Package: wflo (via r-universe)

September 15, 2024

Title Data Set and Helper Functions for Wind Farm Layout Optimization Problems

Version 1.9

Date 2024-03-18

Description Provides a convenient data set, a set of helper functions, and a benchmark function for economically (profit) driven wind farm layout optimization. This enables researchers in the field of the NP-hard (non-deterministic polynomial-time hard) problem of wind farm layout optimization to focus on their optimization methodology contribution and also provides a realistic benchmark setting for comparability among contributions. See Croonenbroeck, Carsten & Hennecke, David (2020) <doi:10.1016/j.energy.2020.119244>.

Depends R (>= 3.5.0)

Imports plotrix, progress, emstreeR, terra, sf

Suggests MASS, nloptr, pso, rgenoud, snow, doSNOW, foreach, parallel

License GPL-3

Encoding UTF-8

LazyData true

NeedsCompilation no

Author Carsten Croonenbroeck [aut, cre], David Hennecke [ctb]

Maintainer Carsten Croonenbroeck

`<carsten.croonenbroeck@uni-rostock.de>`

Repository CRAN

Date/Publication 2024-03-18 20:50:02 UTC

Contents

`AcquireData` *Downloads the larger data set for entire Germany.*

Description

Convenience function that downloads the larger data set covering the entire area of Germany and saves it to the specified directory. This data set can replace the smaller sub sample data set that is part of this package.

Usage

```
AcquireData(Folder)
```

Arguments

`Folder`	must be a character string containing the folder location to where the file will be saved. For instance, use `getwd` to save to the current working directory.

Details

This function will interactively lead the user through the downloading process and also gives advice on how to use the larger data set. Make sure that R can write to the specified directory.

Value

`AcquireData` returns nothing.

Author(s)

Carsten Croonenbroeck

See Also

Use `FarmVars` for the settings object and `FarmData` for the smaller, built-in data set that is part of this package.

Examples

```
## Not run:
AcquireData(tempdir())
# Will download the data file to the specified directory.

## End(Not run)
```

`AirDensity` *Provides convenient computations for air density.*

Description

Implements a set of computations and constants to compute air density as a function of altitude (a.s.l.), temperature and relative humidity.

Usage

```
AirDensity(Altitude = 0, Temperature = 20, phi = 0.76)
```

Arguments

`Altitude`	altitude in meters above sea level.
`Temperature`	temperature in degrees Celsius.
`phi`	relative humidity, must be within [0, 1].

Details

The function first computes air pressure at user provided target altitude based on the barometric formula. Then, saturation vapor pressure is computed using the Magnus formula. With both, the density of air is computed afterward.

Value

`AirDensity` returns a single value representing air density in kg per cubic meter.

Note

This function returns valid values for negative altitudes, but only for temperature values between -45 and +60 degrees Celsius.

Author(s)

Carsten Croonenbroeck

See Also

`WindspeedLog` and `WindspeedHellmann` to compute wind speed at target heights using two slightly different established models.

Examples

```
AirDensity()
AirDensity(300, 25, 0.6)
AirDensity(0, 0, 0) # Standard conditions according to DIN 1343:1990,
                    # returns the expected value of 1.292.
```

`Area` *Computes the overlap area for the partial Jensen wake.*

Description

The partial Jensen wake model requires the overlap area of the rotor disc at the downwind location with the wake disc. This function computes it.

Usage

```
Area(a, b, d)
```

Arguments

`a`	must be a single value. Provide the radius of the first circle in meters.
`b`	must be a single value. Provide the radius of the second circle in meters.
`d`	must be a single value. Provide the distance between the two circles' centers.

Details

If a turbine is downwind another turbine, the wake cone of that upwind turbine may only partially cover the rotor disc of the second turbine. For the partial wake model it is necessary to compute the covered area.

Value

`Area` returns covered area in square meters.

Author(s)

Carsten Croonenbroeck

See Also

`JensenTrapezoid` to check whether there are wake effects present. `FarmVars` for the data object.

Examples

```
Area(60, 40, 50)
# Returns 2930.279.
```

`Cost` *Stub for a turbine's cost function.*

Description

A function that returns the yearly installation costs for a given set of turbines (provide x and y for the turbines' locations). In its present form it only returns the 'UnitCost' value from the `FarmVars` settings object per turbine.

Usage

```
Cost(x, y)
```

Arguments

`x`	can be a single value or a numeric vector of values, contains the 'x' location(s) of turbines.
`y`	can be a single value or a numeric vector of values, contains the 'y' location(s) of turbines.

Details

Note that x and y should both be of length n, i.e. the numbers of values they contain should match the number of turbines in the current wind farm layout problem.
This function is a stub and can and should be replaced by something reasonable in an actual wind farm layout problem.

Value

Cost returns a vector of values. The number of elements matches the length of x and y.

Author(s)

Carsten Croonenbroeck

See Also

Profit to see where to use Cost, Yield for a similar function for yearly yield.

Examples

```
## Returns a vector of two, c(100000, 100000).
Cost(c(0.5, 0.7), c(0.2, 0.3))

## Replace the function by another function
## also called 'Cost', embedded in environment e.
## Also, see the vignette.
## Not run:
e$Cost <- function(x, y) #x, y \in R^n
{
 retVal <- rep(e$FarmVars$UnitCost, min(length(x), length(y)))
 retVal[x > 0.5] <- retVal[x > 0.5] * 2
 return(retVal)
}
set.seed(1357)
NumTurbines <- 4 # For example.
Result <- pso::psoptim(par = runif(NumTurbines * 2), fn = Profit,
  lower = rep(0, NumTurbines * 2), upper = rep(1, NumTurbines * 2))
Result
rm(Cost, envir = e)

## End(Not run)
```

---

e *Environment for data and variables.*

Description

Environment that contains variables object FarmVars and, if downloaded, full data set FarmData.

Usage

```
e
```

Details

`e` contains data and variables, see `FarmVars` and `FarmData`.

Value

`e` returns the environment for `FarmVars` and `FarmData`.

Author(s)

Carsten Croonenbroeck

See Also

`AcquireData` for downloading the full data set.

Examples

```
e
## Gives the environment object.
```

`FarmData` *Data set for wind farm layout optimization.*

Description

This list object contains four matrices covering adjusted yield, wind speed, wind direction and standard deviations of wind directions in Germany.

Usage

```
FarmData
```

Format

A `list` object containing seven matrices, each containing 25 x 25 values at a raster resolution of 200 x 200 m (note that for the larger data set downloadable using `AcquireData`, each matrix contains 4400 x 3250 values):

`$AdjustedYield`: Yield is average annual energy production (AEP). According to FGW technical guidelines, AEP is adjusted due to different location qualities to obtain a better guess at the marketable energy output. Interpret these values directly as 'megawatt hours per year'.

`$WindSpeed`: Average wind speeds in meters per second.

`$WindDirection`: Average wind directions in degrees (azimuth system).

`$SDDirection`: Standard deviations of wind directions in degrees (azimuth system).

`$Elevation`: Terrain relief (elevation in meters).

`$Slope`: Terrain slope in degrees.

`$SlopeDirection`: Direction of hillside in degrees.

Note

If the full mode data set is present, it is loaded into the environment `e`. The built-in data set `FarmData` contains matrices of dimension 25 x 25, while the full data set `e$FarmData` consists of 4400 x 3250 matrices.

Author(s)

Carsten Croonenbroeck
David Hennecke

Source

DWD Climate Data Center (CDC): ftp://opendata.dwd.de/climate_environment/CDC/grids_germany/multi_annual/wind_parameters/resol_200x200/

FGW Technical Guidelines: https://wind-fgw.de/shop/technical-guidelines/?lang=en

Croonenbroeck, C. & Hennecke, D. Does the German renewable energy act provide a fair incentive system for onshore wind power? - A simulation analysis, Energy Policy, 2020, 114, 111663, 1-15, https://doi.org/10.1016/j.enpol.2020.111663

Examples

```
# 'Profit' uses this data set internally:
NumTurbines <- 4
set.seed(1357)
Result <- optim(par = runif(NumTurbines * 2), fn = Profit, method = "L-BFGS-B",
  lower = rep(0, NumTurbines * 2), upper = rep(1, NumTurbines * 2))
Result
PlotResult(Result)
```

`FarmVars` *Variables object for wind farm specifications.*

Description

This list object collects all necessary variables for wind farm layout optimizations within this package. The object is loaded upon loading the package, embedded into a user writeable environment `e` and provides a set of default values.

Usage

```
FarmVars
```

Format

A `list` object containing:

`$UnitCost`: This contains yearly costs per turbine in actual currency. Mostly, this will be installation costs, maintenance cost and others. If, for example, total turbine installation costs are 2 mio. EUR and the projected life span of the turbine is 20 years, then the yearly cost is roughly 100,000 EUR, which is also the default value for this item. Refine the number at will to encompass annuity (taking interest effects into consideration), salvage, or the mentioned maintenance cost.

