Package ‘eRTG3D’

July 27, 2020

Title     Empirically Informed Random Trajectory Generation in 3-D
Version   0.6.3
URL       https://munterfinger.github.io/eRTG3D/, https://github.com/munterfinger/eRTG3D

Description Creates realistic random trajectories in a 3-D space between two given fix points, so-called conditional empirical random walks (CERWs). The trajectory generation is based on empirical distribution functions extracted from observed trajectories (training data) and thus reflects the geometrical movement characteristics of the mover. A digital elevation model (DEM), representing the Earth’s surface, and a background layer of probabilities (e.g. food sources, uplift potential, waterbodies, etc.) can be used to influence the trajectories.


Depends R (>= 3.5.0)
Imports CircStats (>= 0.2-6), ggplot2 (>= 3.1.1), pbapply (>= 1.4-1), plotly (>= 4.9.0), raster (>= 2.9-5), rasterVis (>= 0.45), tiff (>= 0.1-5)
License GPL (>= 3)
Encoding UTF-8
LazyData true
RoxygenNote 7.1.0
Suggests knitr (>= 1.23), pander (>= 0.6.3), gridExtra (>= 2.3), plyr (>= 1.8.4), rmarkdown (>= 1.13), sf (>= 0.7-4), sp (>= 1.3-1), testthat (>= 2.1.0), covr (>= 3.2.1)
VignetteBuilder knitr
NeedsCompilation no
Maintainer Merlin Unterfinger <info@munterfinger.ch>
BugReports https://github.com/munterfinger/eRTG3D/issues

Author Merlin Unterfinger [aut, cre] (<https://orcid.org/0000-0003-2020-2366>),
  Kamran Safi [ctb, ths],
  George Technitis [ctb, ths],
  Robert Weibel [ths]

Repository CRAN

Date/Publication 2020-07-26 23:10:18 UTC

R topics documented:

- chiMaps
- dem
- dem2track.extent
- dist2point.3d
- dist2target.3d
- eRTG3D
- filter.dead.ends
- get.densities.3d
- get.glideRatio.3d
- get.section.densities.3d
- get.track.densities.3d
- is.sf.3d
- lift2target.3d
- logRasterStack
- movingMedian
- n.sim.cond.3d
- n.sim.glidingSoaring.3d
- niclas
- parpbapply
- parpblapply
- parpbsapply
- plot2d
- plot3d
- plot3d.densities
- plot3d.multiplot
- plot3d.tldCube
- plotRaster
- qProb.3d
- reproduce.track.3d
- saveImageSlices
- sf2df.3d
- sim.cond.3d
- sim.crw.3d
- sim.glidingSoaring.3d
- sim.uncond.3d
- test.eRTG.3d
- test.verification.3d
chiMaps

Calculates the chi maps for one rasterStack or all raster all the raster pairs stored in two rasterStacks. As observed values, the first stack is used. The expected value is either set to the mean of the first stack, or if given to be the values of the second stack.

Usage

chiMaps(stack1, stack2 = NULL, verbose = FALSE)

Arguments

stack1 rasterStack

stack2 rasterStack NULL or containing the same number of rasterLayers and has equal extent and resolution.

verbose logical: print currently processed height band in raster stack?

Value

A rasterStack containing the chi maps.

Examples

chiMaps(raster::stack(dem))
dem

Example digital elevation model (DEM)

Description

This is data to be included in the package and can be used to test its functionality. The dem data is a RasterLayer and has a resolution of 90 meters. It is the topography of the Swiss midlands. The complete dataset can be downloaded directly from http://srtm.csi.cgiar.org/srtmdata/.

References

http://srtm.csi.cgiar.org/srtmdata/

dem2track.extent

Crops the DEM to the extent of the track with a buffer

Description

Crops the DEM to the extent of the track with a buffer

Usage

dem2track.extent(DEM, track, buffer = 100)

Arguments

DEM a raster containing a digital elevation model, covering the extent as the track
track data.frame with x,y,z coordinates of the original track
buffer buffer with, by default set to 100

Value

A the cropped digital elevation model as a raster layer.

Examples

dem2track.extent(dem, niclas)
dist2point.3d  

**Description**  
Distance of each track point to a given point

**Usage**  
dist2point.3d(track, point, groundDistance = FALSE)

**Arguments**  
- **track**: a list containing data.frames with x,y,z coordinates or a data.frame  
- **point**: a vector with x, y or x, y, z coordinates  
- **groundDistance**: logical: calculate only ground distance in x-y plane?

**Value**  
Returns the distance of each track point to the point.

**Examples**  
dist2point.3d(niclas, c(0,0,0))

dist2target.3d  

**Description**  
Calculates the distance between every point in the track and the last point (target).

**Usage**  
dist2target.3d(track)

**Arguments**  
- **track**: a track data.frame containing x, y and z coordinates

**Value**  
A numeric vector with the distances to target

**Examples**  
dist2target.3d(niclas)
Description

The empirically informed random trajectory generator in three dimensions (eRTG3D) is an algorithm to generate realistic random trajectories in a 3-D space between two given fix points in space, so-called Conditional Empirical Random Walks. The trajectory generation is based on empirical distribution functions extracted from observed trajectories (training data) and thus reflects the geometrical movement characteristics of the mover. A digital elevation model (DEM), representing the Earth’s surface, and a background layer of probabilities (e.g. food sources, uplift potential, waterbodies, etc.) can be used to influence the trajectories.

Details

See the packages site on GitHub, detailed information about the algorithm in this Master’s Thesis, or test the algoritm online in the eRTG3D Simulator.

filter.dead.ends

Description

Function to filter out tracks that have found a dead end

Usage

filter.dead.ends(cerwList)

Arguments

cerwList                  list of data.frames and NULL entries

Value

A list that is only containing valid tracks.

