayer` representing a grid of 1974 raster cells, each cell is a 30*30 meters rectangle. Data source is the Shuttle Radar Topography Mission (SRTM) 1 Arc-Second Global dataset.

References

https://lta.cr.usgs.gov/SRTM1Arc

`boston_block` *Polygonal layer of a building block in Boston*

Description

A `SpatialPolygons` object representing the boundaries of a building block in Central Boston.

Usage

`boston_block`

Format

A `SpatialPolygons` with a single feature.

boston_build *Polygonal layer of three buildings in Boston*

Description

A `SpatialPolygonsDataFrame` object representing the outlines of three buildings located in Central Boston. The attribute `height_m` contains building height, in meters.

Usage

```
boston_build
```

Format

A `SpatialPolygonsDataFrame` with 10 features and 4 attributes:

objectid Building part ID

build_id Building ID

part_floor Number of floors for part

height_m Building height, in meters

boston_park *Polygonal layer of a park in Boston*

Description

A `SpatialPolygons` object representing the boundaries of a park in Central Boston.

Usage

```
boston_park
```

Format

A `SpatialPolygons` with a single feature.

boston_sidewalk — *Polygonal layer of sidewalks in Boston*

Description

A `SpatialLinesDataFrame` object representing sidewalks in Central Boston.

Usage

```
boston_sidewalk
```

Format

A `SpatialLinesDataFrame` with 78 features.

build — *Polygonal layer of four buildings in Rishon*

Description

A `SpatialPolygonsDataFrame` object representing the outlines of four buildings located in Rishon-Le-Zion. The attribute `BLDG_HT` contains building height, in meters.

Usage

```
build
```

Format

A `SpatialPolygonsDataFrame` with 4 features and 2 attributes:

build_id Building ID

BLDG_HT Building height, in meters

classifyAz *Classify azimuth of line segments*

Description

Classify azimuth of line segments

Usage

```
classifyAz(sl)
```

Arguments

sl	A SpatialLines* object

Value

A numeric vector with the segment azimuth values (in decimal degrees)

Examples

```
build_seg = toSeg(build[1, ])
az = classifyAz(build_seg)
plot(build_seg, col = rainbow(4)[cut(az, c(0, 90, 180, 270, 360))])
raster::text(
  rgeos::gCentroid(build_seg, byid = TRUE),
  round(az)
)
```

coefDirect *Coefficient of Direct Normal Irradiance reduction*

Description

This function calculates the coefficient of reduction in Direct Normal Irradiance load due to angle of incidence. For example, a coefficient of 1 is obtained when the sun is perpendicular to the surface.

Usage

```
coefDirect(type, facade_az, solar_pos)
```

Arguments

type	character, specifying surface type. All values must be either "roof" or "facade"
facade_az	Facade azimuth, in decimal degrees from North. Only relevant for type="facade"
solar_pos	A matrix with two columns representing sun position(s); first column is the solar azimuth (in decimal degrees from North), second column is sun elevation (in decimal degrees); rows represent different positions (e.g. at different times of day)

Value

Numeric vector of coefficients, to be multiplied by the direct beam radiation values. The vector length is the same as the length of the longest input (see **Note** below)

Note

All four arguments are recycled to match each other's length. For example, you may specify a single type value of "roof" or "facade" and a single facade_az value, but multiple sun_az and sun_elev values, for calculating the coefficients for a single location given different positions of the sun, etc.

Examples

```
# Basic usage
coefDirect(type = "facade", facade_az = 180, solar_pos = matrix(c(210, 30), ncol = 2))

# Demonstration - Direct beam radiation coefficient on 'facades'
sun_az = seq(270, 90, by = -5)
sun_elev = seq(0, 90, by = 5)
solar_pos = expand.grid(sun_az = sun_az, sun_elev = sun_elev)
solar_pos$coef = coefDirect(type = "facade", facade_az = 180, solar_pos = as.matrix(solar_pos))[1, ]
coef = reshape2::acast(solar_pos, sun_az ~ sun_elev, value.var = "coef")
image(
  180 - sun_az, sun_elev, coef,
  col = rev(heat.colors(10)),
  breaks = seq(0, 1, 0.1),
  asp = 1,
  xlab = "Facade azimuth - Sun azimuth (deg)",
  ylab = "Sun elevation (deg)",
  main = "Facade - Coefficient of Direct Normal Irradiance"
)
contour(180 - sun_az, sun_elev, coef, add = TRUE)

# Demonstration - Direct beam radiation coefficient on 'roofs'
solar_pos$coef = coefDirect(type = "roof", facade_az = 180, solar_pos = as.matrix(solar_pos))[1, ]
coef = reshape2::acast(solar_pos, sun_az ~ sun_elev, value.var = "coef")
image(
  180 - sun_az, sun_elev, coef,
  col = rev(heat.colors(10)),
  breaks = seq(0, 1, 0.1),
  asp = 1,
```

```
  xlab = "Facade azimuth - Sun azimuth (deg)",
  ylab = "Sun elevation (deg)",
  main = "Roof - Coefficient of Direct Normal Irradiance"
)
contour(180 - sun_az, sun_elev, coef, add = TRUE)
```

deg2rad *Degrees to radians*

Description

Degrees to radians

Usage

```
deg2rad(deg)
```

Arguments

deg	Angle in degrees

Value

`numeric` Angle in radians

Examples

```
deg2rad(360) == 2*pi
```

inShadow *Logical shadow calculation (is given point shaded?) for 3D points considering sun position and obstacles*

Description

This function determines whether each given point in a set of 3D points (`location`), is shaded or not, taking into account:

- Obstacles outline (`obstacles`), given by a polygonal layer with a height attribute (`obstacles_height_field`), or alternatively a `Raster*` which is considered as a grid of ground locations
- Sun position (`solar_pos`), given by azimuth and elevation angles

Alternatively, the function determines whether each point is in shadow based on a raster representing shadow height `shadowHeightRaster`, in which case `obstacles`, `obstacles_height_field` and `solar_pos` are left unspecified.

