Package 'DHARMa'

April 20, 2020

Title Residual Diagnostics for Hierarchical (Multi-Level / Mixed) Regression Models

Version 0.3.0

Date 2020-04-19

Description The 'DHARMa' package uses a simulation-based approach to create readily interpretable scaled (quantile) residuals for fitted (generalized) linear mixed models. Currently supported are linear and generalized linear (mixed) models from 'lme4' (classes 'lmerMod', 'glmerMod'), 'glmmTMB' and 'spaMM', generalized additive models ('gam' from 'mgcv'), 'glm' (including 'negbin' from 'MASS', but excluding quasi-distributions) and 'lm' model classes. Moreover, externally created simulations, e.g. posterior predictive simulations from Bayesian software such as 'JAGS', 'STAN', or 'BUGS' can be processed as well. The resulting residuals are standardized to values between 0 and 1 and can be interpreted as intuitively as residuals from a linear regression. The package also provides a number of plot and test functions for typical model misspecification problems, such as over/underdispersion, zero-inflation, and residual spatial and temporal autocorrelation.

Depends R (>= 3.0.2)

Imports stats, graphics, utils, grDevices, parallel, doParallel, foreach, gap, lmtest, ape, sfsmisc, MASS, lme4, mgcv, glmmTMB (>= 1.0.0), spaMM (>= 2.6.0), qgam (>= 1.3.2)

Suggests knitr, testthat, KernSmooth

License GPL (>= 3)

URL http://florianhartig.github.io/DHARMa/

BugReports https://github.com/florianhartig/DHARMa/issues

LazyData true

RoxygenNote 7.0.2.9000

VignetteBuilder knitr

Encoding UTF-8

NeedsCompilation no

Author Florian Hartig [aut, cre] (<https://orcid.org/0000-0002-6255-9059>)

Maintainer Florian Hartig <florian.hartig@biologie.uni-regensburg.de>

Repository CRAN

Date/Publication 2020-04-20 15:30:02 UTC

R topics documented:

createData *Simulate test data*

Description

This function creates synthetic dataset with various problems such as overdispersion, zero-inflation, etc.

Usage

```
createData(sampleSize = 100, intercept = 0, fixedEffects = 1,
  quadraticFixedEffects = NULL, numGroups = 10, randomEffectVariance = 1,
  overdispersion = 0, family = poisson(), scale = 1, cor = 0,
  roundPoissonVariance = NULL, pZeroInflation = 0, binomialTrials = 1,
  temporalAutocorrelation = 0, spatialAutocorrelation = 0,
  factorResponse = F, replicates = 1)
```

Arguments

sampleSize	sample size of the dataset
intercept	intercept (linear scale)
fixedEffects	vector of fixed effects (linear scale)
quadraticFixedEffects	vector of quadratic fixed effects (linear scale)
numGroups	number of groups for the random effect
randomEffectVariance	variance of the random effect (intercept)
overdispersion	if this is a numeric value, it will be used as the sd of a random normal variate that is added to the linear predictor. Alternatively, a random function can be provided that takes as input the linear predictor.
family	family
scale	scale if the distribution has a scale (e.g. sd for the Gaussian)
cor	correlation between predictors
roundPoissonVariance	if set, this creates a uniform noise on the possion response. The aim of this is to create heteroscedasticity
pZeroInflation	probability to set any data point to zero
binomialTrials	Number of trials for the binomial. Only active if family == binomial
temporalAutocorrelation	strength of temporalAutocorrelation
spatialAutocorrelation	strength of spatial Autocorrelation
factorResponse	should the response be transformed to a factor (inteded to be used for 0/1 data)
replicates	number of datasets to create

Examples

```
testData = createData(sampleSize = 500, intercept = 2, fixedEffects = c(1),
  overdispersion = 0, family = poisson(), quadraticFixedEffects = c(-3),
  randomEffectVariance = 0)

par(mfrow = c(1,2))
plot(testData$Environment1, testData$observedResponse)
hist(testData$observedResponse)

# with zero-inflation

testData = createData(sampleSize = 500, intercept = 2, fixedEffects = c(1),
  overdispersion = 0, family = poisson(), quadraticFixedEffects = c(-3),
  randomEffectVariance = 0, pZeroInflation = 0.6)

par(mfrow = c(1,2))
plot(testData$Environment1, testData$observedResponse)
hist(testData$observedResponse)

# binomial with multiple trials

testData = createData(sampleSize = 40, intercept = 2, fixedEffects = c(1),
                  overdispersion = 0, family = binomial(), quadraticFixedEffects = c(-3),
                      randomEffectVariance = 0, binomialTrials = 20)

plot(observedResponse1 / observedResponse0 ~ Environment1, data = testData, ylab = "Proportion 1")


# spatial / temporal correlation

testData = createData(sampleSize = 100, family = poisson(), spatialAutocorrelation = 3,
                      temporalAutocorrelation = 3)

plot(log(observedResponse) ~ time, data = testData)
plot(log(observedResponse) ~ x, data = testData)
```

`createDHARMa` *Create a DHARMa object from hand-coded simulations or Bayesian posterior predictive simulations*

Description

Create a DHARMa object from hand-coded simulations or Bayesian posterior predictive simulations

Usage

```
createDHARMa(simulatedResponse, observedResponse,
  fittedPredictedResponse = NULL, integerResponse = F, seed = 123)
```

Arguments

`simulatedResponse`
: matrix of observations simulated from the fitted model - row index for observations and colum index for simulations

`observedResponse`
: true observations

`fittedPredictedResponse`
: optional fitted predicted response. For Bayesian posterior predictive simulations, using the median posterior prediction as fittedPredictedResponse is recommended. If not provided, the mean simulatedResponse will be used.

`integerResponse`
: if T, noise will be added at to the residuals to maintain a uniform expectations for integer responses (such as Poisson or Binomial). Unlike in `simulateResiduals`, the nature of the data is not automatically detected, so this MUST be set by the user appropriately

`seed`
: the random seed to be used within DHARMa. The default setting, recommended for most users, is keep the random seed on a fixed value 123. This means that you will always get the same randomization and thus teh same result when running the same code. NULL = no new seed is set, but previous random state will be restored after simulation. FALSE = no seed is set, and random state will not be restored. The latter two options are only recommended for simulation experiments. See vignette for details.

Details

The use of this function is to convert simulated residuals (e.g. from a point estimate, or Bayesian p-values) to a DHARMa object, to make use of the plotting / test functions in DHARMa

Note

Either scaled residuals or (simulatedResponse AND observed response) have to be provided

Examples

```
## READING IN HAND-CODED SIMULATIONS

testData = createData(sampleSize = 50, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1, data = testData, family = "poisson")

# in DHARMA, using the simulate.glm function of glm
sims = simulateResiduals(fittedModel)
plot(sims, quantreg = FALSE)

# Doing the same with a handcode simulate function.
# of course this code will only work with a 1-par glm model
simulateMyfit <- function(n=10, fittedModel){
  int = coef(fittedModel)[1]
  slo = coef(fittedModel)[2]
  pred = exp(int + slo * testData$Environment1)
```

```
  predSim = replicate(n, rpois(length(pred), pred))
  return(predSim)
}

sims = simulateMyfit(250, fittedModel)

dharmaRes <- createDHARMa(simulatedResponse = sims,
                          observedResponse = testData$observedResponse,
                       fittedPredictedResponse = predict(fittedModel, type = "response"),
                          integer = TRUE)
plot(dharmaRes, quantreg = FALSE)

## A BAYESIAN EXAMPLE

## Not run:

  # This example shows how to check the residuals for a
  # Bayesian fit of a process-based vegetation model, using
  # THe BayesianTools package

  library(BayesianTools)

  # Create input data for the model
  PAR <- VSEMcreatePAR(1:1000)
  plotTimeSeries(observed = PAR)

  # load reference parameter definition (upper, lower prior)
  refPars <- VSEMgetDefaults()
  # this adds one additional parameter for the likelihood standard deviation (see below)
  refPars[12,] <- c(2, 0.1, 4)
  rownames(refPars)[12] <- "error-sd"

  # create some simulated test data
  # generally recommended to start with simulated data before moving to real data
 referenceData <- VSEM(refPars$best[1:11], PAR) # model predictions with reference parameters
  referenceData[,1] = 1000 * referenceData[,1]
  # this adds the error - needs to conform to the error definition in the likelihood
  obs <- referenceData + rnorm(length(referenceData), sd = refPars$best[12])

  parSel = c(1:6, 12) # parameters to calibrate

  # here is the likelihood
  likelihood <- function(par, sum = TRUE){
    # set parameters that are not calibrated on default values
    x = refPars$best
    x[parSel] = par
    predicted <- VSEM(x[1:11], PAR) # replace here VSEM with your model
    predicted[,1] = 1000 * predicted[,1] # this is just rescaling
    diff <- c(predicted[,1:4] - obs[,1:4]) # difference betweeno observed and predicted
   # univariate normal likelihood. Note that there is a parameter involved here that is fit
    llValues <- dnorm(diff, sd = x[12], log = TRUE)
    if (sum == FALSE) return(llValues)
    else return(sum(llValues))
```

```
}

# optional, you can also directly provide lower, upper in the createBayesianSetup, see help
prior <- createUniformPrior(lower = refPars$lower[parSel],
                            upper = refPars$upper[parSel], best = refPars$best[parSel])

bayesianSetup <- createBayesianSetup(likelihood, prior, names = rownames(refPars)[parSel])

# settings for the sampler, iterations should be increased for real applicatoin
settings <- list(iterations = 10000, nrChains = 2)

out <- runMCMC(bayesianSetup = bayesianSetup, sampler = "DEzs", settings = settings)

plot(out)
summary(out)
gelmanDiagnostics(out) # should be below 1.05 for all parameters to demonstrate convergence

# Posterior predictive simulations

# Create a function to create posterior predictive simulations
createPredictions <- function(par){
  # set the parameters that are not calibrated on default values
  x = refPars$best
  x[parSel] = par
  predicted <- VSEM(x[1:11], PAR) * 1000
  out = rnorm(length(predicted), mean = predicted, sd = par[7])
  return(out)
}

posteriorSample = getSample(out, numSamples = 1000)
posteriorPredictiveSims = apply(posteriorSample, 1, createPredictions)

dim(posteriorPredictiveSims)
library(DHARMa)
x = createDHARMa(t(posteriorPredictiveSims))
plot(x)

## End(Not run)
```

