# Package 'windfarmGA'

September 14, 2026

**Title** Genetic Algorithm for Wind Farm Layout Optimization

**Version** 5.0.0

**Maintainer** Sebastian Gatscha <sebastian_gatscha@gmx.at>

**Description** The genetic algorithm is designed to optimize wind farms of any shape. Each layout is encoded as n unique grid-cell identifiers. It requires a predefined amount of turbines, a unified rotor radius and an average wind speed value for each incoming wind direction. A terrain effect model can be included that downloads an 'SRTM' elevation model and loads a Corine Land Cover raster to approximate surface roughness.

**Depends** R (>= 4.1.0)

**Imports** Rcpp, terra, sf, RColorBrewer, calibrate, grDevices, graphics, magrittr, methods, stats, utils

**LinkingTo** Rcpp

**LazyData** TRUE

**License** MIT + file LICENSE

**URL** https://ysosirius.github.io/windfarmGA/index.html

**BugReports** https://github.com/YsoSirius/windfarmGA/issues

**Encoding** UTF-8

**RoxygenNote** 7.3.2

**Suggests** testthat (>= 3.0.0), foreach, parallel, doParallel, progress, stars, raster, leaflet, elevatr (>= 0.99.0), ggplot2, plotly, shiny, gstat, rworldmap

**X-schema.org-keywords** windfarm-layout, optimization, genetic-algorithm, renewable-energy, r, rstats, r-package

**Config/testthat/edition** 3

**Config/testthat/parallel** true

**NeedsCompilation** yes

**Author** Sebastian Gatscha [aut, cre, cph]

**Repository** CRAN

**Date/Publication** 2026-09-14 06:40:02 UTC

# Contents

---

`windfarmGA-package` *windfarmGA: Genetic Algorithm for Wind Farm Layout Optimization*

---

## Description

The genetic algorithm is designed to optimize wind farms of any shape. Each layout is encoded as n unique grid-cell identifiers. It requires a predefined amount of turbines, a unified rotor radius and an average wind speed value for each incoming wind direction. A terrain effect model can be included that downloads an 'SRTM' elevation model and loads a Corine Land Cover raster to approximate surface roughness.

## Details


A package to optimize small wind farms with irregular shapes using a genetic algorithm. Each individual is `n` unique grid-cell IDs (set-crossover, swap-mutation). It requires a fixed amount of turbines, a fixed rotor radius and an average wind speed value for each incoming wind direction. A terrain effect model can be included which downloads a digital elevation model and a Corine Land Cover raster to approximate surface roughness. Further information can be found at the description of the function `genetic_algorithm`.

## Author(s)

**Maintainer**: Sebastian Gatscha `<sebastian_gatscha@gmx.at>`

## See Also

Useful links:

- Documentation Github.io
- Documentation
- Master Thesis
- Shiny App
- Report Issues

## as_windfarmGA *Mark a genetic_algorithm result*

### Description

Attach the windfarmGA class so print() and plot() dispatch. New runs from genetic_algorithm() already have this class. print() shows run inputs, the wind table, and each generation that set a new fitness maximum.

### Usage

```
as_windfarmGA(x)

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

## S3 method for class 'windfarmGA'
plot(x, y, ...)
```

### Arguments

| | |
|---|---|
| x | A result matrix from genetic_algorithm(). |
| ... | Passed to plot_windfarmGA(). |
| y | The site polygon (same as area in genetic_algorithm()). |

### Value

x with class windfarmGA.

## barometric_height *Calculates Air Density, Air Pressure and Temperature according to the Barometric Height Formula*

### Description

Calculates air density, temperature and air pressure respective to certain heights according to the International standard atmosphere and the barometric height formula.

### Usage

```
barometric_height(data, height, po = 101325, ro = 1.225)
```

### Arguments

| | |
|---|---|
| data | A data.frame containing the height values |
| height | Column name of the height values |
| po | Standardized air pressure at sea level (101325 Pa) |
| ro | Standardized air density at sea level (1,225 kg per m3) |

### Value

Returns a data.frame with height values and corresponding air pressures, air densities and temperatures in Kelvin and Celsius.

### See Also

Other Wind Energy Calculation Functions: calculate_energy(), circle_intersection(), get_dist_angles(), turbine_influences()

### Examples

```
data <- matrix(seq(0, 5000, 500))
barometric_height(data)
plot.ts(barometric_height(data))
```

---

## big_shape *A POLYGON with an area of ~70 km2*

---

### Description

A POLYGON with an area of ~70 km2

### Usage

```
big_shape
```

### Format

An object of class sf (inherits from data.frame) with 1 rows and 1 columns.

`calculate_energy` *Calculate Energy Outputs of Individuals*

## Description

Calculate the energy output and efficiency rates of an individual in the current population under all given wind directions and speeds. If the terrain effect model is activated, the main calculations to model those effects will be done in this function.

## Usage

```
calculate_energy(
  layout,
  reference_height,
  rotor_height,
  surface_roughness,
  wake_angle,
  wake_distance,
  area,
  rotor,
  wind,
  elevation = NULL,
  terrain = FALSE,
  ccl_raster = NULL,
  weibull = FALSE,
  park_center = NULL,
  plot = FALSE
)
```

## Arguments

| | |
|---|---|
| `layout` | One individual: matrix with X/Y (and typically cell IDs). |
| `reference_height` | Height at which `wind$ws` was measured. |
| `rotor_height` | Hub height in metres. |
| `surface_roughness` | Roughness length in metres. Per-cell when `terrain` is on. |
| `wake_angle` | Angle (degrees) beyond which wake influence is ignored. |
| `wake_distance` | Distance (metres) beyond which wake effects are ignored. |
| `area` | Site polygon (`sf`, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| `rotor` | Rotor radius in metres. |
| `wind` | Wind data.frame with `ws`, `wd` and optional `probab`. See `windata_format()`. |
| `elevation` | Terrain list from `terrain_model()`. Unused when `terrain` is `FALSE`. |

| | |
|---|---|
| terrain | Terrain model (elevation + land cover). TRUE downloads a DEM via elevatr. Pass a DEM raster to skip the download. Per-cell values are computed once and stored in the result as terrainModel for plot_result() / random_search(). |
| ccl_raster | Land-cover roughness raster from terrain_model(). |
| weibull | If TRUE, hub-height speed comes from Weibull rasters; wind$ws is ignored. |
| park_center | Optional numeric of length 2 (x, y) used as rotation origin. Computed from the polygon bounding box when missing. |
| plot | If TRUE, the process will be plotted. |

## Value

Returns a list of an individual of the current generation with resulting wake effects, energy outputs, efficiency rates for every wind direction. The length of the list corresponds to the number of different wind directions.

## See Also

Other Wind Energy Calculation Functions: barometric_height(), circle_intersection(), get_dist_angles(), turbine_influences()

## Examples

```
## Create a random Polygon
library(sf)
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4498482, 4498482, 4499991, 4499991, 4498482),
    c(2668272, 2669343, 2669343, 2668272, 2668272)
  ))),
  crs = 3035
))

## Create a uniform and unidirectional wind data.frame and plot the
## resulting wind rose
data.in <- data.frame(ws = 12, wd = 0)
windrosePlot <- plot_windrose(
  data = data.in, spd = data.in$ws,
  dir = data.in$wd, dirres = 10, spdmax = 20
)

## Assign the rotor radius and a factor of the radius for grid spacing.
rotor <- 50
fcr <- 3
resGrid <- grid_area(
  area = area, size = rotor * fcr, prop = 1,
  plot_grid = TRUE
)
## Assign the indexed data frame to new variable. Element 2 of the list
## is the grid, saved as Simple Feature Polygons.
resGrid1 <- resGrid[[1]]
```

```
## Create an initial population with the indexed Grid, 15 turbines and
## 100 individuals.
initpop <- init_population(grid = resGrid1, n = 15, n_start = 100)

## Calculate the expected energy output of the first individual of the
## population.
par(mfrow = c(1, 2))
plot(area)
points(initpop[[1]][, "X"], initpop[[1]][, "Y"], pch = 20, cex = 2)
plot(resGrid[[2]], add = TRUE)
resCalcEn <- calculate_energy(
  layout = initpop[[1]], reference_height = 50,
  rotor_height = 50, surface_roughness = 0.14, wake_angle = 20,
  wake_distance = 100000, wind = data.in,
  rotor = 50, area = area, terrain = FALSE,
  weibull = FALSE
)
resCalcEn <- as.data.frame(resCalcEn)
plot(area, main = resCalcEn[, "Energy_Output_Red"][[1]])
points(x = resCalcEn[, "Bx"], y = resCalcEn[, "By"], pch = 20)


## Create a variable and multidirectional wind data.frame and plot the
## resulting wind rose
data.in10 <- data.frame(ws = runif(10, 1, 25), wd = runif(10, 0, 360))
windrosePlot <- plot_windrose(
  data = data.in10, spd = data.in10$ws,
  dir = data.in10$wd, dirres = 10, spdmax = 20
)

## Calculate the energy outputs for the first individual with more than one
## wind direction.
resCalcEn <- calculate_energy(
  layout = initpop[[1]], reference_height = 50,
  rotor_height = 50, surface_roughness = 0.14, wake_angle = 20,
  wake_distance = 100000, wind = data.in10,
  rotor = 50, area = area, terrain = FALSE,
  weibull = FALSE
)
```

---

`circle_intersection` *Get area of intersecting circles*

---

## Description

Calculate the intersection area of two circles with different radii and different heights

### Usage

```
circle_intersection(r1, r2, h1, h2, dx)
```

### Arguments

| | |
|---|---|
| r1 | The radius of circle 1 |
| r2 | The radius of circle 2 |
| h1 | The height of the circle center 1 |
| h2 | The height of the circle center 2 |
| dx | The distance on the x-axis between both centers |

### Value

A numeric vector; one intersection area per pair. Scalars stay length 1.