Usage

```
## S4 method for signature 'SpatialPoints,Raster,missing,missing'
inShadow(
  location,
  shadowHeightRaster,
  obstacles,
  obstacles_height_field,
  solar_pos
)

## S4 method for signature 'SpatialPoints,missing,ANY,ANY'
inShadow(
  location,
  shadowHeightRaster,
  obstacles,
  obstacles_height_field,
  solar_pos = solarpos2(location, time),
  time = NULL,
  ...
)

## S4 method for signature 'Raster,missing,ANY,ANY'
inShadow(
  location,
  shadowHeightRaster,
  obstacles,
  obstacles_height_field,
  solar_pos = solarpos2(pnt, time),
  time = NULL,
  ...
)
```

Arguments

`location` A `SpatialPoints*` or `Raster*` object, specifying the location(s) for which to calculate logical shadow values. If `location` is `SpatialPoints*`, then it can have 2 or 3 dimensions. A 2D `SpatialPoints*` is considered as a point(s) on the ground, i.e. 3D point(s) where $z = 0$. In a 3D `SpatialPoints*` the 3rd dimension is assumed to be elevation above ground z (in CRS units). `Raster*` cells are considered as ground locations

`shadowHeightRaster` Raster representing shadow height

`obstacles` A `SpatialPolygonsDataFrame` object specifying the obstacles outline

`obstacles_height_field` Name of attribute in `obstacles` with extrusion height for each feature

`solar_pos` A `matrix` with two columns representing sun position(s); first column is the solar azimuth (in degrees from North), second column is sun elevation (in degrees); rows represent different positions (e.g. at different times of day)

`time`	When both `shadowHeightRaster` and `solar_pos` are unspecified, `time` can be passed to automatically calculate `solarpos` based on the time and the centroid of `location`, using function `maptools::solarpos`. In such case `location` must have a defined CRS (not `NA`). The `time` value must be a `POSIXct` or `POSIXlt` object.
`...`	Other parameters passed to `shadowHeight`, such as `parallel`

Value

Returned object is either a logical `matrix` or a `Raster*` with logical values -

- If input `location` is a `SpatialPoints*`, then returned object is a `matrix` where rows represent spatial locations (`location` features), columns represent solar positions (`solar_pos` rows) and values represent shadow state
- If input `location` is a `Raster*`, then returned object is a `RasterLayer` or `RasterStack`, where raster layers represent solar positions (`solar_pos` rows) and pixel values represent shadow state

In both cases the logical values express shadow state:

- `TRUE` means the location is in shadow
- `FALSE` means the location is not in shadow
- `NA` means the location 3D-intersects an obstacle

Note

For a correct geometric calculation, make sure that:

- The layers `location` and `obstacles` are projected and in same CRS
- The values in `obstacles_height_field` of `obstacles` are given in the same distance units as the CRS (e.g. meters when using UTM)

Examples

```
# Method for 3D points - Manually defined

opar = par(mfrow = c(1, 3))

# Ground level
location = sp::spsample(
  rgeos::gBuffer(rgeos::gEnvelope(build), width = 20),
  n = 80,
  type = "regular"
)
solar_pos = as.matrix(tmy[9, c("sun_az", "sun_elev")])
s = inShadow(
  location = location,
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  solar_pos = solar_pos
```

```
)
plot(location, col = ifelse(s[, 1], "grey", "yellow"), main = "h=0")
plot(build, add = TRUE)

# 15 meters above ground level
coords = coordinates(location)
coords = cbind(coords, z = 15)
location1 = SpatialPoints(coords, proj4string = CRS(proj4string(location)))
solar_pos = as.matrix(tmy[9, c("sun_az", "sun_elev")])
s = inShadow(
  location = location1,
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  solar_pos = solar_pos
)
plot(location, col = ifelse(s[, 1], "grey", "yellow"), main = "h=15")
plot(build, add = TRUE)

# 30 meters above ground level
coords = coordinates(location)
coords = cbind(coords, z = 30)
location2 = SpatialPoints(coords, proj4string = CRS(proj4string(location)))
solar_pos = as.matrix(tmy[9, c("sun_az", "sun_elev")])
s = inShadow(
  location = location2,
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  solar_pos = solar_pos
)
plot(location, col = ifelse(s[, 1], "grey", "yellow"), main = "h=30")
plot(build, add = TRUE)

par(opar)

# Shadow on a grid covering obstacles surface
## Not run:

# Method for 3D points - Covering building surface

obstacles = build[c(2, 4), ]
location = surfaceGrid(
  obstacles = obstacles,
  obstacles_height_field = "BLDG_HT",
  res = 2,
  offset = 0.01
)
solar_pos = tmy[c(9, 16), c("sun_az", "sun_elev")]
solar_pos = as.matrix(solar_pos)
s = inShadow(
  location = location,
  obstacles = obstacles,
  obstacles_height_field = "BLDG_HT",
  solar_pos = solar_pos
```

```
)
location$shadow = s[, 1]
plotGrid(location, color = c("yellow", "grey")[as.factor(location$shadow)], size = 0.5)
location$shadow = s[, 2]
plotGrid(location, color = c("yellow", "grey")[as.factor(location$shadow)], size = 0.5)

# Method for ground locations raster

ext = as(raster::extent(build) + 20, "SpatialPolygons")
location = raster::raster(ext, res = 2)
proj4string(location) = proj4string(build)
obstacles = build[c(2, 4), ]
solar_pos = tmy[c(9, 16), c("sun_az", "sun_elev")]
solar_pos = as.matrix(solar_pos)
s = inShadow(                ## Using 'solar_pos'
  location = location,
  obstacles = obstacles,
  obstacles_height_field = "BLDG_HT",
  solar_pos = solar_pos,
  parallel = 3
)
time = as.POSIXct(tmy$time[c(9, 16)], tz = "Asia/Jerusalem")
s = inShadow(               ## Using 'time'
  location = location,
  obstacles = obstacles,
  obstacles_height_field = "BLDG_HT",
  time = time,
  parallel = 3
)
plot(s)

# Method for pre-calculated shadow height raster

ext = as(raster::extent(build), "SpatialPolygons")
r = raster::raster(ext, res = 1)
proj4string(r) = proj4string(build)
r[] = rep(seq(30, 0, length.out = ncol(r)), times = nrow(r))
location = surfaceGrid(
  obstacles = build[c(2, 4), ],
  obstacles_height_field = "BLDG_HT",
  res = 2,
  offset = 0.01
)
s = inShadow(
  location = location,
  shadowHeightRaster = r
)
location$shadow = s[, 1]
r_pnt = raster::as.data.frame(r, xy = TRUE)
coordinates(r_pnt) = names(r_pnt)
proj4string(r_pnt) = proj4string(r)
r_pnt = SpatialPointsDataFrame(
  r_pnt,
```

```
    data.frame(
      shadow = rep(TRUE, length(r_pnt)),
      stringsAsFactors = FALSE
      )
   )
  pnt = rbind(location[, "shadow"], r_pnt)
  plotGrid(pnt, color = c("yellow", "grey")[as.factor(pnt$shadow)], size = 0.5)

  # Automatically calculating 'solar_pos' using 'time' - Points
  location = sp::spsample(
    rgeos::gBuffer(rgeos::gEnvelope(build), width = 20),
    n = 500,
    type = "regular"
  )
  time = as.POSIXct("2004-12-24 13:30:00", tz = "Asia/Jerusalem")
  s = inShadow(
    location = location,
    obstacles = build,
    obstacles_height_field = "BLDG_HT",
    time = time
  )
  plot(location, col = ifelse(s[, 1], "grey", "yellow"), main = time)
  plot(build, add = TRUE)


  ## End(Not run)
```