`$Price`: This denotes the average sale price of electricity produced at the wind farm per megawatt hour. Defaults to 100, since 100 EUR per megawatt hour is a good (still rough) estimate in the European electricity market.

`$StartPoint`: Denotes the point at which to start to 'cut out' a sample field from the data set. Since the full data set covers the entire region of Germany at a 200 x 200 meters resolution, the entire area is too big for a wind farm. For an actual wind farm layout optimization problem, a smaller region is indicated. This package will conveniently provide a square area of 5 km x 5 km in size. Note: Concerning the larger data set covering the entire area of Germany (conveniently downloadable using `AcquireData`), this area con be found at point 2000:2000 (full data covering a range of 4400 x 3250 points) and from there, spans a five kilometers square, i.e. covers the raster at 2000:2024, 2000:2024. Change the default value of 1 to 2000 to find that area or to something else for cross checking your algorithm with data covering a different area. However, for comparability, evaluate your algorithm result at the default of 2000 or using the built-in data set.

`$Width`: Denotes the width (and, since the test area is a square, also the height) of the test area. Defaults to 25, which leads to a 5 km x 5 km square as a sample wind field, as the raster resolution is 200 x 200 m.

`$MeterMinDist`: Denotes the minimum distance from one turbine to another in meters. Defaults to 500 meters, which is a rather conservative value.

`$EndPoint`: The complement to '$StartPoint'. Is computed based on '$StartPoint' and '$Width' as follows: FarmVars$StartPoint + FarmVars$Width - 1. By default, $StartPoint = 1 and $Width = 25, so $EndPoint = 1 + 25 - 1 = 25.

`$MeterWidth`: Contains the width (and, since the test area is a square, also the height) of the test area in meters. Since by default, FarmVars$Width = 25 and the raster resolution is 200 x 200 m, this defaults to $MeterWidth = 200 * 25 = 5000 meters (5 km).

`$MinDist`: Contains the minimum distance in problem space. Since the problem space in a convenient unit square, this defaults to $MinDist = MeterMinDist / MeterWidth = 500 / 5000 = 0.1.

`$z`: Contains the hub height of the turbine type under investigation. Default value is 100 meters.

`$z0`: Contains the terrain's roughness length required for Jensen's wake model. In most studies, a default value of 0.1 is agreed to be a good guess for onshore wind farms, so this is the default value here.

`$r0`: Contains the rotor radius for the turbine type under investigation. For instance, a Vestas V90 turbine has a rotor diameter of 90 meters (hence the name), so the radius is 45 mesters, which is also the default value here.

`Partial`: Selects whether to incorporate partial Jensen wake or not. Defaults to TRUE.

`$BenchmarkSolution`: Contains the best setup for the standard benchmark field at a task of placing 20 turbines. This result has been generated by optimizer `genoud()` from package `rgenoud`. It should serve as a benchmark or starting point for improved solutions. e$FarmVars$BenchmarkSolution is taken from Croonenbroeck & Hennecke (2020). Note, however, that the computed profit is a little less than the 12,053,607 EUR reported there, since, starting from version 1.6, wflo by default takes partial Jensen wake effects into account. This does however in no way constrain the representativeness of e$FarmVars$BenchmarkSolution.

Source

Carsten Croonenbroeck

References

Croonenbroeck, C. & Hennecke, D. A Comparison of Optimizers in a Unified Standard for Optimization on Wind Farm Layout Optimization, Energy, 2020, 119244, 1-15, https://doi.org/10.1016/j.energy.2020.119244

Examples

```
# Inspect the benchmark solution by
Result <- list(par = e$FarmVars$BenchmarkSolution)
Profit(Result$par)
PlotResult(Result)
ShowWakePenalizers(Result)
```

`GaussWS` *Convenience function to look-up values from a object returned by GenerateGauss.*

Description

Dependent on the resolution at which `GenerateGauss` was set to use, the resolution coordinates (array indices) cannot immediately be interpreted as being measured in meters. This function returns wind speeds from the Gauss object accoring to x, y, z simply provided in meters.

Usage

```
GaussWS(Gauss, x, y, z)
```

Arguments

`Gauss`	must be an object returned by `GenerateGauss`.
`x`	must be a single value. Provide desired distance in meters.
`y`	must be a single value. Provide desired distance in meters. May be negative.
`z`	must be a single value. Provide desired distance in meters.

Details

The Gaussian wake model is loosely based on the initial contribution by Bastankhah & Porte-Agel (2014).

Value

`GaussWS` returns a single number which can be considered a wind speed in the wake of a turbine at location x, y, and z.

Author(s)

Carsten Croonenbroeck

References

Bastankhah, M., & Porte-Agel, F. (2014). A new analytical model for wind-turbine wakes. Renewable Energy, 70, 116-123.

See Also

Use `GenerateGauss` to compute the three-dimensional tensor array object containing the wind speed data. See `QuickGauss3D` for the same algorithm, immediately returning the wind speed at one single point only.

Examples

```
## Not run:
GaussWS(Gauss, 100, 1, 100)
GaussWS(Gauss, 80, -40, 90)
GaussWS(Gauss, 200, 40, 150)

## End(Not run)
```

GenerateGauss *For an incoming wind speed at reference height, this function computes a 3D tensor object containing Gaussian model based wind speeds.*

Description

This function pre-computes Gaussian wind speeds and stores them in a 3D array, similar to voxels. Using that 'table', wind speeds can be looked up very quickly, which makes Gaussian wake feasible during WFLO runs.

Usage

```
GenerateGauss(u = 8, refHeight = 10, maxX = 500, resY = 100,
resZ = 100, Verbose = TRUE)
```

Arguments

`u`	measured wind speed at reference height. Will mostly be measured in meters per second.
`refHeight`	reference height in meters. This is the height at which the incoming wind speed u is measured.
`maxX`	the number of steps down the x axis for which to compute the model.
`resY`	the number of steps along the y axis for which to compute the model. Note that as y may take negative values, the resolution space should be chosen not too small, here. If, e.g., resY = 100, this means that y may take values from -50 to 50, which may be too low a resolution in some cases.
`resZ`	the number of steps up the z axis for which to compute the model.
`Verbose`	selectes whether the function displays status reports during computation, as it may take some time, dependent on the resolution setting.

Details

Users may choose to compute a rather fine resolution run over night and then save the returned object so it can be loaded in future sessions. The Gaussian wake model is loosely based on the initial contribution by Bastankhah & Porte-Agel (2014).

Value

`GenerateGauss` returns the three-dimensional array containing wind speeds.

Note

Note that the model assumes that along the x axis, x = 0 is the turbine location. x expands along the wind direction downwind. y denoted whether a point is 'left' or 'right' the x axis. Thus, the x-z plane is the plane along the x axis and perpendicular to the ground. The z axis is hight, starting at 0 = ground level.

Author(s)

Carsten Croonenbroeck

References

Bastankhah, M., & Porte-Agel, F. (2014). A new analytical model for wind-turbine wakes. Renewable Energy, 70, 116-123.

See Also

Use `GaussWS` for a convenience function to look-up the values from the returned array. See `QuickGauss3D` for the same algorithm, immediately returning the wind speed at one single point only.

Examples

```
## Not run:
Gauss <- GenerateGauss(maxX = 500, resY = 1000, resZ = 1000)

## End(Not run)
```

`Geo2Ari` *Converts degrees between the arithmetic system and the azimuth system (and vice versa).*

Description

Use this function to convert degrees from the arithmetic system (0° being east, ascending counterclockwise) into the azimuth system (0° being north, ascending clockwise) and vice versa.

Usage

```
Geo2Ari(g)
```

Arguments

`g`	contains a single value degree (usually between 0° and 360°, decimal fractions allowed).

Details

`g` may contain degrees from both systems, the function turns the data into the respective other system.

Value

`Geo2Ari` returns a single value of degrees.

Author(s)

Carsten Croonenbroeck

See Also

`GetDirInfo` for further degree information.

Examples

```
Geo2Ari(0)
## In an arithmetic system, 0° means 'east', while 'east' in
## azimuth notation is 90°. This call returns 90.
Geo2Ari(90)
## In an azimuth system, 90° means 'east', while 'east' in
## arithmetic notation is 0°. This call returns 0.
Geo2Ari(Geo2Ari(123))
## Returns 123.
```

`GetAngle` *Returns the angle between two turbines.*

Description

As seen from a point in the wind farm, computes the angle to another point in that farm.

Usage

```
GetAngle(x1, y1, x2, y2)
```

Arguments

`x1`	must be a single value. Provide the `x` location of the first turbine.
`y1`	must be a single value. Provide the `y` location of the first turbine.
`x2`	must be a single value. Provide the `x` location of the second turbine.
`y2`	must be a single value. Provide the `y` location of the second turbine.

Details

From point 2's point of view, computes the angle to point 1.

