

# Package ‘eRTG3D’

May 25, 2019

**Title** Empirically Informed Random Trajectory Generation in 3-D

**Version** 0.5.9

**URL** <https://github.com/munterfinger/eRTG3D>

**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. 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.

**Depends** R (>= 3.6.0)

**Imports** CircStats (>= 0.2-6), doParallel (>= 1.0.14), ggplot2 (>= 3.1.1), gridExtra (>= 2.3), parallel (>= 3.6.0), pbmcapply (>= 1.4.1), plyr (>= 1.8.4), plotly (>= 4.9.0), raster (>= 2.9-5), rasterVis (>= 0.45), sp (>= 1.3-1), tiff (>= 0.1-5)

**License** GPL (>=3)

**Encoding** UTF-8

**LazyData** true

**RoxygenNote** 6.1.1

**Suggests** knitr (>= 1.23), pander (>= 0.6.3), rmarkdown (>= 1.13), sf (>= 0.7-4)

**VignetteBuilder** knitr

**Author** Merlin Unterfinger [aut, cre],  
Kamran Safi [aut],  
George Technitis [aut],  
Robert Weibel [aut]

**Maintainer** Merlin Unterfinger <info@munterfinger.ch>

## R topics documented:

chiMaps . . . . .	2
dem . . . . .	3
dem2track.extent . . . . .	3
dist2point.3d . . . . .	4
dist2target.3d . . . . .	4
eRTG3D . . . . .	5
filter.dead.ends . . . . .	5

get.densities.3d . . . . .	6
get.glideRatio.3d . . . . .	7
get.section.densities.3d . . . . .	7
get.track.densities.3d . . . . .	8
is.sf.3d . . . . .	9
lift2target.3d . . . . .	9
logRasterStack . . . . .	10
movingMedian . . . . .	10
n.sim.cond.3d . . . . .	11
n.sim.glidingSoaring.3d . . . . .	12
niclas . . . . .	13
plot2d . . . . .	13
plot3d . . . . .	14
plot3d.densities . . . . .	15
plot3d.multiplot . . . . .	15
plot3d.tldCube . . . . .	16
plotRaster . . . . .	17
qProb.3d . . . . .	17
reproduce.track.3d . . . . .	18
saveImageSlices . . . . .	19
sf2df.3d . . . . .	19
sim.cond.3d . . . . .	20
sim.crw.3d . . . . .	21
sim.glidingSoaring.3d . . . . .	21
sim.uncond.3d . . . . .	22
test.eRTG.3d . . . . .	23
test.verifcation.3d . . . . .	24
track.extent . . . . .	25
track.properties.3d . . . . .	25
track.split.3d . . . . .	26
track2sf.3d . . . . .	26
transformCRS.3d . . . . .	27
turn2target.3d . . . . .	27
turnLiftStepHist . . . . .	28
voxelCount . . . . .	28

## Index 30

---

chiMaps

*Chi maps of two variables*

---

### Description

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)
```

**Arguments**

stack1	rasterStack
stack2	rasterStack NULL or containing the same number of rasterLayers and has euqal extent and resolution.

**Value**

A rasterStack containing the chi maps.

**Examples**

```
chiMaps(raster::stack(dem))
```

---

dem	<i>Example digital elevation model (DEM)</i>
-----	--

---

**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 [www.cgiar-csi.org](http://www.cgiar-csi.org).

**References**

<http://www.cgiar-csi.org/data/>

---

dem2track.extent	<i>Crops the DEM to the extent of the track with a buffer</i>
------------------	---

---

**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	<i>Distance of each track point to a given point</i>
---------------	--

---

**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	<i>Distance to target</i>
----------------	---------------------------

---

**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)
```

eRTG3D

*eRTG3D: Empirically Informed Random Trajectory Generator in 3-D***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. 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.

**Details**

See the README on [GitHub](#), detailed information about the algorithm in this [Master's Thesis](#), or test the algorithm online in the [eRTG3D Simulator](#)

`filter.dead.ends`*Remove 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))
```

---

get.densities.3d

---

*Extract tldCube and autodifference approximation functions*


---

## 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	stepLength 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 ( <a href="#">turnLiftStepHist</a> )

## Value

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

## Examples

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

---

get.glideRatio.3d	<i>Calculate glide ratio</i>
-------------------	------------------------------

---

**Description**

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

**Usage**

```
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**

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

---

get.section.densities.3d	<i>Extract tldCube and autodifferences functions from track sections</i>
--------------------------	--

---

**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.

**Usage**

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

**Arguments**

trackSections	list of track sections got by the <a href="#">track.split.3d</a> 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 ( <a href="#">turnLiftStepHist</a> )

**Value**

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

**Examples**

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

---

```
get.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 ( <a href="#">turnLiftStepHist</a> )

**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	<i>Tests if the object is a simple feature collection (class: 'sf', data.frame')</i>
----------	--

---

**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**

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

---

lift2target.3d	<i>Lift angle to target</i>
----------------	-----------------------------

---

**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**

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

---

logRasterStack	<i>Converts a rasterStack to logarithmic scale</i>
----------------	--

---

**Description**

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

**Usage**

```
logRasterStack(rStack, standartize = FALSE, InfVal = 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 rpeplaced with

**Value**

A rasterStack in logarithmic scale

**Examples**

```
logRasterStack(raster::stack(dem))
```

---

movingMedian	<i>Moving median in one dimension</i>
--------------	---------------------------------------

---

**Description**

Applies a twosided moving median window on a vector, where the window paramter 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 vetor is filled with a one-sided median. There might be border effects.