Examples

filter.dead.ends(list(niclas, niclas))
Description

Creates a list consisting of the three dimensional probability distribution cube for turning angle, lift angle and step length (turnLiftStepHist) as well as the uni-dimensional distributions of the differences of the turn angles, lift angles and step lengths with a lag of 1 to maintain minimal level of autocorrelation in each of the terms. Additionally also the distribution of the flight height over the ellipsoid (absolute) and the distribution of flight height over the topography (relative) can be included.

Usage

get.densities.3d(
  turnAngle,
  liftAngle,
  stepLength,
  deltaLift,
  deltaTurn,
  deltaStep,
  gradientAngle = NULL,
  heightEllipsoid = NULL,
  heightTopo = NULL,
  maxBin = 25
)

Arguments

- turnAngle: turn angles of the track (t)
- liftAngle: lift angles of the track (l)
- stepLength: step length of the track (d)
- deltaLift: auto differences of the turn angles (diff(t))
- deltaTurn: auto differences of the lift angles (diff(l))
- deltaStep: auto differences of the step length (diff(d))
- gradientAngle: NULL or the gradient angles of the track
- heightEllipsoid: flight height over the ellipsoid (absolute) or NULL to exclude this distribution
- heightTopo: flight height over the topography (relative) or NULL to exclude this distribution
- maxBin: numeric scalar, maximum number of bins per dimension of the tld-cube (turnLiftStepHist)

Value

A list containing the tldCube and the autodifferences functions (and additionally the flight height distribution functions)
Examples

```r
niclas <- track.properties.3d(niclas)[2:nrow(niclas),]
P <- get.densities.3d(turnAngle=niclas$t, liftAngle=niclas$l, stepLength=niclas$d,
  deltaY=diff(niclas$t), deltaY=diff(niclas$l), deltaY=diff(niclas$d),
  gradientAngle = NULL, heightEllipsoid = NULL, heightTopo = NULL, maxBin = 25)
```

get.glideRatio.3d  
Calculate glide ratio

Description

Calculates the ratio between horizontal movement and vertical movement. The value expresses the distance covered forward movement per distance movement in sinking.

Usage

```r
get.glideRatio.3d(track)
```

Arguments

- `track`  
a track data.frame containing x, y and z coordinates of a gliding section

Value

The ratio between horizontal and vertical movement.

Examples

```r
get.glideRatio.3d(niclas)
```

ges.section.densities.3d

Extract tldCube and autodifferences functions from track sections

Description

Creates a list consisting of the 3 dimensional probability distribution cube for turning angle, lift angle and step length (turnLiftStepHist) as well as the uni-dimensional distributions of the differences of the turning angles, lift angles and step lengths with a lag of 1 to maintain minimal level of autocorrelation in each of the terms.
get.track.densities.3d

Usage

get.section.densities.3d(
  trackSections,
  gradientDensity = TRUE,
  heightDistEllipsoid = TRUE,
  DEM = NULL,
  maxBin = 25
)

Arguments

trackSections    list of track sections got by the track.split.3d function
gradientDensity  logical: Should a distribution of the gradient angle be extracted and later used
                 in the simulations?
heightDistEllipsoid  logical: Should a distribution of the flight height over ellipsoid be extracted and
                          later used in the sim.cond.3d()?
DEM              a raster containing a digital elevation model, covering the same extent as the
                 track sections
maxBin           numeric scalar, maximum number of bins per dimension of the tld-cube (turnLiftStepHist)

Value

A list containing the tldCube and the autodifferences functions (and additionally the height distribution function)

Examples

g.get.section.densities.3d(list(niclas[1:10, ], niclas[11:nrow(niclas), ]))

g.track.densities.3d

Extract tldCube and autodifferences functions from a consistent track

Description

Get densities creates a list consisting of the 3 dimensional probability distribution cube for turning angle, lift angle and step length (turnLiftStepHist) as well as the uni-dimensional distributions of the differences of the turning angles, lift angles and step lengths with a lag of 1 to maintain minimal level of autocorrelation in each of the terms.
Usage

get.track.densities.3d(
    track,
    gradientDensity = TRUE,
    heightDistEllipsoid = TRUE,
    DEM = NULL,
    maxBin = 25
)

Arguments

- **track**: a data.frame with 3 columns containing the x,y,z coordinates
- **gradientDensity**: logical: Should a distribution of the gradient angle be extracted and later used in the simulations?
- **heightDistEllipsoid**: logical: Should a distribution of the flight height over ellipsoid be extracted and later used in the sim.cond.3d()?  
- **DEM**: a raster containing a digital elevation model, covering the same extent as the track
- **maxBin**: numeric scalar, maximum number of bins per dimension of the tld-cube (turnLiftStepHist)

Value

A list containing the tldCube and the autodifferences functions (and additionally the height distribution function)

Note

The time between the acquisition of fix points of the track must be constant, otherwise this leads to distorted statistic distributions, which increases the probability of dead ends. In this case please check track.split.3d and get.section.densities.3d

Examples

get.track.densities.3d(niclas, heightDist = TRUE)

---

is.sf.3d

Tests if the object is a simple feature collection (class: 'sf, data.frame')

Description

Tests if the object is a simple feature collection (class: 'sf, data.frame')
**Usage**

`is.sf.3d(track)`

**Arguments**

- `track` any object to test

**Value**

A logical: TRUE if is a simple feature collection (class: 'sf, data.frame') of the sf package, FALSE otherwise.

