Package 'DHARMa'

November 7, 2026

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

Version 0.5.1

Date 2026-11-07

Description 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', 'GLMMadaptive', 'spaMM', and 'brms' (simple models); phylogenetic linear models from 'phylolm' (classes 'phylolm' and 'phyloglm'); 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, temporal and phylogenetic autocorrelation.

Depends R (>= 3.5.0)

Imports stats, graphics, utils, grDevices, Matrix, parallel, gap, lmtest, ape, qgam (>= 1.3.2), lme4

Suggests knitr, testthat (>= 3.0.0), rmarkdown, KernSmooth, sfsmisc, MASS, mgcv, mgcViz (>= 0.1.9), spaMM (>= 3.2.0), GLMMadaptive, glmmTMB (>= 1.1.2.3), phylolm (>= 2.6.5), brms

Enhances rstan, phyr, rjags, BayesianTools

License GPL (>= 3)

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

LazyData TRUE

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

VignetteBuilder knitr

Encoding UTF-8

Config/testthat/edition 3

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author Florian Hartig [aut, cre] (ORCID:
<https://orcid.org/0000-0002-6255-9059>),
Lukas Lohse [ctb],
Melina de Souza Leite [ctb] (ORCID:
<https://orcid.org/0000-0003-0505-0667>),
Cosmina Werneke [ctb] (ORCID: <https://orcid.org/0009-0008-7629-2173>)

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

Repository CRAN

Date/Publication 2026-11-07 10:10:02 UTC

Contents

DHARMa-package . . . 3
benchmarkRuntime . . . 4
createData . . . 5
createDHARMa . . . 7
getData . . . 9
getFamily . . . 10
getFitted . . . 11
getFittedResponse . . . 12
getFixedEffects . . . 14
getObservedResponse . . . 15
getPearsonResiduals . . . 16
getPredictorNames . . . 18
getQuantile . . . 19
getRandomState . . . 20
getRefit . . . 22
getResiduals . . . 23
getSimulations . . . 25
hist.DHARMa . . . 28
hurricanes . . . 29
outliers . . . 31
plot.DHARMa . . . 32
plot.DHARMaBenchmark . . . 33
plotConventionalResiduals . . . 34
plotQQunif . . . 34
plotResiduals . . . 36
plotResidualsAll . . . 39
plotSimulatedResiduals . . . 40
print.DHARMa . . . 40
recalculateResiduals . . . 41
residuals.DHARMa . . . 43
runBenchmarks . . . 45

simulateLRT . . . 47
simulateResiduals . . . 49
testCategorical . . . 53
testDispersion . . . 55
testGeneric . . . 58
testOutliers . . . 60
testOverdispersion . . . 62
testOverdispersionParametric . . . 63
testPDistribution . . . 63
testPhylogeneticAutocorrelation . . . 64
testQuantiles . . . 66
testResiduals . . . 68
testSimulatedResiduals . . . 69
testSpatialAutocorrelation . . . 70
testTemporalAutocorrelation . . . 73
testUniformity . . . 78
testZeroInflation . . . 80
transformQuantiles . . . 82

Index **83**

`DHARMa-package` *DHARMa: Residual Diagnostics for Hierarchical (Multi-Level / Mixed) Regression Models*

Description

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', 'GLMMadaptive', 'spaMM', and 'brms' (simple models); phylogenetic linear models from 'phylolm' (classes 'phylolm' and 'phyloglm'); 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, temporal and phylogenetic autocorrelation.

Details

To get started with the package, look at the vignette and start with simulateResiduals

Funding

Development and maintenance of the DHARMa package has been supported by the **Deutsche Forschungsgemeinschaft (DFG)**, project number 528747641 (2024-now).

Author(s)

Maintainer: Florian Hartig <florian.hartig@biologie.uni-regensburg.de> (ORCID)

Authors:

- Florian Hartig <florian.hartig@biologie.uni-regensburg.de> (ORCID)

Other contributors:

- Lukas Lohse [contributor]
- Melina de Souza Leite (ORCID) [contributor]
- Cosmina Werneke (ORCID) [contributor]

References

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

See Also

Useful links:

- https://florianhartig.github.io/DHARMa/
- Report bugs at https://github.com/florianhartig/DHARMa/issues

Examples

```
# simulating a dataset
testData = createData(sampleSize = 200, family = poisson(),
                      fixedEffects = c(1,1),
                      randomEffectVariance = 1, numGroups = 10)
# fitting glm model
fittedModel <- glm(observedResponse ~ Environment1 + Environment2,
                   family = "poisson", data = testData)

# simulating residuals
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

#plotting residuals
plot(simulationOutput, quantreg = FALSE)
```

---

benchmarkRuntime *Benchmark runtimes of several functions*

Description

Benchmark runtimes of several functions

Usage

```
benchmarkRuntime(createModel, evaluationFunctions, n)
```

Arguments

createModel	a function that creates and returns a fitted model.
evaluationFunctions	a list of functions that are to be evaluated on the fitted models.
n	number of replicates.

Details

This is a small helper function designed to benchmark runtimes of several operations that are to be performed on a list of fitted models. In the example, this is used to benchmark the runtimes of several DHARMa tests.

Author(s)

Florian Hartig

Examples

```
createModel = function(){
  testData = createData(family = poisson(), overdispersion = 1,
                        randomEffectVariance = 0)
  fittedModel <- glm(observedResponse ~ Environment1, data = testData, family = poisson())
  return(fittedModel)
}

a = function(m){
  testUniformity(m, plot = FALSE)$p.value
}

b = function(m){
  testDispersion(m, plot = FALSE)$p.value
}

c = function(m){
  testDispersion(m, plot = FALSE, type = "PearsonChisq")$p.value
}


evaluationFunctions = list(a,b, c)

benchmarkRuntime(createModel, evaluationFunctions, 2)
```

---

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 = FALSE, replicates = 1, hasNA = FALSE)
```

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`	a family function for the error distribution and link function to be used in the model to simulate data from. (See `stats::family()` for details of family functions for GLMs.)
`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 Poisson 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 temporal autocorrelation.
`spatialAutocorrelation`	strength of spatial autocorrelation.
`factorResponse`	should the response be transformed to a factor (intended to be used for 0/1 data).
`replicates`	number of datasets to create.
`hasNA`	should an NA be added to the environmental predictor (for test purposes).

Examples

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

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

# with zero-inflation

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

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

# 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 = FALSE, seed = 123,
  method = c("PIT", "traditional"), rotation = NULL)
```

Arguments

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

`observedResponse`
: true observations.

`fittedPredictedResponse`
: optional fitted predicted response, on the scale of the observed response (e.g. predict(..., type = "response"); for binomial models with more than one trial, the expected number of successes). This is used e.g. by testDispersion. 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 expectation 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 to keep the random seed on a fixed value 123. This means that you will always get the same randomization and thus the 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.

`method`
: the quantile randomization method used. The two options implemented at the moment are probability integral transform (PIT-) residuals (current default), and the "traditional" randomization procedure, that was used in DHARMa until version 0.3.0. For details, see getQuantile.

`rotation`
: optional rotation of the residual space to remove residual autocorrelation. See details in simulateResiduals, section *residual auto-correlation* for an extended explanation, and getQuantile for syntax.

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.

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 handcoded 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)
```

`getData` *Get model data*

Description

Wrapper to get the data that was used to fit a model.

Usage

```
getData(object, ...)

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

## S3 method for class 'merMod'
getData(object, ...)

## S3 method for class 'brmsfit'
getData(object, ...)
```

Arguments

`object`	a fitted model.
`...`	additional parameters to be passed on.

Note

The data is retrieved from the environment where the model was fit, which is usually your current environment. If you delete the data from the environment or change it after fitting the model this function will probably not work or get the wrong dataset, respectively.

Author(s)

Florian Hartig

See Also

getObservedResponse, getSimulations, getRefit, getFixedEffects, getFitted

---

`getFamily` *Get model family*

Description

Wrapper to get the family of a fitted model.

Usage

```
getFamily(object, ...)

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

## S3 method for class 'phylolm'
getFamily(object, ...)

## S3 method for class 'phyloglm'
getFamily(object, ...)
```

Arguments

`object`	a fitted model.
`...`	additional parameters to be passed on.

Author(s)

Florian Hartig

See Also

getObservedResponse, getSimulations, getRefit, getFixedEffects, getFitted

getFitted *Get fitted/predicted values*

Description

Wrapper to get the fitted/predicted response of the model at the response scale.

Usage

```
getFitted(object, ...)

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

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

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

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

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

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

## S3 method for class 'brmsfit'
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 extract the fitted values, which is implemented via predict(model, type = "response") for most model classes.

If you implement this function for a new model class, you should include an option to modify which random effects (REs) are included in the predictions. If this option is not available, it is essential that predictions are provided marginally/unconditionally, i.e. without the RE estimates (because of https://github.com/florianhartig/DHARMa/issues/43), which corresponds to re-form = ~0 in lme4.

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)

# predictions on the scale of the observed response
# (differs from getFitted only for k/n binomial models)
getFittedResponse(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))

# get data frame that was used to fit the model
getData(fittedModel)
```

---

getFittedResponse *Get fitted/predicted values on the scale of the observed response*

Description

Wrapper to get the fitted/predicted values of the model on the same scale as the observed response (getObservedResponse) and the simulations (getSimulations).

Usage

```
getFittedResponse(object, ...)

## Default S3 method:
```

```
getFittedResponse(object, ...)