Value

`GetAngle` returns a single number between 0 and 360. If both points are identical, the return value is 0.

Note

Note that this function returns an angle in arithmetic degrees notation. To convert to azimuth notation, use `Geo2Ari`.

Author(s)

Carsten Croonenbroeck

See Also

Use `JensenAngle` to compute the wake cone and with it, use `JensenTrapezoid` to see if another turbine B is in turbine A's wake.

Examples

```
GetAngle(0.2, 0.2, 0.1, 0.1)
## Looking from point (0.1, 0.1) at point (0.2, 0.2), the angle is 45° (arithmetic).
```

`GetArrow` *Simple helper function for* `PlotResult`.

Description

Given a point, an angle information (in arithmetic degrees), and a length information, computes start and end points of an arrow usable via the `arrows` function.

Usage

```
GetArrow(BaseX, BaseY, Degrees, Frac = 25)
```

Arguments

`BaseX`	must be a single value containing the x value of a point which later will be the center of the arrow.
`BaseY`	must be a single value containing the y value of a point which later will be the center of the arrow.
`Degrees`	must be a single value containing the desired rotation degree of the arrow.
`Frac`	must be a single value containing the length of the arrow. Default is 25 and for convenience, this parameter should in most cases be identical to FarmVars$Width.

Details

This function will be used internally by `PlotResult`.

Value

`GetArrow` returns a vector of four values representing x and y for the start point and x and y for the end point of an arrow (in that order).

Author(s)

Carsten Croonenbroeck

See Also

Use PlotResult to visualize the optimization result. See FarmVars for the data object.

Examples

```
GetArrow(0.5, 0.5, 45)
#At c(0.5, 0.5), generates an arrow pointing in north-eastern direction.
```

GetDirInfo — *Returns average wind direction and direction standard deviation for a turbine's location.*

Description

For a turbine's location represented by `x` and `y`, looks up the wind direction from the matrix `Dirs` and the corresponding standard deviation from matrix `SDs`. Internally transforms coordinates of `x` and `y` from problem space (usually unit square) to the matrix space of `Dirs` and `SDs`, respectively.

Usage

```
GetDirInfo(x, y, Dirs, SDs)
```

Arguments

`x`	must be a single value containing the 'x' location of a turbine in problem space.
`y`	must be a single value containing the 'y' location of a turbine in problem space.
`Dirs`	a matrix containing average yearly wind directions. Usually, the third element of the list object FarmData will be used as this matrix.
`SDs`	a matrix containing average yearly wind direction standard deviations. Usually, the fourth element of the list object FarmData will be used as this matrix.

Value

`GetDirInfo` returns a vector of two elements, the first being the average wind direction in degrees and the second being the corresponding standard deviation. Note that degrees are meant in the arithmetic degrees system ($0°$ being east, ascending counterclockwise). To transform into an azimuth system ($0°$ being north, ascending clockwise), use function Geo2Ari. Also note that wind directions are meant to denote 'where the wind is going to' rather than 'where the wind is coming from'.

Author(s)

Carsten Croonenbroeck

See Also

Profit to see where to use GetDirInfo, Yield for a similar function for adjusted yield. See FarmVars for the data object.

Examples

```
Dirs <- FarmData[[3]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
e$FarmVars$StartPoint:e$FarmVars$EndPoint]
SDs <- FarmData[[4]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
e$FarmVars$StartPoint:e$FarmVars$EndPoint]
GetDirInfo(0.5, 0.7, Dirs, SDs)
## Returns wind direction and standard deviation for the given location
## and provided the matrices given.
```

GetFarmFromLonLat *Accesses full FarmData set and returns a compatible list object for the requested location.*

Description

This function accepts a pair longitude (decimal system, east) and latitude (decimal system, north) coordinates as well a a desired edge length in m and if valid, returns a list object similar to FarmData, but at the specified location and with the specified size.

Usage

```
GetFarmFromLonLat(Top, Left, EdgeLen, DoPlot = FALSE)
```

Arguments

Top	longitude value (decimal system, east).
Left	latitude value (decimal system, north).
EdgeLen	edge length in meters. Note that the returned farm data object is always a square area and thus, contains square matrices.
DoPlot	optionally plots the annual energy production (AEP) 'landscape' in the returned object using a terrain.colors coloring scheme, together with a color key.

Details

Requires that the full FarmData dataset is loaded. See AcquireData on how to obtain it.

Value

Returns a list object following the structure of FarmData, but only containing the farm area starting from the top-left point specified and as many 'tiles' required to meet the desired edge length at 200 m tiles resolution.

Author(s)

Carsten Croonenbroeck

See Also

See `Index2GK` to convert index coordinates to Gauss-Kruger coordinates, and `GK2LonLat` to convert Gauss-Kruger coordinates to longitude/latitude coordinates.

Examples

```
# This will return a farm at the specified location, edge length 5,000 m (5 km).
# Requires full data set to be loaded.
## Not run:
MyFarm <- GetFarmFromLonLat(51.49594, 11.58818, 5000, DoPlot = FALSE)

## End(Not run)
```

GK2Index *Converts Gauss-Kruger coordinates to FarmData indices.*

Description

This function accepts Gauss-Kruger (GK) coordinates that are within the FarmData space and converts them to a pair of indices that point to locations within the FarmData matrices.

Usage

```
GK2Index(X, Y)
```

Arguments

X	a numeric value for the 'right' value in the GK coordinate system.
Y	a numeric value for the 'top' value in the GK coordinate system.

Details

Assumes that the GK zone is EPSG 31467. References to indices valid if the full data set is loaded, see `FarmData` and `AcquireData`.

Value

Returns index values between 1 and 3,250 for X and between 1 and 4,400 for Y.

Author(s)

Carsten Croonenbroeck

See Also

Index2GK for the inverse.

Examples

```
GK2Index(3280000, 5230000) # Will return c(1, 4400), the lower left point.
GK2Index(3929800, 6109800) # Will return c(3250, 1), the top right point.
```

`GK2LonLat` *Converts Gauss-Kruger coordinates to longitude/latitude coordinates.*

Description

This function accepts a pair of Gauss-Kruger (GK) coordinates and converts them to longitude (decimal system, east) and latitude (decimal system, north) coordinates by performing re-projection.

Usage

```
GK2LonLat(X, Y)
```

Arguments

`X`	a numeric value for the 'right' value in the GK coordinate system.
`Y`	a numeric value for the 'top' value in the GK coordinate system.

Details

Assumes that the GK zone is EPSG 31467.

Value

Returns a vector of two values where longitude is first, latitude second.

Author(s)

Carsten Croonenbroeck

See Also

LonLat2GK for the inverse.

Examples

```
LonLat <- GK2LonLat(3702793, 5998319) # Will return c(12.09750, 54.07547).
```

Height *Returns the elevation of a turbine's location.*

Description

For a turbine's location represented by x and y, looks up the elevation from the matrix Elev. Internally transforms coordinates of x and y from problem space (usually unit square) to the matrix space of Elev.

Usage

```
Height(x, y, Elev)
```

Arguments

x	must be a single value containing the 'x' location of a turbine in problem space.
y	must be a single value containing the 'y' location of a turbine in problem space.
Elev	a matrix containing heights. Usually, the fifth element of the list object FarmData will be used as this matrix.

Details

Height is a convenience function for looking up heights as required e.g. for a partial Jensen wake model, independent from the actual size of the area under investigation.

Value

Height returns a single value, the elevation in meters.

Author(s)

Carsten Croonenbroeck

See Also

Profit to see where to use Height. See FarmData for the data set.

Examples

```
## Returns adjusted yield for the given location.
P <- c(0.5868695, 0.9722714)
Height(P[1], P[2], FarmData[[5]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
    e$FarmVars$StartPoint:e$FarmVars$EndPoint])
```

ImposeVectorField *Simple helper function for* PlotResult.

Description

Draws a set of arrows over an existing plot. Will be used internally by PlotResult to visualize the vector field of wind directions over a plot of the wind farm.

Usage

```
ImposeVectorField(xNum = 5, yNum = 5, ColMain = "black",
  ColBand = "white", Frac = 25, DoSDs = TRUE)
```

Arguments

xNum	must be a single value containing the desired number of arrows horizontally.
yNum	must be a single value containing the desired number of arrows vertically.
ColMain	must be a single value containing the desired color for the main wind direction arrow.
ColBand	must be a single value containing the desired color for the secondary arrows representing the standard deviation arrows. Only used if `DoSDs = TRUE`.
Frac	must be a single value containing the desired length information for the arrows. Will be passed to `GetArrow` internally.
DoSDs	must be `TRUE` or `FALSE`. If `TRUE`, will not only draw one arrow for the main wind direction per point, but also two additional arrows (using color `ColBand`) representing the main wind direction plus/minus half the standard deviation.

Details

This function will be used internally by `PlotResult`.
`ImposeVectorField` requires `FarmData` to be present and an existing plot to impose the vector field over.