**Examples**

```r
is.sf.3d(niclas)
is.sf.3d(track2sf.3d(track = niclas, CRS = "+init=epsg:2056"))
```

---

**lift2target.3d**

*Lift angle to target*

**Description**

Calculates the lift angle between every point in the track and the last point (target).

**Usage**

`lift2target.3d(track)`

**Arguments**

- `track` a track data.frame containing x, y and z coordinates

**Value**

A numeric vector with the lift angles to target

**Examples**

```r
lift2target.3d(niclas)
```
logRasterStack  

*Converts a rasterStack to logarithmic scale*

**Description**

Avoids the problem of -Inf occurring for log(0).

**Usage**

\[
\text{logRasterStack}(rStack, \text{standartize} = \text{FALSE}, \text{InfVal} = \text{NA})
\]

**Arguments**

- `rStack`: rasterStack to convert to logarithmic scale
- `standartize`: logical: standartize cube between 0 and 1
- `InfVal`: the value that Inf and -Inf should be replaced with

**Value**

A rasterStack in logarithmic scale

**Examples**

\[
\text{logRasterStack}(\text{raster::stack(dem)})
\]

movingMedian  

*Moving median in one dimension*

**Description**

Applies a twosided moving median window on a vector, where the window parameter is the total size of the window. The value in the window middle is the index where the median of the window is written. Therefore the window size has to be an uneven number. The border region of the vector is filled with a one-sided median. There might be border effects.

**Usage**

\[
\text{movingMedian}(\text{data}, \text{window})
\]

**Arguments**

- `data`: numeric vector
- `window`: uneven number for the size of the moving window
n.sim.cond.3d

Value

A numeric vector.

Examples

movingMedian(sequence(1:10), window = 5)

n.sim.cond.3d  Conditional Empirical Random Walks (CERW) in 3-D

Description

Creates multiple conditional empirical random walks, with a specific starting and ending point, geometrically similar to the initial trajectory by applying sim.cond.3d multiple times.

Usage

n.sim.cond.3d(
  n.sim,
  n.locs,
  start = c(0, 0, 0),
  end = start,
  a0,
  g0,
  densities,
  qProbs,
  error = FALSE,
  parallel = FALSE,
  DEM = NULL,
  BG = NULL
)

Arguments

n.sim  number of CERWs to simulate
n.locs  length of the trajectory in locations
start  numeric vector of length 3 with the coordinates of the start point
end  numeric vector of length 3 with the coordinates of the end point
a0  initial incoming heading in radian
g0  initial incoming gradient/polar angle in radian
densities  list object returned by the get.densities.3d function
qProbs  list object returned by the qProb.3d function
error  logical: add random noise to the turn angle, lift angle and step length to account for errors measurements?
**Description**

Creates conditional empirical random walks in gliding mode, between a start and end point. The walk is performed on a MODE layer and, if provided, additionally on a background and digital elevation layer. The gliding is simulated with `sim.cond.3d` and soaring with `sim.uncond.3d`, therefore soaring is not restricted towards the target and can happen completely free as long as there are good thermal conditions. It is important to extract for every mode in the MODE raster layer a corresponding densities object with `get.densities.3d` and pass them to the function.

**Usage**

```r
n.sim.glidingSoaring.3d(
  n.sim = 1,
  parallel = FALSE,
  MODE,
  dGliding,
  dSoaring,
  qGliding,
  start = c(0, 0, 0),
  n.locs = 3,
  f <- 1500
)
```

**Examples**

```r
niclas <- track.properties.3d(niclas)
n.locs <- 3
P <- get.track.densities.3d(niclas)
f <- 1500
start <- Reduce(c, niclas[1, 1:3])
end <- Reduce(c, niclas[n.locs, 1:3])
a0 <- niclas$a[1]
g0 <- niclas$g[1]
uerw <- sim.uncond.3d(n.locs*f, start=start, a0=a0, g0=g0, densities=P)
Q <- qProb.3d(uerw, n.locs)
n.sim.cond.3d(n.sim=2, n.locs=n.locs, start=start, end=end, a0=a0, g0=g0, densities = P, qProbs = Q)
```

**Value**

A list containing the CERWs or `NULL`s if dead ends have been encountered.
end = start,
a0,
g0,
error = TRUE,
smoothTransition = TRUE,
glideRatio = 20,
DEM = NULL,
BG = NULL,
verbose = FALSE
)

Arguments

- **n.sim** number of simulations to produce
- **parallel** logical: run computations in parallel (n-1 cores)? Or numeric: the number of nodes (maximum: n - 1 cores)
- **MODE** raster layer containing the number/index of the mode, which should be used at each location
- **dGliding** density object returned by the `get.densities.3d` function for gliding mode
- **dSoaring** density object returned by the `get.densities.3d` function for soaring mode
- **qGliding** the Q probabilites for the steps in gliding mode (`qProb.3d`)
- **start** numeric vector of length 3 with the coordinates of the start point
- **end** numeric vector of length 3 with the coordinates of the end point
- **a0** initial incoming heading in radian
- **g0** initial incoming gradient/polar angle in radian
- **error** logical: add random noise to the turn angle, lift angle and step length to account for errors measurements?
- **smoothTransition** logical: should the transitions between soaring and the following gliding sections be smoothed? Recommended to avoid dead ends
- **glideRatio** ratio between vertical and horizontal movement, by default set to 15 meters forward movement per meter vertical movement
- **DEM** raster layer containing a digital elevation model, covering the area between start and end point
- **BG** a background raster layer that can be used to inform the choice of steps
- **verbose** logical: print current mode used?

Value

A list containing 'soaring-gliding' trajectories or NULLs if dead ends have been encountered.