## S3 method for class 'brmsfit'
getFittedResponse(object, ...)
```

Arguments

object	a fitted model.
...	additional parameters to be passed on, usually to getFitted.

Details

For most models, this is identical to getFitted. The exception are binomial and beta-binomial models with more than one trial per observation (k/n models), specified either as cbind(successes, failures) or as proportions with the number of trials provided via weights. For these models, getObservedResponse and getSimulations return the number of successes, whereas getFitted returns the predicted proportions. getFittedResponse returns the predicted proportions multiplied by the number of trials, i.e. the expected number of successes.

The fitted response is used by testDispersion (refit = FALSE) to calculate the raw residuals (observed or simulated response minus fitted response).

See Also

getFitted, getObservedResponse, getSimulations

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)

# predictions on the scale of the observed response
# (differs from getFitted only for k/n binomial models)
getFittedResponse(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))
```

```
# get data frame that was used to fit the model
getData(fittedModel)
```

`getFixedEffects` *Extract fixed effects of a supported model*

Description

A wrapper to extract fixed effects of a supported model.

Usage

```
getFixedEffects(object, ...)

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

## S3 method for class 'MixMod'
getFixedEffects(object, ...)

## S3 method for class 'brmsfit'
getFixedEffects(object, ...)
```

Arguments

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

Note

Note that for brms models, the mean is used as the measure of central tendency for the fixed effects as in brms::fixef.

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)
```

```
# predictions on the scale of the observed response
# (differs from getFitted only for k/n binomial models)
getFittedResponse(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))

# get data frame that was used to fit the model
getData(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, ...)

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

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

Arguments

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

Details

The purpose of this function is to safely extract the observed 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)

# predictions on the scale of the observed response
# (differs from getFitted only for k/n binomial models)
getFittedResponse(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))

# get data frame that was used to fit the model
getData(fittedModel)
```

---

getPearsonResiduals *Get Pearson residuals*

Description

Wrapper to get the Pearson residuals of a fitted model.

Usage

```
getPearsonResiduals(object, ...)

## Default S3 method:
getPearsonResiduals(object, ...)
```

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

## S3 method for class 'brmsfit'
getPearsonResiduals(object, ...)
```

Arguments

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

Details

The purpose of this wrapper is to extract the Pearson residuals of a fitted model.

Note

Pearson residuals are not supported for brms models.

Author(s)

Florian Hartig

See Also

getObservedResponse, getSimulations, 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)

# predictions on the scale of the observed response
# (differs from getFitted only for k/n binomial models)
getFittedResponse(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))

# get data frame that was used to fit the model
getData(fittedModel)
```

`getPredictorNames` *Get predictor names*

Description

Wrapper to get the names of the predictors used in the model

Usage

```
getPredictorNames(object, ...)

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

## S3 method for class 'MixMod'
getPredictorNames(object, ...)

## S3 method for class 'brmsfit'
getPredictorNames(object, ...)
```

Arguments

`object`	a fitted model.
`...`	additional parameters to be passed on.

Author(s)

Florian Hartig

See Also

getObservedResponse, getSimulations, getRefit, getFixedEffects, getFitted

getQuantile *Calculate Residual Quantiles*

Description

Calculates residual quantiles from a given simulation.

Usage

```
getQuantile(simulations, observed, integerResponse, method = c("PIT",
  "traditional"), rotation = NULL)
```

Arguments

simulations	a matrix with simulations from a fitted model. Rows = observations, columns = replicate simulations.
observed	a vector with the observed data.
integerResponse	is the response integer-valued? Only has an effect for method = "traditional".
method	the quantile randomization method used. See details.
rotation	optional rotation of the residuals. You can either provide as a known or estimated covariance matrix (e.g. when fitting an AR1 model), or use the argument "estimated", in which case the residual covariance will be approximated by simulations. See comments in details.

Details

The function calculates residual quantiles from the simulated data. For continuous distributions, this will simply be the value of the ecdf.

Randomization procedure for discrete data

For discrete data, there are two options implemented.

The current default (available since DHARMa 0.3.1) are probability integral transform (PIT-) residuals (Smith, 1985; Dunn & Smyth, 1996; see also Warton, et al., 2017).

Before DHARMa 0.3.1, a different randomization procedure was used, in which a U(-0.5, 0.5) distribution was added on observations and simulations for discrete distributions. For a completely discrete distribution, the two procedures should deliver equivalent results, but the second method has the disadvantage that (a) one has to know if the distribution is discrete (DHARMa tries to recognize this automatically), and (b) that it leads to inefficiencies for some distributions such as the Tweedie, which are partly continuous, partly discrete (see e.g. issue #168 on DHARMa GitHub page).

Rotation (optional)

The getQuantile function includes an additional option to rotate residuals prior to calculating the quantile residuals. This option should ONLY be used when the fitted model includes a particular residual covariance structure, such as an AR1 or a spatial or phylogenetic CAR model.

For these models, residuals calculated from unconditional simulations will include the specified covariance structure, which will trigger e.g. temporal autocorrelation tests and can inflate type I errors of other tests. The idea of the rotation is to rotate the residual space according to the covariance structure of the fitted model, such that the rotated residuals are conditional independent (provided the fitted model is correct).

If the residual covariance of the fitted model at the response scale can be extracted (e.g. when fitting gls type models), it would be best to extract it and provide this covariance matrix to the rotation option. If that is not the case, providing the argument "estimated" to rotation will estimate the covariance from the data simulated by the model. This is probably without alternative for GLMMs, where the covariance at the response scale is likely not known / provided, but note that this approximation will tend to have considerable error and may be slow to compute for high-dimensional data. If you try to estimate the rotation from simulations, you should set n as high as possible! See testTemporalAutocorrelation for a practical example.

The rotation of residuals implemented here is similar to the Variogram.lme() and Variongram.gls() functions in nlme package using the argument resType = "normalized".

References

Smith, J. Q. "Diagnostic checks of non-standard time series models." Journal of Forecasting 4.3 (1985): 283-291.

Dunn, P.K., & Smyth, G.K. (1996). Randomized quantile residuals. Journal of Computational and Graphical Statistics 5, 236-244.

Warton, David I., Loïc Thibaut, and Yi Alice Wang. "The PIT-trap—A “model-free” bootstrap procedure for inference about regression models with discrete, multivariate responses." PloS one 12.7 (2017).

`getRandomState` *Record and restore a random state*

Description

The aim of this function is to record, manipulate and restore a random state.

Usage

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

Arguments

`seed`	seed argument to set.seed(), typically a number. Additional options: NULL = no seed is set, but return includes function for restoring random seed. F = function does nothing, i.e. neither seed is changed, nor does the returned function do anything.

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 after function execution, as well as a function to restore the previous state before the function execution.

Author(s)

Florian Hartig

Examples

```
set.seed(13)
runif(1)

# testing the function in standard settings
currentSeed = .Random.seed
x = getRandomState(123)
runif(1)
x$restoreCurrent()
all(.Random.seed == currentSeed)

# if no seed was set in env, 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
runif(1) # re-create a seed

# with seed = false
currentSeed = .Random.seed
x = getRandomState(FALSE)
runif(1)
x$restoreCurrent()
all(.Random.seed == currentSeed)

# with seed = NULL
currentSeed = .Random.seed
x = getRandomState(NULL)
runif(1)
x$restoreCurrent()
all(.Random.seed == currentSeed)
```

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 'merMod'
getRefit(object, newresp, ...)

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

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

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

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

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

## S3 method for class 'brmsfit'
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)

# predictions on the scale of the observed response
# (differs from getFitted only for k/n binomial models)
getFittedResponse(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))

# get data frame that was used to fit the model
getData(fittedModel)
```

---

getResiduals *Get model residuals*

Description

Wrapper to get the residuals of a fitted model.

Usage

```
getResiduals(object, ...)

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

## S3 method for class 'MixMod'
getResiduals(object, ...)

## S3 method for class 'brmsfit'
getResiduals(object, ...)
```

Arguments

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

Details

The purpose of this wrapper is to standardize the extraction of model residuals. Similar to some other functions, a key question is whether to calculate those conditional or unconditional on the fitted Random Effects.

Author(s)

Florian Hartig

See Also

getObservedResponse, getSimulations, 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)

# predictions on the scale of the observed response
# (differs from getFitted only for k/n binomial models)
getFittedResponse(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))

# get data frame that was used to fit the model
getData(fittedModel)
```

`getSimulations` *Get model simulations*

Description

Wrapper to simulate from a fitted model.

Usage

```
getSimulations(object, nsim = 1, simulateREs = c("conditional",
  "unconditional", "user-specified"), type = c("normal", "refit"), ...)

## Default S3 method:
getSimulations(object, nsim = 1,
  simulateREs = c("conditional", "unconditional", "user-specified"),
  type = c("normal", "refit"), ...)

## S3 method for class 'negbin'
getSimulations(object, nsim = 1,
  simulateREs = c("conditional", "unconditional", "user-specified"),
  type = c("normal", "refit"), ...)

## S3 method for class 'gam'
getSimulations(object, nsim = 1,
  simulateREs = c("conditional", "unconditional", "user-specified"),
  type = c("normal", "refit"), mgcViz = TRUE, ...)

## S3 method for class 'merMod'
getSimulations(object, nsim = 1,
  simulateREs = c("conditional", "unconditional", "user-specified"),
  type = c("normal", "refit"), ...)

## S3 method for class 'glmmTMB'
getSimulations(object, nsim = 1,
  simulateREs = c("conditional", "unconditional", "user-specified"),
  type = c("normal", "refit"), ...)
```

```
## S3 method for class 'HLfit'
getSimulations(object, nsim = 1,
  simulateREs = c("conditional", "unconditional", "user-specified"),
  type = c("normal", "refit"), ...)

## S3 method for class 'MixMod'
getSimulations(object, nsim = 1,
  simulateREs = c("conditional", "unconditional", "user-specified"),
  type = c("normal", "refit"), ...)

## S3 method for class 'phylolm'
getSimulations(object, nsim = 1,
  simulateREs = c("conditional", "unconditional", "user-specified"),
  type = c("normal", "refit"), ...)