`plotGrid` *Interactive plot for 3D spatial points*

Description

This is a wrapper around `scatterplot3js` from package `threejs`. The function adjusts the x, y and z axes so that 1:1:1 proportion are kept and z=0 corresponds to ground level.

Usage

```
plotGrid(grid, color = c("grey", "red")[as.factor(grid$type)], size = 0.2, ...)
```

Arguments

`grid`	A three-dimensional `SpatialPoints*` object
`color`	Point color, either a single value or vector corresponding to the number of points. The default values draws "facade" and "roof" points in different colors, assuming these classes appear in a column named `type`, as returned by function `surfaceGrid`
`size`	Point radius, default is `0.1`
`...`	Additional parameters passed to `scatterplot3js`

Value

An htmlwidget object that is displayed using the object's show or print method. If you don't see your widget plot, try printing it with the print function. (Same as for `threejs::scatterplot3js`)

Examples

```
## Not run:
grid = surfaceGrid(
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  res = 1,
  offset = 0.01
)
plotGrid(grid)

## End(Not run)
```

`rad2deg` *Radians to degrees*

Description

Radians to degrees

Usage

```
rad2deg(rad)
```

Arguments

`rad`	Angle in radians

Value

`numeric` Angle in degrees

Examples

```
rad2deg(2*pi) == 360
```

radiation *Estimation of Direct and Diffuse Radiation Load on Extruded Polygon Surfaces*

Description

This is a wrapper function for calculating total diffuse, direct and total radiation load per unit area on extruded polygon surfaces. The function operates on obstacle geometry and a set of sun positions with associated meteorological estimates for direct and diffuse radiation (see Details below).

Usage

```
radiation(
  grid,
  obstacles,
  obstacles_height_field,
  solar_pos = solarpos2(obstacles, time),
  time = NULL,
  solar_normal,
  solar_diffuse,
  radius = Inf,
  returnList = FALSE,
  parallel = getOption("mc.cores")
)
```

Arguments

`grid` A 3D `SpatialPointsDataFrame` layer, such as returned by function `surfaceGrid`, specifying the locations where radiation is to be estimated. The layer must include an attribute named `type`, with possible values being `"roof"` or `"facade"`, expressing surface orientation per 3D point. The layer must also include an attribute named `facade_az`, specifying facade azimuth (only for `"facade"` points, for `"roof"` points the value should be `NA`). The `type` and `facade_az` attributes are automatically created when creating the grid with the `surfaceGrid` function

`obstacles` A `SpatialPolygonsDataFrame` object specifying the obstacles outline, inducing self- and mutual-shading on the grid points

`obstacles_height_field` Name of attribute in `obstacles` with extrusion height for each feature