Value

`ImposeVectorField` returns nothing.

Author(s)

Carsten Croonenbroeck

See Also

Use `PlotResult` to visualize the optimization result.

Examples

```
plot(c(0, 1), c(0, 1))
ImposeVectorField(ColBand = "red")
```

Index2GK *Converts FarmData indices to Gauss-Kruger coordinates.*

Description

This function accepts indices that are within the FarmData space and converts them to a pair of Gauss-Kruger (GK) coordinates.

Usage

```
Index2GK(X, Y)
```

Arguments

X	a numeric value for the 'X' value in the FarmData space.
Y	a numeric value for the 'Y' value in the FarmData space.

Details

Assumes that the GK zone is EPSG 31467. References to indices are valid if the full data set is loaded, see `FarmData` and `AcquireData`.

Value

For index values between 1 and 3,250 for X and between 1 and 4,400 for Y, returns Gauss-Kruger coordinates valid in the EPSG 31467 coordinate system.

Author(s)

Carsten Croonenbroeck

See Also

`GK2Index` for the inverse.

Examples

```
Index2GK(1, 4400) # Will return c(3280000, 5230000), the lower left point.
Index2GK(3250, 1) # Will return c(3929800, 6109800), the top right point.
```

JensenAngle — *For a given distance x, computes the wake cone generated by a turbine.*

Description

In Jensen's wake model, a wake cone is generated by a turbine based on the turbine's hub height z, its rotor radius r_0, the terrain's roughness length z_0 and the distance x between the turbine under investigation and a certain second point. z, r_0 and z_0 are taken from the `FarmVars` settings object, while x is to be provided.

Usage

```
JensenAngle(x)
```

Arguments

`x`	must be a single value. Provide distance in meters.

Details

If a second turbine (B) is in a first turbine (A)'s wake, turbine B's power output must be penalized due to turbulence. This can be performed via Jensen's wake model, but first, it should be checked whether B is actually in A's wake. This function computes the angle of the wake cone generated by turbine A. This can be used together with function `JensenTrapezoid` `PointInPolygon` and to see if B is in the wake of A.

Value

`JensenAngle` returns the (onesided) angle of the wake cone generated by a turbine for distance `x`.

Author(s)

Carsten Croonenbroeck

References

Jensen, N. O. (1983). A note on wind generator interaction. Roskilde: Risø National Laboratory. Risø-M, No. 2411

See Also

`JensenTrapezoid` to check whether there are wake effects present. `FarmVars` for the data object.

Examples

```
JensenAngle(500)
## For a distance of 500 m, the function returns a wake cone angle of 9.2233 degrees
## (given standard values for z, z_0 and r_0 in the FarmVars settings object).
```

`JensenFactor` *For a given distance* `x`*, computes the penalty factor for a turbine's wake.*

Description

In Jensen's wake model, a wake cone is generated by a turbine based on the turbine's hub height z, its rotor radius r_0, the terrain's roughness length z_0 and the distance x between the turbine under investigation and a certain second point. z, r_0 and z_0 are taken from the `FarmVars` settings object, while x is to be provided.

Usage

```
JensenFactor(x)
```

Arguments

`x`	must be a single value. Provide distance in meters.

Details

If a second turbine (B) is in a first turbine (A)'s wake, turbine B's power output must be penalized due to turbulence. This function computes the penalty factor for that wake.

Value

`JensenFactor` returns the a single number between 0 and 1, which can immediately be used a a penalty factor.

Note

Note that for Jensen's multiple wake model, wake penalties from several potentially influencing upwind turbines must be computed. As the factors are between 0 and 1, they can conveniently multiplied by each other afterward. If there are four turbines (A, B, C, D) inside a wind farm, you are investigating turbine D and it turns out that D is in the wake of B and C, but not A, then compute penalty for B (assume 0.7) and C (assume 0.8). Now, the penalty factors for A, B and C are 1, 0.7 and 0.8, so 1 * 0.7 * 0.8 = 0.56 is the penalty for turbine D.

Author(s)

Carsten Croonenbroeck

References

Jensen, N. O. (1983). A note on wind generator interaction. Roskilde: Risø National Laboratory. Risø-M, No. 2411

See Also

Use `JensenAngle` to compute the wake cone and with it, use `JensenTrapezoid` to see if another turbine B is in turbine A's wake. Apply `JensenFactor` only if this is the case.

Examples

```
JensenFactor(500)
## Provided that turbine B is in turbine A's wake and that the two turbines are 500 meters apart,
## turbine B must be penalized by a factor of 0.7952.
```

`JensenTrapezoid` *Computes the four corner points of a Jensen trapezoid (or cone).*

Description

Provided a wind direction, a point and a downwind length, computes the corner points of a Jensen trapezoid (in the literature oftentimes called a 'cone').

Usage

```
JensenTrapezoid(WindDir, Point, x, Margin = TRUE)
```

Arguments

`WindDir`	unique value, a wind direction in degrees.
`Point`	a vector of two containing x and y coordinates of a point. Usually, both will be between 0 and 1.
`x`	downwind distance of the cone in meters.
`Margin`	specified whether to add a small margin to the distance parameter (i.e. internally, compute x * 1.1). Defaults to TRUE.

Value

`JensenTrapezoid` returns a matrix of four columns (the four corner points) and two rows (x and y coordinates).

Author(s)

Carsten Croonenbroeck

References

Jensen, N. O. (1983). A note on wind generator interaction. Roskilde: Riso National Laboratory. Riso-M, No. 2411

See Also

JensenAngle to compute 'cone' angle information, PointInPolygon for a test whether a point is inside a polygon.

Examples

```
MyTrapezoid <- JensenTrapezoid(45, c(0.5, 0.5), 500)
```

LonLat2GK *Converts longitude/latitude coordinates to Gauss-Kruger coordinates.*

Description

This function accepts a pair longitude (decimal system, east) and latitude (decimal system, north) coordinates and converts them to Gauss-Kruger (GK) coordinates by performing re-projection.

Usage

```
LonLat2GK(Lon, Lat)
```

Arguments

`Lon`	longitude value (decimal system, east).
`Lat`	latitude value (decimal system, north).

Details

The resulting values are in GK zone EPSG 31467.

Value

Returns a vector of two values where X is first, Y second.

Author(s)

Carsten Croonenbroeck

See Also

GK2LonLat for the inverse.

Examples

```
LonLat2GK(12.09750, 54.07548) # Will return c(3702793, 5998320).
```

`MosettiTurbineCost` *Returns Mosetti's cost model for a number of turbines.*

Description

This function returns the dimensionless cost based on Mosetti et al. (1994).

Usage

```
MosettiTurbineCost(N)
```

Arguments

`N`	number of turbines for which cost is to be computed.

Details

Wind farm cost does not increase linearly with the number of turbines. Economies of scale lead to disproportionate cost increases. This function returns the dimensionless cost based on Mosetti et al. (1994).

Value

`MosettiTurbineCost` returns a dimensionless cost factor.

Author(s)

Carsten Croonenbroeck

References

G. Mosetti, C. Poloni, B. Diviacco (1994). Optimization of wind turbine positioning in large wind-farms by means of a genetic algorithm, Journal of Wind Engineering and Industrial Aerodynamics, 51(1), 105-116.

See Also

`Cost` for a plug-in cost function for profit.

Examples

```
MosettiTurbineCost(1)
MosettiTurbineCost(20)
```

PairPenalty — *Returns the Jensen wake penalty factor for a pair of turbines.*

Description

As seen from a turbine in the wind farm, computes the wake penalty factor for another turbine in that farm.

Usage

```
PairPenalty(x1, y1, x2, y2, Dirs, SDs)
```

Arguments

`x1`	must be a single value. Provide the x location of the first turbine.
`y1`	must be a single value. Provide the y location of the first turbine.
`x2`	must be a single value. Provide the x location of the second turbine.
`y2`	must be a single value. Provide the y location of the second turbine.
`Dirs`	a matrix containing average yearly wind directions. Usually, the third element of the list object `FarmData` will be used as this matrix.
`SDs`	a matrix containing average yearly wind direction standard deviations. Usually, the fourth element of the list object `FarmData` will be used as this matrix.

Details

First, this function uses `GetAngle` to compute the angle between the two points provided, as seen from point 2's point of view. It then obtains the wind direction at point 2 using `GetDirInfo`. After that, the distance between the two points is computed. With it, the wake cone is computed using `JensenAngle` to check whether point 2 is in point 1's wake using `JensenTrapezoid`. If that is the case, `JensenFactor` is used to compute the penalty factor.
Note that the penalty is the deduction to wind speed. It applies to wind power by its third power, so the user is responsible to take it to its cube himself. `Profit` does that automatically internally.

Value

`PairPenalty` returns a single number between 0 and 1. If point 2 is not in the wake of point 1, the function returns 1.

Author(s)

Carsten Croonenbroeck

See Also

Use `JensenFactor` to see how this function operates. See `FarmVars` for the data object.