## S3 method for class 'phyloglm'
getSimulations(object, nsim = 1,
  simulateREs = c("conditional", "unconditional", "user-specified"),
  type = c("normal", "refit"), ...)

## S3 method for class 'brmsfit'
getSimulations(object, nsim = 1,
  simulateREs = c("conditional", "unconditional", "user-specified"),
  type = c("normal", "refit"), ...)
```

Arguments

`object`	a fitted model.
`nsim`	number of simulations.
`simulateREs`	which hierarchical levels should be re-simulated. If `conditional`, the simulations are done conditional on all fitted random effects (default). If `unconditional`, all hierarchical levels are re-simulated, including the random effects. With `user-specified`, the default simulate function of the respective fitted model object is used. See details and simulateResiduals.
`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.
`mgcViz`	whether simulations should be created with mgcViz (if mgcViz is available)

Details

The purpose of this function is to wrap or implement the simulate function of different model classes to return simulations from fitted models in a standardized way.

One important parameter is simulateRE, which controls which hierarchical levels are held constant (conditioned on), and which are re-simulated. The default as of DHARMa 0.5.0 is to simulate "conditional" on all fitted random effects. The setting "unconditional" re-simulates all REs.

With simulateREs = "user-specified", users can supply additional parameters to the simulate function of the respective model class and thus condition on specific REs or structures. The exact behavior will depend on the regression package. For details, please see vignette, or consult the help of the different packages. If choosing simulateREs = "user-specified" with no additional parameters, the default simulation function of the respective regression package is used. This corresponds to the DHARMa behavior prior to 0.5.0.

If the model was fit with weights and the respective model class does not include the weights in the simulations, getSimulations will throw a warning. The background is if weights are used on the likelihood directly, then what is fitted is effectively a pseudo likelihood, and there is no way to directly simulate from the specified likelihood. Whether or not residuals can be used in this case depends very much on what is tested and how weights are used. I'm sorry to say that it is hard to give a general recommendation, you have to consult someone that understands how weights are processed in the respective model class.

Binomial models with a cbind(successes, failures) response and prior weights are not supported; getSimulations stops for them.

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)

# predictions on the scale of the observed response
# (differs from getFitted only for k/n binomial models)
getFittedResponse(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))

# get data frame that was used to fit the model
getData(fittedModel)
```

`hist.DHARMa` *Histogram of DHARMa residuals*

Description

The function produces a histogram from a DHARMa output. Outliers are marked in red.

Usage

```
## S3 method for class 'DHARMa'
hist(x, breaks = seq(-0.02, 1.02, len = 53),
  col = c(.Options$DHARMaSignalColor, rep("lightgrey", 50),
  .Options$DHARMaSignalColor), 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`	color for histogram bars.
`main`	plot title.
`xlab`	plot x-axis label.
`cex.main`	plot cex.main.
`...`	other arguments to be passed on to hist().

Details

The function calls hist() to create a histogram of the scaled residuals. Outliers are marked red as default but it can be changed by setting `options(DHARMaSignalColor = "red")` to a different color. See `getOption("DHARMaSignalColor")` for the current setting.

See Also

plot.DHARMa, plotResiduals

Examples

```
testData = createData(sampleSize = 200, family = poisson(),
                      fixedEffects = c(1,1),
                      randomEffectVariance = 1, numGroups = 10)
testData$Environment2[1] = NA
fittedModel <- glm(observedResponse ~ Environment1 + Environment2,
                   family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

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

plot(simulationOutput)
# for all functions, quantreg = T (default) will be more informative
# but slower. Alternative:

plot(simulationOutput, quantreg = FALSE)

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

plotQQunif(simulationOutput = simulationOutput,
           testDispersion = FALSE,
           testUniformity = FALSE,
           testOutliers = FALSE)

hist(simulationOutput)
```

---

hurricanes *Hurricanes*

Description

A dataset on hurricane strength and fatalities in the US between 1950 and 2012. The data originates from the study by Jung et al., PNAS, 2014, who claim that the masculinity / femininity of a hurricane name has a causal effect on fatalities, presumably through a different perception of danger caused by the names.

Format

A 'data.frame': 92 obs. of 14 variables

Year Year of the hurricane (1950-2012)

Name Name of the hurricane

MasFem Masculinity-femininity rating of the hurricane's name in the range 1 = very masculine, 11 = very feminine.

MinPressure_before Minimum air pressure (909-1002).

Minpressure_Updated_2014 Updated minimum air pressure (909-1003).

Gender_MF Binary gender categorization based on MasFem (male = 0, female = 1).

Category Strength of the hurricane in categories (1:7). (1 = not at all, 7 = very intense).

alldeaths Human deaths occurred (1:256).

NDAM Normalized damage in millions (1:75.000). The raw (dollar) amounts of property damage caused by hurricanes were obtained, and the unadjusted dollar amounts were normalized to 2013 monetary values by adjusting them to inflation, wealth and population density.

Elapsed_Yrs Elapsed years since the occurrence of hurricanes (1:63).

Source MWR/wikipedia ()

ZMasFem Scaled (MasFem)

ZMinPressure_A Scaled (Minpressure_Updated_2014)

ZNDAM Scaled (NDAM) ...

References

Jung, K., Shavitt, S., Viswanathan, M., & Hilbe, J. M. (2014). Female hurricanes are deadlier than male hurricanes. Proceedings of the National Academy of Sciences, 111(24), 8782-8787.

Examples

```
## Not run:
# Loading hurricanes dataset

library(DHARMa)

data(hurricanes)
str(hurricanes)

# this is the model fit by Jung et al.
library(glmmTMB)
originalModelGAM = glmmTMB(alldeaths ~ scale(MasFem) *
                             (scale(Minpressure_Updated_2014) + scale(NDAM)),
                           data = hurricanes, family = nbinom2)

# no significant deviation in the general DHARMa plot
res <- simulateResiduals(originalModelGAM)
plot(res)

# but residuals ~ NDAM looks funny, which was pointed
# out by Bob O'Hara in a blog post after publication of the paper
plotResiduals(res, hurricanes$NDAM)

# we also find temporal autocorrelation
res2 = recalculateResiduals(res, group = hurricanes$Year)
testTemporalAutocorrelation(res2, time = sort(unique(hurricanes$Year)))

# task: try to address these issues - in many instances, this will
```

```
# make the MasFem predictor n.s.

## End(Not run)
```

`outliers` *Return outliers*

Description

Returns the outliers of a DHARMa object.

Usage

```
outliers(object, lowerQuantile = 0, upperQuantile = 1,
  return = c("index", "logical"))
```

Arguments

`object`	an object with simulated residuals created by simulateResiduals.
`lowerQuantile`	lower threshold for outliers. Default is zero = outside simulation envelope.
`upperQuantile`	upper threshold for outliers. Default is 1 = outside simulation envelope.
`return`	whether to return indices of outliers or a logical vector.

Details

First of all, note that the standard definition of outlier in the DHARMa plots and outlier tests is an observation that is outside the simulation envelope. How far outside that is depends a lot on how many simulations you do. If you have 100 data points and do 100 simulations, you would expect to have about two "outliers" on average (one at each margin), even with a perfectly fitting model. This is in fact what the outlier test tests.

Thus, keep in mind that for a small number of simulations, outliers are mostly a technical term: these are points that are outside our simulations, but we don't know how far away they are.

If you are seriously interested in HOW FAR outside the expected distribution a data point is, you should increase the number of simulations in simulateResiduals to be sure to get the tail of the data distribution correctly. In this case, it may make sense to adjust lowerQuantile and upperQuantile, e.g. to 0.025, 0.975, which would define outliers as values outside the central 95% of the distribution.

Also, note that outliers are particularly concerning if they have a strong influence on the model fit. One could test the influence, for example, by removing them from the data, or by some measures of leverage, e.g. generalizations for Cook's distance as in Pinho, L. G. B., Nobre, J. S., & Singer, J. M. (2015). Cook's distance for generalized linear mixed models. Computational Statistics & Data Analysis, 82, 126–136. doi:10.1016/j.csda.2014.08.008. At the moment, however, no such function is provided in DHARMa.

`plot.DHARMa` *DHARMa standard residual plots*

Description

This S3 function creates standard plots for the simulated residuals contained in an object of class DHARMa, using plotQQunif (left panel) and plotResiduals (right panel).

Usage

```
## S3 method for class 'DHARMa'
plot(x, title = "DHARMa residual", ...)
```

Arguments

`x`	an object of class DHARMa with simulated residuals created by simulateResiduals.
`title`	the title for both panels (plotted via mtext, outer = TRUE).
`...`	further options for plotResiduals. Consider in particular parameters quantreg, rank and asFactor. xlab, ylab and main passed here only change the right panel (the residual plot); use plotResiduals and plotQQunif to fully control the labels of the individual plots.

Details

The function creates a plot with two panels. The left panel is a uniform qq plot (calling plotQQunif), and the right panel shows residuals against predicted values (calling plotResiduals), with outliers as a star/asterisk and highlighted in red if outlier test is significant (default color but see Note).

Very briefly, we would expect that a correctly specified model shows:

a) a straight 1-1 line, as well as non-significance of the displayed tests in the qq-plot (left) -> evidence for a correct overall residual distribution (for more details on the interpretation of this plot, see plotQQunif)

b) visual homogeneity of residuals in both vertical and horizontal direction, as well as non-significance of quantile tests in the res ~ predictor plot (for more details on the interpretation of this plot, see plotResiduals)

Deviations from these expectations can be interpreted similar to a linear regression. See the vignette for detailed examples.

Note

The color for highlighting outliers and significant tests can be changed by setting `options(DHARMaSignalColor = "red")` to a different color. See `getOption("DHARMaSignalColor")` for the current setting. This is convenient for a color-blind friendly display, since red and black are difficult for some people to separate.

See Also

plotResiduals, plotQQunif

Examples