**Usage**

```
movingMedian(data, window)
```

**Arguments**

data	numeric vector
window	uneven number for the size of the moving window

**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, multicore = 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 <a href="#">get.densities.3d</a> function
qProbs	list object returned by the <a href="#">qProb.3d</a> function
error	logical: add random noise to the turn angle, lift angle and step length to account for errors measurements?
multicore	logical: run computations in parallel (n-1 cores)?
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 list containing the CERWs or NULLs if dead ends have been encountered.

## Examples

```

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)

```

---

n.sim.glidingSoaring.3d

*Simulates multiple 'gliding & soaring' tracks with a given number of gliding steps*

---

## 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

```

n.sim.glidingSoaring.3d(n.sim = 1, multicore = FALSE, MODE, dGliding,
  dSoaring, qGliding, start = c(0, 0, 0), end = start, a0, g0,
  error = TRUE, smoothTransition = TRUE, glideRatio = 20,
  DEM = NULL, BG = NULL)

```

## Arguments

n.sim	number of simulations to produce
multicore	logical: should simulations be spread to the available number of cores?
MODE	raster layer containing the number/index of the mode, which should be used at each location
dGliding	density object returned by the <a href="#">get.densities.3d</a> function for gliding mode
dSoaring	density object returned by the <a href="#">get.densities.3d</a> function for soaring mode
qGliding	the Q probabilities for the steps in gliding mode ( <a href="#">qProb.3d</a> )
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

### 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

```
## Not run:
n.sim.glidingSoaring.3d(locsVec, start = c(0,0,0), end=start, a0, g0, dList, qList, MODE)

## End(Not run)
```

---

niclas	<i>Example track data.frame</i>
--------	---------------------------------

---

### 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>

---

plot2d	<i>Plot function to plot the 3-D tracks in 2-D plane</i>
--------	--

---

### 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)
```

**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)
BG	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**

```
plot2d(niclas)
```

---

plot3d	<i>Plot track(s) with a surface of a digital elevation model in three dimensions</i>
--------	--

---

**Description**

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

**Usage**

```
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)
timesHeight	multiply the height scale by a scalar (by default set to 10)

**Value**

Plots a plotly object

**Examples**

```
plot3d(niclas)
```

---

plot3d.densities	<i>Density plots of turn angle, lift angle and step length</i>
------------------	--

---

### 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

```
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

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

---

plot3d.multiplot	<i>Multiple plot function for ggplot objects</i>
------------------	--

---

### 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**

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

**Value**

Nothing, plots the ggplot2 objects.

**Examples**

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

---

<code>plot3d.tldCube</code>	<i>Visualize turn-lift-step histogram</i>
-----------------------------	---

---

**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**

<code>tldCube</code>	<code>tldCube</code> ; the ouptut from <a href="#">turnLiftStepHist</a> or <a href="#">get.densities.3d</a>
----------------------	---

**Value**

Plots a plotly object

**Examples**

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



---

plotRaster	<i>Plots a rasterLayer or rasterStack</i>
------------	---

---

**Description**

Plots a rasterLayer or rasterStack

**Usage**

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

**Arguments**

r	rasterLayer or rasterStack
title	title text of plot(s)
centerColorBar	logical: center colobar 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**

```
plotRaster(dem)
```

---

qProb.3d	<i>Q probabilities for n steps</i>
----------	------------------------------------

---

**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**

```
qProb.3d(sim, n.locs, multicore = FALSE, maxBin = 25)
```

**Arguments**

sim	the result of <a href="#">sim.uncond.3d</a> , 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
multicore	logical: run computations in parallel (n-1 cores)?
maxBin	numeric scalar, maximum number of bins per dimension of the tld-cube ( <a href="#">turn-LiftStepHist</a> )

**Value**

A list containing the Q - tldCubes for every step

**Examples**

```
qProb.3d(niclas, 3)
```

---

reproduce.track.3d	<i>Reproduce a track with the eRTG3D</i>
--------------------	--

---

**Description**

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

**Usage**

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

**Arguments**

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

**Value**

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

**Examples**

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

---

saveImageSlices	<i>Export a dataCube as image slice sequence</i>
-----------------	--

---

**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**

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

**Arguments**

rStack	rasterStack to be saved to Tiff image slices
filename	name of the image slices
dir	directory, where the slices should be stored
NaVal	numeric value that should represent NA values in the Tiff image, default is NaVal = 0

**Value**

Saves the Tiff image files.

**Examples**