### See Also

Other Wind Energy Calculation Functions: barometric_height(), calculate_energy(), get_dist_angles(), turbine_influences()

---

## crossover *Crossover Method*

---

### Description

Legacy binary crossover. The GA loop uses set_crossover on unique grid-cell IDs instead. This function still permutes 0/1 chromosomes (EQU/RAN) and typically needs trimton afterwards.

### Usage

```
crossover(se6, u, uplimit, crossPart = c("EQU", "RAN"), verbose, seed)
```

### Arguments

| | |
|---|---|
| se6 | Legacy binary selection: a list with a grid-ID column plus 0/1 layout columns, and a fitness row. Current selection returns ID matrices; use set_crossover in the GA loop. |
| u | The crossover point rate |
| uplimit | The upper limit of allowed permutations |
| crossPart | The crossover method. Either "EQU" or "RAN" |
| verbose | If TRUE, will print out further information |
| seed | Set a seed for comparability. Default is NULL |

## Value

Returns a binary coded matrix of all permutations and all grid cells, where 0 indicates no turbine and 1 indicates a turbine in the grid cell.

## See Also

Other Genetic Algorithm Functions: `fitness()`, `genetic_algorithm()`, `init_population()`, `mutation()`, `selection()`, `set_crossover()`, `swap_mutation()`, `trimton()`

## Examples

```
## Create two random parents with an index and random binary values
Parents <- data.frame(
  ID = 1:20,
  bin = sample(c(0, 1), 20, replace = TRUE, prob = c(70, 30)),
  bin.1 = sample(c(0, 1), 20, replace = TRUE, prob = c(30, 70))
)

## Create random Fitness values for both individuals
FitParents <- data.frame(ID = 1, Fitness = 1000, Fitness.1 = 20)

## Assign both values to a list
CrossSampl <- list(Parents, FitParents)
## Cross their data at equal locations with 2 crossover parts
crossover(CrossSampl, u = 1.1, uplimit = 300, crossPart = "EQU")

## with 3 crossover parts and equal locations
crossover(CrossSampl, u = 2.5, uplimit = 300, crossPart = "EQU")

## or with random locations and 5 crossover parts
crossover(CrossSampl, u = 4.9, uplimit = 300, crossPart = "RAN")
```

---

# explore_result *Shiny explorer for a GA result*

---

## Description

One-page viewer for a `genetic_algorithm()` result: Leaflet map of that generation's best layout (downwind wake cones, optional terrain layers), a wind rose, and one plotly figure with fitness, operator rates and population. New fitness maxima are marked; click a marker to jump to that generation. The cell heatmap is not included: use `plot_cell_heatmap()` or `plot_generation()` offline. Requires Suggests `shiny`. Plotly is used when installed.

## Usage

```
explore_result(result, area)
```

### Arguments

| | |
|---|---|
| result | The output of genetic_algorithm |
| area | Site polygon (sf, SpatialPolygons, or coordinate matrix). Must be projected in metres. |

### Value

The Shiny app object, invisibly. Called for its side effect.

### Examples

```
## Not run:
explore_result(resultrect, sp_polygon)

## End(Not run)
```

---

## fitness *Evaluate the Individual Fitness values*

---

### Description

The fitness of all individuals in the current population is calculated after their energy output has been evaluated in calculate_energy. This function reduces the resulting energy outputs to a single fitness value for each individual.

### Usage

```
fitness(
  population,
  reference_height,
  rotor_height,
  surface_roughness,
  area,
  rotor,
  wind,
  elevation = NULL,
  terrain = FALSE,
  ccl_raster = NULL,
  weibull = FALSE,
  parallel = FALSE,
  n_cluster = 2
)
```

## Arguments

| | |
|---|---|
| population | A list of individuals (layouts with X/Y and cell IDs). |
| reference_height | Height at which wind$ws was measured. |
| rotor_height | Hub height in metres. |
| surface_roughness | Roughness length in metres. Per-cell when terrain is on. |
| area | Site polygon (sf, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| rotor | Rotor radius in metres. |
| wind | Wind data as returned by windata_format() (list(df, probab)). |
| elevation | Terrain list from terrain_model() (elevation, orography, roughness). Unused when terrain is FALSE. |
| terrain | Terrain model (elevation + land cover). TRUE downloads a DEM via elevatr. Pass a DEM raster to skip the download. Per-cell values are computed once and stored in the result as terrainModel for plot_result() / random_search(). |
| ccl_raster | Land-cover roughness raster from terrain_model(). |
| weibull | Raster of estimated wind speeds, or FALSE. |
| parallel | Parallel fitness (parallel + doParallel). |
| n_cluster | Worker count when parallel is TRUE. |

## Value

Returns a list with every individual, consisting of X & Y coordinates, rotor radii, the runs and the selected grid cell IDs, and the resulting energy outputs, efficiency rates and fitness values.

## See Also

Other Genetic Algorithm Functions: crossover(), genetic_algorithm(), init_population(), mutation(), selection(), set_crossover(), swap_mutation(), trimton()

## Examples

```
## Create a random rectangular shapefile
library(sf)
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4498482, 4498482, 4499991, 4499991, 4498482),
    c(2668272, 2669343, 2669343, 2668272, 2668272)
  ))),
  crs = 3035
))

## Create a uniform and unidirectional wind data.frame and plots the
## resulting wind rose
## Uniform wind speed and single wind direction
```

```
wind <- data.frame(ws = 12, wd = 0)
# windrosePlot <- plot_windrose(data = wind, spd = wind$ws,
#                dir = wind$wd, dirres=10, spdmax=20)

## Calculate a Grid and an indexed data.frame with coordinates and
## grid cell IDs.
Grid1 <- grid_area(area = area, size = 200, prop = 1)
Grid <- Grid1[[1]]
AmountGrids <- nrow(Grid)

wind <- list(wind, probab = 100)
startsel <- init_population(Grid, 10, 20)
fit <- fitness(
  population = startsel, reference_height = 100, rotor_height = 100,
  surface_roughness = 0.3, area = area, rotor = 20,
  wind = wind, terrain = FALSE, parallel = FALSE
)
```

---

`ga_options` *Get or set windfarmGA options*

---

## Description

Print every `windfarmGA.*` session option, or set some of them. Names may be given with or without the `windfarmGA.` prefix.

## Usage

```
ga_options(...)
```

## Arguments

| | |
|---|---|
| `...` | Named options to set, or a single named list. With no arguments, print the current values and return them invisibly. Setting is silent. |

## Value

A named list of `windfarmGA.*` options, invisibly.

## See Also

`options()`

## Examples

```
ga_options()
ga_options(immigrants = 3, local_search_tries = 6)
```

## generation_layouts *Layouts evaluated in one generation*

### Description

Return every individual that was fitness-evaluated in a given generation (`allCoords`). Duplicate layouts (same cell IDs) are flagged.

### Usage

```
generation_layouts(result, generation = NULL)
```

### Arguments

| | |
|---|---|
| `result` | The output of `genetic_algorithm` |
| `generation` | Generation index (1 = first). Default is the last generation in `result`. |

### Value

A list with `turbines` (one row per turbine), `layouts` (one row per individual) and `generation`.

### See Also

Other Plotting Functions: `plot_cell_heatmap()`, `plot_cloud()`, `plot_development()`, `plot_evolution()`, `plot_generation()`, `plot_parkfitness()`, `plot_population()`, `plot_result()`, `plot_windfarmGA()`, `plot_windrose()`, `population_census()`, `random_search_single()`

### Examples

```
generation_layouts(resultrect, generation = 10)
```

## genetic_algorithm *Run a Genetic Algorithm to optimize a wind farm layout*

### Description

Run a Genetic Algorithm to optimize the layout of wind turbines on a given area. The algorithm works with a fixed amount of turbines, a fixed rotor radius and a mean wind speed value for every incoming wind direction.

## Usage

```
genetic_algorithm(
  area,
  wind,
  n,
  rotor,
  rotor_height,
  grid_method = "rectangular",
  fcr = 5,
  reference_height = rotor_height,
  surface_roughness = 0.3,
  proportionality = 1,
  iteration = 20,
  mutation_rate = NULL,
  terrain = FALSE,
  elitism = TRUE,
  n_elite = 3,
  selection_mode = "VAR",
  crs = NULL,
  ccl = NULL,
  ccl_roughness = NULL,
  weibull = FALSE,
  weibull_src = NULL,
  parallel = FALSE,
  n_cluster = 2,
  verbose = FALSE,
  plot = FALSE,
  on_generation = NULL
)
```

## Arguments

| | |
|---|---|
| area | Site polygon (sf, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| wind | Wind data.frame with ws, wd and optional probab. See windata_format(). |
| n | Number of turbines (fixed; every individual has n unique cell IDs). |
| rotor | Rotor radius in metres. |
| rotor_height | Hub height in metres. |
| grid_method | "rectangular" or "h" / "hexagon". |
| fcr | Grid spacing factor. Cell size is fcr * rotor. |
| reference_height | Height at which wind$ws was measured. |
| surface_roughness | Roughness length in metres. Per-cell when terrain is on. |
| proportionality | Minimum fraction of a grid cell that must overlap the site (prop in grid_area()). |

| | |
|---|---|
| iteration | Generation budget. |
| mutation_rate | Swap probability per turbine. NULL means 2/n. |
| terrain | Terrain model (elevation + land cover). TRUE downloads a DEM via elevatr. Pass a DEM raster to skip the download. Per-cell values are computed once and stored in the result as terrainModel for plot_result() / random_search(). |
| elitism | Archive the best layout and breed elite children. |
| n_elite | Base elite count (grows/shrinks with search phase). |
| selection_mode | "VAR" (parent share follows fitness) or "FIX" (50 %). |
| crs | CRS if area has none (EPSG code or PROJ string). |
| ccl | Path to a Corine Land Cover raster when terrain is on. |
| ccl_roughness | Path to the CLC legend CSV (Rauhigkeit_z column). |
| weibull | If TRUE, hub-height speed comes from Weibull rasters; wind$ws is ignored. |
| weibull_src | list(k, a) shape and scale rasters (e.g. Global Wind Atlas combined-Weibull-k / combined-Weibull-A). |
| parallel | Parallel fitness (parallel + doParallel). |
| n_cluster | Worker count when parallel is TRUE. |
| verbose | Print a line per generation. |
| plot | Plot the current best layout each generation. |
| on_generation | Optional callback after each generation: function(generation, iteration, energy, efficiency, f Used by Shiny for progress. Errors in the callback are ignored. |