Examples

```
Dirs <- FarmData[[3]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
e$FarmVars$StartPoint:e$FarmVars$EndPoint]
SDs <- FarmData[[4]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
e$FarmVars$StartPoint:e$FarmVars$EndPoint]
PairPenalty(0.9, 0.8, 0.6, 0.9, Dirs, SDs)
## Weak wake penalty
PairPenalty(0.1, 0.1, 0.6, 0.9, Dirs, SDs)
## No wake penalty
```

`PartialJensen` *Computes the partial Jensen wake effect.*

Description

If a rotor disc of a first turbine is only partially covered by a wake cone of a second turbine, the penalty must be adjusted accordingly. This function returns partial penalties, if partial wake applies, or full penalty, if not.

Usage

```
PartialJensen(x2, y2, x1, y1, Dirs, SDs, Elev, Z = NULL, DrawTop = FALSE,
DrawFront = FALSE, DrawSide = FALSE)
```

Arguments

x1	must be a single value. Provide the x location of the first turbine.
y1	must be a single value. Provide the y location of the first turbine.
x2	must be a single value. Provide the x location of the second turbine.
y2	must be a single value. Provide the y location of the second turbine.
Dirs	a matrix containing average yearly wind directions. Usually, the third element of the list object FarmData will be used as this matrix.
SDs	a matrix containing average yearly wind direction standard deviations. Usually, the fourth element of the list object FarmData will be used as this matrix.
Elev	a matrix containing elevations. Usually, the fifth element of the list object FarmData will be used as this matrix.
Z	accepts a vector of two representing the heights of the turbines at the two points in meters. If NULL (the default), PartialJensen uses e$FarmVars$z as the heights information for both points. In all cases, the heights are added to the terrain topography information from Elev.
DrawTop	If TRUE, draws a top view of the partial wake situation. Defaults to FALSE. Should not be TRUE at the same time as DrawFront or DrawSide.
DrawFront	If TRUE, draws a front view of the partial wake situation. Defaults to FALSE. Should not be TRUE at the same time as DrawTop or DrawSide.
DrawSide	If TRUE, draws a side view of the partial wake situation. Defaults to FALSE. Should not be TRUE at the same time as DrawTop or DrawFront.

Details

The function first checks whether there is partial coverage. If so, it adjusts the penalty internally given by `PairPenalty` and if not, returns the full penalty. Therefore, the function is imposed over an existing Jensen model and refines it.
Note that the penalty is the deduction to wind speed. It applies to wind power by its third power, so the user is responsible to take it to its cube himself. `Profit` does that automatically internally.

Value

`PartialJensen` returns a penality between 0 and 1.

Author(s)

Carsten Croonenbroeck

References

Frandsen, S. (1992). On the wind speed reduction in the center of large clusters of wind turbines. Journal of Wind Engineering and Industrial Aerodynamics, 39(1-3), pp. 251-265, https://doi.org/10.1016/0167-6105(92)90551-K.

See Also

`JensenTrapezoid` to check whether there are wake effects present. `FarmVars` for the data object. `PairPenalty` for the non-partial wake penalty.

Examples

```
P1 <- c(0.5868695, 0.9722714)
P2 <- c(0.4827957, 0.9552658)

if (exists("FarmData", envir = e, inherits = FALSE))
{
  Dirs <- e$FarmData[[3]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
    e$FarmVars$StartPoint:e$FarmVars$EndPoint]
  SDs <- e$FarmData[[4]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
    e$FarmVars$StartPoint:e$FarmVars$EndPoint]
  Elev <- e$FarmData[[5]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
    e$FarmVars$StartPoint:e$FarmVars$EndPoint]
} else
{
  Dirs <- FarmData[[3]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
    e$FarmVars$StartPoint:e$FarmVars$EndPoint]
  SDs <- FarmData[[4]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
    e$FarmVars$StartPoint:e$FarmVars$EndPoint]
  Elev <- FarmData[[5]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
    e$FarmVars$StartPoint:e$FarmVars$EndPoint]
}

## First, compute the non-partial penalty:
Penal <- PairPenalty(P2[1], P2[2], P1[1], P1[2], Dirs, SDs)
```

```
## Then, correct it for partial coverage:
Penal2 <- PartialJensen(P2[1], P2[2], P1[1], P1[2], Dirs, SDs, Elev)

## Now draw a top view:
Penal2 <- PartialJensen(P2[1], P2[2], P1[1], P1[2], Dirs, SDs, Elev, DrawTop = TRUE)

## Now draw a front view:
Penal2 <- PartialJensen(P2[1], P2[2], P1[1], P1[2], Dirs, SDs, Elev, DrawFront = TRUE)
## Note that the downwind point is somewhat elevated, seems 'right'
## from the upwind point of view, and far away (rotor disc seems smaller).

## Now draw a side view:
Penal2 <- PartialJensen(P2[1], P2[2], P1[1], P1[2], Dirs, SDs, Elev, DrawSide = TRUE)

## For elevation, see
Height(P1[1], P1[2], Elev) # (upwind point)
## and
Height(P2[1], P2[2], Elev) # (downwind point)

## Instead, for the next example, the downwind point is closer
## to the upwind point (larger impression of rotor disc), 'left'
## of it and lower in terms of elevation:
x1 <- 0.5
y1 <- 0.5
x2 <- 0.45
y2 <- 0.478
Penal <- PairPenalty(x2, y2, x1, y1, Dirs, SDs)
Penal2 <- PartialJensen(x2, y2, x1, y1, Dirs, SDs, Elev, DrawTop = TRUE)
Penal2 <- PartialJensen(x2, y2, x1, y1, Dirs, SDs, Elev, DrawFront = TRUE)
## For elevation, see
Height(x1, y1, Elev) # (upwind point)
## and
Height(x2, y2, Elev) # (downwind point)
```

`PlotResult` *Visualizes the wind farm layout optimization result.*

Description

This function draws the adjusted yields of the wind farm under investigation using `image`, superimposes a contour plot using `contour` and arrows for the wind directions using `ImposeVectorField` and then draws the points for the turbines' locations (aligned to their respective raster grid centers).

Usage

```
PlotResult(Result, ImageData = NULL, DoLabels = FALSE, Labels = "IDs")
```

Arguments

`Result`	must be an optimization result as returned by an optimizer such as `optim`. Usually, this will be a `list` at least containing '$par'. Most optimizers comply with this R standard. If your optimizer does not, wrap its result into a list containing an object '$par' containing the optimization result (i.e., a vector of points).
`ImageData`	a matrix containing the data for the background, processed via `image`. If not provided (or NULL), uses the data in `FarmData[[1]]` or, if present, `e$FarmData[[1]]`. Defaults to NULL.
`DoLabels`	a boolean that indicates whether labels should be plotted next to the points. Defaults to FALSE.
`Labels`	a vector of length n = N / 2, can be numeric values or strings. Defines the labels to be shown if `DoLabels` = `TRUE`. Defaults to "IDs", in which case the points are sequently numbered from 1:n. `Labels` is ignored if `DoLabels` = `FALSE`.

Details

For maximum convenience and compatibility with numeric optimizers of most kinds, this function expects nothing but the usual optimization result `list`. This, however, requires that the `FarmData` dataset as well as the `FarmVars` object is present defining additional settings.

Value

`PlotResult` returns nothing.

Author(s)

Carsten Croonenbroeck

See Also

Use `Profit` to obtain an optimization result.

Examples

```
#Will not provide a very good result
NumTurbines <- 4
set.seed(1357)
Result <- optim(par = runif(NumTurbines * 2), fn = Profit,
  method = "L-BFGS-B", lower = rep(0, NumTurbines * 2),
  upper = rep(1, NumTurbines * 2))
Result
PlotResult(Result)
```

PointInPolygon — *Checks whether a point is inside a polygon.*

Description

Uses the famous Jensen test to check whether a provided point is inside a polygon, the latter defined by a set of points.

Usage

```
PointInPolygon(PointMat, TestPoint)
```

Arguments

`PointMat`	a 2 x 4 matrix, four corner points of each x and y coordinates.
`TestPoint`	a vector of two containing x and y coordinates of a point to test.

Value

`PointInPolygon` returns `TRUE` if the point is inside the polygon, or `FALSE`, else.

Author(s)

Carsten Croonenbroeck

References

Jensen, N. O. (1983). A note on wind generator interaction. Roskilde: Risø National Laboratory. Risø-M, No. 2411

See Also

`JensenTrapezoid` to compute a Jensen trapezoid.