`solar_pos` A `matrix` with two columns representing sun position(s); first column is the solar azimuth (in decimal degrees from North), second column is sun elevation (in decimal degrees); rows represent different sun positions corresponding to the `solar_normal` and the `solar_diffuse` estimates. For example, if `solar_normal` and `solar_diffuse` refer to hourly measurements in a Typical Meteorological Year (TMY) dataset, then `solar_pos` needs to contain the corresponding hourly sun positions

time
: When solar_pos is unspecified, time can be passed to automatically calculate solar_pos based on the time and the centroid of obstacles, using function maptools::solarpos. In such case obstacles must have a defined CRS (not NA). The time value must be a POSIXct or POSIXlt object

solar_normal
: Direct Normal Irradiance (e.g. in Wh/m^2), at sun positions corresponding to solar_pos. Must be a vector with the same number of elements as the number of rows in solar_pos

solar_diffuse
: Diffuse Horizontal Irradiance (e.g. in Wh/m^2), at sun positions corresponding to solar_pos. Must be a vector with the same number of elements as the number of rows in solar_pos

radius
: Effective search radius (in CRS units) for considering obstacles when calculating shadow and SVF. The default is to use a global search, i.e. radius=Inf. Using a smaller radius can be used to speed up the computation when working on large areas. Note that the search radius is not specific per grid point; instead, a buffer is applied on all grid points combined, then "dissolving" the individual buffers, so that exactly the same obstacles apply to all grid points

returnList
: Logical, determines whether to return summed radiation over the entire period per 3D point (default, FALSE), or to return a list with all radiation values per time step (TRUE)

parallel
: Number of parallel processes or a predefined socket cluster. With parallel=1 uses ordinary, non-parallel processing. Parallel processing is done with the parallel package

Details

Input arguments for this function comprise the following:

- An extruded polygon obstacles layer (obstacles and obstacles_height_field) inducing shading on the queried grid
- A grid of 3D points (grid) where radiation is to be estimated. May be created from the 'obstacles' layer, or a subset of it, using function surfaceGrid. For instance, in the code example (see below) radiation is estimated on a grid covering just one of four buildings in the build layer (the first building), but all four buildings are taken into account for evaluating self- and mutual-shading by the buildings.
- Solar positions matrix (solar_pos)
- Direct and diffuse radiation meteorological estimate vectors (solar_normal and solar_diffuse)

Given these inputs, the function goes through the following steps:

- Determining whether each grid point is shaded, at each solar position, using inShadow
- Calculating the coefficient of Direct Normal Irradiance reduction, using coefDirect
- Summing direct radiation considering (1) mutual shading, (2) direst radiation coefficient and (3) direct radiation estimates
- Calculating the Sky View Factor (SVF) for each point, using SVF
- Summing diffuse radiation load considering (1) SVF and (2) diffuse radiation estimates
- Summing total (direct + diffuse) radiation load

Value

If returnList=FALSE (the default), then returned object is a data.frame, with rows corresponding to grid points and four columns corresponding to the following estimates:

- svf Computed Sky View Factor (see function SVF)
- direct Total direct radiation for each grid point
- diffuse Total diffuse radiation for each grid point
- total Total radiation (direct + diffuse) for each grid point

Each row of the data.frame gives summed radiation values for the entire time period in solar_pos, solar_normal and solar_diffuse

If returnList=TRUE then returned object is a list with two elements:

- direct Total direct radiation for each grid point
- diffuse Total diffuse radiation for each grid point

Each of the elements is a matrix with rows corresponding to grid points and columns corresponding to time steps in solar_pos, solar_normal and solar_diffuse

Examples

```
# Create surface grid
grid = surfaceGrid(
  obstacles = build[1, ],
  obstacles_height_field = "BLDG_HT",
  res = 2
)

solar_pos = tmy[, c("sun_az", "sun_elev")]
solar_pos = as.matrix(solar_pos)

# Summed 10-hour radiation estimates for two 3D points
rad1 = radiation(
  grid = grid[1:2, ],
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  solar_pos = solar_pos[8:17, , drop = FALSE],
  solar_normal = tmy$solar_normal[8:17],
  solar_diffuse = tmy$solar_diffuse[8:17],
  returnList = TRUE
)
rad1

## Not run:

# Same, using 'time' instead of 'solar_pos'

rad2 = radiation(
  grid = grid[1:2, ],
  obstacles = build,
```

```
  obstacles_height_field = "BLDG_HT",
  time = as.POSIXct(tmy$time[8:17], tz = "Asia/Jerusalem"),
  solar_normal = tmy$solar_normal[8:17],
  solar_diffuse = tmy$solar_diffuse[8:17],
  returnList = TRUE
)
rad2

# Differences due to the fact that 'tmy' data come with their own
# solar positions, not exactly matching those calulated using 'maptools::solarpos'
rad1$direct - rad2$direct
rad1$diffuse - rad2$diffuse


## End(Not run)

## Not run:

### Warning! The calculation below takes some time.

# Annual radiation estimates for entire surface of one building
rad = radiation(
  grid = grid,
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  solar_pos = solar_pos,
  solar_normal = tmy$solar_normal,
  solar_diffuse = tmy$solar_diffuse,
  parallel = 3
)

# 3D plot of the results
library(plot3D)
opar = par(mfrow=c(1, 3))

scatter3D(
  x = coordinates(grid)[, 1],
  y = coordinates(grid)[, 2],
  z = coordinates(grid)[, 3],
  colvar = rad$direct / 1000,
  scale = FALSE,
  theta = 55,
  pch = 20,
  cex = 1.35,
  clab = expression(paste("kWh / ", m^2)),
  main = "Direct radiation"
)
scatter3D(
  x = coordinates(grid)[, 1],
  y = coordinates(grid)[, 2],
  z = coordinates(grid)[, 3],
  colvar = rad$diffuse / 1000,
  scale = FALSE,
```

```
  theta = 55,
  pch = 20,
  cex = 1.35,
  clab = expression(paste("kWh / ", m^2)),
  main = "Diffuse radiation"
)
scatter3D(
  x = coordinates(grid)[, 1],
  y = coordinates(grid)[, 2],
  z = coordinates(grid)[, 3],
  colvar = rad$total / 1000,
  scale = FALSE,
  theta = 55,
  pch = 20,
  cex = 1.35,
  clab = expression(paste("kWh / ", m^2)),
  main = "Total radiation"
)

par(opar)

## End(Not run)
```

`ray` *Line between two points*

Description

The function connects two points into a line segment.