## Details

A terrain effect model can be included in the optimization process. Therefore, a digital elevation model will be downloaded automatically via the elevatr::get_elev_raster function. A land cover raster can also downloaded automatically from the EEA-website, or the path to a raster file can be passed to ccl. The algorithm uses an adapted version of the Raster legend ("clc_legend.csv"), which is stored in the package directory '~/inst/extdata'. To use other values for the land cover roughness lengths, insert a column named **"Rauhigkeit_z"** to the .csv file, assign a surface roughness length to all land cover types. Be sure that all rows are filled with numeric values and save the file with **";"** separation. Assign the path of the file to the input variable ccl_roughness of this function.

Fitness is $EnergyOverall \times (EfficAllDir/100)^w$ with w = getOption("windfarmGA.fitness_efficiency_weight"). Hub-height wind speeds use a logarithmic profile unless options(windfarmGA.wind_profile = "power") restores the legacy power law. Power uses options(windfarmGA.Cp) (default 0.45) and optional cut-in / rated / cut-out speeds. Layouts are encoded as n unique grid-cell IDs (set crossover and swap mutation). Selection defaults to VAR (percentage follows fitness progress). Mutation, immigrants and unused-cell injection prefer rarely visited cells. Crossover is spatial with probability options(windfarmGA.spatial_crossover) (default 0.5). Evaluated layouts are cached. A flat global max is not a stop signal. The run ends at iteration, or earlier only after options(windfarmGA.stall_generations) consecutive generations with no new layout, no newly visited cell and no new best fitness (set to 0 to disable). Operator rates cycle like seasons, still only selection / set-crossover / swap-mutation: explore (rates rise on stall) until options(windfarmGA.refine_min_gen) (default 18) and options(windfarmGA.refine_after)

(default 12) generations without a new max at coverage $\geq 0.35$; then refine (inject toward 0.15, mutation toward 2/n, selection toward about 45\ `options(windfarmGA.refine_hold)` generations in refine (default 25), a short disturbance pulse of `options(windfarmGA.explore_pulse)` generations (default 10) raises the same rates even if new maxes are still trickling in, then refine resumes. Elites get a short local search each generation: one turbine slides to a neighbouring empty cell (not a random cell anywhere on the grid).

## Value

The result is a matrix with aggregated values per generation; the best individual regarding energy and efficiency per generation, some fuzzy control variables per generation, a list of all fitness values per generation, the amount of individuals after each process, a matrix of all energy, efficiency and fitness values per generation, the selection and crossover parameters, a matrix with the generational difference in maximum and mean energy output, a matrix with the given inputs, a dataframe with the wind information, the mutation rate per generation and a matrix with all tested wind farm layouts.

## See Also

Other Genetic Algorithm Functions: `crossover()`, `fitness()`, `init_population()`, `mutation()`, `selection()`, `set_crossover()`, `swap_mutation()`, `trimton()`

## Examples

```
## Not run:
## Create a random rectangular shapefile
library(sf)

area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4498482, 4498482, 4499991, 4499991, 4498482),
    c(2668272, 2669343, 2669343, 2668272, 2668272)
  ))),
  crs = 3035
))

## Create a uniform and unidirectional wind data.frame and plot the
## resulting wind rose
data.in <- data.frame(ws = 12, wd = 0)
windrosePlot <- plot_windrose(
  data = data.in, spd = data.in$ws,
  dir = data.in$wd, dirres = 10, spdmax = 20
)

## Runs an optimization run for 20 iterations with the
## given shapefile (area), the wind data.frame (data.in),
## 12 turbines (n) with rotor radii of 30m and hub height of 100m.
result <- genetic_algorithm(
  area = area,
  n = 12,
  wind = data.in,
  rotor = 30,
  rotor_height = 100
```

```
)
plot_windfarmGA(result = result, area = area)

## End(Not run)
```

---

`get_dist_angles` *Calculate distances and angles of possibly influencing turbines*

---

## Description

Calculate distances and angles for a turbine and all it's potentially influencing turbines.

## Usage

```
get_dist_angles(t, o, wnkl, dist, area, plot_angles = FALSE)
```

## Arguments

| | |
|---|---|
| `t` | A data.frame of the current individual with X and Y coordinates |
| `o` | A numeric value indicating the index of the current turbine |
| `wnkl` | Wake opening angle in degrees. Turbines outside this cone are ignored. |
| `dist` | A numeric value indicating the distance, after which the wake effects are considered to be eliminated. |
| `area` | Site polygon |
| `plot_angles` | Plot distances and angles |

## Value

Returns a matrix with the distances, angles and heights of potentially influencing turbines

## See Also

Other Wind Energy Calculation Functions: `barometric_height()`, `calculate_energy()`, `circle_intersection()`, `turbine_influences()`

## Examples

```
library(sf)

## Exemplary input Polygon with 2km x 2km:
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4498482, 4498482, 4499991, 4499991, 4498482),
    c(2668272, 2669343, 2669343, 2668272, 2668272)
  ))),
  crs = 3035
))
```

```
## Create a random windfarm with 10 turbines
t <- st_coordinates(st_sample(area, 10))
t <- cbind(t, "Z" = 1)
wnkl <- 20
dist <- 100000

## Evaluate and plot for every turbine all other potentially influencing turbines
potInfTur <- list()
for (i in 1:(length(t[, 1]))) {
  potInfTur[[i]] <- get_dist_angles(
    t = t, o = i, wnkl = wnkl,
    dist = dist, area = area, plot_angles = TRUE
  )
}
potInfTur
```

---

`get_grids` *Map layouts to grid coordinates*

---

## Description

Map a population of layouts to grid coordinates. Accepts either an integer matrix of unique cell IDs (n turbines x individuals) or a legacy binary matrix (`n_gridcells` x individuals).

## Usage

```
get_grids(layouts, grid)
```

## Arguments

| | |
|---|---|
| `layouts` | Binary matrix (legacy) or integer matrix of grid IDs (n turbines x individuals) |
| `grid` | Indexed grid from `grid_area()` |

## Value

Returns a list of all individuals with X and Y coordinates and the grid cell ID.

## See Also

Other Helper Functions: `grid_area()`, `hexa_area()`, `isSpatial()`, `permutations()`, `read_power_curve()`, `splitAt()`, `wind_from_series()`, `wind_from_uv()`, `windata_format()`

## Examples

```
## Create a random rectangular shapefile
library(sf)
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(0, 0, 2000, 2000, 0),
    c(0, 2000, 2000, 0, 0)
  ))),
  crs = 3035
))

## Calculate a Grid and an indexed data.frame with coordinates and
## grid cell Ids.
Grid1 <- grid_area(area = area, size = 200, prop = 1)
Grid <- Grid1[[1]]

startsel <- init_population(Grid, 10, 20)
wind <- data.frame(ws = 12, wd = 0)
wind <- list(wind, probab = 100)
fit <- fitness(
  population = startsel, reference_height = 100, rotor_height = 100,
  surface_roughness = 0.3, area = area, rotor = 20,
  wind = wind, terrain = FALSE
)
allparks <- do.call("rbind", fit)

## SELECTION (n unique cell IDs per individual)
selec6best <- selection(fit, Grid, 2, TRUE, 6, "VAR")

## Set-crossover and swap-mutation keep exactly n turbines.
cross_ids <- set_crossover(selec6best[[1]], Grid[, "ID"], uplimit = 20)
mut_ids <- swap_mutation(cross_ids, Grid[, "ID"], p = 0.2)

## Look up XY coordinates for the next fitness evaluation.
getRectV <- get_grids(mut_ids, Grid)
fit <- fitness(
  population = getRectV, reference_height = 100, rotor_height = 100,
  surface_roughness = 0.3, area = area, rotor = 20,
  wind = wind, terrain = FALSE
)
head(fit)
```

---

`grid_area` *Make a grid from a Simple Feature Polygon*

---

## Description

Create a grid from a given polygon with a certain resolution and proportionality. The grid cell centroids represent possible wind turbine locations.

## Usage

```
grid_area(area, size = 500, prop = 1, plot_grid = FALSE)
```

## Arguments

| | |
|---|---|
| area | Simple Feature polygon of the site |
| size | Cell size of the grid in metres |
| prop | Minimum fraction of a cell that must overlap the site |
| plot_grid | Draw the grid |

## Value

Returns a list with 2 elements. List element 1 will have the grid cell IDS, and the X and Y coordinates of the centers of each grid cell. List element 2 is the grid as Simple Feature Polygons, which is used for plotting purposes.

## Note

The grid of the genetic algorithm will have a resolution of `rotor * fcr`. See the arguments of `genetic_algorithm()`.