```
testData = createData(sampleSize = 200, family = poisson(),
                      fixedEffects = c(1,1),
                      randomEffectVariance = 1, numGroups = 10)
testData$Environment2[1] = NA
fittedModel <- glm(observedResponse ~ Environment1 + Environment2,
                   family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

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

plot(simulationOutput)
# for all functions, quantreg = T (default) will be more informative
# but slower. Alternative:

plot(simulationOutput, quantreg = FALSE)

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

plotQQunif(simulationOutput = simulationOutput,
           testDispersion = FALSE,
           testUniformity = FALSE,
           testOutliers = FALSE)

hist(simulationOutput)
```

---

plot.DHARMaBenchmark *Plots DHARMa benchmarks*

Description

The function plots the result of an object of class DHARMaBenchmark, created by runBenchmarks.

Usage

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

Arguments

x	object of class DHARMaBenchmark, created by runBenchmarks.
...	parameters to pass to the plot function.

Details

The function will create two types of plots, depending on whether the run contains only a single value (or no value) of the control parameter, or whether a vector of control values is provided:

If a single or no value of the control parameter is provided, the function will create boxplots of the estimated p-values, with the number of significant p-values plotted to the left.

If a control parameter is provided, the function will plot the proportion of significant p-values against the control parameter, with 95% CIs based on the performed replicates displayed as confidence bands.

See Also

runBenchmarks

`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. Optionally, tests for uniformity, outliers and dispersion can be added.

Usage

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

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 gap::qqunif.

Details

The function calls qqunif() from the R package gap to create a quantile-quantile plot for a uniform distribution, and overlays tests for particular distributional problems as specified. When tests are displayed, significant p-values are highlighted in the color red by default. This can be changed by setting `options(DHARMaSignalColor = "red")` to a different color. See `getOption("DHARMaSignalColor")` for the current setting.

See Also

plot.DHARMa, plotResiduals

Examples

```
testData = createData(sampleSize = 200, family = poisson(),
                      fixedEffects = c(1,1),
                      randomEffectVariance = 1, numGroups = 10)
testData$Environment2[1] = NA
fittedModel <- glm(observedResponse ~ Environment1 + Environment2,
                   family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

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

plot(simulationOutput)
# for all functions, quantreg = T (default) will be more informative
# but slower. Alternative:

plot(simulationOutput, quantreg = FALSE)

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

plotQQunif(simulationOutput = simulationOutput,
           testDispersion = FALSE,
           testUniformity = FALSE,
           testOutliers = FALSE)

hist(simulationOutput)
```

`plotResiduals` *Generic res ~ pred scatter plot with spline or quantile regression on top*

Description

The function creates a generic residual plot with either spline or quantile regression to highlight patterns in the residuals. Outliers are marked in star/asterisk form and highlighted in red if the outlier test is significant.

Usage

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

Arguments

`simulationOutput`	an 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(s) against which the residuals should be plotted. Default is to use the predicted(simulationOutput). Recommended to specify as formula. Use `form = ~.` to plot against all predictors. See details.
`quantreg`	whether to perform a quantile regression based on testQuantiles or a smooth spline around the mean. Default NULL chooses T for nObs < 10000, 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 quantiles should be plotted. Default is 0.25, 0.5, 0.75.
`absoluteDeviation`	if T, switch from displaying normal quantile residuals to absolute deviation from the mean expectation of 0.5 (calculated as 2 * abs(res - 0.5)). The purpose of this is to test explicitly for heteroskedasticity, see details.
`...`	additional arguments to plot / boxplot. Ignored if several panels are plotted.

Details

The function plots residuals against a predictor (by default against the fitted value, extracted from the DHARMa object, or any other predictor). The shaded gray areas indicate 95% confidence intervals of the quantile estimates (1.96 * standard error).

Outliers are drawn if simulationOutput is a DHARMa object and highlighted in red if the outlier test is significant (for information on definition and interpretation of outliers, see testOutliers). See the note below to change the highlighting color of the outliers.

To provide a visual aid for detecting deviations from uniformity in the y-direction, the plot function calculates an (optional) quantile regression of the residuals, by default for the 0.25, 0.5 and 0.75 quantiles. Since the residuals should be uniformly distributed for a correctly specified model, the theoretical expectations for these regressions are straight lines at 0.25, 0.5 and 0.75, shown as dashed black lines on the plot. However, even for a perfect model, some deviation from these expectations is to be expected by chance, especially if the sample size is small. The function therefore tests whether the deviation of the fitted quantile regression from the expectation is significant, using testQuantiles. If so, the significant quantile regression is highlighted in red (as default) and a warning is displayed in the plot. See the note below to change the color of significant quantile lines.

Overdispersion typically manifests itself as Q1 (0.25) deviating towards 0 and Q3 (0.75) deviating towards 1. Heteroskedasticity manifests itself as non-parallel quantile lines. To diagnose heteroskedasticity and overdispersion, it can be helpful to additionally plot the absolute deviation of the residuals from the mean expectation of 0.5, using the option absoluteDeviation = T. In this case, we would again expect Q1-Q3 quantile lines at 0.25, 0.5, 0.75, but greater dispersion (also locally in the case of heteroskedasticity) always manifests itself in deviations towards 1.

The quantile regression can take some time to calculate, especially for larger datasets. For this reason, quantreg = F can be set to generate a smooth spline instead. This is the default for n > 10000.

If form is specified as a formula, e.g. `form = ~ your_predictor`, NAs will be handled automatically (recommended). If `form = ~.`, separate plots for all predictors in the model are produced. If `form = ~ predictor1 + predictor2`, a separate plot for each specified predictor is produced. If an additional grouping variable is specified, e.g. `form = ~ predictor|group`, a separate plot for the predictor within each grouping level is produced. Be careful with this command: if you have many levels, the plot may not display nicely. If you are interested in a specific group, you can use `form = ~ predictor|group == "group_level"`, and for several groups `form = ~ predictor|group %in% c("level1", "level2")`, which produces a separate plot for each of these levels. If form is **not** a formula, e.g. `form = data$predictor`, NAs are **not** handled automatically. For phyr and gamm4$mer models, you need to specify form in this way.

If the predictor 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 (within-group). To test if deviations from those expectations are significant, KS-tests per group and a Levene test for homogeneity of variances are performed. See testCategorical for details.

Value

If quantile tests are performed, the function returns them invisibly. If several panels are plotted, the function returns NULL.

Note

If nObs > 10,000, the scatter plot is replaced by graphics::smoothScatter().

The color for highlighting outliers and quantile lines/splines with significant tests can be changed by setting `options(DHARMaSignalColor = "red")` to a different color. See `getOption("DHARMaSignalColor")` for the current setting. This is convenient for a color-blind friendly display, since red and black are difficult for some people to distinguish.

See Also

plotQQunif, testQuantiles, testOutliers

Examples

```
testData = createData(sampleSize = 200, family = poisson(),
                      fixedEffects = c(1,1),
                      randomEffectVariance = 1, numGroups = 10)
testData$Environment2[1] = NA
fittedModel <- glm(observedResponse ~ Environment1 + Environment2,
                   family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

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

# Default in DHARMa is to show predictions rank transformed
# if you want plots based on raw predictions, use rank = F
plotResiduals(simulationOutput, rank = FALSE)

# smooth scatter plot - default for large datasets with n > 10,000
plotResiduals(simulationOutput, rank = TRUE, smoothScatter = TRUE)



  # It is very advisable to plot the residual against all predictors
  # the following syntax uses the predictor values from the fitted model
  plotResiduals(simulationOutput, form = ~ Environment1)

  # plot against all predictors
  plotResiduals(simulationOutput, form = ~.)

  # if pred is a factor, or if asFactor = TRUE, will produce a boxplot
  plotResiduals(simulationOutput, form = ~group)

  # plot residuals against multiple predictors at once
  plotResiduals(simulationOutput, form = ~Environment1 + Environment2)

  # plot residuals against a predictor for all respective group levels
  plotResiduals(simulationOutput, form = ~Environment1|group, quantreg = FALSE)

  # plot residuals against a predictor for a specific group level, here group 1
  plotResiduals(simulationOutput, form = ~Environment1|group == "1")
```

```
# plot residuals against a predictor for selected group levels, here groups 1 and 2
plotResiduals(simulationOutput, form = ~Environment1|group %in% c("1", "2"), quantreg = FALSE)


# alternatively, you can plot against a variable from the global environment
# in this case, the model function automatically removed one row of observations
# because there was an NA in Environment2. When using the variable Environment1
# from the global environment, we have to remove this observation as well
plotResiduals(simulationOutput,
              form = testData$Environment1[complete.cases(testData)])


# to diagnose overdispersion and heteroskedasticity it can be useful to
# display residuals as absolute deviation from the expected mean 0.5
plotResiduals(simulationOutput, absoluteDeviation = TRUE)
```

`plotResidualsAll` *Create multiple residual plots simultaneously (deprecated)*

Description

DEPRECATED, use plotResiduals with `form = ~.`, `~a + b`, `~x | group` or `~x | group %in% c(...)` instead

Usage

```
plotResidualsAll(simulationOutput, predictorNames,
  predictorType = c("predictor", "groupLevel"), form = NULL, ...)
```

Arguments

`simulationOutput`	an object with simulated residuals created by simulateResiduals.
`predictorNames`	names of the predictors / group levels to be plotted.
`predictorType`	whether multiple plots should be created based on different predictor variables (predictor) or a single predictor and each corresponding group level (groupLevel).
`form`	optional predictor (and grouping variable) against which the residuals should be plotted.
`...`	further options for plotResiduals.

Note

This function is deprecated. Use plotResiduals

plotSimulatedResiduals *DHARMa standard residual plots (deprecated)*

Description

DEPRECATED, use plot() instead

Usage

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

Arguments

simulationOutput	an object with simulated residuals created by simulateResiduals
...	further options for plotResiduals. Consider in particular parameters quantreg, rank and asFactor. xlab, ylab and main passed here only change the right panel (the residual plot); use plotResiduals and plotQQunif to fully control the labels of the individual plots.

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,
  sel = NULL, seed = 123, method = c("PIT", "traditional"),
  rotation = NULL)
```

Arguments

`simulationOutput`	an object with simulated residuals created by simulateResiduals.
`group`	a grouping variable with the same dimensions as the residuals in simulationOutput. If specified as formula, e.g. group = ~group, NAs are handled automatically (recommended).
`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 anymore if you choose general functions, such as sd etc.
`sel`	an optional vector for selecting the data to be aggregated.
`seed`	the random seed to be used within DHARMa. The default setting, recommended for most users, is to keep the random seed on a fixed value of 123. This means that you will always get the same randomization and thus the 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.
`method`	the quantile randomization method used. The two options implemented at the moment are probability integral transform (PIT-) residuals (current default), and the "traditional" randomization procedure, that was used in DHARMa until version 0.3.0. For residuals simulated with refit = TRUE, "traditional" is always used, as in simulateResiduals; rotation is then not available. For details, see getQuantile.
`rotation`	optional rotation of the residual space to remove residual autocorrelation. See details in simulateResiduals, section *residual auto-correlation* for an extended explanation, and getQuantile for syntax.