Usage

```
ray(from, to)
```

Arguments

`from`	A `SpatialPoints*` object specifying origin.
`to`	A `SpatialPoints*` object specifying destination.

Value

A `SpatialLines` object.

Examples

```
ctr = rgeos::gCentroid(build)
angles = seq(0, 359, 20)
sun = mapply(
  shadow:::.sunLocation,
```

```
  sun_az = angles,
  MoreArgs = list(
    location = ctr,
    sun_elev = 10)
)
rays = mapply(ray, MoreArgs = list(from = ctr), to = sun)
rays$makeUniqueIDs = TRUE
rays = do.call(rbind, rays)
plot(rays)
sun = do.call(rbind, sun)
text(sun, as.character(angles))
```

---

shadow — `shadow`*: R Package for Geometric Shade Calculations*

Description

Main functions for calculating:

- `shadowHeight`, Shadow height at individual points or continuous surface
- `shadowFootprint`, Polygonal layer of shadow footprints on the ground
- `SVF`, Sky View Factor (SVF) value at individual points or continuous surface

Typical inputs for these functions include:

- `location`, Queried location(s)
- `obstacles`, A polygonal layer of obstacles (e.g. buildings) outline, with height attributes `obstacles_height_field`
- `solar_pos`, Solar position (i.e. sun azimuth and elevation angles)

The package also provides functions for related preliminary calculations, such as:

- `toSeg`, Converting polygons to line segments
- `classifyAz`, Finding segment azimuth
- `shiftAz`, Shifting segments by azimuth and distance
- `ray`, Constructing a line between two points

shadowFootprint *Shadow footprint on the ground*

Description

Creates a polygonal layer of shadow footprints on the ground, taking into account:

- Obstacles outline (`obstacles`), given by a polygonal layer with a height attribute (`obstacles_height_field`)
- Sun position (`solar_pos`), given by azimuth and elevation angles

The calculation method was inspired by Morel Weisthal's MSc thesis at the Ben-Gurion University of the Negev.

Usage

```
## S4 method for signature 'SpatialPolygonsDataFrame'
shadowFootprint(
  obstacles,
  obstacles_height_field,
  solar_pos = solarpos2(obstacles, time),
  time = NULL,
  b = 0.01
)
```

Arguments

`obstacles`	A `SpatialPolygonsDataFrame` object specifying the obstacles outline
`obstacles_height_field`	Name of attribute in `obstacles` with extrusion height for each feature
`solar_pos`	A `matrix` with one row and two columns; first column is the solar azimuth (in decimal degrees from North), second column is sun elevation (in decimal degrees)
`time`	When `solar_pos` is unspecified, `time` can be passed to automatically calculate `solar_pos` based on the time and the centroid of `obstacles`, using function `maptools::solarpos`. In such case `obstacles` must have a defined CRS (not `NA`). The `time` value must be a `POSIXct` or `POSIXlt` object
`b`	Buffer size for shadow footprints of individual segments of a given polygon; used to eliminate minor internal holes in the resulting shadow polygon.

Value

A `SpatialPolygonsDataFrame` object representing shadow footprint, plus buildings outline. Object length is the same as that of the input `obstacles`, with an individual footprint feature for each obstacle.

References

Weisthal, M. (2014). Assessment of potential energy savings in Israel through climate-aware residential building design (MSc Thesis, Ben-Gurion University of the Negev). https://www.dropbox.com/s/bztnh1fi9znmswj/Thesis_Morel_Weisthal.pdf?dl=1

Examples

```
location = rgeos::gCentroid(build)
time = as.POSIXct("2004-12-24 13:30:00", tz = "Asia/Jerusalem")
solar_pos = maptools::solarpos(
  matrix(c(34.7767978098526, 31.9665936050395), ncol = 2),
  time
  )
footprint1 =               ## Using 'solar_pos'
  shadowFootprint(
    obstacles = build,
    obstacles_height_field = "BLDG_HT",
    solar_pos = solar_pos
    )
footprint2 =               ## Using 'time'
  shadowFootprint(
    obstacles = build,
    obstacles_height_field = "BLDG_HT",
    time = time
    )
all.equal(footprint1, footprint2)
footprint = footprint1
plot(footprint, col = adjustcolor("lightgrey", alpha.f = 0.5))
plot(build, add = TRUE, col = "darkgrey")
```

---

shadowHeight — *Shadow height calculation considering sun position and obstacles*

Description

This function calculates shadow height at given points or complete grid (`location`), taking into account:

- Obstacles outline (`obstacles`), given by a polygonal layer with a height attribute (`obstacles_height_field`)
- Sun position (`solar_pos`), given by azimuth and elevation angles

Usage

```
## S4 method for signature 'SpatialPoints'
shadowHeight(
  location,
  obstacles,
```

```
  obstacles_height_field,
  solar_pos = solarpos2(location, time),
  time = NULL,
  b = 0.01,
  parallel = getOption("mc.cores"),
  filter_footprint = FALSE
)

## S4 method for signature 'Raster'
shadowHeight(
  location,
  obstacles,
  obstacles_height_field,
  solar_pos = solarpos2(pnt, time),
  time = NULL,
  b = 0.01,
  parallel = getOption("mc.cores"),
  filter_footprint = FALSE
)
```