Examples

```
set.seed(1357)
Angle <- runif(n = 1, min = 0, max = 360)
Point <- runif(n = 2, min = 0, max = 1)
Dist <- rnorm(n = 1, mean = 200, sd = 200)

MyTrapezoid <- JensenTrapezoid(Angle, Point, Dist)

plot(x = MyTrapezoid[1, ], y = MyTrapezoid[2, ], xlab = "", ylab = "")
lines(x = c(MyTrapezoid[1, 1], MyTrapezoid[1, 2]), y = c(MyTrapezoid[2, 1], MyTrapezoid[2, 2]))
lines(x = c(MyTrapezoid[1, 2], MyTrapezoid[1, 3]), y = c(MyTrapezoid[2, 2], MyTrapezoid[2, 3]))
lines(x = c(MyTrapezoid[1, 3], MyTrapezoid[1, 4]), y = c(MyTrapezoid[2, 3], MyTrapezoid[2, 4]))
lines(x = c(MyTrapezoid[1, 4], MyTrapezoid[1, 1]), y = c(MyTrapezoid[2, 4], MyTrapezoid[2, 1]))
```

```
NumTest <- 50

xTest <- runif(n = NumTest, min = min(MyTrapezoid[1, ]), max = max(MyTrapezoid[1, ]))
yTest <- runif(n = NumTest, min = min(MyTrapezoid[2, ]), max = max(MyTrapezoid[2, ]))

for (i in 1:NumTest)
{
 ThisPoint <- c(xTest[i], yTest[i])
 if (PointInPolygon(MyTrapezoid, ThisPoint))
 {
  points(ThisPoint[1], ThisPoint[2], pch = 16, col = "green")
 } else
 {
  points(ThisPoint[1], ThisPoint[2], pch = 16, col = "red")
 }
}
```

Profit — *Computes the economic profit for a given wind farm layout configuration*

Description

This is the bread-and-butter function to this package. It takes a set of points (turbine locations) and based on the adjusted yields in the field specified via the FarmVars settings object, checks whether the layout is valid, computes the Jensen penalties, generates the total marketable power production of this layout, takes cost and sale price into consideration and finally computes the farm's economic profit. Note, however, that since most numeric optimizers by default operate as a minimizer, `Profit` returns the negative profit, i.e. a negative number for positive profit values and a positive number if cost is greater than revenue.

Usage

```
Profit(X)
```

Arguments

`X`	must be a numeric vector containing an even number of values, at least two. If the length of the vector is N, then n = N / 2 is the number of points. The first values 1,...,n are interpreted as x coordinates and the subsequent n + 1,...,N values are the y coordinates.

Details

For maximum convenience and compatibility with numeric optimizers of most kinds, this function expects nothing but the problem vector (the set of points) as a parameter. This, however, requires that the FarmData dataset as well as the FarmVars object is present defining additional settings.
As `Profit` returns the negative profit for compatibility with minimizers, reverse the sign for actual profit values.

Profit requires the optimizer to accept box constraints, as Profit can not compute values outside the wind farm boundary (the function's domain). If your optimizer does not accept box constraints, embed Profit into a wrapper function that returns the sum of costs if it least one point is outside the boundary.

Value

Profit returns a single number. The result is the negative profit for any valid setting of points and consequently, the sum of all costs for invalid settings.

Author(s)

Carsten Croonenbroeck

See Also

Calls Yield and Cost internally. Use PlotResult to visualize the optimization result.

Examples

```
#Will not provide a very good result
NumTurbines <- 4
set.seed(1357)
Result <- optim(par = runif(NumTurbines * 2), fn = Profit,
  method = "L-BFGS-B", lower = rep(0, NumTurbines * 2),
  upper = rep(1, NumTurbines * 2))
Result
PlotResult(Result)
##########################################
#Will provide a somewhat better result
#Necessary to install pso
## Not run:
NumTurbines <- 4
Result <- pso::psoptim(par = runif(NumTurbines * 2), fn = Profit,
  lower = rep(0, NumTurbines * 2), upper = rep(1, NumTurbines * 2))
Result
PlotResult(Result)

## End(Not run)
##########################################
#Simple wrapper function for optimizers not accepting box constraints
NumTurbines <- 4
lower <- rep(0, NumTurbines * 2)
upper <- rep(1, NumTurbines * 2)
Wrapper <- function(X)
{
 xSel <- seq(from = 1, to = length(X) - 1, by = 2)
 x <- X[xSel]
 y <- X[xSel + 1]

 if (any(x < lower) | any(x > upper) | any(y < lower) | any(y > upper))
 {
  return(sum(rep(e$FarmVars$UnitCost, length(x))))
 }

 return(Profit(X))
}
## Not run:
set.seed(1357)
Result <- optim(par = runif(NumTurbines * 2), fn = Wrapper, method = "SANN")
Result
PlotResult(Result)
## End(Not run)
```

ProfitContributors *Computes profit contributions for all points in a setup solution.*

Description

This function takes an optimizer result (or simple vector object) and computes the profit contribution for each point.

Usage

```
ProfitContributors(Result)
```

Arguments

`Result`	must be an optimization result as returned by an optimizer such as `optim` or a vector. Usually, this will be a `list` at least containing '$par'.

Details

Neglecting a possibly invalid setup (caused by at least one point), computes the profit contributions. If the setup provided is valid, the sum of contributions is identical to the (absulute value of the) returned value of `Profit`.

Value

`ProfitContributors` returns a matrix of two columns and n rows, whith n being the number of turbines. The first column is a sequence of 1:n, representing the turbine IDs, while the second column contains the actual profit contributions.

Author(s)

Carsten Croonenbroeck

See Also

See `ValidSetup` to see how a setup is categorized as valid or not. Use `PlotResult` to visualize the optimization result.

Examples

```
#Prints a result and uses the profit contributions as labels.
NumTurbines <- 4
set.seed(1235)
Result <- optim(par = runif(NumTurbines * 2), fn = Profit,
  method = "L-BFGS-B", lower = rep(0, NumTurbines * 2),
  upper = rep(1, NumTurbines * 2))
MyLabels <- ProfitContributors(Result)
MyLabels
#PlotResult(Result, DoLabels = TRUE, Labels = MyLabels[, 2])

# Given a valid setup, this should be TRUE.
#identical(abs(Profit(Result$par)), sum(MyLabels[, 2]))
```

`QuickGauss3D` *For an incoming wind speed at reference height, this function computes a 3D Gaussian model based wind speed.*

Description

This function computes Gaussian wind speeds.

Usage

```
QuickGauss3D(x, y, z, u = 8, refHeight = 10)
```

Arguments

`x`	must be a single value. Provide distance in meters.
`y`	must be a single value. Provide distance in meters.
`z`	must be a single value. Provide distance in meters.
`u`	must be a single value. Provide wind speed in meters per second.
`refHeight`	must be a single value. Provide reference height (the height at which wind speed u was measured) in meters.

Details

The Gaussian wake model is loosely based on the initial contribution by Bastankhah & Porte-Agel (2014).

Value

`QuickGauss3D` returns a single number which can be considered a wind speed in the wake of a turbine at location x, y, and z.

Note

Note that the model assumes that along the x axis, x = 0 is the turbine location. x expands along the wind direction downwind. y denoted whether a point is 'left' or 'right' the x axis. Thus, the x-z plane is the plane along the x axis and perpendicular to the ground. The z axis is hight, starting at 0 = ground level.

Author(s)

Carsten Croonenbroeck

References

Bastankhah, M., & Porte-Agel, F. (2014). A new analytical model for wind-turbine wakes. Renewable Energy, 70, 116-123.

See Also

Use `GenerateGauss` to compute the three-dimensional tensor array object containing the wind speed data. Use `GaussWS` for a convenience function to look-up the values from the returned array.

Examples

```
QuickGauss3D(100, 1, 100)
QuickGauss3D(200, -40, 120)
QuickGauss3D(50, 40, 70)
```

`ShowWakePenalizers` *Visualizes the points causing/'suffering' from wake effects.*

Description

After optimization, this function draws a reduced 'field' and shows which points cause wake penalties on which points. 'Causers' are displayed in grey, 'sufferers', i.e. points that are in (possibly multiple) wakes of (several) causers are displayed in gold.