DHARMa *DHARMa - Residual Diagnostics for HierArchical (Multi-level / Mixed) Regression Models*

Description

The 'DHARMa' package uses a simulation-based approach to create readily interpretable scaled (quantile) residuals for fitted (generalized) linear mixed models. Currently supported are linear and generalized linear (mixed) models from 'lme4' (classes 'lmerMod', 'glmerMod'), 'glmmTMB' and 'spaMM', generalized additive models ('gam' from 'mgcv'), 'glm' (including 'negbin' from 'MASS', but excluding quasi-distributions) and 'lm' model classes. Moreover, externally created

simulations, e.g. posterior predictive simulations from Bayesian software such as 'JAGS', 'STAN', or 'BUGS' can be processed as well. The resulting residuals are standardized to values between 0 and 1 and can be interpreted as intuitively as residuals from a linear regression. The package also provides a number of plot and test functions for typical model misspecification problems, such as over/underdispersion, zero-inflation, and residual spatial and temporal autocorrelation.

Details

See index / vignette for details

See Also

simulateResiduals

Examples

```
vignette("DHARMa", package="DHARMa")
```

---

getFitted *Get model fitted*

Description

Wrapper to get the fitted value a fitted model

Usage

```
getFitted(object, ...)

## Default S3 method:
getFitted(object, ...)

## S3 method for class 'gam'
getFitted(object, ...)
```

Arguments

object	a fitted model
...	additional parameters to be passed on, usually to the simulate function of the respective model class

Details

The purpose of this wrapper is to standardize extract the fitted values

Author(s)

Florian Hartig

See Also

getObservedResponse, getSimulations, getRefit, getFixedEffects

Examples

```
testData = createData(sampleSize = 400, family = gaussian())

fittedModel <- lm(observedResponse ~ Environment1 , data = testData)

# response that was used to fit the model
getObservedResponse(fittedModel)

# predictions of the model for these points
getFitted(fittedModel)

# extract simulations from the model as matrix
getSimulations(fittedModel, nsim = 2)

# extract simulations from the model for refit (often requires different structure)
x = getSimulations(fittedModel, nsim = 2, type = "refit")

getRefit(fittedModel, x[[1]])

getRefit(fittedModel, getObservedResponse(fittedModel))
```

---

getFixedEffects *Extract fixed effects of a supported model*

Description

A wrapper to extract fixed effects of a supported model

Usage

```
getFixedEffects(fittedModel)
```

Arguments

fittedModel	a fitted model

See Also

getObservedResponse, getSimulations, getRefit, getFitted

Examples

```
testData = createData(sampleSize = 400, family = gaussian())

fittedModel <- lm(observedResponse ~ Environment1 , data = testData)

# response that was used to fit the model
getObservedResponse(fittedModel)

# predictions of the model for these points
getFitted(fittedModel)

# extract simulations from the model as matrix
getSimulations(fittedModel, nsim = 2)

# extract simulations from the model for refit (often requires different structure)
x = getSimulations(fittedModel, nsim = 2, type = "refit")

getRefit(fittedModel, x[[1]])

getRefit(fittedModel, getObservedResponse(fittedModel))
```

---

getObservedResponse *Get model response*

Description

Extract the response of a fitted model

Usage

```
getObservedResponse(object, ...)

## Default S3 method:
getObservedResponse(object, ...)

## S3 method for class 'HLfit'
getObservedResponse(object, ...)
```

Arguments

`object`	a fitted model
`...`	additional parameters

Details

The purpose of this function is to savely extract the response (dependent variable) of the fitted model classes

Author(s)

Florian Hartig

See Also

getRefit, getSimulations, getFixedEffects, getFitted

Examples

```
testData = createData(sampleSize = 400, family = gaussian())

fittedModel <- lm(observedResponse ~ Environment1 , data = testData)

# response that was used to fit the model
getObservedResponse(fittedModel)

# predictions of the model for these points
getFitted(fittedModel)

# extract simulations from the model as matrix
getSimulations(fittedModel, nsim = 2)

# extract simulations from the model for refit (often requires different structure)
x = getSimulations(fittedModel, nsim = 2, type = "refit")

getRefit(fittedModel, x[[1]])

getRefit(fittedModel, getObservedResponse(fittedModel))
```

---

getRandomState *Record and restore a random state*

Description

The aim of this function is to record, manipualate and restor a random state

Usage

```
getRandomState(seed = NULL)
```

Arguments

seed	seed argument to set.seed(). NULL = no seed, but random state will be restored. F = random state will not be restored

Details

This function is intended for two (not mutually exclusive tasks)

a) record the current random state

b) change the current random state in a way that the previous state can be restored

Value

a list with various infos about the random state that after function execution, as well as a function to restore the previous state before the function execution

Author(s)

Florian Hartig

Examples

```
# testing the function in standard settings

set.seed(13)
runif(1)
x = getRandomState(123)
runif(1)
x$restoreCurrent()
runif(1)

# values outside set /restore are identical to

set.seed(13)
runif(2)

# if no seed is set, this will also be restored

rm(.Random.seed) # now, there is no random seed
x = getRandomState(123)
exists(".Random.seed")  # TRUE
runif(1)
x$restoreCurrent()
exists(".Random.seed") # False

# with seed = false

x = getRandomState(seed = FALSE)
exists(".Random.seed")
runif(1)
x$restoreCurrent()
exists(".Random.seed")
```

---

getRefit *Get model refit*

Description

Wrapper to refit a fitted model

Usage

```
getRefit(object, newresp, ...)

## Default S3 method:
getRefit(object, newresp, ...)

## S3 method for class 'lm'
getRefit(object, newresp, ...)

## S3 method for class 'glmmTMB'
getRefit(object, newresp, ...)

## S3 method for class 'HLfit'
getRefit(object, newresp, ...)
```

Arguments

object	a fitted model
newresp	the new response that should be used to refit the model
...	additional parameters to be passed on to the refit or update class that is used to refit the model

Details

The purpose of this wrapper is to standardize the refit of a model. The behavior of this function depends on the supplied model. When available, it uses the refit method, otherwise it will use update. For glmmTMB: since version 1.0, glmmTMB has a refit function, but this didn't work, so I switched back to this implementation, which is a hack based on the update function.

Author(s)

Florian Hartig

See Also

getObservedResponse, getSimulations, getFixedEffects

Examples

```
testData = createData(sampleSize = 400, family = gaussian())

fittedModel <- lm(observedResponse ~ Environment1 , data = testData)

# response that was used to fit the model
getObservedResponse(fittedModel)

# predictions of the model for these points
getFitted(fittedModel)
```

```
# extract simulations from the model as matrix
getSimulations(fittedModel, nsim = 2)

# extract simulations from the model for refit (often requires different structure)
x = getSimulations(fittedModel, nsim = 2, type = "refit")

getRefit(fittedModel, x[[1]])

getRefit(fittedModel, getObservedResponse(fittedModel))
```

`getSimulations` *Get model simulations*

Description

Wrapper to simulate from a fitted model

Usage

```
getSimulations(object, nsim = 1, type = c("normal", "refit"), ...)

## Default S3 method:
getSimulations(object, nsim = 1, type = c("normal", "refit"), ...)

## S3 method for class 'glmmTMB'
getSimulations(object, nsim = 1, type = c("normal", "refit"), ...)

## S3 method for class 'HLfit'
getSimulations(object, nsim = 1, type = c("normal", "refit"), ...)
```

Arguments

`object`	a fitted model
`nsim`	number of simulations
`type`	if simulations should be prepared for getQuantile or for refit
`...`	additional parameters to be passed on, usually to the simulate function of the respective model class

Details

The purpose of this wrapper for for the simulate function is to standardize the simulations from a model in a standardized way

Value

a matrix with simulations

Author(s)