Note

The MODE raster layer must be in the following structure: Gliding pixels have the value 1 and soaring pixel the values 2. NA’s are not allowed in the raster.
Examples

n.sim.glidingSoaring.3d(locsVec, start = c(0,0,0), end=start, a0, g0, dList, qList, MODE)

---

niclas  
*Example track data.frame*

---

Description

This is data to be included in the package and can be used to test its functionality. The track consists of x, y and z coordinates and represents the movement of a stork called niclas in the Swiss midlands.

References

[https://www.movebank.org](https://www.movebank.org)

---

parpbapply  
*Parallel apply with progressbar*

---

Description

Function detects the operating system and chooses the approximate kind of process for parallelizing the task: Windows: PSOCKCluster, Unix: Forking.

Usage

parpbapply(
  X,
  FUN,
  MARGIN,
  packages = NULL,
  export = NULL,
  envir = environment(),
  nNodes = parallel::detectCores() - 1
)
Arguments

X: an array, including a matrix.
FUN: function, the function to be applied to each element of X
MARGIN: a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where X has named dimnames, it can be a character vector selecting dimension names.
packages: character vector, Only relevant for Windows: the packages needed in the function provided, eg. c("MASS", "data.table")
export: character vector, Only relevant for Windows: the variables needed in the function provided, eg. c("df", "vec")
envir: environment, Only relevant for Windows: Environment from which the variables should be exported from
nNodes: numeric, Number of processes to start (unix: best to fit with the available Cores)

Value

Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

Examples

```r
n <- 1000
df <- data.frame(
  x = seq(1, n, 1),
  y = -seq(1, n, 1)
)
a <- parpbapply(X = df, FUN = sum, MARGIN = 1, nNodes = 2)
```

Description

Function detects the operating system and chooses the approximate kind of process for parallelizing the task: Windows: PSOCKCluster, Unix: Forking.

Usage

```r
parpbapply(
  X,
  FUN,
  packages = NULL,
  export = NULL,
  envir = environment(),
  nNodes = parallel::detectCores() - 1
)`
Arguments

X  a vector (atomic or list) or an expression object. Other objects (including classed objects) will be coerced by base::as.list

FUN  function, the function to be applied to each element of X

packages  character vector, Only relevant for Windows: the packages needed in the function provided, eg. c("MASS", "data.table")

export  character vector, Only relevant for Windows: the variables needed in the function provided, eg. c("df", "vec")

envir  environment, Only relevant for Windows: Environment from which the variables should be exported from

nNodes  numeric, Number of processes to start (unix: best to fit with the available Cores)

Value

A list with the results.

Examples

```r
square <- function(x){x*x}
l <- parpbsapply(X = 1:1000, FUN = square, export = c("square"), nNodes = 2)
```

Description

Function detects the operating system and chooses the approximate kind of process for parallelizing the task: Windows: PSOCKCluster, Unix: Forking.

Usage

```r
parpbsapply(
  X,
  FUN,
  packages = NULL,
  export = NULL,
  envir = environment(),
  nNodes = parallel::detectCores() - 1
)
```
plot2d

Arguments

X    a vector (atomic or list) or an expression object. Other objects (including classed objects) will be coerced by base::as.list.
FUN  function, the function to be applied to each element of X
packages character vector, Only relevant for Windows: the packages needed in the function provided, e.g. c("MASS", "data.table")
export character vector, Only relevant for Windows: the variables needed in the function provided, e.g. c("df", "vec")
envir environment, Only relevant for Windows: Environment from which the variables should be exported from
nNodes numeric, Number of processes to start (unix: best to fit with the available Cores)

Value

A vector with the results.

Examples

square <- function(x){x*x}
s <- parpbsapply(X = 1:1000, FUN = square, export = c("square"), nNodes = 2)

plot2d

Plot function to plot the 3-D tracks in 2-D plane

Description

Plot function to plot the 3-D tracks in 2-D plane

Usage

plot2d(
  origTrack,
  simTrack = NULL,
  titleText = character(1),
  DEM = NULL,
  BG = NULL,
  padding = 0.1,
  alpha = 0.7,
  resolution = 500
)
**plot3d**

Plot track(s) with a surface of a digital elevation model in three dimensions

**Usage**

```r
plot3d(
  origTrack,
  simTrack = NULL,
  titleText = character(1),
  DEM = NULL,
  padding = 0.1,
  timesHeight = 10
)
```

**Arguments**

- **origTrack**: a list containing data.frames with x,y,z coordinates or a data.frame
- **simTrack**: a list containing data.frames with x,y,z coordinates or a data.frame
- **titleText**: string with title of the plot
- **DEM**: an object of type RasterLayer, needs overlapping extent with the line(s)
- **padding**: adds a pad to the 2-D space in percentage (by default set to 0.1)
- **alpha**: a number between 0 and 1, to specify the transparency of the simulated line(s)
- **resolution**: number of pixels the rasters are downsampled to (by default set to 500 pixels)

**Value**

A ggplot2 object.

**Examples**

```r
plot2d(niclas)
```
**plot3d.densities**

**Value**
Plots a plotly object

**Examples**

```r
plot3d.densities(niclas)
```

---

**plot3d.densities**  
*Density plots of turn angle, lift angle and step length*

**Description**
The function takes either one track or two tracks. The second track can be a list of tracks (eg. the output of `n.sim.cond.3d`). Then the densities of turn angle, lift angle and step length of all the simulations is taken. Additionally the autodifferences parameter can be set to true, then the densities of the autodifferences in turn angle, lift angle and step length are visualized.

**Usage**