```
## Not run:
saveImageSlices(rstack, filename = "image")

## End(Not run)
```

---

sf2df.3d	<i>Converts a sf data.frame to a normal dataframe</i>
----------	---

---

**Description**

Converts a sf data.frame to a normal dataframe

**Usage**

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

**Arguments**

track	An object of type 'sf, data.frame'
-------	------------------------------------

**Value**

A data.frame.

## Examples

```
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 <a href="#">get.densities.3d</a> function
qProbs	list object returned by the <a href="#">qProb.3d</a> 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

```
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)
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**

```
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**

```
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**

```
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)
```

## Arguments

MODE	raster layer containing the number/index of the mode, which should be used at each location
dGliding	density object returned by the <a href="#">get.densities.3d</a> function for gliding mode
dSoaring	density object returned by the <a href="#">get.densities.3d</a> function for soaring mode
qGliding	the Q probabilities for the steps in gliding mode ( <a href="#">qProb.3d</a> )
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

## 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

```
## Not run:
sim.glidingSoaring.3d(locsVec, start = c(0,0,0), end=start, a0, g0, dList, qList, MODE)

## End(Not run)
```

---

sim.uncond.3d

---

*Unconditional Empirical Random Walk (UERW) in 3-D*


---

## 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)
```

**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 <a href="#">get.densities.3d</a> 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)[,`

**Examples**

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

---

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**

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

**Arguments**

multicore	logical: test with multicore?
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**

```
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**

```
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 deviatio should tend to 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**

```
test.verification.3d(niclas, niclas)
```



---

track.extent	<i>Extent of track(s)</i>
--------------	---------------------------

---

**Description**

Extent of track(s)

**Usage**

```
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**

```
track.extent(niclas, zAxis = TRUE)
```

---

track.properties.3d	<i>Track properties of a 3-D track</i>
---------------------	--

---

**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	<i>This function splits the by outliers in the time lag.</i>
----------------	--

---

**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.

**Examples**

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

---

track2sf.3d	<i>Converts a track to a 'sf, data.frame'</i>
-------------	---

---

**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	<i>Transform coordinates reference system (CRS) of a 3-D track</i>
-----------------	--

---

### 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	<i>Turn angle to target</i>
----------------	-----------------------------

---

### 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	<i>Three dimensional histogram</i>
------------------	------------------------------------

---

### 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

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

---

voxelCount	<i>Apply voxel counting on a point cloud</i>
------------	--

---

### 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

```
voxelCount(points, extent, xyRes, zRes = xyRes, zMin, zMax,
  standartize = 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)
zMin	minimum z value
zMax	maximum height value
standartize	logical: standartize the values?

**Value**

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

**Examples**

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

# Index

## \*Topic **data**

dem, [3](#)

niclas, [13](#)

chiMaps, [2](#)

dem, [3](#)

dem2track.extent, [3](#)

dist2point.3d, [4](#)

dist2target.3d, [4](#)

eRTG3D, [5](#)

eRTG3D-package (eRTG3D), [5](#)

filter.dead.ends, [5](#)

get.densities.3d, [6](#), [11](#), [12](#), [16](#), [18](#), [20–23](#)

get.glideRatio.3d, [7](#)

get.section.densities.3d, [7](#), [9](#)

get.track.densities.3d, [8](#)

is.sf.3d, [9](#)

lift2target.3d, [9](#)

logRasterStack, [10](#)

movingMedian, [10](#)

n.sim.cond.3d, [11](#), [15](#)

n.sim.glidingSoaring.3d, [12](#)

niclas, [13](#)

plot2d, [13](#)

plot3d, [14](#)

plot3d.densities, [15](#)

plot3d.multiplot, [15](#)

plot3d.tldCube, [16](#)

plotRaster, [17](#)

qProb.3d, [11](#), [12](#), [17](#), [20](#), [22](#)

reproduce.track.3d, [18](#)

saveImageSlices, [19](#)

sf2df.3d, [19](#)

sim.cond.3d, [11](#), [12](#), [20](#), [21](#)

sim.crw.3d, [21](#)

sim.glidingSoaring.3d, [21](#)

sim.uncond.3d, [12](#), [17](#), [21](#), [22](#)

sp, [27](#)

spTransform, [27](#)

st\_transform, [27](#)

test.eRTG.3d, [23](#)

test.verification.3d, [24](#)

track.extent, [25](#)

track.properties.3d, [25](#)

track.split.3d, [7](#), [9](#), [26](#)

track2sf.3d, [26](#)

transformCRS.3d, [27](#)

turn2target.3d, [27](#)

turnLiftStepHist, [6–8](#), [16–18](#), [28](#)

voxelCount, [28](#)