Arguments

location
: A SpatialPoints* or Raster* object, specifying the location(s) for which to calculate shadow height

obstacles
: A SpatialPolygonsDataFrame object specifying the obstacles outline

obstacles_height_field
: Name of attribute in obstacles with extrusion height for each feature

solar_pos
: A matrix with two columns representing sun position(s); first column is the solar azimuth (in decimal degrees from North), second column is sun elevation (in decimal degrees); rows represent different positions (e.g. at different times of day)

time
: When solar_pos is unspecified, time can be passed to automatically calculate solar_pos based on the time and the centroid of location, using function maptools::solarpos. In such case location must have a defined CRS (not NA). The time value must be a POSIXct or POSIXlt object

b
: Buffer size when joining intersection points with building outlines, to determine intersection height

parallel
: Number of parallel processes or a predefined socket cluster. With parallel=1 uses ordinary, non-parallel processing. Parallel processing is done with the parallel package

filter_footprint
: Should the points be filtered using shadowFootprint before calculating shadow height? This can make the calculation faster when there are many point which are not shaded

Value

Returned object is either a numeric matrix or a Raster* -

- If input `location` is a `SpatialPoints*`, then returned object is a `matrix`, where rows represent spatial locations (`location` features), columns represent solar positions (`solar_pos` rows) and values represent shadow height
- If input `location` is a `Raster*`, then returned object is a `RasterLayer` or `RasterStack` where layers represent solar positions (`solar_pos` rows) and pixel values represent shadow height

In both cases the numeric values express shadow height -

- `NA` value means no shadow
- A **valid number** expresses shadow height, in CRS units (e.g. meters)
- `Inf` means complete shadow (i.e. sun below horizon)

Note

For a correct geometric calculation, make sure that:

- The layers `location` and `obstacles` are projected and in same CRS
- The values in `obstacles_height_field` of `obstacles` are given in the same distance units as the CRS (e.g. meters when using UTM)

Examples

```
# Single location
location = rgeos::gCentroid(build)
## Not run:
location_geo = spTransform(location, "+proj=longlat +datum=WGS84")
## End(Not run)
location_geo = matrix(c(34.7767978098526, 31.9665936050395), ncol = 2)
time = as.POSIXct("2004-12-24 13:30:00", tz = "Asia/Jerusalem")
solar_pos = maptools::solarpos(location_geo, time)
plot(build, main = time)
plot(location, add = TRUE)
sun = shadow:::.sunLocation(location = location, sun_az = solar_pos[1,1], sun_elev = solar_pos[1,2])
sun_ray = ray(from = location, to = sun)
build_outline = as(build, "SpatialLinesDataFrame")
inter = rgeos::gIntersection(build_outline, sun_ray)
plot(sun_ray, add = TRUE, col = "yellow")
plot(inter, add = TRUE, col = "red")
shadowHeight(
  location = location,
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  solar_pos = solar_pos
)

# Automatically calculating 'solar_pos' using 'time'
shadowHeight(
  location = location,
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  time = time
```

```
)

## Not run:

# Two points - three times
location0 = rgeos::gCentroid(build)
location1 = raster::shift(location0, 0, -15)
location2 = raster::shift(location0, -10, 20)
locations = rbind(location1, location2)
time = as.POSIXct("2004-12-24 13:30:00", tz = "Asia/Jerusalem")
times = seq(from = time, by = "1 hour", length.out = 3)
shadowHeight(                            ## Using 'solar_pos'
  location = locations,
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  solar_pos = maptools::solarpos(location_geo, times)
)
shadowHeight(                            ## Using 'time'
  location = locations,
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  time = times
)

# Grid - three times
time = as.POSIXct("2004-12-24 13:30:00", tz = "Asia/Jerusalem")
times = seq(from = time, by = "1 hour", length.out = 3)
ext = as(raster::extent(build), "SpatialPolygons")
r = raster::raster(ext, res = 2)
proj4string(r) = proj4string(build)
x = Sys.time()
shadow1 = shadowHeight(
  location = r,
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  time = times,
  parallel = 3
)
y = Sys.time()
y - x
x = Sys.time()
shadow2 = shadowHeight(
  location = r,
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  solar_pos = solarpos2(r, times),
  parallel = 3
)
y = Sys.time()
y - x
shadow = shadow1
opar = par(mfrow = c(1, 3))
for(i in 1:raster::nlayers(shadow)) {
```

```
  plot(shadow[[i]], col = grey(seq(0.9, 0.2, -0.01)), main = raster::getZ(shadow)[i])
  raster::contour(shadow[[i]], add = TRUE)
  plot(build, border = "red", add = TRUE)
}
par(opar)


## End(Not run)
```

---

shiftAz *Shift features by azimuth and distance*

Description

Shift features by azimuth and distance

Usage

```
shiftAz(object, az, dist)
```

Arguments

object	The object to be shifted.
az	Shift azimuth, in decimal degrees.
dist	Shift distance, in `object` projection units.

Value

The shifted `object`.

Examples

```
s = c(270, 90, 180, 0)
build_shifted = shiftAz(build, az = s, dist = 2.5)
plot(build)
plot(build_shifted, add = TRUE, border = "red")
raster::text(rgeos::gCentroid(build, byid = TRUE), s)
```

`solarpos2` *Calculate solar position(s) for location and time*

Description

This is a wrapper function around `maptools::solarpos`, adapted for accepting location as a `Spatial*` layer or a `Raster`. The function calculates layer centroid, transforms it to lon-lat, then calls `maptools::solarpos` to calculate solar position(s) for that point at the given time(s)

Usage

```
solarpos2(location, time)
```

Arguments

`location`	A `Spatial*` or a `Raster` object
`time`	A `SpatialLines*` or a `SpatialPolygons*` object

Value

A `matrix` with two columns representing sun position(s); first column is the solar azimuth (in decimal degrees from North), second column is sun elevation (in decimal degrees); rows represent different times corresponding to `time`

Examples

```
time = as.POSIXct("2004-12-24 13:30:00", tz = "Asia/Jerusalem")
solarpos2(build, time)
```

`surfaceGrid` *Create grid of 3D points covering the 'facades' and 'roofs' of obstacles*

Description

The function creates a grid of 3D points covering the given obstacles at specified resolution. Such a grid can later on be used to quantify the shaded / non-shaded proportion of the obstacles surface area.