```r
plot3d.densities(
  track1,
  track2 = NULL,
  autodifferences = FALSE,
  scaleDensities = FALSE
)
```

**Arguments**

- `track1`: a list containing a data.frame with x,y,z coordinates or a data.frame
- `track2`: a list containing a data.frame with x,y,z coordinates or a data.frame
- `autodifferences`: logical: should the densities of the autodifferences in turn angle, lift angle and step length are visualized.
- `scaleDensities`: logical: should densities be scaled between 0 and 1, then sum of the area under the curve is not 1 anymore!

**Value**
A ggplot2 object.

**Examples**

```r
plot3d.densities(niclas)
```
plot3d.multiplot

Multiple plot function for ggplot objects

Description

If the layout is something like matrix(c(1,2,3,3),nrow=2,byrow=TRUE), then plot 1 will go in the upper left, 2 will go in the upper right, and 3 will go all the way across the bottom.

Usage

plot3d.multiplot(..., plotlist = NULL, cols = 1, layout = NULL)

Arguments

... ggplot objects
plotlist a list of ggplot objects
cols number of columns in layout
layout a matrix specifying the layout. If present, cols is ignored.

Value

Nothing, plots the ggplot2 objects.

Examples

plot3d.multiplot(plot2d(niclas), plot2d(niclas), plot2d(niclas))

plot3d.tldCube

Visualize turn-lift-step histogram

Description

Creates a three dimensional scatterplot of the possibles next steps, based on the tldCube, which was extracted from a track.

Usage

plot3d.tldCube(tldCube)

Arguments

tldCube tldCube; the ouput from turnLiftStepHist or get.densities.3d

Value

Plots a plotly object
**Examples**

```r
P <- get.track.densities.3d(niclas)
suppressWarnings(plot3d.tldCube(P$tldCube))
```

---

### plotRaster

**Plots a rasterLayer or rasterStack**

**Description**

Plots a rasterLayer or rasterStack

**Usage**

```r
plotRaster(r, title = character(0), centerColorBar = FALSE, ncol = NULL)
```

**Arguments**

- **r**: rasterLayer or rasterStack
- **title**: title text of plot(s)
- **centerColorBar**: logical: center colorbar around 0 and use RdBuTheme()?
- **ncol**: number of columns to plot a stack, by default estimated by the square root

**Value**

Plots the rasters

**Examples**

```r
plotRaster(dem)
```

---

### qProb.3d

**Q probabilities for n steps**

**Description**

Calculates the Q probability, representing the pull to the target. The number of steps on which the Q prob will be quantified is number of total segments less than one (the last step is defined by the target itself).

**Usage**

```r
qProb.3d(sim, n.locs, parallel = FALSE, maxBin = 25)
```
Arguments

- **sim**: the result of `sim.uncond.3d`, or a data frame with at least x,y,z-coordinates, the arrival azimuth and the arrival gradient.
- **n.locs**: number of total segments to be modeled, the length of the desired conditional empirical random walk.
- **parallel**: logical: run computations in parallel (n-1 cores)? Or numeric: the number of nodes (maximum: n - 1 cores).
- **maxBin**: numeric scalar, maximum number of bins per dimension of the tld-cube (turn-LiftStepHist).

Value

A list containing the Q - tldCubes for every step.

Examples

```r
gProb.3d(niclas, 3)
```

---

**reproduce.track.3d**  
Reproduce a track with the eRTG3D

---

Description

Simulates n tracks with the geometrical properties of the original track, between the same start and end point.

Usage

```r
reproduce.track.3d(
  track,
  n.sim = 1,
  parallel = FALSE,
  error = TRUE,
  DEM = NULL,
  BG = NULL,
  filterDeadEnds = TRUE,
  plot2d = FALSE,
  plot3d = FALSE,
  maxBin = 25,
  gradientDensity = TRUE
)
```
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>track</td>
<td>data.frame with x,y,z coordinates of the original track</td>
</tr>
<tr>
<td>n.sim</td>
<td>number of simulations that should be done</td>
</tr>
<tr>
<td>parallel</td>
<td>logical: run computations in parallel (n-1 cores)? Or numeric: the number of</td>
</tr>
<tr>
<td></td>
<td>nodes (maximum: n - 1 cores)</td>
</tr>
<tr>
<td>error</td>
<td>logical: add error term to movement in simulation?</td>
</tr>
<tr>
<td>DEM</td>
<td>a raster containing a digital elevation model, covering the same extent as</td>
</tr>
<tr>
<td></td>
<td>the track</td>
</tr>
<tr>
<td>BG</td>
<td>a raster influencing the probabilities.</td>
</tr>
<tr>
<td>filterDeadEnds</td>
<td>logical: Remove tracks that ended in a dead end?</td>
</tr>
<tr>
<td>plot2d</td>
<td>logical: plot tracks on 2-D plane?</td>
</tr>
<tr>
<td>plot3d</td>
<td>logical: plot tracks in 3-D?</td>
</tr>
<tr>
<td>maxBin</td>
<td>numeric scalar, maximum number of bins per dimension of the tld-cube (turn-</td>
</tr>
<tr>
<td></td>
<td>LiftStepHist)</td>
</tr>
<tr>
<td>gradientDensity</td>
<td>logical: Should a distribution of the gradient angle be extracted and used in</td>
</tr>
<tr>
<td></td>
<td>the simulations (get.densities.3d)?</td>
</tr>
</tbody>
</table>

Value

A list or data.frame containing the simulated track(s) (CERW).

Examples

```r
reproduce.track.3d(niclas[1:10, ])
```

Description

Exports a dataCube of type `rasterStack` as Tiff image sequence. Image sequences are a common structure to represent voxel data and most of the specific software to visualize voxel data is able to read it (e.g. blender)

Usage

```r
saveImageSlices(rStack, filename, dir, NaVal = 0)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rStack</td>
<td>rasterStack to be saved to Tiff image slices</td>
</tr>
<tr>
<td>filename</td>
<td>name of the image slices</td>
</tr>
<tr>
<td>dir</td>
<td>directory, where the slices should be stored</td>
</tr>
<tr>
<td>NaVal</td>
<td>numeric value that should represent NA values in the Tiff image, default is</td>
</tr>
<tr>
<td></td>
<td>NaVal = 0</td>
</tr>
</tbody>
</table>
Value

Saves the Tiff image files.