Florian Hartig

See Also

getObservedResponse, getRefit, getFixedEffects, getFitted

Examples

```
testData = createData(sampleSize = 400, family = gaussian())

fittedModel <- lm(observedResponse ~ Environment1 , data = testData)

# response that was used to fit the model
getObservedResponse(fittedModel)

# predictions of the model for these points
getFitted(fittedModel)

# extract simulations from the model as matrix
getSimulations(fittedModel, nsim = 2)

# extract simulations from the model for refit (often requires different structure)
x = getSimulations(fittedModel, nsim = 2, type = "refit")

getRefit(fittedModel, x[[1]])

getRefit(fittedModel, getObservedResponse(fittedModel))
```

---

hasWeigths *Check weights*

Description

Checks if a model was fit with the weight argument

Usage

```
hasWeigths(object, ...)
```

Arguments

object	a fitted model
...	additional parameters

Details

The purpose of this function is to check if a model was fit with the weights

Author(s)

Florian Hartig

`hist.DHARMa` *Histogram of DHARMa residuals*

Description

The function produces a histogram from a DHARMa output

Usage

```
## S3 method for class 'DHARMa'
hist(x, breaks = seq(-0.02, 1.02, len = 53), col = c("red",
  rep("lightgrey", 50), "red"), main = "Hist of DHARMa residuals",
  xlab = "Residuals (outliers are marked red)", cex.main = 1, ...)
```

Arguments

`x`	a DHARMa simulation output (class DHARMa)
`breaks`	breaks for hist() function
`col`	col for hist bars
`main`	plot main
`xlab`	plot xlab
`cex.main`	plot cex.main
`...`	other arguments to be passed on to hist

See Also

`plotSimulatedResiduals`, `plotResiduals`

Examples

```
testData = createData(sampleSize = 200, family = poisson(),
                      randomEffectVariance = 1, numGroups = 5)
fittedModel <- glm(observedResponse ~ Environment1,
                   family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

######### main plotting function #############

# for all functions, quantreg = T will be more
# informative, but slower

plot(simulationOutput, quantreg = FALSE)
```

```
############  Distribution  ######################

plotQQunif(simulationOutput = simulationOutput)

hist(simulationOutput )

############  residual plots  ###############

# rank transformation, using a simulationOutput
plotResiduals(simulationOutput, rank = TRUE, quantreg = FALSE)

# smooth scatter plot - usually used for large datasets, default for n > 10000
plotResiduals(simulationOutput, rank = TRUE, quantreg = FALSE, smoothScatter = TRUE)

# residual vs predictors, using explicit values for pred, residual
plotResiduals(simulationOutput, form = testData$Environment1,
              quantreg = FALSE)

# if pred is a factor, or asFactor = T, will produce a boxplot
plotResiduals(simulationOutput, form = testData$group,
              quantreg = FALSE, asFactor = TRUE)

# All these options can also be provided to the main plotting function
plot(simulationOutput, quantreg = FALSE, rank = FALSE)

# If you want to plot summaries per group, use
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, asFactor = TRUE) # we see one residual point per RE
```

`plot.DHARMa` *DHARMa standard residual plots*

Description

This function creates standard plots for the simulated residuals

Usage

```
## S3 method for class 'DHARMa'
plot(x, rank = TRUE, ...)
```

Arguments

`x`	an object with simualted residuals created by `simulateResiduals`
`rank`	if T (default), the values of pred will be rank transformed. This will usually make patterns easier to spot visually, especially if the distribution of the predictor is skewed.

... further options for plotResiduals. Consider in particular parameters quantreg, rank and asFactor. xlab, ylab and main cannot be changed when using plotSimulatedResiduals, but can be changed when using plotResiduals.

Details

The function creates two plots. To the left, a qq-uniform plot to detect deviations from overall uniformity of the residuals (calling plotQQunif), and to the right, a plot of residuals against predicted values (calling plotResiduals). Outliers are highlighted in red (for more on oultiers, see testOutliers). For a correctly specified model, we would expect

a) a straight 1-1 line in the uniform qq-plot -> evidence for an overal uniform (flat) distribution of the residuals

b) uniformity of residuals in the vertical direction in the res against predictor plot

Deviations of this can be interpreted as for a linear regression. See the vignette for detailed examples.

To provide a visual aid in detecting deviations from uniformity in y-direction, the plot of the residuals against the predited values also performs an (optional) quantile regression, which provides 0.25, 0.5 and 0.75 quantile lines across the plots. These lines should be straight, horizontal, and at y-values of 0.25, 0.5 and 0.75. Note, however, that some deviations from this are to be expected by chance, even for a perfect model, especially if the sample size is small. See further comments on this plot, it's interpreation and options, in plotResiduals

The quantile regression can take some time to calculate, especially for larger datasets. For that reason, quantreg = F can be set to produce a smooth spline instead. This is default for n > 2000.

See Also

plotResiduals, plotQQunif

Examples

```
testData = createData(sampleSize = 200, family = poisson(),
                      randomEffectVariance = 1, numGroups = 5)
fittedModel <- glm(observedResponse ~ Environment1,
                   family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

######### main plotting function #############

# for all functions, quantreg = T will be more
# informative, but slower

plot(simulationOutput, quantreg = FALSE)

#############  Distribution  ######################

plotQQunif(simulationOutput = simulationOutput)

hist(simulationOutput )
```

```
############  residual plots  ###############

# rank transformation, using a simulationOutput
plotResiduals(simulationOutput, rank = TRUE, quantreg = FALSE)

# smooth scatter plot - usually used for large datasets, default for n > 10000
plotResiduals(simulationOutput, rank = TRUE, quantreg = FALSE, smoothScatter = TRUE)

# residual vs predictors, using explicit values for pred, residual
plotResiduals(simulationOutput, form = testData$Environment1,
              quantreg = FALSE)

# if pred is a factor, or asFactor = T, will produce a boxplot
plotResiduals(simulationOutput, form = testData$group,
              quantreg = FALSE, asFactor = TRUE)

# All these options can also be provided to the main plotting function
plot(simulationOutput, quantreg = FALSE, rank = FALSE)

# If you want to plot summaries per group, use
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, asFactor = TRUE) # we see one residual point per RE
```

`plotConventionalResiduals` *Conventional residual plot*

Description

Convenience function to draw conventional residual plots

Usage

```
plotConventionalResiduals(fittedModel)
```

Arguments

`fittedModel`	a fitted model object

plotQQunif *Quantile-quantile plot for a uniform distribution*

Description

The function produces a uniform quantile-quantile plot from a DHARMa output

Usage

```
plotQQunif(simulationOutput, testUniformity = T, testOutliers = T,
  testDispersion = T, ...)
```

Arguments

simulationOutput	a DHARMa simulation output (class DHARMa)
testUniformity	if T, the function testUniformity will be called and the result will be added to the plot
testOutliers	if T, the function testOutliers will be called and the result will be added to the plot
testDispersion	if T, the function testDispersion will be called and the result will be added to the plot
...	arguments to be passed on to qqunif

Details

the function calls qqunif from the R package gap to create a quantile-quantile plot for a uniform distribution.

See Also

plotSimulatedResiduals, plotResiduals

Examples

```
testData = createData(sampleSize = 200, family = poisson(),
                      randomEffectVariance = 1, numGroups = 5)
fittedModel <- glm(observedResponse ~ Environment1,
                   family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

######### main plotting function #############

# for all functions, quantreg = T will be more
# informative, but slower

plot(simulationOutput, quantreg = FALSE)
```

```
############  Distribution  ######################

plotQQunif(simulationOutput = simulationOutput)

hist(simulationOutput )

############  residual plots  ###############

# rank transformation, using a simulationOutput
plotResiduals(simulationOutput, rank = TRUE, quantreg = FALSE)

# smooth scatter plot - usually used for large datasets, default for n > 10000
plotResiduals(simulationOutput, rank = TRUE, quantreg = FALSE, smoothScatter = TRUE)

# residual vs predictors, using explicit values for pred, residual
plotResiduals(simulationOutput, form = testData$Environment1,
              quantreg = FALSE)

# if pred is a factor, or asFactor = T, will produce a boxplot
plotResiduals(simulationOutput, form = testData$group,
              quantreg = FALSE, asFactor = TRUE)

# All these options can also be provided to the main plotting function
plot(simulationOutput, quantreg = FALSE, rank = FALSE)

# If you want to plot summaries per group, use
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, asFactor = TRUE) # we see one residual point per RE
```

`plotResiduals` *Generic residual plot with either spline or quantile regression*

Description

The function creates a generic residual plot with either spline or quantile regression to highlight patterns in the residuals. Outliers are highlighted in red

Usage

```
plotResiduals(simulationOutput, form = NULL, quantreg = NULL, rank = F,
  asFactor = NULL, smoothScatter = NULL, quantiles = c(0.25, 0.5, 0.75),
  ...)
```

Arguments

simulationOutput	on object, usually a DHARMa object, from which residual values can be extracted. Alternatively, a vector with residuals or a fitted model can be provided, which will then be transformed into a DHARMa object.
form	optional predictor against which the residuals should be plotted. Default is to used the predicted(simulationOutput)
quantreg	whether to perform a quantile regression on 0.25, 0.5, 0.75 on the residuals. If F, a spline will be created instead. Default NULL chooses T for nObs < 2000, and F otherwise.
rank	if T, the values provided in form will be rank transformed. This will usually make patterns easier to spot visually, especially if the distribution of the predictor is skewed. If form is a factor, this has no effect.
asFactor	should a numeric predictor provided in form be treated as a factor. Default is to choose this for < 10 unique values, as long as enough predictions are available to draw a boxplot.
smoothScatter	if T, a smooth scatter plot will plotted instead of a normal scatter plot. This makes sense when the number of residuals is very large. Default NULL chooses T for nObs < 10000, and F otherwise.
quantiles	for a quantile regression, which quanties should be plotted
...	additional arguments to plot / boxplot.