## See Also

Other Helper Functions: `get_grids()`, `hexa_area()`, `isSpatial()`, `permutations()`, `read_power_curve()`, `splitAt()`, `wind_from_series()`, `wind_from_uv()`, `windata_format()`

## Examples

```
## Exemplary input Polygon with 2km x 2km:
library(sf)
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(0, 0, 2000, 2000, 0),
    c(0, 2000, 2000, 0, 0)
  ))),
  crs = 3035
))

## Create a Grid
grid_area(area, 200, 1, TRUE)
grid_area(area, 400, 1, TRUE)

## Examplary irregular input Polygon
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(0, 0, 2000, 3000, 0),
    c(20, 200, 2000, 0, 20)
  ))),
  crs = 3035
))
```

```
## Create a Grid
grid_area(area, 200, 1, TRUE)
grid_area(area, 200, 0.1, TRUE)
grid_area(area, 400, 1, TRUE)
grid_area(area, 400, 0.1, TRUE)
```

---

hexa_area *Polygon to Hexagonal Grids*

---

## Description

The function takes a Simple Feature Polygon and a size argument and creates a list with an indexed matrix with coordinates and a Simple Feature object, that consists of hexagonal grids.

## Usage

```
hexa_area(area, size = 500, plot_grid = FALSE)
```

## Arguments

| | |
|---|---|
| area | Simple Feature polygon of the site |
| size | Cell size of the grid in metres |
| plot_grid | Draw the grid |

## Value

Returns a list with 2 elements. List element 1 will have the grid cell IDS, and the X and Y coordinates of the centers of each grid cell. List element 2 is the grid as Simple Feature Polygons, which is used for plotting purposes.

## See Also

Other Helper Functions: get_grids(), grid_area(), isSpatial(), permutations(), read_power_curve(), splitAt(), wind_from_series(), wind_from_uv(), windata_format()

## Examples

```
library(sf)
## Exemplary input Polygon with 2km x 2km:
Poly <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4498482, 4498482, 4499991, 4499991, 4498482),
    c(2668272, 2669343, 2669343, 2668272, 2668272)
  ))),
  crs = 3035
))
HexGrid <- hexa_area(Poly, 100, TRUE)
```

## hole_shape *A POLYGON with a hole*

### Description

A POLYGON with a hole

### Usage

```
hole_shape
```

### Format

An object of class `sf` (inherits from `data.frame`) with 1 rows and 2 columns.

## init_population *Create a random initial Population*

### Description

Create `n_start` random sub-selections from the indexed grid and assign binary variable 1 to selected grids. This function initiates the genetic algorithm with a first random population and will only be needed in the first iteration.

### Usage

```
init_population(grid, n, n_start = 100)
```

### Arguments

| | |
|---|---|
| `grid` | Indexed grid from `grid_area()` (X, Y, cell IDs). |
| `n` | A numeric value indicating the amount of required turbines. |
| `n_start` | A numeric indicating the amount of randomly generated initial individuals. Default is 100. |

### Value

Returns a list of `n_start` initial individuals, each consisting of `n` turbines. Resulting list has the x and y coordinates, the grid cell ID and a binary variable of 1, indicating a turbine in the grid cell.

### See Also

Other Genetic Algorithm Functions: `crossover()`, `fitness()`, `genetic_algorithm()`, `mutation()`, `selection()`, `set_crossover()`, `swap_mutation()`, `trimton()`

## Examples

```
library(sf)
## Exemplary input Polygon with 2km x 2km:
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4498482, 4498482, 4499991, 4499991, 4498482),
    c(2668272, 2669343, 2669343, 2668272, 2668272)
  ))),
  crs = 3035
))

Grid <- grid_area(area, 200, 1, TRUE)

## Create 5 individuals with 10 wind turbines each.
firstPop <- init_population(grid = Grid[[1]], n = 10, n_start = 5)
```

---

# isSpatial *Transform to Simple Feature Polygons*

---

## Description

Helper Function, which transforms SpatialPolygons or coordinates in matrix/data.frame - form to a Simple Feature Polygon

## Usage

```
isSpatial(area, crs = NULL)
```

## Arguments

| | |
|---|---|
| area | Site as SpatialPolygon, Simple Feature polygon, or coordinates as matrix/data.frame |
| crs | CRS assigned to matrix / data.frame coordinates |

## Details

If the columns are named, it will look for common abbreviation to match x/y or long/lat columns. If the columns are not named, the first 2 numeric columns are taken.

## Value

A Simple Feature Polygon

## See Also

Other Helper Functions: get_grids(), grid_area(), hexa_area(), permutations(), read_power_curve(), splitAt(), wind_from_series(), wind_from_uv(), windata_format()

## Examples

```
library(sf)
df <- rbind(
  c(4498482, 2668272), c(4498482, 2669343),
  c(4499991, 2669343), c(4499991, 2668272)
)
isSpatial(df)

area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4498482, 4498482, 4499991, 4499991, 4498482),
    c(2668272, 2669343, 2669343, 2668272, 2668272)
  ))),
  crs = 3035
))
isSpatial(st_coordinates(area), 3035)
```

---

# multi_shape *A MULTIPOLYGON with 3 Polygons*

---

## Description

A MULTIPOLYGON with 3 Polygons

## Usage

```
multi_shape
```

## Format

An object of class `sf` (inherits from `data.frame`) with 1 rows and 1 columns.

---

# mutation *Mutation Method*

---

## Description

Legacy bit-flip mutation on 0/1 chromosomes. The GA loop uses `swap_mutation` so that every individual keeps exactly `n` turbines.

## Usage

```
mutation(a, p, seed = NULL)
```

### Arguments

| | |
|---|---|
| a | The binary matrix of all individuals |
| p | The mutation rate |
| seed | Set a seed for comparability. Default is NULL |

### Value

Returns a binary matrix with mutated genes.

### See Also

Other Genetic Algorithm Functions: crossover(), fitness(), genetic_algorithm(), init_population(), selection(), set_crossover(), swap_mutation(), trimton()

### Examples

```
## Create 4 random individuals with binary values
a <- cbind(
  bin0 = sample(c(0, 1), 20, replace = TRUE, prob = c(70, 30)),
  bin1 = sample(c(0, 1), 20, replace = TRUE, prob = c(30, 70)),
  bin2 = sample(c(0, 1), 20, replace = TRUE, prob = c(30, 70)),
  bin3 = sample(c(0, 1), 20, replace = TRUE, prob = c(30, 70))
)
a

## Mutate the individuals with a low percentage
aMut <- mutation(a, 0.1, NULL)
## Check which values are not like the originals
a == aMut

## Mutate the individuals with a high percentage
aMut <- mutation(a, 0.4, NULL)
## Check which values are not like the originals
a == aMut
```

---

## package_installed *Is the package installed or not*

---

### Description

Is the package installed or not

### Usage

```
is_foreach_installed()

is_parallel_installed()

is_doparallel_installed()

is_ggplot2_installed()

is_leaflet_installed()

is_elevatr_installed()

is_plotly_installed()

is_shiny_installed()
```

### Value

An invisible boolean value, indicating if the package is installed or not.

---

## permutations *Enumerate the Combinations or Permutations of the Elements of a Vector*

---

### Description

permutations enumerates the possible permutations. The function is forked and minified from gtools::permutations

### Usage

```
permutations(n, r, v = 1:n)
```

### Arguments

| | |
|---|---|
| n | Size of the source vector |
| r | Size of the target vectors |
| v | Source vector. Defaults to 1:n |

### Value

Returns a matrix where each row contains a vector of length r.

### Author(s)

Original versions by Bill Venables. Extended to handle repeats.allowed by Gregory R. Warnes

## References

Venables, Bill. "Programmers Note", R-News, Vol 1/1, Jan. 2001. https://cran.r-project.org/doc/Rnews/

## See Also

Other Helper Functions: get_grids(), grid_area(), hexa_area(), isSpatial(), read_power_curve(), splitAt(), wind_from_series(), wind_from_uv(), windata_format()

---

`plot_cell_heatmap` *Heatmap of probed grid cells*

---

## Description

Count how often each grid cell appeared in an evaluated layout over all generations (`allCoords` in the GA result). Never-tried cells stay light gray. Useful to see whether the search covered the area or stuck to a few sites.

## Usage

```
plot_cell_heatmap(result, area, log = TRUE, plot = TRUE)
```

## Arguments

| | |
|---|---|
| `result` | The output of genetic_algorithm |
| `area` | Site polygon (`sf`, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| `log` | If `TRUE`, color the counts on a `log1p` scale so a few elite cells do not dominate the palette. Default is `TRUE` |
| `plot` | Deprecated alias for `plot`. |

## Value

A `data.frame` with grid ID, coordinates and visit count (`n_probed`), returned invisibly.

## See Also

Other Plotting Functions: generation_layouts(), plot_cloud(), plot_development(), plot_evolution(), plot_generation(), plot_parkfitness(), plot_population(), plot_result(), plot_windfarmGA(), plot_windrose(), population_census(), random_search_single()

## Examples

```
plot_cell_heatmap(resultrect, sp_polygon)
```

## plot_cloud *Per-generation fitness / efficiency / energy*

### Description

Summarise every evaluated individual. With `pl = TRUE` draw three panels of points (max as a line). The returned table has min / mean / max / sd per generation.

### Usage

```
plot_cloud(result, pl = FALSE)
```

### Arguments

| | |
|---|---|
| `result` | The output of `genetic_algorithm` |
| `pl` | Draw the three panels? Default is FALSE |

### Value

A data.frame with fitness, efficiency and energy summaries

### See Also

Other Plotting Functions: `generation_layouts()`, `plot_cell_heatmap()`, `plot_development()`, `plot_evolution()`, `plot_generation()`, `plot_parkfitness()`, `plot_population()`, `plot_result()`, `plot_windfarmGA()`, `plot_windrose()`, `population_census()`, `random_search_single()`

### Examples

```
plcdf <- plot_cloud(resulthex, TRUE)
```

## plot_development *When the best layout improved*

### Description

Change of the generation-best fitness. Green = new record, orange = flat, red = worse. Blue line is the share of grid cells visited so far.