Usage

```
ShowWakePenalizers(Result, Cones = TRUE, All = FALSE, VectorField = TRUE,
ImposeMST = FALSE, NS = FALSE, NSOnly = FALSE, Exaggerate = TRUE,
DoubleLine = TRUE, Frames = 100, Alpha = 0.5, MaxContrast = TRUE, Soften = TRUE)
```

Arguments

`Result`	must be an optimization result as returned by an optimizer such as `optim`. Usually, this will be a `list` at least containing '$par'. Most optimizers comply with this R standard. If your optimizer does not, wrap its result into a list containing an object '$par' containing the optimization result (i.e., a vector of points).
`Cones`	triggers whether the Jensen cones are to be displayed as well. Defaults to `TRUE`.
`All`	triggers whether all Jensen cones are to be displayed, instead of just causers and sufferers. Defaults to `FALSE`, as `TRUE` results in a mess most of the times.
`VectorField`	controls whether or not to impose the wind directions vector field (arrows). Defaults to `TRUE`.
`ImposeMST`	if `TRUE`, computes and imposes a minimum spanning tree for the vertices, useful for cable installation purposes. Defaults to `FALSE`.
`NS`	if `TRUE`, imposes a computationally intensive Navier-Stokes based wake simulation image over the plot. Defaults to `FALSE`.
`NSOnly`	if `TRUE`, imposes just the Navier-Stokes wake simulation (no Jensen wakes). Defaults to `FALSE`.
`Exaggerate`	if `TRUE`, displays turbines as squares in the Navier-Stokes visualization. The turbine is then still to scale, but in reality the rotor is a disc instead of a cube, of course. This overly emphasizes wakes, which may be desired in most cases. Thus, defaults to `TRUE`. Set to `FALSE` for a more realistic look of the rotor discs, but this will make the wakes appear less severe.
`DoubleLine`	applies only if `Exaggerate` == `TRUE`. The rotor disc is then a disc instead of a cube (or line instead of square), but the line will be two instead of one pixel wide. This option is a trade-off between `Exaggerate` == `TRUE` and `Exaggerate` == `FALSE`. Set to `TRUE` (the default) if `Exaggerate` == `FALSE` to apply. Note that only if `Exaggerate` == `FALSE` and `DoubleLine` == `FALSE`, the rotor disc is a one pixel wide line, the most realistic setting which however shows only weak wakes.
`Frames`	number of frames to pre-compute for the Navier-Stokes simulation. 100 is a good value. Less makes the computation faster, but more inaccurate. More than 100 in only necessary for very low wind speeds. Thus, defaults to `100`.
`Alpha`	The Navier-Stokes wake simulation is imposed semi-transparent over the usual image. This controls the transparency. Alpha = 0 means transparent, Alpha = 1 means opaque. Defaults to `0.5`.
`MaxContrast`	increases the contrast to maximum (this is equivalent to histogram stretching). Defaults to `TRUE`.
`Soften`	applies a Gaussian soften filter to the Navier-Stokes image to have a more homogeneous visualization. Defaults to `TRUE`.

Details

Enables the researcher to inspect the optimization result in an a-posteriori analysis of Jensen's wake model. Parts of the Navier-Stokes code are inspired by Stam (2003).

Value

ShowWakePenalizers invisibly returns a square matrix (dimension: 'number of turbines') of wake penalty pairs: Columns are sufferers, rows are causers. For each row/column pair unequal to one, the turbine in row i sheds wake on the turbine in column j. The actual numbers are those returned by JensenFactor.

Author(s)

Carsten Croonenbroeck

References

Stam, S. (2003). Real-Time Fluid Dynamics for Games. https://www.researchgate.net/publication/2560062_Real-Time_Fluid_Dynamics_for_Games, 1-17

See Also

Use PlotResult to visualize the optimization result.

Examples

```
## Not run:
#Will show that turbine 1 sheds wake on turbine 5.
NumTurbines <- 8
set.seed(1357)
Result <- optim(par = runif(NumTurbines * 2), fn = Profit,
  method = "L-BFGS-B", lower = rep(0, NumTurbines * 2),
  upper = rep(1, NumTurbines * 2))
ShowWakePenalizers(Result)

###############

#This may take some time. A progress bar is displayed.
Result = list(par = e$FarmVars$BenchmarkSolution)
ShowWakePenalizers(Result, All = TRUE, NS = TRUE, VectorField = FALSE, Alpha = 0.7)

###############

#Generates a few frames of an animation.
Result <- list(par = e$FarmVars$BenchmarkSolution)
MakeFrames <- function(Frames)
{
 for (i in 100:(100 + Frames - 1))
 {
  bmp(file = paste(i, ".bmp", sep = ""))
  ShowWakePenalizers(Result, All = TRUE, NS = TRUE, Frames = i)
  dev.off()
 }
}
system.time(MakeFrames(30)) #May take an hour.
```

```
## End(Not run)
```

`SwitchProfile` *Helper function for ex-post wind direction dependence analysis*

Description

This function helps with analzing an already computed turbine setup. As `Profit` (together with an optimizer) finds an optimal setup given actual wind speeds in that area, the question arises whether that setup is 'robust' against changing wind directions. If for a given setup, there is no or only little wake dependence (analyze e.g. with `ShowWakePenalizers`), this may change if wind direction changes. Given a different wind direction, a situation may come up at which there is an upwind turbine and a local set of, say, two, three, or even more downwind turbines. As these downwind turbines are in the wake of the upwind turbines ('receiving' only reduced wind speeds), it may be beneficial to shut down the upwind turbine in order to have the downwind turbine receive full wind, thus possibly generating greater profit.

`SwitchProfile` expects a 'bit' pattern of turbines (so-called 'profiles') at which '1' means that a turbine is to be used, while '0' means, the turbine is to be shut down. For a total of four tubines in a setup, profile `c(1, 1, 1, 1)` means that all turbines are used, while profile `c(1, 0, 0, 1)` means that turbines 1 and 4 are used, turbines 2 and 3 are to be shut down. If a wind farm is 'robust' against wind direction changes, it should return profiles of all ones for all wind directions.

In the examples we show a short code sample that uses `SwitchProfile` to loop through 360 integer wind direction degrees and optimizes actual profiles. It is up to the researcher to use alternative optimizers in order to find (possibly) better profiles.

Usage

```
SwitchProfile(On, Result)
```

Arguments

`On`	must be a numeric vector of length `n`, where `n` is the number of turbines in a setup.
`Result`	the actual setup as an optimizer result. Must be a list that contains the turbine locations is a slot "$par". For example, use `Result = list(par = e$FarmVars$BenchmarkSolution)`, see `FarmVars` for details.

Details

For compatibility with non-integer solution optimizers, `SwitchProfile` expects not only values equal to 0 or 1, but will assume that values below 0.5 are meant to be 0, and values >= 0.5 are coerced to 1.

Value

`SwitchProfile` returns a single number. The result is the negative profit for the profile given. The result can be optimized in order to find the maximum profit profile, which should be identical to a profile of all ones if the setup is robust against wind direction changes.

Author(s)

Carsten Croonenbroeck

See Also

Use `Profit` for the target function used internally.