Details

The function plots residuals against a predictor (by default against the fitted value, extracted from the DHARMa object, or any other predictor).

Outliers are highlighted in red (for information on definition and interpretation of outliers, see testOutliers).

To provide a visual aid in detecting deviations from uniformity in y-direction, the plot function calculates an (optional) quantile regression, which compares the empirical 0.25, 0.5 and 0.75 quantiles (default) in y direction (red solid lines) with the theoretical 0.25, 0.5 and 0.75 quantiles (dashed black line).

Assymptotically (i.e. for lots of data / residuals), if the model is correct, theoretical and the empirical quantiles should be identical (i.e. dashed and solid lines should match). A p-value for the deviation is calculated for each quantile line. Significant deviations are highlighted by red color.

If form is a factor, a boxplot will be plotted instead of a scatter plot. The distribution for each factor level should be uniformly distributed, so the box should go from 0.25 to 0.75, with the median line at 0.5. Again, chance deviations from this will increases when the sample size is smaller. You can run null simulations to test if the deviations you see exceed what you would expect from random variation. If you want to create box plots for categorical predictors (e.g. because you only have a small number of unique numberic predictor values), you can convert your predictor with as.factor(pred)

Note

The quantile regression can take some time to calculate, especially for larger datasets. For that reason, quantreg = F can be set to produce a smooth spline instead.

See Also

plotQQunif

Examples

```
testData = createData(sampleSize = 200, family = poisson(),
                      randomEffectVariance = 1, numGroups = 5)
fittedModel <- glm(observedResponse ~ Environment1,
                   family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

######### main plotting function #############

# for all functions, quantreg = T will be more
# informative, but slower

plot(simulationOutput, quantreg = FALSE)

#############  Distribution  ######################

plotQQunif(simulationOutput = simulationOutput)

hist(simulationOutput )

#############  residual plots  ###############

# rank transformation, using a simulationOutput
plotResiduals(simulationOutput, rank = TRUE, quantreg = FALSE)

# smooth scatter plot - usually used for large datasets, default for n > 10000
plotResiduals(simulationOutput, rank = TRUE, quantreg = FALSE, smoothScatter = TRUE)

# residual vs predictors, using explicit values for pred, residual
plotResiduals(simulationOutput, form = testData$Environment1,
              quantreg = FALSE)

# if pred is a factor, or asFactor = T, will produce a boxplot
plotResiduals(simulationOutput, form = testData$group,
              quantreg = FALSE, asFactor = TRUE)

# All these options can also be provided to the main plotting function
plot(simulationOutput, quantreg = FALSE, rank = FALSE)

# If you want to plot summaries per group, use
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, asFactor = TRUE) # we see one residual point per RE
```

plotSimulatedResiduals *DHARMa standard residual plots*

Description

DEPRECATED, use plot() instead

Usage

```
plotSimulatedResiduals(simulationOutput, ...)
```

Arguments

simulationOutput	an object with simualted residuals created by simulateResiduals
...	further options for plotResiduals. Consider in particular parameters quantreg, rank and asFactor. xlab, ylab and main cannot be changed when using plotSimulatedResiduals, but can be changed when using plotResiduals.

Note

This function is deprecated. Use plot.DHARMa

See Also

plotResiduals, plotQQunif

print.DHARMa *Print simulated residuals*

Description

Print simulated residuals

Usage

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

Arguments

x	an object with simulated residuals created by simulateResiduals
...	optional arguments for compatibility with the generic function, no function implemented

recalculateResiduals *Recalculate residuals with grouping*

Description

The purpose of this function is to recalculate scaled residuals per group, based on the simulations done by simulateResiduals

Usage

```
recalculateResiduals(simulationOutput, group = NULL, aggregateBy = sum,
  seed = 123)
```

Arguments

simulationOutput	an object with simulated residuals created by simulateResiduals
group	group of each data point
aggregateBy	function for the aggregation. Default is sum. This should only be changed if you know what you are doing. Note in particular that the expected residual distribution might not be flat any more if you choose general functions, such as sd etc.
seed	the random seed to be used within DHARMa. The default setting, recommended for most users, is keep the random seed on a fixed value 123. This means that you will always get the same randomization and thus teh same result when running the same code. NULL = no new seed is set, but previous random state will be restored after simulation. FALSE = no seed is set, and random state will not be restored. The latter two options are only recommended for simulation experiments. See vignette for details.

Value

an object of class DHARMa, similar to what is returned by simulateResiduals, but with additional outputs for the new grouped calculations. Note that the relevant outputs are 2x in the object, the first is the grouped calculations (which is returned by $name access), and later another time, under identical name, the original output. Moreover, there is a function 'aggregateByGroup', which can be used to aggregate predictor variables in the same way as the variables calculated here

Examples

```
library(lme4)

testData = createData(sampleSize = 200, overdispersion = 0.5, family = poisson())
fittedModel <- glmer(observedResponse ~ Environment1 + (1|group),
                     family = "poisson", data = testData)

simulationOutput <- simulateResiduals(fittedModel = fittedModel)
```

```
# standard plot
plot(simulationOutput)

# one of the possible test, for other options see ?testResiduals
testOutliers(simulationOutput)

# for various other plots and tests, see the help / vignette

# the calculated residuals can be accessed via
residuals(simulationOutput)

# transform residuals to other pdf, see ?residuals.DHARMa for details
residuals(simulationOutput, quantileFunction = qnorm, outlierValues = c(-7,7))

# calculating summaries per group
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, quantreg = FALSE)

# create residuals with refitting, see ?simulateResiduals for details
# n=10 is very low, set higher when using this for real
simulationOutput <- simulateResiduals(fittedModel = fittedModel,
                                      n = 10, refit = TRUE)
plot(simulationOutput, quantreg = FALSE)
```

`residuals.DHARMa` *Return residuals of a DHARMa simulation*

Description

Return residuals of a DHARMa simulation

Usage

```
## S3 method for class 'DHARMa'
residuals(object, quantileFunction = NULL, outlierValues = NULL, ...)
```

Arguments

`object`	an object with simulated residuals created by `simulateResiduals`
`quantileFunction`	optional - a quantile function to transform the uniform 0/1 scaling of DHARMa to another distribution
`outlierValues`	if a quantile function with infinite support (such as dnorm) is used, residuals that are 0/1 are mapped to -Inf / Inf. outlierValues allows to convert -Inf / Inf values to an optional min / max value.
`...`	optional arguments for compatibility with the generic function, no function implemented

Details

the function accesses the slot $scaledResiduals in a fitted DHARMa object, and optionally transforms the standard DHARMa quantile residuals (which have a uniform distribution) to a particular pdf.

Note

some of the papers on simulated quantile residuals transforming the residuals (which are natively uniform) back to a normal distribution. I presume this is because of the larger familiarity of most users with normal residuals. Personally, I never considered this desirable, for the reasons explained in https://github.com/florianhartig/DHARMa/issues/39, but with this function, I wanted to give users the option to plot normal residuals if they so wish.

Examples

```
library(lme4)

testData = createData(sampleSize = 200, overdispersion = 0.5, family = poisson())
fittedModel <- glmer(observedResponse ~ Environment1 + (1|group),
                     family = "poisson", data = testData)

simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# standard plot
plot(simulationOutput)

# one of the possible test, for other options see ?testResiduals
testOutliers(simulationOutput)

# for various other plots and tests, see the help / vignette

# the calculated residuals can be accessed via
residuals(simulationOutput)

# transform residuals to other pdf, see ?residuals.DHARMa for details
residuals(simulationOutput, quantileFunction = qnorm, outlierValues = c(-7,7))

# calculating summaries per group
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, quantreg = FALSE)

# create residuals with refitting, see ?simulateResiduals for details
# n=10 is very low, set higher when using this for real
simulationOutput <- simulateResiduals(fittedModel = fittedModel,
                                      n = 10, refit = TRUE)
plot(simulationOutput, quantreg = FALSE)
```

runBenchmarks *Benchmark calculations*

Description

This function runs statistical benchmarks, including Power / Type I error simulations for an arbitrary test with a control parameter

Usage

```
runBenchmarks(calculateStatistics, controlValues = NULL, nRep = 10,
  alpha = 0.05, parallel = F, ...)
```

Arguments

calculateStatistics	the statistics to be benchmarked. Should return one value, or a vector of values. If controlValues are given, must accept a paramteter control
controlValues	a vector with a control parameter (e.g. to vary the strength of a problem the test should be specific to)
nRep	number of replicates per level of the controlValues
alpha	significance level
parallel	whether to use parallel computations. Possible values are F, T (sets the cores automatically to number of available cores -1), or an integer number for the number of cores that should be used for the cluster
...	additional parameters to calculateStatistics