Usage

```
surfaceGrid(obstacles, obstacles_height_field, res, offset = 0.01)
```

Arguments

obstacles — A SpatialPolygonsDataFrame object specifying the obstacles outline

obstacles_height_field — Name of attribute in obstacles with extrusion height for each feature

res — Required grid resolution, in CRS units

offset — Offset between grid points and facade (horizontal distance) or between grid points and roof (vertical distance).

Value

A 3D SpatialPointsDataFrame layer, including all attributes of the original obstacles each surface point corresponds to, followed by six new attributes:

- obs_id Unique consecutive ID for each feature in obstacles
- type Either "facade" or "roof"
- seg_id Unique consecutive ID for each facade segment (only for 'facade' points)
- xy_id Unique consecutive ID for each ground location (only for 'facade' points)
- facade_az The azimuth of the corresponding facade, in decimal degrees (only for 'facade' points)

Note

The reason for introducing an offset is to avoid ambiguity as for whether the grid points are "inside" or "outside" of the obstacle. With an offset all grid points are "outside" of the building and thus not intersecting it. offset should be given in CRS units; default is 0.01.

See Also

Function plotGrid to visualize grid.

Examples

```
grid = surfaceGrid(
  obstacles = build,
  obstacles_height_field = "BLDG_HT",
  res = 2
)
plot(grid)
plot(grid, pch = 1, lwd = 0.1, col = "black", add = TRUE)

# When 'res/2' is larger then height, facade will be left unsampled
build_small = build
build_small$BLDG_HT = 1
grid = surfaceGrid(
  obstacles = build_small,
  obstacles_height_field = "BLDG_HT",
  res = 2
)
```

```
plot(grid)
plot(grid, pch = 1, lwd = 0.1, col = "black", add = TRUE)
table(grid$type)

grid = surfaceGrid(
  obstacles = build_small,
  obstacles_height_field = "BLDG_HT",
  res = 2.00001  # res/2 > h
)
plot(grid)
plot(grid, pch = 1, lwd = 0.1, col = "black", add = TRUE)
table(grid$type)

# When input already contains 'obs_id', 'type', 'seg_id', 'xy_id', 'facade_az' or 'ZZZ'
build2 = build
build2$ZZZ = 1
grid = surfaceGrid(
  obstacles = build2,
  obstacles_height_field = "BLDG_HT",
  res = 2
)
```

SVF *Sky View Factor (SVF) calculation*

Description

Calculates the Sky View Factor (SVF) at given points or complete grid (`location`), taking into account obstacles outline (`obstacles`) given by a polygonal layer with a height attribute (`obstacles_height_field`).

Usage

```
## S4 method for signature 'SpatialPoints'
SVF(
  location,
  obstacles,
  obstacles_height_field,
  res_angle = 5,
  b = 0.01,
  parallel = getOption("mc.cores")
)

## S4 method for signature 'Raster'
SVF(
  location,
  obstacles,
  obstacles_height_field,
  res_angle = 5,
```

```
  b = 0.01,
  parallel = getOption("mc.cores")
)
```

Arguments

`location` A `SpatialPoints*` or `Raster*` object, specifying the location(s) for which to calculate logical shadow values. If `location` is `SpatialPoints*`, then it can have 2 or 3 dimensions. A 2D `SpatialPoints*` is considered as a point(s) on the ground, i.e. 3D point(s) where $z = 0$. In a 3D `SpatialPoints*` the 3rd dimension is assumed to be elevation above ground z (in CRS units). `Raster*` cells are considered as ground locations

`obstacles` A `SpatialPolygonsDataFrame` object specifying the obstacles outline

`obstacles_height_field` Name of attribute in `obstacles` with extrusion height for each feature

`res_angle` Circular sampling resolution, in decimal degrees. Default is 5 degrees, i.e. 0, 5, 10... 355.

`b` Buffer size when joining intersection points with building outlines, to determine intersection height

`parallel` Number of parallel processes or a predefined socket cluster. With `parallel=1` uses ordinary, non-parallel processing. Parallel processing is done with the `parallel` package

Value

A numeric value between 0 (sky completely obstructed) and 1 (sky completely visible).

- If input `location` is a `SpatialPoints*`, then returned object is a `vector` where each element representing the SVF for each point in `location`
- If input `location` is a `Raster*`, then returned object is a `RasterLayer` where cell values express SVF for each ground location

Note

SVF calculation for each view direction follows the following equation -

$$1 - (sin(\beta))^2$$

Where β is the highest elevation angle (see equation 3 in Gal & Unger 2014).

References

Erell, E., Pearlmutter, D., & Williamson, T. (2012). Urban microclimate: designing the spaces between buildings. Routledge.

Gal, T., & Unger, J. (2014). A new software tool for SVF calculations using building and tree-crown databases. Urban Climate, 10, 594-606.