### Usage

```
plot_development(result)
```

### Arguments

| | |
|---|---|
| `result` | The output of `genetic_algorithm` |

## Value

Returns NULL. Used for plotting

## See Also

Other Plotting Functions: generation_layouts(), plot_cell_heatmap(), plot_cloud(), plot_evolution(), plot_generation(), plot_parkfitness(), plot_population(), plot_result(), plot_windfarmGA(), plot_windrose(), population_census(), random_search_single()

## Examples

```
plot_development(resultrect)
```

---

# plot_evolution *Energy and efficiency over generations*

---

## Description

Max and mean park efficiency and energy yield on one page.

## Usage

```
plot_evolution(result, ask = FALSE, spar = 0.1)
```

## Arguments

| | |
|---|---|
| result | The output of genetic_algorithm |
| ask | Unused, kept so existing calls do not break. |
| spar | Unused, kept so existing calls do not break. |

## Value

Returns NULL. Used for plotting

## See Also

Other Plotting Functions: generation_layouts(), plot_cell_heatmap(), plot_cloud(), plot_development(), plot_generation(), plot_parkfitness(), plot_population(), plot_result(), plot_windfarmGA(), plot_windrose(), population_census(), random_search_single()

## Examples

```
plot_evolution(resultrect)
```

plot_generation *Plot all layouts of one generation*

## Description

Occupancy map of one generation, then the distinct layouts as small maps. In an interactive session each page waits for Enter so you can step through every distinct layout (n_show maps per page). Non-interactive calls only draw the first n_show maps.

## Usage

```
plot_generation(
  result,
  area,
  generation = NULL,
  n_show = 6,
  interactive = NULL,
  ask = NULL
)
```

## Arguments

| | |
|---|---|
| result | The output of genetic_algorithm |
| area | Site polygon (sf, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| generation | Generation index (1 = first). Default is the last generation. |
| n_show | Distinct layouts per page (and, if ask is FALSE, how many to draw in total). Default is 6. Set to 0 to skip the maps. |
| interactive | Use plotly for the occupancy map when available. |
| ask | If TRUE, wait for Enter and page through all distinct layouts. Default is TRUE in an interactive session. |

## Value

The list from generation_layouts, invisibly.

## See Also

Other Plotting Functions: generation_layouts(), plot_cell_heatmap(), plot_cloud(), plot_development(), plot_evolution(), plot_parkfitness(), plot_population(), plot_result(), plot_windfarmGA(), plot_windrose(), population_census(), random_search_single()

## Examples

```
plot_generation(resultrect, sp_polygon, generation = 10)
```

plot_leaflet *Plot a wind warm with leaflet*

## Description

Plot a resulting wind farm on a leaflet map. Wakes are downwind cones (Jensen search angle), not circles. Terrain rasters from `result$terrainModel` are optional overlay layers.

## Usage

```
plot_leaflet(
  result,
  area,
  which = 1,
  orderitems = TRUE,
  grid = NULL,
  wind = NULL,
  terrain = NULL
)
```

## Arguments

| | |
|---|---|
| result | The output of genetic_algorithm |
| area | Site polygon (sf, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| which | A numeric value, indicating which individual to plot. The default is 1. Combined with orderitems = TRUE this will show the best performing wind farm. |
| orderitems | A logical value indicating whether the results should be ordered by energy values TRUE or chronologically FALSE |
| grid | Optional grid polygons. By default they are rebuilt from result and area. You can pass the polygon element of grid_area() or hexa_area(). |
| wind | Optional wind table (ws, wd, probab). Defaults to the table stored in result. |
| terrain | Optional output of leaflet_prepare_terrain(). Defaults to result$terrainModel when present. |

## Value

Returns a leaflet map.

## Examples

```
## Not run:
## Plot the best wind farm on a leaflet map (ordered by energy values)
plot_leaflet(result = resulthex, area = sp_polygon, which = 1)

## Plot the last wind farm (ordered by chronology).
```

```
plot_leaflet(
  result = resulthex, area = sp_polygon, orderitems = FALSE,
  which = 1
)

## Plot the best wind farm on a leaflet map with the rectangular Grid
Grid <- grid_area(sp_polygon, size = 150, prop = 0.4)
plot_leaflet(
  result = resultrect, area = sp_polygon, which = 1,
  grid = Grid[[2]]
)

## Plot the last wind farm with hexagonal Grid
Grid <- hexa_area(sp_polygon, size = 75)
plot_leaflet(
  result = resulthex, area = sp_polygon, which = 1,
  grid = Grid[[2]]
)

## End(Not run)
```

---

`plot_parkfitness` *Fitness and operator rates*

---

## Description

Fitness (max / mean / min) and the three operator rates. Rates are percentages: **Selection** = share of the population used as parents, **Crossover inject** = unused cells mixed into children, **Mutation** = chance that a turbine is swapped to a free cell. The legend sits outside and follows the typical vertical order of the lines (selection highest, then inject, mutation lowest). Plotly hover is used only when `ask` is `FALSE` and plotly is installed.

## Usage

```
plot_parkfitness(result, spar = 0.1, interactive = NULL, ask = NULL)
```

## Arguments

| | |
|---|---|
| `result` | The output of `genetic_algorithm` |
| `spar` | Unused, kept so existing calls do not break. |
| `interactive` | Use plotly when `ask` is `FALSE` and plotly is installed. |
| `ask` | If `TRUE`, wait for Enter between the fitness page and the rates page. Default is `TRUE` in an interactive session. |

## Value

A plotly object, or a list of ggplots, invisibly.

## See Also

Other Plotting Functions: generation_layouts(), plot_cell_heatmap(), plot_cloud(), plot_development(), plot_evolution(), plot_generation(), plot_population(), plot_result(), plot_windfarmGA(), plot_windrose(), population_census(), random_search_single()

## Examples

```
plot_parkfitness(resulthex)
```

---

# plot_population *Plot population size, cells and efficiency*

---

## Description

Three pages: (1) individuals / parents / elites, (2) unique grid cells this generation, in the elites, and ever probed, (3) park efficiency. Each page waits for Enter in an interactive session. If the grid has e.g. 70 cells and the whole population still touches every cell, "this generation" and "ever probed" both sit at 70; the elite line is then the one that shrinks toward the best sites.

## Usage

```
plot_population(result, interactive = NULL, ask = NULL)
```

## Arguments

| | |
|---|---|
| result | The output of genetic_algorithm |
| interactive | Use plotly when ask is FALSE and plotly is installed. |
| ask | If TRUE, wait for Enter between pages (Plots pane). Default is TRUE in an interactive session. |

## Value

A plotly object, or a list of ggplots, invisibly. The census table is attached as attribute census.

## See Also

Other Plotting Functions: generation_layouts(), plot_cell_heatmap(), plot_cloud(), plot_development(), plot_evolution(), plot_generation(), plot_parkfitness(), plot_result(), plot_windfarmGA(), plot_windrose(), population_census(), random_search_single()

## Examples

```
plot_population(resultrect)
```

## plot_power_curve *Manufacturer power curve*

### Description

Linear interpolation of a two-column table (`ws`, `power` in kW). Set it with `ga_options(power_curve = curve)` or `options(windfarmGA.power_curve = curve)`. While a curve is set, park energy is the sum of those kW values instead of Cp * v^3. Cut-in / rated / cut-out still apply only when no table is set. If hub wind stays on the rated plateau after wakes, every layout looks the same - use wind in the rising part of the curve. Supply your own table (manufacturer data); the package does not ship copyrighted curves.

### Usage

```
plot_power_curve(curve = NULL, plot = TRUE)
```

### Arguments

| | |
|---|---|
| `curve` | A data.frame with wind speed and power. Default is the current `windfarmGA.power_curve` option. |
| `plot` | If `TRUE`, draw the curve. Default is `TRUE`. |

### Value

The interpolated table, invisibly.

### Examples

```
curve <- data.frame(
  ws = c(0, 3, 4, 8, 12, 25, 26),
  power = c(0, 0, 80, 1200, 2000, 2000, 0)
)
plot_power_curve(curve)
```

## plot_random_search *Plot the result of a randomized output.*

### Description

Plotting method for the results of `random_search_single` and `random_search`.

### Usage

```
plot_random_search(resultRS, result, area, best)
```

### Arguments

| | |
|---|---|
| resultRS | The result of the random functions random_search_single and random_search. |
| result | The output of genetic_algorithm |
| area | Site polygon (sf, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| best | How many best candidates to plot. Default is 1. |

### Value

Returns NULL. Used for plotting

### See Also

Other Randomization: random_search(), random_search_single()

### Examples

```
library(sf)
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4498482, 4498482, 4499991, 4499991, 4498482),
    c(2668272, 2669343, 2669343, 2668272, 2668272)
  ))),
  crs = 3035
))

Res <- random_search(result = resultrect, area = area)
plot_random_search(resultRS = Res, result = resultrect, area = area, best = 2)
```

---

## plot_result *Plot the best results*

---

### Description

Draw the best layout(s) on the site. Turbine labels are the total wake in percent. Default is the single best energy layout.