Note

The benchmark function in DHARMa are intended for development purposes, and for users that want to test / confirm the properties of functions in DHARMa. If you are running an applied data analysis, they are probably of little use.

simulateResiduals *Create simulated residuals*

Description

The function creates scaled residuals by simulating from the fitted model

Usage

```
simulateResiduals(fittedModel, n = 250, refit = F, integerResponse = NULL,
  plot = F, seed = 123, ...)
```

Arguments

fittedModel	a fitted model of a class supported by DHARMa
n	number of simulations. Default is 100. A more save value would be 250 or even 1000. The smaller the number, the higher the stochastic error on the residuals. Also, for very small n, discretization artefacts can influence the tests.
refit	if FALSE, new data will be simulated and scaled residuals will be created by comparing observed data with new data. If TRUE, the model will be refit on the simulated data (parametric bootstrap), and scaled residuals will be created by comparing observed with refitted residuals.
integerResponse	if TRUE, noise will be added at to the residuals to maintain a uniform expectations for integer responses (such as Poisson or Binomial). Usually, the model will automatically detect the appropriate setting, so there is no need to adjust this setting.
plot	if TRUE, plotSimulatedResiduals will be directly run after the simulations have terminated
seed	the random seed to be used within DHARMa. The default setting, recommended for most users, is keep the random seed on a fixed value 123. This means that you will always get the same randomization and thus teh same result when running the same code. NULL = no new seed is set, but previous random state will be restored after simulation. FALSE = no seed is set, and random state will not be restored. The latter two options are only recommended for simulation experiments. See vignette for details.
...	parameters to pass to the simulate function of the model object. An important use of this is to specify whether simulations should be conditional on the current random effect estimates, e.g. via re.form. Note that not all models support syntax to specify conditionao or unconditional simulations. See also details

Details

There are a number of important considerations when simulating from a more complex (hierarchical) model:

Re-simulating random effects / hierarchical structure: in a hierarchical model, we have several stochastic processes aligned on top of each other. Specifically, in a GLMM, we have a lower level stochastic process (random effect), whose result enters into a higher level (e.g. Poisson distribution). For other hierarchical models such as state-space models, similar considerations apply.

In such a situation, we have to decide if we want to re-simulate all stochastic levels, or only a subset of those. For example, in a GLMM, it is common to only simulate the last stochastic level (e.g. Poisson) conditional on the fitted random effects. This is often referred to as a conditional simuation. For controlling how many levels should be re-simulated, the simulateResidual function allows to pass on parameters to the simulate function of the fitted model object. Please refer to the help of the different simulate functions (e.g. ?simulate.merMod) for details. For merMod (lme4) model objects, the relevant parameters are parameters are use.u and re.form

If the model is correctly specified, the simulated residuals should be flat regardless how many hierarchical levels we re-simulate. The most thorough procedure would therefore be to test all

possible options. If testing only one option, I would recommend to re-simulate all levels, because this essentially tests the model structure as a whole. This is the default setting in the DHARMa package. A potential drawback is that re-simulating the lower-level random effects creates more variability, which may reduce power for detecting problems in the upper-level stochastic processes. In particular dispersion tests may produce different results when switching from conditional to unconditional simulations, and often the conditional simulation is more sensitive.

Integer responses: a second complication is the treatment of inter responses. Imaging we have observed a 0, and we predict 30% zeros - what is the quantile that we should display for the residual? To deal with this problem and maintain a uniform response, the option integerResponse adds a uniform noise from -0.5 to 0.5 on the simulated and observed response, which creates a uniform distribution - you can see this via hist(ecdf(runif(10000))(runif(10000))).

DHARMa will try to automatically if the fitted model has an integer or discrete distribution via the family argument. However, in some cases the family does not allow to uniquely identify the distribution type. For example, a tweedie distribution can be inter or continuous. Therefore, DHARMa will additionally check the simulation results for repeated values, and will change the distribution type if repeated values are found (a message is displayed in this case).

Refitting or not: a third issue is how residuals are calculated. simulateResiduals has two options that are controlled by the refit parameter:

1. if refit = FALSE (default), new data is simulated from the fitted model, and residuals are calculated by comparing the observed data to the new data

2. if refit = TRUE, a parametric bootstrap is performed, meaning that the model is refit on the new data, and residuals are created by comparing observed residuals against refitted residuals

The second option is much slower, and only important for running tests that rely on comparing observed to simulated residuals, e.g. the `testOverdispersion` function

Residuals per group: In many situations, it can be useful to look at residuals per group, e.g. to see how much the model over / underpredicts per plot, year or subject. To do this, use `recalculateResiduals`, together with a grouping variable (see also help)

Transformation to other distributions: DHARMa calculates residuals for which the theoretical expectation (assuming a correctly specified model) is uniform. To transfor this residuals to another distribution (e.g. so that a correctly specified model will have normal residuals) see `residuals.DHARMa`.

Value

An S3 class of type "DHARMa", essentially a list with various elements. Implemented S3 functions include plot, print and `residuals.DHARMa`. Residuals returns the calculated scaled residuals.

Note

See `testResiduals` for an overview of residual tests, `plot.DHARMa` for an overview of available plots.

See Also

`testResiduals`, `plot.DHARMa`, `print.DHARMa`, `residuals.DHARMa`, `recalculateResiduals`

Examples

```
library(lme4)

testData = createData(sampleSize = 200, overdispersion = 0.5, family = poisson())
fittedModel <- glmer(observedResponse ~ Environment1 + (1|group),
                     family = "poisson", data = testData)

simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# standard plot
plot(simulationOutput)

# one of the possible test, for other options see ?testResiduals
testOutliers(simulationOutput)

# for various other plots and tests, see the help / vignette

# the calculated residuals can be accessed via
residuals(simulationOutput)

# transform residuals to other pdf, see ?residuals.DHARMa for details
residuals(simulationOutput, quantileFunction = qnorm, outlierValues = c(-7,7))

# calculating summaries per group
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, quantreg = FALSE)

# create residuals with refitting, see ?simulateResiduals for details
# n=10 is very low, set higher when using this for real
simulationOutput <- simulateResiduals(fittedModel = fittedModel,
                                      n = 10, refit = TRUE)
plot(simulationOutput, quantreg = FALSE)
```

---

testDispersion *DHARMa dispersion tests*

Description

This function performs a simulation-based test for over/underdispersion

Usage

```
testDispersion(simulationOutput, alternative = c("two.sided", "greater",
  "less"), plot = T, ...)
```

Arguments

simulationOutput	an object of class DHARMa with simulated quantile residuals, either created via simulateResiduals or by createDHARMa for simulations created outside DHARMa
alternative	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis. Greater corresponds to overdispersion.
plot	whether to plot output
...	arguments to pass on to testGeneric

Details

The function implements two tests, depending on whether it is applied on a simulation with refit = F, or refit = T.

If refit = F, the function tests the sd of the data against the sd of the simulated data.

If refit = T, the function compares the approximate deviance (via squared pearson residuals) with the same quantity from the models refitted with simulated data. Applying this is much slower than the previous alternative, but simulations show that it is slightly more powerful as well. However, given the computational cost, I would suggest that most users will be satisfied with the standard dispersion test.

Note

The dispersion test can be quite different if it is evaluate on conditional or unconditional simulations (see simulateResiduals). The default in DHARMa is to use unconditional simulations, but I recommend trying both.

Author(s)

Florian Hartig

See Also

testResiduals, testUniformity, testOutliers, testZeroInflation, testGeneric, testTemporalAutocorrelation, testSpatialAutocorrelation, testQuantiles

Examples

```
testData = createData(sampleSize = 200, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# the plot function runs 4 tests
# i) KS test i) Dispersion test iii) Outlier test iv) quantile test
plot(simulationOutput, quantreg = TRUE)

# testResiduals tests distribution, dispersion and outliers
```

```
testResiduals(simulationOutput)

####### Individual tests #######

# KS test for correct distribution of residuals
testUniformity(simulationOutput)

# Dispersion test
testDispersion(simulationOutput) # tests under and overdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion

# if model is refitted, a different test will be called
simulationOutput2 <- simulateResiduals(fittedModel = fittedModel, refit = TRUE, seed = 12)
testDispersion(simulationOutput2)

# often useful to test dispersion per group (e.g. binomial data, see vignette)
simulationOutput3 = recalculateResiduals(simulationOutput, group = testData$group)
testDispersion(simulationOutput3)

# Outlier test (number of observations outside simulation envelope)
testOutliers(simulationOutput)

# testing zero inflation
testZeroInflation(simulationOutput)

# testing generic summaries
countOnes <- function(x) sum(x == 1)  # testing for number of 1s
testGeneric(simulationOutput, summary = countOnes) # 1-inflation
testGeneric(simulationOutput, summary = countOnes, alternative = "less") # 1-deficit

means <- function(x) mean(x) # testing if mean prediction fits
testGeneric(simulationOutput, summary = means)

spread <- function(x) sd(x) # testing if mean sd fits
testGeneric(simulationOutput, summary = spread)
```

`testGeneric` *Generic simulation test of a summary statistic*

Description

This function tests if a user-defined summary differs when applied to simulated / observed data.