Details

The function aggregates the observed and simulated data per group according to the function provided by the aggregateBy option. DHARMa residuals are then calculated exactly as for a single data point (see getQuantile for details). The function stops if fewer than 2 residuals remain after grouping and selection, and warns if fewer than 10 remain.

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 = 100, overdispersion = 0.5, family = poisson())
fittedModel <- glmer(observedResponse ~ Environment1 + (1|group),
                     family = "poisson", data = testData)

# simulate residuals (default behavior, conditional on the fitted random effects)
simulationOutput1 <- simulateResiduals(fittedModel = fittedModel)
plot(simulationOutput1, quantreg = FALSE)

# simulate residuals unconditional on the fitted random effects (REs are re-simulated)
simulationOutput2 <- simulateResiduals(fittedModel = fittedModel,
                                       simulateREs = "unconditional")
plot(simulationOutput2, quantreg = FALSE)

# simulate residuals user-specified using lme4 syntax (e.g. conditional only on a specific RE)
simulationOutput3 <- simulateResiduals(fittedModel = fittedModel,
                                       simulateREs = "user-specified",
                                       re.form = ~(1|group))
plot(simulationOutput3, quantreg = FALSE)

# one of the possible test, for other options see ?testResiduals / vignette
testDispersion(simulationOutput1)

# the calculated residuals can be accessed via
residuals(simulationOutput1)

# transform residuals to other pdf, see ?residuals.DHARMa for details
residuals(simulationOutput1, quantileFunction = qnorm, outlierValues = c(-7,7))

# get residuals that are outside the simulation envelope
outliers(simulationOutput1)

# calculating aggregated residuals per group
# group here specified as formula (recommended)
```

```
# but you can also use a variable from your environment, e.g. testData$group
simulationOutput4 = recalculateResiduals(simulationOutput1, group = ~group)
plot(simulationOutput4, quantreg = FALSE)
# we see one residual point per RE

# with group as variable in your environment
# but then you need to exclude NAs by hand
simulationOutput5 = recalculateResiduals(simulationOutput1, group = testData$group)

# calculating residuals only for subset of the data
simulationOutput6 = recalculateResiduals(simulationOutput1, sel = testData$group == 1)
plot(simulationOutput6, quantreg = FALSE)

# or with sel as formula
simulationOutput7 = recalculateResiduals(simulationOutput1, sel = ~Environment1<0.5)

# as condition based on a variable in your environment
simulationOutput8 = recalculateResiduals(simulationOutput1, sel = testData$Environment1<0.5)

# or combine group and sel
simulationOutput9 = recalculateResiduals(simulationOutput1, group = ~group, sel = ~Environment1<0.5)

# or as subset of rows
# (note: order of rows is based on residuals, not on original dataframe)
# so be careful with NAs!
simulationOutput10 = recalculateResiduals(simulationOutput1, sel = 1:20)
```

`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 qnorm) 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 = 100, overdispersion = 0.5, family = poisson())
fittedModel <- glmer(observedResponse ~ Environment1 + (1|group),
                     family = "poisson", data = testData)

# simulate residuals (default behavior, conditional on the fitted random effects)
simulationOutput1 <- simulateResiduals(fittedModel = fittedModel)
plot(simulationOutput1, quantreg = FALSE)

# simulate residuals unconditional on the fitted random effects (REs are re-simulated)
simulationOutput2 <- simulateResiduals(fittedModel = fittedModel,
                                       simulateREs = "unconditional")
plot(simulationOutput2, quantreg = FALSE)

# simulate residuals user-specified using lme4 syntax (e.g. conditional only on a specific RE)
simulationOutput3 <- simulateResiduals(fittedModel = fittedModel,
                                       simulateREs = "user-specified",
                                       re.form = ~(1|group))
plot(simulationOutput3, quantreg = FALSE)

# one of the possible test, for other options see ?testResiduals / vignette
testDispersion(simulationOutput1)

# the calculated residuals can be accessed via
residuals(simulationOutput1)

# transform residuals to other pdf, see ?residuals.DHARMa for details
residuals(simulationOutput1, quantileFunction = qnorm, outlierValues = c(-7,7))

# get residuals that are outside the simulation envelope
outliers(simulationOutput1)
```

```
# calculating aggregated residuals per group
# group here specified as formula (recommended)
# but you can also use a variable from your environment, e.g. testData$group
simulationOutput4 = recalculateResiduals(simulationOutput1, group = ~group)
plot(simulationOutput4, quantreg = FALSE)
# we see one residual point per RE

# with group as variable in your environment
# but then you need to exclude NAs by hand
simulationOutput5 = recalculateResiduals(simulationOutput1, group = testData$group)

# calculating residuals only for subset of the data
simulationOutput6 = recalculateResiduals(simulationOutput1, sel = testData$group == 1)
plot(simulationOutput6, quantreg = FALSE)

# or with sel as formula
simulationOutput7 = recalculateResiduals(simulationOutput1, sel = ~Environment1<0.5)

# as condition based on a variable in your environment
simulationOutput8 = recalculateResiduals(simulationOutput1, sel = testData$Environment1<0.5)

# or combine group and sel
simulationOutput9 = recalculateResiduals(simulationOutput1, group = ~group, sel = ~Environment1<0.5)

# or as subset of rows
# (note: order of rows is based on residuals, not on original dataframe)
# so be careful with NAs!
simulationOutput10 = recalculateResiduals(simulationOutput1, sel = 1:20)
```

`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 = FALSE, exportGlobal = FALSE, ...)
```

Arguments

`calculateStatistics`	the statistics to be benchmarked. Should return one value, or a vector of values. If controlValues are given, must accept a parameter control.
`controlValues`	optionally, a vector with a control parameter (e.g. to vary the strength of a problem the test should be specific to). See help for an example.

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.
exportGlobal	whether the global environment should be exported to the parallel nodes. This will use more memory. Set to true only if your function calculate statistics depends on other functions or global variables.
...	additional parameters to calculateStatistics.

Value

an object with list structure of class DHARMaBenchmark. Contains entry simulations with a matrix of simulations, and entry summaries with an list of summaries (significant (T/F), mean, p-value for KS-test uniformity). Can be plotted with plot.DHARMaBenchmark.

Note

The benchmark functions 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.

Author(s)

Florian Hartig

See Also

plot.DHARMaBenchmark

Examples

```
# define a function that will run a simulation and return a number of statistics, typically p-values
returnStatistics <- function(control = 0){
  testData = createData(sampleSize = 20, family = poisson(), overdispersion = control,
                        randomEffectVariance = 0)
 fittedModel <- glm(observedResponse ~ Environment1, data = testData, family = poisson())
  res <- simulateResiduals(fittedModel = fittedModel, n = 250)
 out <- c(testUniformity(res, plot = FALSE)$p.value, testDispersion(res, plot = FALSE)$p.value)
  return(out)
}

# testing a single return
returnStatistics()

# running benchmark for a fixed simulation, increase nRep for sensible results
out = runBenchmarks(returnStatistics, nRep = 5)

# plotting results depend on whether a vector or a single value is provided for control
plot(out)
```

```
## Not run:

# running benchmark with varying control values
out = runBenchmarks(returnStatistics, controlValues = c(0,0.5,1), nRep = 100)
plot(out)

# running benchmark can be done using parallel cores
out = runBenchmarks(returnStatistics, nRep = 100, parallel = TRUE)
out = runBenchmarks(returnStatistics, controlValues = c(0,0.5,1), nRep = 10, parallel = TRUE)

# Alternative plot function using vioplot, provides nicer pictures

plot.DHARMaBenchmark <- function(x, ...){

  if(length(x$controlValues)== 1){
   vioplot::vioplot(x$simulations[,x$nSummaries:1], las = 2, horizontal = TRUE, side = "right",
                    areaEqual = FALSE,
                    main = "p distribution under H0",
                    ylim = c(-0.15,1), ...)
    abline(v = 1, lty = 2)
    abline(v = c(0.05, 0), lty = 2, col = "red")
    text(-0.1, x$nSummaries:1, labels = x$summaries$propSignificant[-1])

  }else{
    res = x$summaries$propSignificant
    matplot(res$controlValues, res[,-1], type = "l",
            main = "Power analysis", ylab = "Power", ...)
    legend("bottomright", colnames(res[,-1]),
            col = 1:x$nSummaries, lty = 1:x$nSummaries, lwd = 2)