### Usage

```
plot_result(
  result,
  area,
  best = 1,
  plot_en = 1,
  terrain = FALSE,
  plot_grid = TRUE,
```

```
  ccl_roughness = NULL,
  ccl = NULL,
  weibull_src = NULL
)
```

## Arguments

| | |
|---|---|
| result | The output of genetic_algorithm |
| area | Site polygon (sf, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| best | How many distinct best layouts to draw. Default is 1. |
| plot_en | A numeric value that indicates if the best energy or efficiency output should be plotted. 1 plots the best energy solutions and 2 plots the best efficiency solutions |
| terrain | Draw terrain rasters for the best layout. Reuses result$terrainModel when present; TRUE downloads only if nothing is stored. A DEM raster rebuilds the model. |
| plot_grid | If TRUE (default) the used grid is added. You can also pass another Simple Feature object |
| ccl_roughness | Path to the CLC legend CSV (Rauhigkeit_z column). |
| ccl | Path to a Corine Land Cover raster when terrain is on. |
| weibull_src | list(k, a) shape and scale rasters (e.g. Global Wind Atlas combined-Weibull-k / combined-Weibull-A). |

## Value

Returns a data.frame of the best (energy/efficiency) individual during all iterations

## See Also

Other Plotting Functions: generation_layouts(), plot_cell_heatmap(), plot_cloud(), plot_development(), plot_evolution(), plot_generation(), plot_parkfitness(), plot_population(), plot_windfarmGA(), plot_windrose(), population_census(), random_search_single()

## Examples

```
## Not run:
## Add some data examples from the package
library(sf)
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4498482, 4498482, 4499991, 4499991, 4498482),
    c(2668272, 2669343, 2669343, 2668272, 2668272)
  ))),
  crs = 3035
))

## Plot the results of a hexagonal grid optimization
plot_result(resulthex, area, best = 1, plot_en = 1, terrain = FALSE)
```

```
## Plot the results of a rectangular grid optimization
plot_result(resultrect, area, best = 1, plot_en = 1, terrain = FALSE)

## End(Not run)
```

---

`plot_viewshed` *Plot visibility*

---

## Description

Calculate and plot visibility for given points in a given area. `terra::viewshed` needs a projected (metre) raster. Lon/lat DEMs (typical `elevatr` downloads) are projected automatically.

## Usage

```
plot_viewshed(r, turbine_locs, h1 = 0, h2 = 0, plot = TRUE, ...)
```

## Arguments

| | |
|---|---|
| `r` | The elevation SpatRaster |
| `turbine_locs` | Coordinates, SpatialPoint or SimpleFeature Points representing the wind turbines |
| `h1` | A single number or numeric vector giving the extra height offsets for the `turbine_locs` |
| `h2` | The height offset for Point 2 |
| `plot` | Should the result be plotted. Default is `TRUE` |
| `...` | forwarded to `terra::plot` |

## Value

A mosaiced SpatRaster, representing the visibility for all `turbine_locs`

## Examples

```
library(sf)
library(terra)

f <- system.file("ex/elev.tif", package = "terra")
r <- rast(f)
x <- project(r, "EPSG:2169")
shape <- sf::st_as_sf(as.polygons(terra::boundaries(x)))
plot(shape)
st_crs(shape) <- 2169
locs <- st_sample(shape, 10, type = "random")
plot_viewshed(x, locs, h1 = 0, h2 = 0, plot = TRUE)
```

plot_windfarmGA *Plot the results of an optimization run*

## Description

Draw the useful summary plots of a GA run, one after another: best layout, fitness, operator rates, population, cells, efficiency, and the cell heatmap. In an interactive session every page waits for Enter so nothing is overwritten in the Plots pane.

## Usage

```
plot_windfarmGA(
  result,
  area,
  which_plot = "all",
  best = 1,
  plot_en = 1,
  weibull_src = NULL,
  ask = NULL,
  plotly = NULL
)
```

## Arguments

| | |
|---|---|
| result | The output of genetic_algorithm |
| area | Site polygon (sf, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| which_plot | "all" (default) shows result, progress, population and heatmap. Or a character vector ("result", "progress", "population", "heatmap", "evolution") or the numbers 1-4. |
| best | How many distinct best layouts to draw. Default is 1. |
| plot_en | A numeric value that indicates if the best energy or efficiency output should be plotted. 1 plots the best energy solutions and 2 plots the best efficiency solutions |
| weibull_src | list(k, a) shape and scale rasters (e.g. Global Wind Atlas combined-Weibull-k / combined-Weibull-A). |
| ask | If TRUE, wait for Enter between pages. Default is TRUE in an interactive session. |
| plotly | If TRUE, draw fitness and rates with plotly (hover). Used only when ask is FALSE and plotly is installed. |

## Value

Returns NULL. Used for plotting

## See Also

Other Plotting Functions: generation_layouts(), plot_cell_heatmap(), plot_cloud(), plot_development(), plot_evolution(), plot_generation(), plot_parkfitness(), plot_population(), plot_result(), plot_windrose(), population_census(), random_search_single()

## Examples

```
## Not run:
library(sf)
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4498482, 4498482, 4499991, 4499991, 4498482),
    c(2668272, 2669343, 2669343, 2668272, 2668272)
  ))),
  crs = 3035
))

plot_windfarmGA(resulthex, area)
plot_windfarmGA(resultrect, area, which_plot = "progress")

## End(Not run)
```

---

# plot_windrose *Plot a Windrose*

---

## Description

Plot a wind rose of the wind data frame.

## Usage

```
plot_windrose(
  data,
  spd,
  dir,
  spdres = 2,
  dirres = 10,
  spdmin = 1,
  spdmax = 30,
  palette = "YlGnBu",
  spdseq = NULL,
  plot = TRUE
)
```

## Arguments

| | |
|---|---|
| data | A data.frame containing the wind information |
| spd | The column of the wind speeds in "data" |
| dir | The column of the wind directions in "data" |
| spdres | The increment of the wind speed legend. Default is 2 |
| dirres | The size of the wind sectors. Default is 10 |
| spdmin | Minimum wind speed. Default is 1 |
| spdmax | Maximal wind speed. Default is 30 |
| palette | A color palette used for drawing the wind rose |
| spdseq | A wind speed sequence, that is used for plotting |
| plot | Deprecated alias for plot. |

## Value

A ggplot2 wind rose plot, returned invisibly.

## See Also

Other Plotting Functions: generation_layouts(), plot_cell_heatmap(), plot_cloud(), plot_development(), plot_evolution(), plot_generation(), plot_parkfitness(), plot_population(), plot_result(), plot_windfarmGA(), population_census(), random_search_single()

## Examples

```
## Exemplary Input Wind speed and direction data frame
# Uniform wind speed and single wind direction
data.in <- data.frame(ws = 12, wd = 0)
windrosePlot <- plot_windrose(
  data = data.in, spd = data.in$ws,
  dir = data.in$wd
)

# Random wind speeds and random wind directions
data.in <- data.frame(
  ws = sample(1:25, 10),
  wd = sample(1:260, 10)
)
windrosePlot <- plot_windrose(
  data = data.in, spd = data.in$ws,
  dir = data.in$wd
)
```

## population_census *Population size and diversity per generation*

### Description

Counts evaluated individuals, distinct layouts, duplicates dropped before the next generation, selected parents, elites, elite offspring and grid cells (this generation and cumulative).

### Usage

```
population_census(result)
```

### Arguments

| | |
|---|---|
| result | The output of genetic_algorithm |

### Value

A data.frame with one row per generation.

### See Also

Other Plotting Functions: generation_layouts(), plot_cell_heatmap(), plot_cloud(), plot_development(), plot_evolution(), plot_generation(), plot_parkfitness(), plot_population(), plot_result(), plot_windfarmGA(), plot_windrose(), random_search_single()

### Examples

```
population_census(resultrect)
```

## random_search *Randomize the output of the Genetic Algorithm*

### Description

Jitter the best GA layouts inside their grid cells and re-evaluate energy. Use this as a short post-search after genetic_algorithm(). Terrain and Weibull follow the GA flags when terrain / weibull are NULL. Terrain rasters from the GA are reused when stored in result; pass a DEM to rebuild. Weibull rasters are not stored; pass weibull_src again if needed.

## Usage

```
random_search(
  result,
  area,
  runs = 20,
  best = 1,
  plot = FALSE,
  max_dist = 2.2,
  terrain = NULL,
  weibull = NULL,
  weibull_src = NULL,
  ccl = NULL,
  ccl_roughness = NULL
)
```

## Arguments

| | |
|---|---|
| result | The resulting matrix of the function genetic_algorithm |
| area | Site polygon (sf, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| runs | How many jittered layouts to try per best start. Default is 20. |
| best | How many distinct best layouts to refine. Default is 1. |
| plot | Draw the random-search layouts |
| max_dist | A numeric value multiplied by the rotor radius to perform collision checks. Default is 2.2 |
| terrain | NULL (default) follows the GA and reuses result$terrainModel. TRUE downloads only if nothing is stored. A DEM rebuilds the model. FALSE skips terrain. |
| weibull | NULL follows the GA flag. Weibull rasters are not stored; pass weibull_src (or a speed raster as weibull) again. Giving weibull_src is enough; you do not also need weibull = TRUE. |
| weibull_src | list(k, a) shape and scale rasters (e.g. Global Wind Atlas combined-Weibull-k / combined-Weibull-A). |
| ccl | Path to a Corine Land Cover raster when terrain is on. |
| ccl_roughness | Path to the CLC legend CSV (Rauhigkeit_z column). |

## Value

Returns a list.

## See Also

Other Randomization: plot_random_search(), random_search_single()

## Examples

```
new <- random_search(resultrect, sp_polygon, runs = 20, best = 4)
plot_random_search(resultRS = new, result = resultrect, area = sp_polygon, best = 2)
```

---

`random_search_single` *Randomize the location of a single turbine*

---

## Description

Perform a random search for a single turbine, to further optimize the output of the wind farm layout.