Examples

```
## Individual locations
location0 = rgeos::gCentroid(build)
location1 = raster::shift(location0, 0, -15)
location2 = raster::shift(location0, -10, 20)
locations = rbind(location1, location2)
svfs = SVF(
  location = locations,
  obstacles = build,
  obstacles_height_field = "BLDG_HT"
)
plot(build)
plot(locations, add = TRUE)
raster::text(locations, round(svfs, 2), col = "red", pos = 3)

## Not run:

## Grid
ext = as(raster::extent(build), "SpatialPolygons")
r = raster::raster(ext, res = 5)
proj4string(r) = proj4string(build)
pnt = raster::rasterToPoints(r, spatial = TRUE)
svfs = SVF(
    location = r,
    obstacles = build,
    obstacles_height_field = "BLDG_HT",
    parallel = 3
  )
plot(svfs, col = grey(seq(0.9, 0.2, -0.01)))
raster::contour(svfs, add = TRUE)
plot(build, add = TRUE, border = "red")

## 3D points
ctr = rgeos::gCentroid(build)
heights = seq(0, 28, 1)
loc3d = data.frame(
    x = coordinates(ctr)[, 1],
    y = coordinates(ctr)[, 2],
    z = heights
)
coordinates(loc3d) = ~ x + y + z
proj4string(loc3d) = proj4string(build)
svfs = SVF(
    location = loc3d,
    obstacles = build,
    obstacles_height_field = "BLDG_HT",
    parallel = 3
)
plot(heights, svfs, type = "b", xlab = "Elevation (m)", ylab = "SVF", ylim = c(0, 1))
abline(v = build$BLDG_HT, col = "red")

## Example from Erell et al. 2012 (p. 19 Fig. 1.2)
```

```
# Geometry
pol1 = rgeos::readWKT("POLYGON ((0 100, 1 100, 1 0, 0 0, 0 100))")
pol2 = rgeos::readWKT("POLYGON ((2 100, 3 100, 3 0, 2 0, 2 100))")
pol = sp::rbind.SpatialPolygons(pol1, pol2, makeUniqueIDs = TRUE)
pol = sp::SpatialPolygonsDataFrame(pol, data.frame(h = c(1, 1)), match.ID = FALSE)
pnt = rgeos::readWKT("POINT (1.5 50)")
plot(pol, col = "grey", xlim = c(0, 3), ylim = c(45, 55))
plot(pnt, add = TRUE, col = "red")

# Fig. 1.2 reproduction
h = seq(0, 2, 0.1)
svf = rep(NA, length(h))
for(i in 1:length(h)) {
  pol$h = h[i]
 svf[i] = SVF(location = pnt, obstacles = pol, obstacles_height_field = "h", res_angle = 1)
}
plot(h, svf, type = "b", ylim = c(0, 1))

# Comparison with SVF values from the book
test = c(1, 0.9805806757, 0.9284766909, 0.8574929257, 0.7808688094,
0.7071067812, 0.6401843997, 0.5812381937, 0.52999894, 0.4856429312,
0.4472135955, 0.4138029443, 0.3846153846, 0.3589790793, 0.336336397,
0.316227766, 0.2982749931, 0.282166324, 0.2676438638, 0.2544932993,
0.242535625)
range(test - svf)


## End(Not run)
```

`tmy` *Typical Meteorological Year (TMY) solar radiation in Tel-Aviv*

Description

A table with hourly solar radiation estimates for a typical meteorological year in Tel-Aviv.

- `time` Time, as `character` in the `"%Y-%m-%d %H:%M:%S"` format, e.g. `"2000-01-01 06:00:00"`, referring to local time
- `sun_az` Sun azimuth, in decimal degrees from North
- `sun_elev` Sun elevation, in decimal degrees
- `solar_normal` Direct Normal Irradiance, in Wh/m^2
- `solar_diffuse` Diffuse Horizontal Irradiance, in Wh/m^2
- `dbt` Dry-bulb temperature, in Celsius degrees
- `ws` Wind speed, in m/s

Usage

```
tmy
```

Format

A `data.frame` with 8760 rows and 7 columns.

References

https://energyplus.net/weather-location/europe_wmo_region_6/ISR//ISR_Tel.Aviv-Bet.Dagan.401790_MSI

`tmy2` *Typical Meteorological Year (TMY) solar radiation in Beer-Sheva*

Description

A table with hourly solar radiation estimates for a typical meteorological year in Beer-Sheva.

- `time` Time, as `character` in the `"%Y-%m-%d %H:%M:%S"` format, e.g. `"2000-01-01 06:00:00"`, referring to local time
- `sun_az` Sun azimuth, in decimal degrees from North
- `sun_elev` Sun elevation, in decimal degrees
- `solar_normal` Direct Normal Irradiance, in Wh/m^2
- `solar_diffuse` Diffuse Horizontal Irradiance, in Wh/m^2
- `dbt` Dry-bulb temperature, in Celsius degrees
- `ws` Wind speed, in m/s

Usage

```
tmy2
```

Format

A `data.frame` with 8760 rows and 7 columns.

References

https://energyplus.net/weather-location/europe_wmo_region_6/ISR//ISR_Beer.Sheva.401900_MSI

toGMT *Local time to GMT*

Description

The function transforms a POSIXct object in any given time zone to GMT.

Usage

```
toGMT(time)
```

Arguments

time	Time, a POSIXct object.

Value

A a POSIXct object, in GMT.

Examples

```
time = as.POSIXct("1999-01-01 12:00:00", tz = "Asia/Jerusalem")
toGMT(time)
```

toSeg *Split polygons or lines to segments*

Description

Split lines or polygons to separate segments.

Usage

```
toSeg(x)
```

Arguments

x	A SpatialLines* or a SpatialPolygons* object

Value

A SpatialLines object where each segment is represented by a separate feature

References

This function uses a modified version of code from the following 'r-sig-geo' post by Roger Bivand: https://stat.ethz.ch/pipermail/r-sig-geo/2013-April/017998.html

Examples

```
seg = toSeg(build[1, ])
plot(seg, col = sample(rainbow(length(seg))))
raster::text(rgeos::gCentroid(seg, byid = TRUE), 1:length(seg))

# Other data structures
toSeg(geometry(build)) # SpatialPolygons
toSeg(boston_sidewalk) # SpatialLinesDataFrame
toSeg(geometry(boston_sidewalk)) # SpatialLinesDataFrame
```

Index