  }
}


## End(Not run)
```

`simulateLRT` *Simulated likelihood ratio tests for (generalized) linear mixed models*

Description

This function uses the DHARMa model wrappers to generate simulated likelihood ratio tests (LRTs) for (generalized) linear mixed models based on a parametric bootstrap. The motivation for using a simulated LRT rather than a standard ANOVA or AIC for model selection in mixed models is that df for mixed models are not clearly defined, thus standard ANOVA based on Chi2 statistics or AIC are unreliable, in particular for models with large contributions of REs to the likelihood.

Interpretation of the results as in a normal LRT: the null hypothesis is that M0 is correct, the test checks if the increase in likelihood of M1 is higher than expected, using data simulated from M0.

Usage

```
simulateLRT(m0, m1, n = 250, seed = 123, plot = TRUE,
  suppressWarnings = TRUE, saveModels = FALSE, ...)
```

Arguments

m0	null model.
m1	alternative model.
n	number of simulations.
seed	random seed.
plot	whether null distribution should be plotted.
suppressWarnings	whether to suppress warnings that occur during refitting the models to simulated data. See details for explanations.
saveModels	whether to save refitted models.
...	additional parameters to pass on to getSimulations, in particular simulateREs, and to the simulate function of the model object. See getSimulations for details.

Details

The function performs a simulated LRT, which works as follows:

1. H0: Model 0 is correct.
2. Our test statistic is the log LR of M1/M0, i.e. logLik(M1) - logLik(M0). Empirical value will always be >= 0 because in a nested setting, the more complex model cannot have a worse likelihood.
3. To generate an expected distribution of the test statistic under H0, we simulate new response data under M0, refit M0 and M1 on this data, and calculate the LRs.
4. Based on this, calculate p-values etc. in the usual way.

About warnings: warnings such as "boundary (singular) fit: see ?isSingular" will likely occur in this function and are not necessarily the sign of a problem. lme4 warns if RE variances are fit to zero. This is desired / likely in this case, however, because we are simulating data with zero RE variances. Therefore, warnings are turned off per default. For diagnostic reasons, you can turn warnings on, and possibly also inspect fitted models via the parameter saveModels to see if there are any other problems in the re-fitted models.

Data simulations are performed by getSimulations, which is a wrapper for the respective model functions. The simulations follow the same defaults and options as in simulateResiduals: since DHARMa 0.5.0, the default is simulateREs = "conditional", i.e. data are simulated conditional on the fitted random effects of M0. Use simulateREs = "unconditional" to re-simulate all random effects, or simulateREs = "user-specified" without further arguments to reproduce the behavior of DHARMa < 0.5.0 (the default of the simulate function of the respective package); arguments are supplied via ... (see simulateResiduals and getSimulations for details and for which packages support these options). This setting only matters if M0 contains random effects.

Note

The logic of an LRT assumes that M0 is nested in M1, which guarantees that L(M1) >= L(M0). The function does not explicitly check if models are nested and will work as long as data can be simulated from M0 that can be refit with M0 and M1; however, I would strongly advise against using this for non-nested models unless you have a good statistical reason for doing so.

Also, note that LRTs may be unreliable when fit with REML or some other kind of penalized / restricted ML. Therefore, you should fit the model with ML for use in this function.

Author(s)

Florian Hartig

Examples

```
library(DHARMa)
library(lme4)

# create test data
set.seed(123)
dat <- createData(sampleSize = 200, randomEffectVariance = 1)

# define Null and alternative model (should be nested)
m1 = glmer(observedResponse ~ Environment1 + (1|group), data = dat, family = "poisson")
m0 = glm(observedResponse ~ Environment1 , data = dat, family = "poisson")

## Not run:
# run LRT - n should be increased to at least 250 for a real study
out = simulateLRT(m0, m1, n = 10)

# To inspect warnings thrown during the refits:
out = simulateLRT(m0, m1, saveModels = TRUE, suppressWarnings = FALSE, n = 10)
summary(out$saveModels[[2]]$refittedM1) # RE SD = 0, no problem
# If there are warnings that seem problematic,
# could try changing the optimizer or iterations

## End(Not run)
```

---

simulateResiduals *Create simulated residuals*

Description

The function creates scaled residuals by simulating from the fitted model. Residuals can be extracted with residuals.DHARMa. See testResiduals for an overview of residual tests, plot.DHARMa for an overview of available plots.

Usage

```
simulateResiduals(fittedModel, n = 250, simulateREs = c("conditional",
  "unconditional", "user-specified"), refit = FALSE,
  integerResponse = NULL, plot = FALSE, seed = 123, method = c("PIT",
  "traditional"), rotation = NULL, ...)
```

Arguments

`fittedModel`	a fitted model of a class supported by DHARMa.
`n`	number of simulations. The smaller the number, the higher the stochastic error on the residuals. Also, for very small n, discretization artefacts can influence the tests. Default is 250, which is a relatively safe value. You can consider increasing to 1000 to stabilize the simulated values.
`simulateREs`	decides which REs / hierarchical levels should be re-simulated. If `conditional`, simulations are done conditional on all fitted random effects (default). If `unconditional`, all REs or hierarchical levels are re-simulated. Setting `user-specified` allows supplying package-specific arguments to decide which REs are to be re-simulated. If none are supplied, the defaults of the respective fitted model object is used. See details and further comments in getSimulations.
`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 refitted 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 to the residuals to maintain 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, plot.DHARMa will be directly run after the residuals have been calculated.
`seed`	the random seed to be used within DHARMa. The default setting, recommended for most users, is to keep the random seed on a fixed value of 123. This means that you will always get the same randomization and thus the same result when running the same code. If NULL, no new seed is set, but previous random state will be restored after simulation. If 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.
`method`	for refit = FALSE, the quantile randomization method is used. The two options implemented at the moment are probability integral transform (PIT-) residuals (current default), and the "traditional" randomization procedure, that was used in DHARMa until version 0.3.0. refit = T will always use "traditional", regardless of the value of method, and rotation is then not available. For details, see getQuantile.
`rotation`	optional rotation of the residual space prior to calculating the quantile residuals. The main purpose of this is to account for residual covariance as created by temporal, spatial or phylogenetic autocorrelation. See details below, section

residual autocorrelation as well as the help of getQuantile and, for a practical example, testTemporalAutocorrelation.

... further parameters to pass on to the simulate and update function of the model object (via DHARMa::getSimulations and DHARMa::getRefit). If simulateREs = "user-specified", an important use of this is to specify whether simulations should be conditional on all, some or none of the current random effect estimates, e.g. via re.form. Note that not all models support syntax to specify conditional or unconditional simulations. See details and getSimulations.

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 or RE model, we have several stochastic processes on top of each other. For example, in a typical Poisson GLMM, we have a lower level normal distribution (random effect), whose result enters at the next hierarchical level into a Poisson distribution. For other hierarchical models such as state-space models, similar considerations apply. When simulating new data in such a situation, we have to decide if we want to re-simulate all stochastic levels (unconditional), only the top level conditional on all other levels (conditional), or if we want to re-simulate a specific subset of REs while conditioning on the others. As of DHARMa 0.5.0, the default was changed to simulating conditionally on all fitted REs. If this is not what you want, you can switch simulateREs to either "unconditional", or "user-specified", which allows to condition on a user-specified set of REs. With "user-specified", if no additional arguments are set, this results in the package behavior previous to version 0.5.0. Note that the syntax to change the behavior is different for each package. Also, specific packages may lack certain functionalities. For further details, please see vignette, getSimulations and refer to the help of the different simulate functions (e.g. ?simulate.merMod).

When to use conditional and unconditional simulations: If the model is correctly specified, the simulated residuals should be flat regardless which hierarchical levels you re-simulate. Re-simulating all levels (unconditional) can be advantageous because it tests the model structure as a whole. If REs are correlated with a predictor, for example, this will show up in the residuals. A potential drawback is that re-simulating the lower-level random effects creates more variability and correlation structures in the residuals, which may reduce power for detecting problems in the upper-level stochastic processes and, in some cases, can lead to inflated type I error rates through the induced correlations. In particular dispersion tests may produce different results when switching from conditional to unconditional simulations, and often the conditional simulation is more sensitive. The default was therefore switched to conditional in DHARMa 0.5.0 which we consider to be more safe, but it can make sense to overrule this default for specific tests.

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. I advise against using this method per default (see more comments in the vignette), unless you are really sure that you need it.

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 transform these residuals to another distribution (e.g. so that a correctly specified model will have normal residuals) see residuals.DHARMa.

Integer responses: this is only relevant if method = "traditional", in which case it activates the randomization of the residuals. Usually, this does not need to be changed, as DHARMa will try to automatically check 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 integer 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).

Residual autocorrelation: a common problem is residual autocorrelation. Spatial, temporal and phylogenetic autocorrelation can be tested with testSpatialAutocorrelation, testTemporalAutocorrelation and testPhylogeneticAutocorrelation. If simulations are unconditional, residual correlations will be maintained, even if the autocorrelation is addressed by an appropriate CAR structure. This may be a problem, because autocorrelation may create apparently systematic patterns in plots or tests such as testUniformity. To reduce this problem, either simulate conditional on fitted correlated REs, or rotate residuals via the rotation parameter (the latter will likely only work in approximately linear models). See getQuantile for details on the rotation.

Value

an S3 class of type "DHARMa". Implemented S3 functions include plot.DHARMa, print.DHARMa and residuals.DHARMa. For other functions that can be used on a DHARMa object, see section "See Also" below.

See Also

testResiduals, plotResiduals, recalculateResiduals, outliers

Examples

```
library(lme4)

testData = createData(sampleSize = 100, overdispersion = 0.5, family = poisson())
fittedModel <- glmer(observedResponse ~ Environment1 + (1|group),
                     family = "poisson", data = testData)