## Usage

```
random_search_single(
  result,
  area,
  runs = 20,
  plot = FALSE,
  max_dist = 2.2,
  terrain = NULL,
  weibull = NULL,
  weibull_src = NULL,
  ccl = NULL,
  ccl_roughness = NULL,
  turbine = NULL
)
```

## Arguments

| | |
|---|---|
| `result` | The resulting matrix of the function `genetic_algorithm` |
| `area` | Site polygon (`sf`, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| `runs` | How many jittered layouts to try per `best` start. Default is 20. |
| `plot` | Draw the random-search layouts |
| `max_dist` | A numeric value multiplied by the rotor radius to perform collision checks. Default is 2.2 |
| `terrain` | `NULL` (default) follows the GA and reuses `result$terrainModel`. `TRUE` downloads only if nothing is stored. A DEM rebuilds the model. `FALSE` skips terrain. |
| `weibull` | `NULL` follows the GA flag. Weibull rasters are not stored; pass `weibull_src` (or a speed raster as `weibull`) again. Giving `weibull_src` is enough; you do not also need `weibull = TRUE`. |
| `weibull_src` | `list(k, a)` shape and scale rasters (e.g. Global Wind Atlas `combined-Weibull-k` / `combined-Weibull-A`). |

| | |
|---|---|
| ccl | Path to a Corine Land Cover raster when terrain is on. |
| ccl_roughness | Path to the CLC legend CSV (Rauhigkeit_z column). |
| turbine | Grid cell ID of the turbine to move. If NULL, the function asks interactively. |

## Value

Returns a list

## See Also

Other Randomization: plot_random_search(), random_search()

Other Plotting Functions: generation_layouts(), plot_cell_heatmap(), plot_cloud(), plot_development(), plot_evolution(), plot_generation(), plot_parkfitness(), plot_population(), plot_result(), plot_windfarmGA(), plot_windrose(), population_census()

---

# read_power_curve *Read a manufacturer or NREL/IEA power-curve table*

---

## Description

Parse a CSV (or data.frame) with wind speed and power in kW. Understands NREL/IEA headers such as Wind Speed [m/s] and Power [kW]. Optional Ct is returned as an attribute. Does not ship manufacturer curves; pass your own file or an open IEA/NREL reference CSV. See experimental/climate_helpers.R.

## Usage

```
read_power_curve(file)
```

## Arguments

| | |
|---|---|
| file | Path to a CSV, or a data.frame. |

## Value

A data.frame with ws and power. Attribute ct is a matching data.frame when a thrust column is present.

## See Also

Other Helper Functions: get_grids(), grid_area(), hexa_area(), isSpatial(), permutations(), splitAt(), wind_from_series(), wind_from_uv(), windata_format()

## Examples

```
curve <- data.frame(
  `Wind Speed [m/s]` = c(3, 8, 12, 25),
  `Power [kW]` = c(0, 1200, 2000, 2000),
  check.names = FALSE
)
read_power_curve(curve)
```

---

`resulthex` *A resulting matrix of* `genetic_algorithm` *with 10 iterations and a hexagonal grid derived from* `sp_polygon`

---

## Description

A resulting matrix of `genetic_algorithm` with 10 iterations and a hexagonal grid derived from `sp_polygon`

## Usage

```
resulthex
```

## Format

An object of class `matrix` (inherits from `array`) with 10 rows and 13 columns.

---

`resultrect` *A resulting matrix of* `genetic_algorithm` *with 50 generations and a rectangular grid derived from* `sp_polygon`

---

## Description

First 50 generations of a 200-iteration run, kept short so the installed package stays small. Do not recreate with the current GA (`n_start` and population size are larger; the `.rda` grows).

## Usage

```
resultrect
```

## Format

An object of class `matrix` (inherits from `array`) with 50 rows and 13 columns.

`selection` *Selection Method*

## Description

Select a certain amount of individuals and recombine them to parental teams. Add the mean fitness value of both parents to the parental team. Depending on the selected `selection_mode`, the algorithm will either take always 50 percent or a variable percentage of the current population. The variable percentage depends on the evolution of the populations fitness values. With `elitism = TRUE` the best individuals are always included in the mating pool.

## Usage

```
selection(
  fit,
  grid,
  share,
  elitism = TRUE,
  n_elite = 3,
  selection_mode = "VAR",
  verbose = FALSE
)
```

## Arguments

| | |
|---|---|
| `fit` | A list of all fitness-evaluated individuals |
| `grid` | Indexed grid from `grid_area()` |
| `share` | Selection divisor: parents are about `nrow / share` of the population (`2` is about 50 %). |
| `elitism` | Archive the best layout and breed elite children. |
| `n_elite` | Base elite count (grows/shrinks with search phase). |
| `selection_mode` | `"VAR"` (parent share follows fitness) or `"FIX"` (50 %). |
| `verbose` | If TRUE, will print out further information. |

## Value

Returns a list with 2 elements. Element 1 is an integer matrix of selected layouts (`n` turbines x selected individuals), each column a set of unique grid cell IDs. Element 2 is the fitness of each selected individual.

## See Also

Other Genetic Algorithm Functions: `crossover()`, `fitness()`, `genetic_algorithm()`, `init_population()`, `mutation()`, `set_crossover()`, `swap_mutation()`, `trimton()`

## Examples

```
## Exemplary input Polygon with 2km x 2km:
library(sf)
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4498482, 4498482, 4499991, 4499991, 4498482),
    c(2668272, 2669343, 2669343, 2668272, 2668272)
  ))),
  crs = 3035
))

## Calculate a Grid and an indexed data.frame with coordinates and grid cell Ids.
Grid1 <- grid_area(area = area, size = 200, prop = 1)
Grid <- Grid1[[1]]
AmountGrids <- nrow(Grid)

startsel <- init_population(Grid, 10, 20)
wind <- as.data.frame(cbind(ws = 12, wd = 0))
wind <- list(wind, probab = 100)
fit <- fitness(
  population = startsel, reference_height = 100, rotor_height = 100,
  surface_roughness = 0.3, area = area, rotor = 20, wind = wind,
  terrain = FALSE
)
allparks <- do.call("rbind", fit)
## SELECTION
## print the amount of Individuals selected. Check if the amount
## of Turbines is as requested.
selec6best <- selection(fit, Grid, 2, TRUE, 6, "VAR")
selec6best <- selection(fit, Grid, 2, TRUE, 6, "FIX")
selec6best <- selection(fit, Grid, 4, FALSE, 6, "FIX")
```

---

`set_crossover` *Set crossover of turbine layouts*

---

## Description

Combine two layouts of `n` unique grid-cell IDs. Shared sites are kept; remaining sites are sampled from the parents' exclusive cells and, with rate `p_inject`, from grid cells that neither parent uses. Identical parents still get unused cells injected so the search does not freeze. Every child has exactly `n` turbines.

## Usage

```
set_crossover(
  ids,
  grid_ids,
  uplimit = 300,
```

```
  seed = NULL,
  verbose = FALSE,
  p_inject = NULL,
  grid_xy = NULL,
  visit = NULL,
  p_spatial = NULL
)
```

## Arguments

| | |
|---|---|
| ids | Integer matrix with n rows (turbines) and one column per parent |
| grid_ids | All valid grid cell IDs |
| uplimit | Maximum number of children. Default is 300 |
| seed | Set a seed for comparability. Default is NULL |
| verbose | If TRUE, print the number of children |
| p_inject | Fraction of non-shared slots filled from unused grid cells. Default is getOption("windfarmGA.crossove (0.25). At least one unused cell is injected when any are available. |
| grid_xy | Optional matrix/data.frame with columns ID, X, Y. If given, a spatial half-plane crossover is used with probability p_spatial. |
| visit | Named visit counts per grid ID (undersampled cells preferred) |
| p_spatial | Probability of spatial (vs set) crossover when grid_xy is given. Default is getOption("windfarmGA.spatial_crossover") (0.5) |

## Value

Integer matrix of unique grid IDs (n x children)

## See Also

Other Genetic Algorithm Functions: crossover(), fitness(), genetic_algorithm(), init_population(), mutation(), selection(), swap_mutation(), trimton()

## Examples

```
ids <- cbind(c(1, 3, 5, 7), c(1, 4, 5, 9))
set_crossover(ids, grid_ids = 1:20, uplimit = 4, seed = 1)
```

---

# splitAt *Split matrices or numeric vectors at specific indices*

---

## Description

The function is used by the crossover method to split a genetic code at certain intervals. See also crossover.

## Usage

```
splitAt(x, pos)
```

## Arguments

| | |
|---|---|
| x | A numeric variable that represents an individual's binary genetic code |
| pos | A numeric value that indicates where to split the genetic code |

## Value

Returns a list of the split genetic code.

## See Also

Other Helper Functions: get_grids(), grid_area(), hexa_area(), isSpatial(), permutations(), read_power_curve(), wind_from_series(), wind_from_uv(), windata_format()

## Examples

```
splitAt(1:100, 20)
splitAt(as.matrix(1:100), 20)
```

---

# sp_polygon *The rectangular POLYGON used to create* resultrect *&* resulthex

---

## Description

The rectangular POLYGON used to create resultrect & resulthex

## Usage

```
sp_polygon
```

## Format

An object of class sf (inherits from data.frame) with 1 rows and 1 columns.

swap_mutation *Swap mutation of turbine layouts*

## Description

Replace occupied grid cells with unused ones. The number of swaps is `max(min_swaps, Binomial(n, p))`, so each individual explores at least `min_swaps` new cells (default 1). The number of turbines stays `n`.

## Usage

```
swap_mutation(ids, grid_ids, p, seed = NULL, min_swaps = NULL, visit = NULL)
```

## Arguments

| | |
|---|---|
| `ids` | Integer matrix with `n` rows (turbines) and one column per individual |
| `grid_ids` | All valid grid cell IDs |
| `p` | Mutation probability per turbine |
| `seed` | Set a seed for comparability. Default is `NULL` |
| `min_swaps` | Minimum number of swaps per individual. Default is `getOption("windfarmGA.min_swaps")` (1) |
| `visit` | Named visit counts per grid ID. Free cells with fewer visits are more likely to be chosen. Default is `NULL` (uniform) |

## Value

Integer matrix of unique grid IDs, same dimension as `ids`

## See Also

Other Genetic Algorithm Functions: `crossover()`, `fitness()`, `genetic_algorithm()`, `init_population()`, `mutation()`, `selection()`, `set_crossover()`, `trimton()`

## Examples

```
ids <- cbind(c(1, 3, 5, 7), c(2, 4, 6, 8))
swap_mutation(ids, grid_ids = 1:20, p = 0.5, seed = 1)
```

`terrain_model` *Get terrainhic rasters*

## Description

Calculate the SpatRasters needed for the terrain model.