Examples

```r
crws <- lapply(X=seq(1:100), FUN = function(X) {
  sim.crw.3d(nStep = 100, rTurn = 0.99, rLift = 0.99, meanStep = 0.1)
})
points <- do.call("rbind", crws)
extent <- raster::extent(c(-10, 10, -10, 10))
ud <- voxelCount(points, extent, xyRes=5,
  zMin=-10, zMax=10, standartize = TRUE)
saveImageSlices(ud, filename = "saveImageSlices_test", dir = tempdir())
```

---

**sf2df.3d**

Converts a sf data.frame to a normal dataframe

Description

Converts a sf data.frame to a normal dataframe

Usage

```r
sf2df.3d(track)
```

Arguments

- `track` An object of type ‘sf.data.frame’

Value

A data.frame.

Examples

```r
sf2df.3d(track2sf.3d(niclas, "+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs"))
```
sim.cond.3d

Conditional Empirical Random Walk (CERW) in 3-D

Description

Creates a conditional empirical random walk, with a specific starting and ending point, geometrically similar to the initial trajectory (extractMethod: raster overlay method can take "simple" or "bilinear")

Usage

sim.cond.3d(
  n.locs,
  start = c(0, 0, 0),
  end = start,
  a0,
  g0,
  densities,
  qProbs,
  error = FALSE,
  DEM = NULL,
  BG = NULL
)

Arguments

n.locs length of the trajectory in locations
start numeric vector of length 3 with the coordinates of the start point
end numeric vector of length 3 with the coordinates of the end point
a0 initial incoming heading in radian
g0 initial incoming gradient/polar angle in radian
densities list object returned by the get.densities.3d function
qProbs list object returned by the qProb.3d function
error logical: add random noise to the turn angle, lift angle and step length to account for errors measurements?
DEM raster layer containing a digital elevation model, covering the area between start and end point
BG a background raster layer that can be used to inform the choice of steps

Value

A trajectory in the form of data.frame
Examples

```r
niclas <- track.properties.3d(nicas)
n.locs <- 3
P <- get.track.densities.3d(niclas)
f <- 1500
start <- Reduce(c, niclas[1, 1:3])
end <- Reduce(c, niclas[n.locs, 1:3])
a0 <- niclas$a[1]
g0 <- niclas$g[1]
uerw <- sim.uncond.3d(n.locs*f, start=start, a0=a0, g0=g0, densities=P)
Q <- qProb.3d(uerw, n.locs)
sim.cond.3d(n.locs=n.locs, start=start, end=end, a0=a0, g0=g0, densities = P, qProbs = Q)
```

---

**sim.crw.3d**

*Simulation of a three dimensional Correlated Random Walk*

**Description**

Simulation of a three dimensional Correlated Random Walk

**Usage**

```r
sim.crw.3d(nStep, rTurn, rLift, meanStep, start = c(0, 0, 0))
```

**Arguments**

- `nStep` the number of steps of the simulated trajectory
- `rTurn` the correlation on the turn angle
- `rLift` the correlation of the lift angle
- `meanStep` the mean step length
- `start` a vector of length 3 containing the coordinates of the start point of the trajectory

**Value**

A trajectory in the form of data.frame

**Examples**

```r
sim.crw.3d(nStep=10, rTurn=0.9, rLift=0.9, meanStep=1, start = c(0,0,0))
```
sim.glidingSoaring.3d  Simulates 'gliding & soaring' track with a given number of gliding steps

Description

Creates a conditional empirical random walk in gliding mode, between a start and end point. The walk is performed on a MODE layer and, if provided, additionally on a background and digital elevation layer. The gliding is simulated with sim.cond.3d and soaring with sim.uncond.3d, therefore soaring is not restricted towards the target and can happen completely free as long as there are good thermal conditions. It is important to extract for every mode in the MODE raster layer a corresponding densities object with get.densities.3d and pass them to the function.

Usage

```r
sim.glidingSoaring.3d(
  MODE,
  dGliding,
  dSoaring,
  qGliding,
  start = c(0, 0, 0),
  end = start,
  a0,
  g0,
  error = TRUE,
  smoothTransition = TRUE,
  glideRatio = 15,
  DEM = NULL,
  BG = NULL,
  verbose = FALSE
)
```

Arguments

- **MODE**: raster layer containing the number/index of the mode, which should be used at each location
- **dGliding**: density object returned by the `get.densities.3d` function for gliding mode
- **dSoaring**: density object returned by the `get.densities.3d` function for soaring mode
- **qGliding**: the Q probabilities for the steps in gliding mode (qProb.3d)
- **start**: numeric vector of length 3 with the coordinates of the start point
- **end**: numeric vector of length 3 with the coordinates of the end point
- **a0**: initial incoming heading in radian
- **g0**: initial incoming gradient/polar angle in radian
- **error**: logical: add random noise to the turn angle, lift angle and step length to account for errors measurements?
smoothTransition logical: should the transitions between soaring and the following gliding sections be smoothed? Recommended to avoid dead ends

glideRatio ratio between vertical and horizontal movement, by default set to 15 meters forward movement per meter vertical movement

DEM raster layer containing a digital elevation model, covering the area between start and end point

BG a background raster layer that can be used to inform the choice of steps

verbose logical: print current mode used?