Examples

```
Result <- list(par = e$FarmVars$BenchmarkSolution)

Profile1 <- rep(1, times = 20)

#Will be identical to Profit(Result$par):
SwitchProfile(Profile1, Result)

Profile2 <- Profile1
Profile2[8] <- 0 #Disable turbine 8.

#Returns farm profit if turbine 8 is off:
SwitchProfile(Profile2, Result)

##########################################

#Warning, this will overwrite e$FarmData. If not dispensable,
#backup before use.
#Computation may be slow.

## Not run:
Result <- list(par = e$FarmVars$BenchmarkSolution)
e$FarmData <- FarmData

n <- length(e$FarmVars$BenchmarkSolution) / 2

Profiles <- matrix(ncol = n, nrow = 360)
Dom <- cbind(rep(0, n), rep(1, n))

for (j in 1:360)
{
 e$FarmData$WindDirection <- matrix(data = j, ncol = e$FarmVars$Width,
   nrow = e$FarmVars$Width)
 message(paste("Processing wind direction: ", j, " degrees.", sep = ""))
 flush.console()

 Result <- rgenoud::genoud(fn = SwitchProfile, nvars = n, starting.values = runif(n),
   Domains = Dom, boundary.enforcement = 2, MemoryMatrix = FALSE, print.level = 0,
    Result = Result)
  On <- Result$par
  On[On < 0.5] <- 0
  On[On >= 0.5] <- 1
  Profiles[j, ] <- On
}

#Now 'Profiles' is a matrix of 360 rows (degrees) and n columns.
#If the setup is robust against wind direction changes, all values
#in the columns should be ones. If there are zeros left, double
#check these profiles (compare to full turbines setup) manually,
#e.g. using

for (i in 1:nrow(Profiles))
{
 if (any(Profiles[i, ] == 0))
 {
  e$FarmData$WindDirection <- matrix(data = i,
    ncol = e$FarmVars$Width, nrow = e$FarmVars$Width)

  print(paste("Is unrestricted profit (",
    abs(Profit(e$FarmVars$BenchmarkSolution)), ") still
    greater than 'profile profit' (",
    abs(SwitchProfile(Profiles[i, ], Result)), ")? ",
    abs(Profit(e$FarmVars$BenchmarkSolution)) >=
    abs(SwitchProfile(Profiles[i, ], Result)), ".", sep = ""))
 }
}

#Is absolute profile profit actually greater than the absolute value
#function Profit() returns? If yes, then it is beneficial to turn off
#turbines (use profiles) if wind comes from the given angle.

## End(Not run)

##########################################

#As an alternative to using an optimizer to find the optimal profile,
#all possible profiles can be 'looped through' deterministically using
#the following few lines of code.

#Warning, this will overwrite e$FarmData. If not dispensable,
#backup before use.

#Be advised, computation may be very slow, even if run on modern
#machines and even though using parallelization.

## Not run:
Result <- list(par = e$FarmVars$BenchmarkSolution)
e$FarmData <- FarmData

ComputeProfile <- function(i)
{
 e$FarmData$WindDirection <- matrix(data = i, ncol = e$FarmVars$Width,
   nrow = e$FarmVars$Width)

 BestProfit <- n * e$FarmVars$UnitCost
 BestProfile <- rep(0, n)
 for (j in 0:MaxNum)
 {
  ThisProfile <- as.numeric(rev(intToBits(j)))[(32 - (n - 1)):32]
  if (length(which(ThisProfile == 1)) > 1)
  {
   ThisProfit <- SwitchProfile(ThisProfile, Result)
  } else
  {
   ThisProfit <- n * e$FarmVars$UnitCost
  }

  if (ThisProfit < BestProfit)
  {
   BestProfit <- ThisProfit
   BestProfile <- ThisProfile
  }
 }
 return(BestProfile)
}

n <- length(e$FarmVars$BenchmarkSolution) / 2
MaxNum <- (2 ^ n) - 1

NumCores <- parallel::detectCores() - 1
cl <- snow::makeCluster(NumCores)

ParallelProfile <- new.env()
ParallelProfile$FarmVars <- e$FarmVars
ParallelProfile$FarmData <- e$FarmData

snow::clusterExport(cl, list = ls())
snow::clusterExport(cl, list = ls(envir = as.environment("package:wflo")))

doSNOW::registerDoSNOW(cl)

iterations <- 360
pb <- txtProgressBar(max = iterations, style = 3)
progress <- function(n) setTxtProgressBar(pb, n)
opts <- list(progress = progress)
`%dopar%` <- foreach::`%dopar%`
AllProfiles <- foreach::foreach(i = 1:iterations,
  .options.snow = opts) %dopar% ComputeProfile(i)

close(pb)
snow::stopCluster(cl)

## End(Not run)
```

```
#Afterward, check the resulting list of profiles for remaining zeros as above.

##########################################

#Thinking the thought of wind direction dependent layouts further,
#Profit() can be used to optimize a farm setup that takes all wind
#direction wake patterns into account during the optimization
#stage already.

#Warning, this will overwrite \code{e$FarmData}. If not dispensable,
#backup before use.
#Computation may be slow.

## Not run:
e$FarmData <- FarmData

SumProfit <- function(X)
{
 MySum <- as.numeric()
 for (i in 1:360)
 {
  e$FarmData$WindDirection <- matrix(data = i, ncol = e$FarmVars$Width,
    nrow = e$FarmVars$Width)
  MySum[i] <- Profit(X)
 }
 MySum <- sum(MySum)
 return(MySum)
}

n <- 20
Dom <- cbind(rep(0, 2 * n), rep(1, 2 * n))
Result <- rgenoud::genoud(fn = SumProfit, nvars = 2 * n, starting.values = runif(2 * n),
  Domains = Dom, boundary.enforcement = 2, MemoryMatrix = FALSE, print.level = 1)

## End(Not run)
```

`ValidSetup` *Checks whether all turbine locations provided satisfy the minimum distance criterion.*

Description

For a set of turbine locations represented by `x` and `y`, checks whether all possible pairs satisfy the minimum distance criterion.

Usage

```
ValidSetup(x, y)
```

Arguments

x	must be a numeric vector of at least two values, contains the 'x' location(s) of turbines.
y	must be a numeric vector of at least two values, contains the 'y' location(s) of turbines.

Value

`ValidSetup` returns `TRUE` if all pairs of turbines are at least as far away from each other as 'MinDist' from the `FarmVars` settings object requests, or `FALSE`, else.

Author(s)

Carsten Croonenbroeck

Examples

```
ValidSetup(c(0.5, 0.7), c(0.7, 0.9))
## Returns TRUE if FarmVars$MinDist is at its standard value (0.1).
```

`wflo` *Data set and functions for wind farm layout optimization.*

Description

This package makes two contributions to the Wind Farm Layout Optimization (WFLO) research branch: First, it provides a convenient and realistic data set of high resolution and accuracy encompassing wind speeds, wind directions, standard deviations of wind directions, and (adjusted) yields for the entire area of Germany. Second, it supplies a set of helper functions and a benchmark function for economically (profit) driven wind farm layout optimization. This enables researchers in the field of the np-hard problem of wflo to focus on their optimization methodology contribution and also provides a realistic benchmark setting for comparability among contributions.

Author(s)

Carsten Croonenbroeck
David Hennecke

`WindspeedHellmann` *Scales wind speeds to certain heights.*

Description

For a wind speed at given height, returns a scaled wind speed at some different height. Often used to obtain wind speed at hub height.

Usage

```
WindspeedHellmann(v0, HH = 100, refHeight = 10, alpha = 1 / 7)
```

Arguments

`v0`	wind speed (in meters per second) at reference height.
`HH`	the height (in meters) at which wind speed is desired.
`refHeight`	reference height (in meters). The height at which the actual wind speed (v0) was measured.
`alpha`	power law parameter. Usually set to 1/7 for onshore sites.

Details

This function simply implements

$$v = v_0(\frac{HH}{refHeight})^\alpha$$

Value

`WindspeedHellmann` returns a wind speed (in meters per second) at the desired height.

Author(s)

Carsten Croonenbroeck

See Also

`WindspeedLog` for a different way to scale wind speeds to heights.

Examples

```
WindspeedLog(v0 = 6, HH = 80, z0 = 0.1, refHeight = 20)
WindspeedHellmann(v0 = 6, HH = 80, refHeight = 20)
```

`WindspeedLog` *Scales wind speeds to certain heights.*

Description

For a wind speed at given height, returns a scaled wind speed at some different height. Often used to obtain wind speed at hub height.

Usage

```
WindspeedLog(v0, HH = 100, z0 = 0.1, refHeight = 10)
```

Arguments

`v0`	wind speed (in meters per second) at reference height.
`HH`	the height (in meters) at which wind speed is desired.
`z0`	roughness length (in meters). Usually set to 0.1 m for onshore sites.
`refHeight`	reference height (in meters). The height at which the actual wind speed (v0) was measured.

Details

This function simply implements

$$v = v_0 (\frac{\log \frac{HH}{z_0}}{\log \frac{refHeight}{z_0}})$$

Note that this way to scale wind speeds to certain heights is frequently considered deprecated in the literature. Use `WindspeedHellmann` instead.

Value

`WindspeedLog` returns a wind speed (in meters per second) at the desired height.

Author(s)

Carsten Croonenbroeck

See Also

`WindspeedHellmann` for a different way to scale wind speeds to heights.

Examples

```
WindspeedLog(v0 = 6, HH = 80, z0 = 0.1, refHeight = 20)
WindspeedHellmann(v0 = 6, HH = 80, refHeight = 20)
```

`Yield` *Returns yearly yield for a turbine's location.*

Description

For a turbine's location represented by `x` and `y`, looks up the (adjusted) yield from the matrix `Adj`. Internally transforms coordinates of `x` and `y` from problem space (usually unit square) to the matrix space of `Adj`.

Usage

```
Yield(x, y, Adj)
```

Arguments

`x`	must be a single value containing the 'x' location of a turbine in problem space.
`y`	must be a single value containing the 'y' location of a turbine in problem space.
`Adj`	a matrix containing adjusted yields. Usually, the first element of the list object `FarmData` will be used as this matrix.

Details

Adjusted yields are the projected yearly average yields dependent on wind speed, hub height and other settings at each point in the raster data. Annual Energy Production (AEP) at a specific location, weighted by a location quality correction factor, produces adjusted yields. This adjustment returns a better guess on the marketable yield at a specific point. For details on the data, see the data set description to this package.
Note that `Profit` internally multiplies the outcome of `Yield` by `e$FarmVars$Price` to obtain revenue. Users who replace the function by `e$Yield` need to provide that manually, if revenue is desired.

Value

`Yield` returns a single value.

Author(s)

Carsten Croonenbroeck

See Also

`Profit` to see where to use `Yield`, `Cost` for a similar function for yearly cost. `FarmData` for the data set.

Examples

```
## Returns adjusted yield for the given location.
Adj <- FarmData[[1]][e$FarmVars$StartPoint:e$FarmVars$EndPoint,
e$FarmVars$StartPoint:e$FarmVars$EndPoint]
Yield(0.5, 0.7, Adj)

## Replace the function by another function
## also called 'Yield', embedded in environment e.
## Also, see the vignette.
## Not run:
e$Yield <- function(x, y, AEP) #x, y \in R
{
 return(x + y)
}
set.seed(1357)
NumTurbines <- 4 # For example.
Result <- pso::psoptim(par = runif(NumTurbines * 2), fn = Profit,
  lower = rep(0, NumTurbines * 2), upper = rep(1, NumTurbines * 2))
Result
rm(Yield, envir = e)

## End(Not run)
```

Index