## Usage

```
terrain_model(
  terrain = TRUE,
  area,
  ccl,
  ccl_roughness,
  plot = FALSE,
  verbose = FALSE
)
```

## Arguments

| | |
|---|---|
| `terrain` | Terrain model (elevation + land cover). `TRUE` downloads a DEM via `elevatr`. Pass a DEM raster to skip the download. Per-cell values are computed once and stored in the result as `terrainModel` for `plot_result()` / `random_search()`. |
| `area` | Site polygon (`sf`, SpatialPolygons, or coordinate matrix). Must be projected in metres. |
| `ccl` | Path to a Corine Land Cover raster when `terrain` is on. |
| `ccl_roughness` | Path to the CLC legend CSV (`Rauhigkeit_z` column). |
| `plot` | Plot the elevation and roughness rasters |
| `verbose` | Print a line per generation. |

## Value

A list of SpatRasters

## Examples

```
## Not run:
library(sf)
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(4651704, 4651704, 4654475, 4654475, 4651704),
    c(2692925, 2694746, 2694746, 2692925, 2692925)
  ))),
  crs = 3035
))
Polygon_wgs84 <- sf::st_transform(area, st_crs(4326))
srtm <- elevatr::get_elev_raster(locations = Polygon_wgs84, z = 11)
```

```
res <- terrain_model(srtm, area)

## End(Not run)
```

---

`trimton` *Adjust the amount of turbines per windfarm*

---

## Description

Legacy repair for binary chromosomes. The GA loop encodes layouts as `n` unique grid IDs, so this function is not called there. It remains exported for the old 0/1 pipeline (`crossover` / `mutation`).

## Usage

```
trimton(mut, nturb, allparks, nGrids, trimForce, seed)
```

## Arguments

| | |
|---|---|
| `mut` | A binary matrix with the mutated individuals |
| `nturb` | A numeric value indicating the amount of required turbines |
| `allparks` | A data.frame consisting of all individuals of the current generation |
| `nGrids` | A numeric value indicating the total amount of grid cells |
| `trimForce` | If `TRUE`, add or drop turbines using fitness-weighted probabilities. If `FALSE`, choose cells at random. |
| `seed` | Set a seed for comparability. Default is NULL |

## Value

Returns a binary matrix with the correct amount of turbines per individual

## See Also

Other Genetic Algorithm Functions: `crossover()`, `fitness()`, `genetic_algorithm()`, `init_population()`, `mutation()`, `selection()`, `set_crossover()`, `swap_mutation()`

## Examples

```
## Create a random rectangular shapefile
library(sf)
area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(0, 0, 2000, 2000, 0),
    c(0, 2000, 2000, 0, 0)
  ))),
  crs = 3035
))
```

```
## Create a uniform and unidirectional wind data.frame and plots the
## resulting wind rose
## Uniform wind speed and single wind direction
data.in <- as.data.frame(cbind(ws = 12, wd = 0))

## Calculate a Grid and an indexed data.frame with coordinates and grid cell Ids.
Grid1 <- grid_area(area = area, size = 200, prop = 1)
Grid <- Grid1[[1]]
AmountGrids <- nrow(Grid)

startsel <- init_population(Grid, 10, 20)
wind <- as.data.frame(cbind(ws = 12, wd = 0))
wind <- list(wind, probab = 100)
fit <- fitness(
  population = startsel, reference_height = 100, rotor_height = 100,
  surface_roughness = 0.3, area = area, rotor = 20,
  wind = wind, terrain = FALSE
)
allparks <- do.call("rbind", fit)
## selection() returns ID matrices; crossover()/trimton() expect 0/1.
sel <- selection(fit, Grid, 2, TRUE, 6, "FIX")
ids <- sel[[1]]
bins <- matrix(0, nrow(Grid), ncol(ids))
for (j in seq_len(ncol(ids))) {
  bins[match(ids[, j], Grid[, "ID"]), j] <- 1
}
selec6best <- list(
  data.frame(ID = Grid[, "ID"], bins),
  data.frame(ID = 1, t(sel[[2]]))
)
crossOut <- crossover(selec6best, 2, uplimit = 300, crossPart = "RAN")
mut <- mutation(a = crossOut, p = 0.3, NULL)
mut1 <- trimton(
  mut = mut, nturb = 10, allparks = allparks, nGrids = AmountGrids,
  trimForce = FALSE
)
colSums(mut)
colSums(mut1)
```

---

`turbine_influences` *Find potentially influencing turbines*

---

## Description

Find all turbines that could potentially influence another turbine and save them to a list.

## Usage

```
turbine_influences(t, wnkl, dist, area, dirct, plot_angles = FALSE)
```

## Arguments

| | |
|---|---|
| t | A data.frame of the current individual with X and Y coordinates |
| wnkl | Wake opening angle in degrees. Turbines outside this cone are ignored. |
| dist | A numeric value indicating the distance, after which the wake effects are considered to be eliminated. |
| area | Site polygon |
| dirct | Current wind direction |
| plot_angles | Plot distances and angles |

## Value

Returns a list of all individuals of the current generation which could potentially influence other turbines. List includes the relevant coordinates, the distances and angles in between and assigns the Point ID.

## See Also

Other Wind Energy Calculation Functions: barometric_height(), calculate_energy(), circle_intersection(), get_dist_angles()

## Examples

```
## Exemplary input Polygon with 2km x 2km:
library(sf)

area <- sf::st_as_sf(sf::st_sfc(
  sf::st_polygon(list(cbind(
    c(0, 0, 2000, 2000, 0),
    c(0, 2000, 2000, 0, 0)
  ))),
  crs = 3035
))

t <- st_coordinates(st_sample(area, 10))
t <- cbind(t, "Z" = 1)
wnkl <- 20
dist <- 100000
dirct <- 0

res <- turbine_influences(t, wnkl, dist, area, dirct, plot_angles = TRUE)
```

## wake_cones *Downwind wake search cones*

### Description

Build sf polygons for the Jensen search cone of each turbine and wind direction. Colour can encode wake loss (AbschGesamt).

### Usage

```
wake_cones(xy, wind, rotor, area, half_deg = NULL, colors = NULL)
```

### Arguments

| | |
|---|---|
| xy | Matrix or data.frame of turbine X/Y in the site CRS. |
| wind | Wind table with wd (and optional probab), as stored in a genetic_algorithm() result. |
| rotor | Rotor radius in metres (sets cone length together with area). |
| area | Site polygon (sf), projected in metres. |
| half_deg | Half search angle. Default windfarmGA.max_angle. |
| colors | Optional colour per turbine (recycled). |

### Value

An sf polygon layer with turb, wd, prob, farbe.

## windata_format *Transform Winddata*

### Description

Helper Function, which transforms winddata to an acceptable format

### Usage

```
windata_format(df)
```

### Arguments

| | |
|---|---|
| df | The wind data with speeds, direction and optionally a probability column. If not assigned, it will be calculated |

### Value

A list of windspeed and probabilities

## See Also

Other Helper Functions: get_grids(), grid_area(), hexa_area(), isSpatial(), permutations(), read_power_curve(), splitAt(), wind_from_series(), wind_from_uv()

## Examples

```
wind_df <- data.frame(
  ws = c(12, 30, 45),
  wd = c(0, 90, 150),
  probab = 30:32
)
windata_format(wind_df)

wind_df <- data.frame(
  speed = c(12, 30, 45),
  direction = c(90, 90, 150),
  probab = c(10, 20, 60)
)
windata_format(wind_df)

wind_df <- data.frame(
  speed = c(12, 30, 45),
  direction = c(400, 90, 150)
)
windata_format(wind_df)
```

---

# wind_from_series *Wind rose from speed and direction series*

---

## Description

Bin a time series of hub-height speed and meteorological direction into ws / wd / probab.

## Usage

```
wind_from_series(ws, wd, dir_width = 30)
```

## Arguments

| | |
|---|---|
| ws | Wind speed (m/s). |
| wd | Direction in degrees (0 = north, clockwise, where the wind comes from). |
| dir_width | Width of direction bins in degrees. Default is 30. |

## Value

A data.frame with ws, wd, probab.

## See Also

Other Helper Functions: get_grids(), grid_area(), hexa_area(), isSpatial(), permutations(), read_power_curve(), splitAt(), wind_from_uv(), windata_format()

---

wind_from_uv *Wind rose from u/v components*

---

## Description

Bin ERA5-style eastward (u) and northward (v) wind components into the ws / wd / probab table that windata_format() and genetic_algorithm() expect. Direction is meteorological (where the wind comes from; 0 = north). Each direction bin gets the mean speed and the hour share.

## Usage

```
wind_from_uv(u, v, dir_width = 30)
```

## Arguments

| | |
|---|---|
| u | Eastward wind component (m/s). Same length as v. |
| v | Northward wind component (m/s). |
| dir_width | Width of direction bins in degrees. Default is 30. |

## Value

A data.frame with ws, wd, probab (probabilities sum to 100).

## See Also

Other Helper Functions: get_grids(), grid_area(), hexa_area(), isSpatial(), permutations(), read_power_curve(), splitAt(), wind_from_series(), windata_format()

## Examples

```
set.seed(1)
u <- rnorm(200, 2)
v <- rnorm(200, -3)
wind_from_uv(u, v, dir_width = 30)
```

# Index