# simulate residuals (default behavior, conditional on the fitted random effects)
simulationOutput1 <- simulateResiduals(fittedModel = fittedModel)
plot(simulationOutput1, quantreg = FALSE)

# simulate residuals unconditional on the fitted random effects (REs are re-simulated)
simulationOutput2 <- simulateResiduals(fittedModel = fittedModel,
                                       simulateREs = "unconditional")
plot(simulationOutput2, quantreg = FALSE)
```

```
# simulate residuals user-specified using lme4 syntax (e.g. conditional only on a specific RE)
simulationOutput3 <- simulateResiduals(fittedModel = fittedModel,
                                       simulateREs = "user-specified",
                                       re.form = ~(1|group))
plot(simulationOutput3, quantreg = FALSE)

# one of the possible test, for other options see ?testResiduals / vignette
testDispersion(simulationOutput1)

# the calculated residuals can be accessed via
residuals(simulationOutput1)

# transform residuals to other pdf, see ?residuals.DHARMa for details
residuals(simulationOutput1, quantileFunction = qnorm, outlierValues = c(-7,7))

# get residuals that are outside the simulation envelope
outliers(simulationOutput1)

# calculating aggregated residuals per group
# group here specified as formula (recommended)
# but you can also use a variable from your environment, e.g. testData$group
simulationOutput4 = recalculateResiduals(simulationOutput1, group = ~group)
plot(simulationOutput4, quantreg = FALSE)
# we see one residual point per RE

# with group as variable in your environment
# but then you need to exclude NAs by hand
simulationOutput5 = recalculateResiduals(simulationOutput1, group = testData$group)

# calculating residuals only for subset of the data
simulationOutput6 = recalculateResiduals(simulationOutput1, sel = testData$group == 1)
plot(simulationOutput6, quantreg = FALSE)

# or with sel as formula
simulationOutput7 = recalculateResiduals(simulationOutput1, sel = ~Environment1<0.5)

# as condition based on a variable in your environment
simulationOutput8 = recalculateResiduals(simulationOutput1, sel = testData$Environment1<0.5)

# or combine group and sel
simulationOutput9 = recalculateResiduals(simulationOutput1, group = ~group, sel = ~Environment1<0.5)

# or as subset of rows
# (note: order of rows is based on residuals, not on original dataframe)
# so be careful with NAs!
simulationOutput10 = recalculateResiduals(simulationOutput1, sel = 1:20)
```

---

testCategorical *Test for categorical dependencies*

Description

This function tests if there are problems in a res ~ group structure. It performs two tests: test for within-group uniformity, and test for between-group homogeneity of variances.

Usage

```
testCategorical(simulationOutput, catPred, quantiles = c(0.25, 0.5, 0.75),
  plot = TRUE, ...)
```

Arguments

simulationOutput	an object of class DHARMa, either created via simulateResiduals for supported models or by createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.
catPred	a categorical predictor with the same dimensions as the residuals in simulationOutput. If specified as formula, e.g. catPred = ~group, NAs are handled automatically (recommended).
quantiles	whether to draw the quantile lines.
plot	if TRUE, the function will create an additional plot.
...	additional arguments to boxplot.

Details

The function tests for two common problems: are residuals within each group distributed according to model assumptions, and is the variance between groups heterogeneous.

The test for within-group uniformity is performed via multiple KS-tests, with adjustment of p-values for multiple testing. If the plot is drawn, problematic groups are highlighted in red, and a corresponding message is displayed in the plot.

The test for homogeneity of variances is done with a Levene test. A significant p-value means that group variances are not constant. In this case, you should consider modelling variances, e.g. via ~dispformula in glmmTMB.

Author(s)

Florian Hartig

See Also

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

Examples

```
testData = createData(sampleSize = 100, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# the plot function shows 2 plots and 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)

# KS test for correct distribution within and between groups
# group specified as formula (recommended)
testCategorical(simulationOutput, ~group)

# group specified as variable in your environment
testCategorical(simulationOutput, testData$group)

# Dispersion test - for details see ?testDispersion
testDispersion(simulationOutput) # tests under and overdispersion

# Outlier test (number of observations outside simulation envelope)
# Use type = "bootstrap" for exact values, see ?testOutliers
testOutliers(simulationOutput, type = "binomial")

# 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)
```

`testDispersion` *DHARMa dispersion tests*

Description

This function performs simulation-based tests for over / underdispersion. If type = "DHARMa" (default and recommended), simulation-based dispersion tests are performed. Their behavior differs depending on whether simulations are done with refit = F, or refit = T, and whether data is simulated conditional (see below). If type = "PearsonChisq", a chi2 test on Pearson residuals is performed.

Usage

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

Arguments

`simulationOutput`	an object of class DHARMa, either created via simulateResiduals for supported models or by createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.
`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 testing only for overdispersion. It is recommended to keep the default setting (testing for both over and underdispersion).
`plot`	whether to provide a plot for the results.
`type`	which test to run. Default is DHARMa, other options are PearsonChisq (see details).
`...`	arguments to pass on to testGeneric.

Details

Over / underdispersion means that the observed data is more / less dispersed than expected under the fitted model. There is no unique way to test for dispersion problems, and there are a number of different dispersion tests implemented in various R packages. For detailed recommendations see also Leite M.S., Rettelbach, D. and Hartig, F. (2025), preprint available at https://doi.org/10.32942/X23M14.

The testDispersion function implements several dispersion tests:

Simulation-based dispersion tests (type == "DHARMa")

If type = "DHARMa" (default and recommended), simulation-based dispersion tests are performed. Their behavior differs depending on whether simulations are done with refit = F, or refit = T.

Important: for either refit = T or F, the results of type = "DHARMa" dispersion test will differ depending on whether simulations are done conditional (= conditional on fitted random effects) or unconditional (= REs are re-simulated). The default as of DHARMa 0.5.0 is conditional simulations, which substantially increase the power and sensitivity of the dispersion test (for details see simulateResiduals and the DHARMa vignette).

If refit = F, the function uses testGeneric to compare the variance of the observed raw residuals (i.e. var(observed - predicted), displayed as a red line) against the variance of the simulated residuals (i.e. var(simulated - predicted), histogram). Predictions are taken on the scale of the observed response (see getFittedResponse), i.e. for binomial models with more than one trial, as the expected number

of successes. For objects created with createDHARMa, the provided fittedPredictedResponse is used. The variances are scaled by the variance of all simulated values; this affects only the plotted values, not the reported ratio or p-value. A significant ratio > 1 indicates overdispersion, a significant ratio < 1 underdispersion.

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. Given the computational cost, I would suggest that most users will be satisfied with the standard dispersion test. After recalculateResiduals, observed and refitted Pearson residuals are aggregated in the same way before squaring; if the aggregated observed residuals are all (close to) zero (e.g. grouping variable as fixed effect in a linear model, or as the only predictor in a GLM), the function warns and returns NA as statistic and p-value.

Analytical dispersion tests (type == "PearsonChisq")

This is the test described in https://bbolker.github.io/mixedmodels-misc/glmmFAQ.html#overdispersion, identical to performance::check_overdispersion. Works only if the fitted model provides df.residual and Pearson residuals.

The test statistic is biased to lower values under quite general conditions, and will therefore tend to test significant for underdispersion. It is recommended to use this test only for overdispersion, i.e. use alternative == "greater". Also, obviously, it requires that Pearson residuals are available for the chosen model, which will not be the case for all models / packages.

Note

For particular model classes / situations, there may be more powerful and thus preferable options over the DHARMa test. The advantage of the DHARMa test is that it directly targets the spread of the data (unlike other tests such as dispersion/df, which essentially measure fit and may thus be triggered by problems other than dispersion as well), and it makes practically no assumptions about the fitted model, other than the availability of simulations.

Author(s)

Florian Hartig

References

Leite, MS, Rettelbach, D. & Hartig, F. 2025. Dispersion tests in generalised linear mixed-effects models - a methods comparison and practical guide. EcoEvoRxiv Preprint: https://doi.org/10.32942/X23M14.

See Also

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

Examples

```
library(lme4)
set.seed(123)

testData = createData(sampleSize = 100, overdispersion = 0.5, randomEffectVariance = 1)
fittedModel <- glmer(observedResponse ~ Environment1 + (1|group),
```

```
                    family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# default DHARMa dispersion test using simulations conditional on fitted REs
# (more powerful for mixed models)
testDispersion(simulationOutput)
testDispersion(simulationOutput, alternative = "less", plot = FALSE) # only underdispersion
testDispersion(simulationOutput, alternative = "greater", plot = FALSE) # only overdispersion

# DHARMa also implements the popular Pearson-chisq test that is also on the glmmWiki by Ben Bolker
# The issue with this test is that it requires the df of the model, which are not well defined
# for GLMMs. It is biased towards underdispersion, with bias getting larger with the number
# of RE groups. In doubt, only test for overdispersion
testDispersion(simulationOutput, type = "PearsonChisq", alternative = "greater")

# if refit = TRUE, a different test on simulated Pearson residuals will calculated (see help)
simulationOutput2 <- simulateResiduals(fittedModel = fittedModel, refit = TRUE, seed = 12, n = 20)
testDispersion(simulationOutput2)

# often useful to test dispersion per group (in particular for binomial data, see vignette)
simulationOutputAggregated = recalculateResiduals(simulationOutput, group = testData$group)
testDispersion(simulationOutputAggregated)
```

`testGeneric` *Test for a generic summary statistic based on simulated data*

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, either created via simulateResiduals for supported models or by createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.
`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 applies a user-defined summary to the simulated / observed data of a DHARMa object and then performs a hypothesis test using the ratio Obs / Sim as the test statistic.

The summary is applied directly to the data and not to the residuals, but it can easily be remodeled to apply summaries to the residuals by simply defining something like f = function(x) summary (x - predictions), as done in testDispersion.

Note

The summary function you specify will be applied to the data as it appears in your fitted model, which may not always be what you want.

As an example, consider the case where we want to test for n-inflation in k/n data. If you provide your data via cbind (k, n-k), you have to test for n-inflation, but if you provide your data via k/n and weights = n, you should test for 1-inflation. When in doubt, check how the data is represented internally in model.frame(model) or via simulate(model).

Author(s)

Florian Hartig

See Also

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

Examples