Usage

```
testGeneric(simulationOutput, summary, alternative = c("two.sided", "greater",
  "less"), plot = T, methodName = "DHARMa generic simulation test")
```

Arguments

simulationOutput	an object of class DHARMa with simulated quantile residuals, either created via simulateResiduals or by createDHARMa for simulations created outside DHARMa
summary	a function that can be applied to simulated / observed data. See examples below
alternative	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis
plot	whether to plot the simulated summary
methodName	name of the test (will be used in plot)

Details

This function tests if a user-defined summary differs when applied to simulated / observed data. the function can easily be remodeled to apply summaries on the residuals, by simply defining f = function(x) summary (x - predictions), as done in testDispersion

Note

The function that you supply is applied on the data as it is represented in your fitted model, which may not always correspond to how you think. This is important in particular when you use k/n binomial data, and want to test for 1-inflation. As an example, if have k/20 observations, and you provide your data via cbind (y, y-20), you have to test for 20-inflation (because this is how the data is represented in the model). However, if you provide data via y/20, and weights = 20, you should test for 1-inflation. In doubt, check how the data is internally represented in model.frame(model), or via simulate(model)

Author(s)

Florian Hartig

See Also

testResiduals, testUniformity, testOutliers, testDispersion, testZeroInflation, testTemporalAutocorrelati
testSpatialAutocorrelation, testQuantiles

Examples

```
testData = createData(sampleSize = 200, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# the plot function runs 4 tests
# i) KS test i) Dispersion test iii) Outlier test iv) quantile test
```

```
plot(simulationOutput, quantreg = TRUE)

# testResiduals tests distribution, dispersion and outliers
testResiduals(simulationOutput)

####### Individual tests #######

# KS test for correct distribution of residuals
testUniformity(simulationOutput)

# Dispersion test
testDispersion(simulationOutput) # tests under and overdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion

# if model is refitted, a different test will be called
simulationOutput2 <- simulateResiduals(fittedModel = fittedModel, refit = TRUE, seed = 12)
testDispersion(simulationOutput2)

# often useful to test dispersion per group (e.g. binomial data, see vignette)
simulationOutput3 = recalculateResiduals(simulationOutput, group = testData$group)
testDispersion(simulationOutput3)

# Outlier test (number of observations outside simulation envelope)
testOutliers(simulationOutput)

# testing zero inflation
testZeroInflation(simulationOutput)

# testing generic summaries
countOnes <- function(x) sum(x == 1)  # testing for number of 1s
testGeneric(simulationOutput, summary = countOnes) # 1-inflation
testGeneric(simulationOutput, summary = countOnes, alternative = "less") # 1-deficit

means <- function(x) mean(x) # testing if mean prediction fits
testGeneric(simulationOutput, summary = means)

spread <- function(x) sd(x) # testing if mean sd fits
testGeneric(simulationOutput, summary = spread)
```

`testOutliers` *Test for outliers*

Description

This function tests if the number of observations that are strictly greater / smaller than all simulations are larger than expected

Usage

```
testOutliers(simulationOutput, alternative = c("two.sided", "greater",
  "less"), plot = T)
```

Arguments

simulationOutput	an object of class DHARMa with simulated quantile residuals, either created via simulateResiduals or by createDHARMa for simulations created outside DHARMa
alternative	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis
plot	if T, the function will create an additional plot

Details

DHARMa residuals are created by simulating from the fitted model, and comparing the simulated values to the observed data. It can occur that all simulated values are higher or smaller than the observed data, in which case they get the residual value of 0 and 1, respectively. I refer to these values as simulation outliers, or simply outliers.

Because no data was simulated in the range of the observed value, we actually don't know "how much" these values deviate from the model expecation, so the term "outlier" should be used with a grain of salt - it's not a judgement about the probability of a deviation from an expectation, but denotes that we are outside the simulated range. The number of outliers would usually decrease if the number of DHARMa simulations is increased.

The probability of an outlier depends on the number of simulations (in fact, it is 1/(nSim +1) for each side), so whether the existence of outliers is a reason for concern depends also on the number of simulations. The expected number of outliers is therefore binomially distributed, and we can calculate a p-value from that

Author(s)

Florian Hartig

See Also

testResiduals, testUniformity, testDispersion, testZeroInflation, testGeneric, testTemporalAutocorrelatic
testSpatialAutocorrelation, testQuantiles

Examples

```
testData = createData(sampleSize = 200, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# the plot function runs 4 tests
# i) KS test i) Dispersion test iii) Outlier test iv) quantile test
plot(simulationOutput, quantreg = TRUE)
```

```
# testResiduals tests distribution, dispersion and outliers
testResiduals(simulationOutput)

####### Individual tests #######

# KS test for correct distribution of residuals
testUniformity(simulationOutput)

# Dispersion test
testDispersion(simulationOutput) # tests under and overdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion

# if model is refitted, a different test will be called
simulationOutput2 <- simulateResiduals(fittedModel = fittedModel, refit = TRUE, seed = 12)
testDispersion(simulationOutput2)

# often useful to test dispersion per group (e.g. binomial data, see vignette)
simulationOutput3 = recalculateResiduals(simulationOutput, group = testData$group)
testDispersion(simulationOutput3)

# Outlier test (number of observations outside simulation envelope)
testOutliers(simulationOutput)

# testing zero inflation
testZeroInflation(simulationOutput)

# testing generic summaries
countOnes <- function(x) sum(x == 1)  # testing for number of 1s
testGeneric(simulationOutput, summary = countOnes) # 1-inflation
testGeneric(simulationOutput, summary = countOnes, alternative = "less") # 1-deficit

means <- function(x) mean(x) # testing if mean prediction fits
testGeneric(simulationOutput, summary = means)

spread <- function(x) sd(x) # testing if mean sd fits
testGeneric(simulationOutput, summary = spread)
```

`testOverdispersion` *Simulated overdisperstion tests*

Description

Simulated overdisperstion tests

Usage

```
testOverdispersion(simulationOutput, ...)
```

Arguments

simulationOutput	an object of class DHARMa with simulated quantile residuals, either created via simulateResiduals or by createDHARMa for simulations created outside DHARMa
...	additional arguments to testDispersion

Details

Deprecated, switch your code to using the testDispersion function

---

testOverdispersionParametric

Parametric overdisperstion tests

Description

Parametric overdisperstion tests

Usage

```
testOverdispersionParametric(...)
```

Arguments

...	arguments will be ignored, the parametric tests is no longer recommend

Details

Deprecated, switch your code to using the testDispersion function. The function will do nothing, arguments will be ignored, the parametric tests is no longer recommend

testPDistribution *Plot distribution of p-values*

Description

Plot distribution of p-values

Usage

```
testPDistribution(x, plot = T,
  main = "p distribution \n expected is flat at 1", ...)
```

Arguments

x	vector of p values
plot	should the values be plottet
main	title for the plot
...	additional arguments to hist

Author(s)

Florian Hartig

testQuantiles *Test for quantiles*

Description

This function tests

Usage

```
testQuantiles(simulationOutput, predictor = NULL, quantiles = c(0.25, 0.5,
  0.75), plot = T)
```

Arguments

simulationOutput	an object of class DHARMa with simulated quantile residuals, either created via simulateResiduals or by createDHARMa for simulations created outside DHARMa
predictor	an optional predictor variable to be used, instead of the predicted response (default)
quantiles	the quantiles to be tested
plot	if T, the function will create an additional plot

Details

The function fits quantile regressions (via package qgam) on the residuals, and compares their location to the expected location (because of the uniform distributionm, the expected location is 0.5 for the 0.5 quantile).

A significant p-value for the splines means the fitted spline deviates from a flat line at the expected location (p-values of intercept and spline are combined via Benjamini & Hochberg adjustment to control the FDR)

The p-values of the splines are combined into a total p-value via Benjamini & Hochberg adjustment to control the FDR.

Author(s)

Florian Hartig

See Also

testResiduals, testUniformity, testDispersion, testZeroInflation, testGeneric, testTemporalAutocorrelatio
testSpatialAutocorrelation, testOutliers

Examples

```
testData = createData(sampleSize = 200, overdispersion = 0.0, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1, family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# run the quantile test
x = testQuantiles(simulationOutput)
x # the test shows a combined p-value, corrected for multiple testing
x$pvals # pvalues for the individual quantiles
x$qgamFits # access the fitted quantile regression
summary(x$qgamFits[[1]]) # summary of the first fitted quantile

# possible to test user-defined quantiles
testQuantiles(simulationOutput, quantiles = c(0.7))

#  example with missing environmental predictor
fittedModel <- glm(observedResponse ~ 1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)
testQuantiles(simulationOutput, predictor = testData$Environment1)

# the quantile test is automatically performed in
## Not run:
plot(simulationOutput)
plotResiduals(simulationOutput)

## End(Not run)
```

`testResiduals` *DHARMa general residual test*

Description

Calls both uniformity and dispersion test

Usage

```
testResiduals(simulationOutput, plot = T)
```

Arguments

`simulationOutput`	an object of class DHARMa with simulated quantile residuals, either created via `simulateResiduals` or by `createDHARMa` for simulations created outside DHARMa
`plot`	if T, plots functions of the tests are called

Details

This function is a wrapper for the various test functions implemented in DHARMa. Currently, this function calls the `testUniformity` and the `testDispersion` functions. All other tests (see list below) have to be called by hand.