Value

A 'soaring-gliding' trajectory in the form of data.frame

Note

The MODE raster layer must be in the following structure: Gliding pixels have the value 1 and soaring pixel the values 2. NA’s are not allowed in the raster.

Examples

sim.glidingSoaring.3d(locsVec, start = c(0,0,0), end=start, a0, g0, dList, qList, MODE)

sim.uncond.3d(n.locs, start = c(0, 0, 0), a0, g0, densities, error = TRUE)

Description

This function creates unconditional walks with prescribed empirical properties (turning angle, lift angle and step length and the auto-differences of them. It can be used for unconditional walks or to seed the conditional walks with comparably long simulations. The conditional walk connecting a given start with a certain end point by a given number of steps needs an attraction term (the Q probability, see qProb.3d) to ensure that the target is approached and hit. In order to calculate the Q probability for each step the distribution of turns and lifts to target and the distribution of distance to target has to be known. They can be derived from the empirical data (ideally), or estimated from an unconditional process with the same properties. Creates a unconditional empirical random walk, with a specific starting point, geometrically similar to the initial trajectory.

Usage

sim.uncond.3d(n.locs, start = c(0, 0, 0), a0, g0, densities, error = TRUE)
test.eRTG.3d

Arguments

- **n.locs**: the number of locations for the simulated track
- **start**: vector indicating the start point \( c(x, y, z) \)
- **\(a0\)**: initial heading in radian
- **\(g0\)**: initial gradient/polar angle in radian
- **densities**: list object returned by the `get.densities.3d` function
- **error**: logical: add random noise to the turn angle, lift angle and step length to account for errors measurements?

Value

A 3 dimensional trajectory in the form of a data.frame

Note

Simulations connecting start and end points with more steps than 1/10th or more of the number of steps of the empirical data should rather rely on simulated unconditional walks with the same properties than on the empirical data (factor = 1500).

Random initial heading

For a random initial heading \(a0\) use: `sample(atan2(diff(coordinates(track)[,2]),diff(coordinates(track)[,1])),1)`

Examples

```r
sim.uncond.3d(10, start=c(0,0,0), a0=pi/2, g0=pi/2, densities=get.track.densities.3d(niclas))
```

descriptor line

---

test.eRTG.3d  

*Test the functionality of the eRTG3D*

Description

The test simulates a CRW with given parameters and reconstructs it by using the eRTG3D

Usage

```r
test.eRTG.3d(
    parallel = FALSE,
    returnResult = FALSE,
    plot2d = FALSE,
    plot3d = TRUE,
    plotDensities = TRUE
)
```
test.verification.3d

Arguments

- parallel logical: test running parallel?
- returnResult logical: return tracks generated?
- plot2d logical: plot tracks on 2-D plane?
- plot3d logical: plot tracks in 3-D?
- plotDensities logical: plot densities of turning angle, lift angle and step length?

Value

A list containing the original CRW and the simulated track (CERW).

Examples

```r
test.eRTG.3d()
```

---

**test.verification.3d**  
*Statistical Verification of the simulated track*

Description

Uses two-sample Kolmogorov-Smirnov test or the one-sample t-test to compare the geometric characteristics of the original track with the characteristics of the simulated track.

Usage

```r
test.verification.3d(track1, track2, alpha = 0.05, plot = FALSE, test = "ks")
```

Arguments

- track1  data.frame or list of data.frames with x,y,z coordinates of the original track
- track2  data.frame or list of data.frames with x,y,z coordinates of the simulated track
- alpha  scalar: significance level, default alpha = 0.05
- plot logical: plot the densities or differences of turn angle, lift angle and step length of the two tracks?
- test character: either "ks" or "ttest" to choose the kind of test procedure.

Value

Test objects of the 6 two-sample Kolmogorov-Smirnov test conducted.
**Note**

By choosing `test = "ttest"` a random sample, without replacement is taken from the longer track, to shorten it to the length of the longer track. The order of the shorter track is also sampled randomly. Then the two randomly ordered vectors of turn angles, lift angles and step lengths are subtracted from each other. If the both tracks stem from the same distributions the the mean deviation should tend towards zero, therefore the difference is tested two-sided against `mu = 0` with a one-sample t-test.

By setting `test = "ks"` a two-sample Kolmogorov-Smirnov test is carried out on the distributions of turn angles, lift angles and step lengths of the two tracks.

**Examples**

```r
track.extent(niclas, test = "ks")
```

---

<table>
<thead>
<tr>
<th>track.extent</th>
<th>Extent of track(s)</th>
</tr>
</thead>
</table>

**Description**

Extent of track(s)

**Usage**

```r
track.extent(track, zAxis = FALSE)
```

**Arguments**

- `track` a list containing data.frames with x,y,z coordinates or a data.frame
- `zAxis` logical: return also the extent of the Z axis?

**Value**

Returns an extent object of the raster package in the 2–D case and a vector in the 3–D case.

**Examples**

```r
track.extent(niclas, zAxis = TRUE)
```
track.properties.3d  Track properties of a 3-D track

Description
Returns the properties (distances, azimuth, polar angle, turn angle & lift angle) of a track in three dimensions.

Usage
track.properties.3d(track)

Arguments
- track: data.frame with x,y,z coordinates

Value
The data.frame with track properties

Examples
track.properties.3d(niclas)

track.split.3d  This function splits the by outliers in the time lag.