Author(s)

Florian Hartig

See Also

`testUniformity`, `testOutliers`, `testDispersion`, `testZeroInflation`, `testGeneric`, `testTemporalAutocorrelation`
`testSpatialAutocorrelation`, `testQuantiles`

Examples

```
testData = createData(sampleSize = 200, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# the plot function runs 4 tests
# i) KS test i) Dispersion test iii) Outlier test iv) quantile test
plot(simulationOutput, quantreg = TRUE)

# testResiduals tests distribution, dispersion and outliers
testResiduals(simulationOutput)

####### Individual tests #######
```

```
# KS test for correct distribution of residuals
testUniformity(simulationOutput)

# Dispersion test
testDispersion(simulationOutput) # tests under and overdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion

# if model is refitted, a different test will be called
simulationOutput2 <- simulateResiduals(fittedModel = fittedModel, refit = TRUE, seed = 12)
testDispersion(simulationOutput2)

# often useful to test dispersion per group (e.g. binomial data, see vignette)
simulationOutput3 = recalculateResiduals(simulationOutput, group = testData$group)
testDispersion(simulationOutput3)

# Outlier test (number of observations outside simulation envelope)
testOutliers(simulationOutput)

# testing zero inflation
testZeroInflation(simulationOutput)

# testing generic summaries
countOnes <- function(x) sum(x == 1)  # testing for number of 1s
testGeneric(simulationOutput, summary = countOnes) # 1-inflation
testGeneric(simulationOutput, summary = countOnes, alternative = "less") # 1-deficit

means <- function(x) mean(x) # testing if mean prediction fits
testGeneric(simulationOutput, summary = means)

spread <- function(x) sd(x) # testing if mean sd fits
testGeneric(simulationOutput, summary = spread)
```

`testSimulatedResiduals` *Residual tests*

Description

Residual tests

Usage

```
testSimulatedResiduals(simulationOutput)
```

Arguments

simulationOutput
: an object of class DHARMa with simulated quantile residuals, either created via simulateResiduals or by createDHARMa for simulations created outside DHARMa

Details

Deprecated, switch your code to using the testResiduals function

Author(s)

Florian Hartig

---

testSpatialAutocorrelation

Test for spatial autocorrelation

Description

This function performs a standard test for spatial autocorrelation on the simulated residuals

Usage

```
testSpatialAutocorrelation(simulationOutput, x = NULL, y = NULL,
  distMat = NULL, alternative = c("two.sided", "greater", "less"),
  plot = T)
```

Arguments

simulationOutput
: an object of class DHARMa with simulated quantile residuals, either created via simulateResiduals or by createDHARMa for simulations created outside DHARMa

x
: the x coordinate, in the same order as the data points. If not provided, random values will be created

y
: the y coordinate, in the same order as the data points. If not provided, random values will be created

distMat
: optional distance matrix. If not provided, a distance matrix will be calculated based on x and y. See details for explanation

alternative
: a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis

plot
: whether to plot output

Details

The function performs Moran.I test from the package ape, based on the provided distance matrix of the data points.

There are several ways to specify this distance. If a distance matrix (distMat) is provided, calculations will be based on this distance matrix, and x,y coordinates will only used for the plotting (if provided) If distMat is not provided, the function will calculate the euclidian distances between x,y coordinates, and test Moran.I based on these distances.

If no x/y values are provided, random values will be created. The sense of being able to run the test with x/y = NULL (random values) is to test the rate of false positives under the current residual structure (random x/y corresponds to H0: no spatial autocorrelation), e.g. to check if the test has nominal error rates for particular residual structures.

Testing for spatial autocorrelation requires unique x,y values - if you have several observations per location, either use the recalculateResiduals function to aggregate residuals per location, or extract the residuals from the fitted object, and plot / test each of them independently for spatially repeated subgroups (a typical scenario would repeated spatial observation, in which case one could plot / test each time step separately for temporal autocorrelation). Note that the latter must be done by hand, outside testSpatialAutocorrelation.

Note

Important to note for all autocorrelation tests (spatial / temporal): the autocorrelation tests are valid to check for residual autocorrelation in models that don't assume such a correlation (in this case, you can use conditional or unconditional simulations), or if there is remaining residual autocorrelation after accounting for it in a spatial/temporal model (in that case, you have to use conditional simulations), but if checking unconditional simulations from a model with an autocorrelation structure on data that corresponds to this model, they will be significant, even if the model fully accounts for this structure.

This behavior is not really a bug, but rather originates from the definition of the quantile residuals: quantile residuals are calculated independently per data point, i.e. without consideratin of any correlation structure between data points that may exist in the simulations. As a result, the simulated distributions from a unconditional simulaton will typically not reflect the correlation structure that is present in each single simulation, and the same is true for the subsequently calculated quantile residuals.

The bottomline here is that spatial / temporal / other autoregressive models should either be tested based on conditional simulations, or (ideally) custom tests should be used that are not based on quantile residuals, but rather compare the correlation structure in the simulated data with the correlation structure in the observed data.

Author(s)

Florian Hartig

See Also

`testResiduals`, `testUniformity`, `testOutliers`, `testDispersion`, `testZeroInflation`, `testGeneric`, `testTemporalAutocorrelation`, `testQuantiles`

Examples

```
testData = createData(sampleSize = 40, family = gaussian())
fittedModel <- lm(observedResponse ~ Environment1, data = testData)
res = simulateResiduals(fittedModel)

# Standard use
testSpatialAutocorrelation(res, x =  testData$x, y = testData$y)

# If x and y is not provided, random values will be created
testSpatialAutocorrelation(res)

# Alternatively, one can provide a distance matrix
dM = as.matrix(dist(cbind(testData$x, testData$y)))
testSpatialAutocorrelation(res, distMat = dM)

# if there are multiple observations with the same x values,
# create first ar group with unique values for each location
# then aggregate the residuals per location, and calculate
# spatial autocorreation on the new group

res2 = recalculateResiduals(res, group = testData$group)
testSpatialAutocorrelation(res)


# carefull when using REs to account for spatially clustered (but not grouped)
# data. this originates from https://github.com/florianhartig/DHARMa/issues/81


# Assume our data is divided into clusters, where observations are close together
# but not at te same point, and we suspect that observations in clusters are
# autocorrelated

clusters = 100
subsamples = 10
size = clusters * subsamples

testData = createData(sampleSize = size, family = gaussian(), numGroups = clusters )
testData$x  = rnorm(clusters)[testData$group] + rnorm(size, sd = 0.01)
testData$y  = rnorm(clusters)[testData$group] + rnorm(size, sd = 0.01)

# It's a good idea to use a RE to take out the cluster effects. This accounts
# for the autocorrelation within clusters

library(lme4)
fittedModel <- lmer(observedResponse ~ Environment1 + (1|group), data = testData)

# DHARMa default is to re-simulted REs - this means spatial pattern remains
# because residuals are still clustered

res = simulateResiduals(fittedModel)
testSpatialAutocorrelation(res, x =  testData$x, y = testData$y)
```

```
# However, it should disappear if you just calculate an aggregate residuals per cluster
# Because at least how the data are simualted, cluster are spatially independent

res2 = recalculateResiduals(res, group = testData$group)
testSpatialAutocorrelation(res2,
                           x =  aggregate(testData$x, list(testData$group), mean)$x,
                           y = aggregate(testData$y, list(testData$group), mean)$x)

# For lme4, it's also possible to simulated residuals conditional on fitted
# REs (re.form). Conditional on the fitted REs (i.e. accounting for the clusters)
# the residuals should now be indepdendent. The remaining RSA we see here is
# probably due to the RE shrinkage

res = simulateResiduals(fittedModel, re.form = NULL)
testSpatialAutocorrelation(res, x =  testData$x, y = testData$y)
```

`testTemporalAutocorrelation` *Test for temporal autocorrelation*

Description

This function performs a standard test for temporal autocorrelation on the simulated residuals

Usage

```
testTemporalAutocorrelation(simulationOutput, time = NULL,
  alternative = c("two.sided", "greater", "less"), plot = T)
```

Arguments

`simulationOutput`	an object with simulated residuals created by `simulateResiduals`
`time`	the time, in the same order as the data points. If not provided, random values will be created
`alternative`	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis
`plot`	whether to plot output

Details

The function performs a Durbin-Watson test on the uniformly scaled residuals, and plots the residuals against time. The DB test was originally be designed for normal residuals. In simulations, I didn't see a problem with this setting though. The alternative is to transform the uniform residuals to normal residuals and perform the DB test on those.

If no time values are provided, random values will be created. The sense of being able to run the test with time = NULL (random values) is to test the rate of false positives under the current residual

structure (random time corresponds to H0: no spatial autocorrelation), e.g. to check if the test has noninal error rates for particular residual structures (note that Durbin-Watson originally assumes normal residuals, error rates seem correct for uniform residuals, but may not be correct if there are still other residual problems).