Description
The length of timeLag must be the the track’s length minus 1 and represents the time passed between the fix point acquisition.

Usage
track.split.3d(track, timeLag, lag = NULL, tolerance = NULL)

Arguments
- track: track data.frame with x, y and z coordinates
- timeLag: a numeric vector with the time passed between the fix point acquisition
- lag: NULL or a manually chosen lag
- tolerance: NULL or a manually chosen tolerance

Value
A list containing the splitted tracks.
track2sf.3d

Examples

track.split.3d(niclas, timeLag=rep(1, nrow(niclas)-1) + rnorm(nrow(niclas)-1, mean = 0, sd = 0.25))

---

**track2sf.3d**

Converts a track to a 'sf, data.frame'

**Description**

Converts a track to a 'sf, data.frame'

**Usage**

track2sf.3d(track, CRS = NA)

**Arguments**

- **track**: eRTG3D track data.frame or a matrix
- **CRS**: string containing the proj4 code of the CRS

**Value**

A track of type 'sf, data.frame'.

**Examples**

track2sf.3d(niclas, "+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs")

---

**transformCRS.3d**

Transform coordinates reference system (CRS) of a 3-D track

**Description**

Attention: Please use this function for CRS transformations, since it is based on the `st_transform` from the sf package and therefore supports CRS transformations in 3-D. Note: `spTransform` from the sp package only supports transformations in the 2D plane, which will cause distortions in the third dimension.

**Usage**

transformCRS.3d(track, fromCRS, toCRS)

**Arguments**

- **track**: data.frame with x,y,z coordinates
- **fromCRS**: string: proj4 of current CRS
- **toCRS**: string: proj4 of CRS to be converted in
Value
A data.frame containing x,y,z and variables.

Examples
transformCRS.3d(niclas, fromCRS="+init=epsg:2056", toCRS="+init=epsg:4326")

turn2target.3d  Turn angle to target

Description
Calculates the turn angle between every point in the track and the last point (target).

Usage
turn2target.3d(track)

Arguments
track a track data.frame containing x, y and z coordinates

Value
A numeric vector with the turn angles to target

Examples
turn2target.3d(niclas)

turnLiftStepHist  Three dimensional histogram

Description
Derives a three dimensional distribution of a turn angle, lift angle and step length, using the Freedman–Diaconis rule for estimating the number of bins.

Usage
turnLiftStepHist(
  turn,
  lift,
  step,
  printDims = TRUE,
  rm.zeros = TRUE,
  maxBin = 25
)
**Arguments**

- **turn**: numeric vector of turn angles
- **lift**: numeric vector of lift angles
- **step**: numeric vector of step lengths
- **printDims**: logical: should dimensions of tld-Cube be messaged?
- **rm.zeros**: logical: should combinations with zero probability be removed?
- **maxBin**: numeric scalar, maximum number of bins per dimension of the tld-cube.

**Value**

A three dimensional histogram as data.frame

**Examples**

```r
niclas <- track.properties.3d(niclas)[2:nrow(niclas),]
turnLiftStepHist(niclas$t, niclas$l, niclas$d)
```

**Description**

A rasterStack object is created, representing the 3-D voxel cube. The z axis is sliced into regular sections between the maximum and minimum value. For every height slice a raster with points per cell counts is created. Additionally the voxels can be standartized between 0 and 1.

**Usage**

```r
voxelCount(
  points,  
extent, 
  xyRes, 
  zRes = xyRes, 
  zMin, 
  zMax, 
  standartize = FALSE, 
  verbose = FALSE 
)
```

**Arguments**

- **points**: a x, y, z data.frame
- **extent**: a raster extent object of the extent to create the rasters
- **xyRes**: resolution in the ground plane of the created rasters
- **zRes**: resolution in the z axis (by default zRes = xyRes)
voxelCount

zMin  minimum z value
zMax  maximum height value
standartize  logical: standartize the values?
verbose  logical: print currently processed height band in raster stack?

Value

A rasterStack object, representing the 3–D voxel cube.

Examples

voxelCount(niclas, raster::extent(dem), 100, 100, 1000, 1400, standartize = TRUE)
Index

* data
  dem, 4
  niclas, 16
chiMaps, 3
dem, 4
dem2track.extent, 4
dist2point.3d, 5
dist2target.3d, 5
eRTG3D, 6
filter.dead.ends, 6
get.densities.3d, 7, 13–15, 22, 25, 27, 29, 31
get.glideRatio.3d, 8
get.section.densities.3d, 8, 10
get.track.densities.3d, 9
is.sf.3d, 10
lift2target.3d, 11
logRasterStack, 12
movingMedian, 12
n.sim.cond.3d, 13, 21
n.sim.glidingSoaring.3d, 14
niclas, 16
parpbapply, 16
parpbapply, 17
parpbsapply, 18
plot2d, 19
plot3d, 20
plot3d.densities, 21
plot3d.multiplot, 22
plot3d.tldCube, 22
plotRaster, 23
qProb.3d, 13, 15, 23, 27, 29, 30
reproduce.track.3d, 24
saveImageSlices, 25
sf2df.3d, 26
sim.cond.3d, 13, 14, 27, 29
sim.crw.3d, 28
sim.glidingSoaring.3d, 29
sim.uncond.3d, 14, 24, 29, 30
st_transform, 35
test.eRTG.3d, 31
test.verification.3d, 32
track.extent, 33
track.properties.3d, 34
track.split.3d, 9, 10, 34
track2sf.3d, 35
transformCRS.3d, 35
turn2target.3d, 36
turnLiftStepHist, 7–10, 22, 24, 25, 36
voxelCount, 37