Testing for temporal autocorrelation requires unique time values - if you have several observations per time value, either use the recalculateResiduals function to aggregate residuals per time step, or extract the residuals from the fitted object, and plot / test each of them independently for temporally repeated subgroups (typical choices would be location / subject etc.). Note that the latter must be done by hand, outside testSpatialAutocorrelation.

Note

Important to note for all autocorrelation tests (spatial / temporal): the autocorrelation tests are valid to check for residual autocorrelation in models that don't assume such a correlation (in this case, you can use conditional or unconditional simulations), or if there is remaining residual autocorrelation after accounting for it in a spatial/temporal model (in that case, you have to use conditional simulations), but if checking unconditional simulations from a model with an autocorrelation structure on data that corresponds to this model, they will be significant, even if the model fully accounts for this structure.

This behavior is not really a bug, but rather originates from the definition of the quantile residuals: quantile residuals are calculated independently per data point, i.e. without consideratin of any correlation structure between data points that may exist in the simulations. As a result, the simulated distributions from a unconditional simulaton will typically not reflect the correlation structure that is present in each single simulation, and the same is true for the subsequently calculated quantile residuals.

The bottomline here is that spatial / temporal / other autoregressive models should either be tested based on conditional simulations, or (ideally) custom tests should be used that are not based on quantile residuals, but rather compare the correlation structure in the simulated data with the correlation structure in the observed data.

Author(s)

Florian Hartig

See Also

testResiduals, testUniformity, testOutliers, testDispersion, testZeroInflation, testGeneric, testSpatialAutocorrelation, testQuantiles

Examples

```
testData = createData(sampleSize = 40, family = gaussian())
fittedModel <- lm(observedResponse ~ Environment1, data = testData)
res = simulateResiduals(fittedModel)

# Standard use
testTemporalAutocorrelation(res, time =  testData$time)

# If no time is provided, random values will be created
```

```
testTemporalAutocorrelation(res)

# If you have several observations per time step

timeSeries1 = createData(sampleSize = 40, family = gaussian())
timeSeries1$location = 1
timeSeries2 = createData(sampleSize = 40, family = gaussian())
timeSeries2$location = 2
testData = rbind(timeSeries1, timeSeries2)

fittedModel <- lm(observedResponse ~ Environment1, data = testData)
res = simulateResiduals(fittedModel)

# for this, you cannot do testTemporalAutocorrelation(res, time = testData$time)
# because here we would have observations with the same time, i.e.
# zero difference in time. We have two options a) aggregate observations
# b) calculate / test per subset. Testing per subset might also be useful
# if you have several locations, regardless of whether the times are
# identical, because you would expect the autocorrelation structure to be
# independent per location

# testing grouped residuals

res = recalculateResiduals(res, group = testData$time)
testTemporalAutocorrelation(res, time = unique(testData$time))

# plotting and testing per subgroup

# extract subgroup
testData$Residuals = res$scaledResiduals
temp = testData[testData$location == 1,]

# plots and tests
plot(Residuals ~ time, data = temp)
lmtest::dwtest(temp$Residuals ~ 1, order.by = temp$time)
```

`testUniformity` *Test for overall uniformity*

Description

This function tests the overall uniformity of the simulated residuals in a DHARMa object

Usage

```
testUniformity(simulationOutput, alternative = c("two.sided", "less",
  "greater"), plot = T)
```

Arguments

simulationOutput	an object of class DHARMa with simulated quantile residuals, either created via simulateResiduals or by createDHARMa for simulations created outside DHARMa
alternative	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis. See ks.test for details
plot	if T, plots calls plotQQunif as well

Details

The function applies a ks.test for uniformity on the simulated residuals.

Author(s)

Florian Hartig

See Also

testResiduals, testOutliers, testDispersion, testZeroInflation, testGeneric, testTemporalAutocorrelation, testSpatialAutocorrelation, testQuantiles

Examples

```
testData = createData(sampleSize = 200, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# the plot function runs 4 tests
# i) KS test i) Dispersion test iii) Outlier test iv) quantile test
plot(simulationOutput, quantreg = TRUE)

# testResiduals tests distribution, dispersion and outliers
testResiduals(simulationOutput)

####### Individual tests #######

# KS test for correct distribution of residuals
testUniformity(simulationOutput)

# Dispersion test
testDispersion(simulationOutput) # tests under and overdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion

# if model is refitted, a different test will be called
simulationOutput2 <- simulateResiduals(fittedModel = fittedModel, refit = TRUE, seed = 12)
testDispersion(simulationOutput2)
```

```
# often useful to test dispersion per group (e.g. binomial data, see vignette)
simulationOutput3 = recalculateResiduals(simulationOutput, group = testData$group)
testDispersion(simulationOutput3)

# Outlier test (number of observations outside simulation envelope)
testOutliers(simulationOutput)

# testing zero inflation
testZeroInflation(simulationOutput)

# testing generic summaries
countOnes <- function(x) sum(x == 1)  # testing for number of 1s
testGeneric(simulationOutput, summary = countOnes) # 1-inflation
testGeneric(simulationOutput, summary = countOnes, alternative = "less") # 1-deficit

means <- function(x) mean(x) # testing if mean prediction fits
testGeneric(simulationOutput, summary = means)

spread <- function(x) sd(x) # testing if mean sd fits
testGeneric(simulationOutput, summary = spread)
```

`testZeroInflation` *Tests for zero-inflation*

Description

This function compares the observed number of zeros with the zeros expected from simulations.

Usage

```
testZeroInflation(simulationOutput, ...)
```

Arguments

`simulationOutput`	an object of class DHARMa with simulated quantile residuals, either created via `simulateResiduals` or by `createDHARMa` for simulations created outside DHARMa
`...`	further arguments to `testGeneric`

Details

The plot shows the expected distribution of zeros against the observed values, the ratioObsSim shows observed vs. simulated zeros. A value < 1 means that the observed data has less zeros than

expected, a value > 1 means that it has more zeros than expected (aka zero-inflation). Per default, the function tests both sides.

Some notes about common problems / questions:

* Zero-inflation tests after fitting the model are crucial to see if you have zero-inflation. Just because there are a lot of zeros doesn't mean you have zero-inflation, see Warton, D. I. (2005). Many zeros does not mean zero inflation: comparing the goodness-of-fit of parametric models to multivariate abundance data. Environmetrics 16(3), 275-289.

* That being said, zero-inflation tests are often not a reliable guide to decide wheter to add a zi term or not. In general, model structures should be decided on ideally a priori, if that is not possible via model selection techniques (AIC, BIC, WAIC, Bayes Factor). A zero-inflation test should only be run after that decision, and to validate the decision that was taken.

Note

This function is a wrapper for testGeneric, where the summary argument is set to function(x) sum(x == 0)

Author(s)

Florian Hartig

See Also

testResiduals, testUniformity, testOutliers, testDispersion, testGeneric, testTemporalAutocorrelation, testSpatialAutocorrelation, testQuantiles

Examples

```
testData = createData(sampleSize = 200, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# the plot function runs 4 tests
# i) KS test i) Dispersion test iii) Outlier test iv) quantile test
plot(simulationOutput, quantreg = TRUE)

# testResiduals tests distribution, dispersion and outliers
testResiduals(simulationOutput)

####### Individual tests #######

# KS test for correct distribution of residuals
testUniformity(simulationOutput)

# Dispersion test
testDispersion(simulationOutput) # tests under and overdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion
testDispersion(simulationOutput, alternative = "less") # only underdispersion

# if model is refitted, a different test will be called
```

```
simulationOutput2 <- simulateResiduals(fittedModel = fittedModel, refit = TRUE, seed = 12)
testDispersion(simulationOutput2)

# often useful to test dispersion per group (e.g. binomial data, see vignette)
simulationOutput3 = recalculateResiduals(simulationOutput, group = testData$group)
testDispersion(simulationOutput3)

# Outlier test (number of observations outside simulation envelope)
testOutliers(simulationOutput)

# testing zero inflation
testZeroInflation(simulationOutput)

# testing generic summaries
countOnes <- function(x) sum(x == 1)  # testing for number of 1s
testGeneric(simulationOutput, summary = countOnes) # 1-inflation
testGeneric(simulationOutput, summary = countOnes, alternative = "less") # 1-deficit

means <- function(x) mean(x) # testing if mean prediction fits
testGeneric(simulationOutput, summary = means)

spread <- function(x) sd(x) # testing if mean sd fits
testGeneric(simulationOutput, summary = spread)
```

`transformQuantiles` *Transform quantiles to pdf (deprecated)*

Description

The purpose of this function was to transform the DHARMa quantile residuals (which have a uniform distribution) to a particular pdf. Since DHARMa 0.3.0, this functionality is integrated in the `residuals.DHARMa` function. Please switch to using this function.

Usage

```
transformQuantiles(res, quantileFunction = qnorm, outlierValue = 7)
```

Arguments

`res`	an object with simulated residuals created by `simulateResiduals`
`quantileFunction`	optional - a quantile function to transform the uniform 0/1 scaling of DHARMa to another distribution
`outlierValue`	if a quantile function with infinite support (such as dnorm) is used, residuals that are 0/1 are mapped to -Inf / Inf. outlierValues allows to convert -Inf / Inf values to an optional min / max value.

Index