```
testData = createData(sampleSize = 100, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# the plot function shows 2 plots and 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)

# KS test for correct distribution within and between groups
```

```
# group specified as formula (recommended)
testCategorical(simulationOutput, ~group)

# group specified as variable in your environment
testCategorical(simulationOutput, testData$group)

# Dispersion test - for details see ?testDispersion
testDispersion(simulationOutput) # tests under and overdispersion

# Outlier test (number of observations outside simulation envelope)
# Use type = "bootstrap" for exact values, see ?testOutliers
testOutliers(simulationOutput, type = "binomial")

# 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 outside the simulation envelope are larger or smaller than expected

Usage

```
testOutliers(simulationOutput, alternative = c("two.sided", "greater",
  "less"), margin = c("both", "upper", "lower"), type = c("default",
  "bootstrap", "binomial"), nBoot = 100, plot = TRUE,
  plotBootstrap = FALSE, plotBoostrap = NULL)
```

Arguments

`simulationOutput`

an object of class DHARMa, either created via simulateResiduals for supported models or by createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.

alternative	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" (default) compared to the simulated null hypothesis.
margin	whether to test for outliers only at the lower, only at the upper, or both sides (default) of the simulated data distribution.
type	either default, bootstrap or binomial. See details.
nBoot	number of bootstrap replicates. Only used at type = "bootstrap".
plot	if TRUE, the function will create an additional plot.
plotBootstrap	if plot should be produced of outlier frequencies calculated under the bootstrap. Only used at type = "bootstrap".
plotBoostrap	deprecated, use plotBootstrap instead.

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 values, we don't know "how strongly" these values deviate from the model expectation, so the term "outlier" should be used with a grain of salt. It is not a judgment about the magnitude of the residual deviation, but simply a dichotomous sign that we are outside the simulated range. Moreover, the number of outliers will decrease as we increase the number of simulations.

To test if the outliers are a concern, testOutliers implements 2 options (bootstrap, binomial), which can be chosen via the parameter "type". The third option (default) chooses bootstrap for integer-valued distributions with nObs <= 500, and else binomial. For residuals recalculated with recalculateResiduals, nObs is the number of recalculated residuals.

The binomial test considers that under the null hypothesis that the model is correct, and for continuous distributions (i.e. data and the model distribution are identical and continuous), the probability that a given observation is higher than all simulations is 1/(nSim +1), and binomial distributed. The testOutlier function can test this null hypothesis via type = "binomial". In principle, it would be nice if we could extend this idea to integer-valued distributions, which are randomized via the PIT procedure (see simulateResiduals); the rate of "true" outliers is more difficult to calculate, and in general not 1/(nSim +1). The testOutlier function implements a small tweak that calculates the rate of residuals that are closer than 1/(nSim+1) to the 0/1 border, which roughly occur at a rate of nData /(nSim +1). This approximate value, however, is generally not exact, and may be particularly off non-bounded integer-valued distributions (such as Poisson or Negative Binomial).

For this reason, the testOutlier function implements an alternative procedure that uses the bootstrap to generate a simulation-based expectation for the outliers. It is recommended to use the bootstrap for integer-valued distributions (and integer-valued only, because it has no advantage for continuous distributions, ideally with reasonably high values of nSim and nBoot (I recommend at least 1000 for both)). Because of the high runtime, however, this option is switched off for type = default when nObs > 500.

Both binomial and bootstrap generate a null expectation, and then test for an excess or lack of outliers. Per default, testOutliers() looks for both, so if you get a significant p-value, you have to

check if you have too many or too few outliers. An excess of outliers is to be interpreted as too many values outside the simulation envelope. This could be caused by overdispersion, or by what we classically call outliers. A lack of outliers would be caused, for example, by underdispersion.

Note that for type = "bootstrap", the reported confidence interval is the 2.5% - 97.5% quantile range of the outlier frequencies simulated under H0, not a confidence interval for the observed outlier frequency.

Author(s)

Florian Hartig

See Also

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

---

`testOverdispersion` *Simulated overdisperstion tests (deprecated)*

Description

Simulated overdisperstion tests (deprecated)

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 (deprecated)*

Description

Parametric overdisperstion tests (deprecated)

Usage

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

Arguments

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

Details

Deprecated, switch your code to using the testDispersion function.

testPDistribution *Plot distribution of p-values.*

Description

Plot distribution of p-values.

Usage

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

Arguments

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

Author(s)

Florian Hartig

`testPhylogeneticAutocorrelation` *Test for phylogenetic autocorrelation*

Description

This function performs a Moran's I test for phylogenetic autocorrelation on the calculated quantile residuals.

Usage

```
testPhylogeneticAutocorrelation(simulationOutput, tree,
  alternative = c("two.sided", "greater", "less"))
```

Arguments

`simulationOutput`	an object of class DHARMa, either created via simulateResiduals for supported models or via createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.
`tree`	a phylogenetic tree object.
`alternative`	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis of no phylogenetic correlation.

Details

The function performs Moran.I test from the package ape on the DHARMa residuals, based on the phylogenetic distance matrix internally created from the provided tree. The residuals are matched to the tree by position, so the observations (rows of the data used to fit the model) must be in the same order as tree$tip.label. For custom distance matrices, you can use testSpatialAutocorrelation.

Note

Standard DHARMa simulations from models with (temporal / spatial / phylogenetic) conditional autoregressive terms will still have the respective temporal / spatial / phylogenetic correlation in the DHARMa residuals, unless the package you are using is modelling the autoregressive terms as explicit REs and is able to simulate conditional on the fitted REs. This has two consequences:

1. If you check the residuals for such a model, they will still show significant autocorrelation, even if the model fully accounts for this structure.
2. Because the DHARMa residuals for such a model are not statistically independent anymore, other tests (e.g. dispersion, uniformity) may have inflated type I error, i.e. you will have a higher likelihood of spurious residual problems.

There are three (non-exclusive) routes to address these issues when working with spatial / temporal / phylogenetic autoregressive models:

1. Simulate conditional on the fitted CAR structures (see conditional simulations in the help of simulateResiduals).
2. Rotate simulations prior to residual calculations (see parameter rotation in simulateResiduals).
3. Use custom tests / plots that explicitly compare the correlation structure in the simulated data to the correlation structure in the observed data.

Author(s)

Florian Hartig

See Also

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

Examples

```
## Not run:

  library(DHARMa)
  library(phylolm)

  set.seed(123)
  tre = rcoal(60)
  b0 = 0; b1 = 1;
  x <- runif(length(tre$tip.label), 0, 1)
  y <- b0 + b1*x +
    rTrait(n = 1, phy = tre,model="BM",
           parameters = list(ancestral.state = 0, sigma2 = 10))
  dat = data.frame(trait = y, pred = x)

  fit = lm(trait ~ pred, data = dat)
  res = simulateResiduals(fit, plot = T)

  testPhylogeneticAutocorrelation(res, tree = tre)


  fit = phylolm(trait ~ pred, data = dat, phy = tre, model = "BM")
  summary(fit)

  # phylogenetic autocorrelation still present in residuals
  res = simulateResiduals(fit, plot = T)

  # with "rotation" the residual autcorrelation is gone, see ?simulateResiduals.
  res = simulateResiduals(fit, plot = T, rotation = "estimated")


## End(Not run)
```

`testQuantiles` *Test for quantiles*

Description

This function fits quantile regressions on the residuals, and compares their location to the expected location.

Usage

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

Arguments

`simulationOutput`	an object of class DHARMa, either created via simulateResiduals for supported models or by createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.
`predictor`	an optional predictor variable to be used, instead of the predicted response (default). See details.
`rank`	if TRUE, the values provided in predictor will be rank transformed. This will usually make patterns easier to spot visually, especially if the distribution of the predictor is skewed.
`quantiles`	the quantiles to be tested.
`plot`	if TRUE, 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 distribution, 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.

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

Predictor can be a formula (e.g. predictor = ~predictor), in which case NAs are handled automatically (recommended). Predictor can also be a variable in your environment (e.g. predictor = data$predictor), but then you need to remove rows that were excluded by the model due to NAs by hand. A formula must contain exactly one variable of the data used to fit the model, e.g. ~x or ~log(x); transformations such as log() are applied, and other objects in the formula (e.g. ~log(x + k)) are taken from the environment of the formula (plotResiduals currently ignores transformations and uses the untransformed variable). To plot residuals against several predictors or by group, see

the argument form in plotResiduals. Predictor must be numeric or an ordered factor (used in the order of its levels); for categorical predictors, use testCategorical.

When plotting (plot = TRUE), the shaded gray areas indicate 95% confidence intervals of the quantile estimates (1.96 * standard error).

Author(s)

Florian Hartig

See Also

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

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

## Not run:
# accessing results of the test
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)

# specified as formula (recommended)
testQuantiles(simulationOutput, predictor = ~Environment1)

# or as variable in your environment
testQuantiles(simulationOutput, predictor = testData$Environment1)

plot(simulationOutput)
plotResiduals(simulationOutput)

## End(Not run)
```

`testResiduals` *DHARMa general residual test*

Description

Calls uniformity, dispersion and outliers tests.

Usage

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

Arguments

`simulationOutput`	an object of class DHARMa, either created via simulateResiduals for supported models or by createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.
`plot`	if TRUE, plot 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 functions testUniformity, testDispersion, and testOutliers. All other tests (see list below) have to be called by hand.

Author(s)

Florian Hartig

See Also

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

Examples

```
testData = createData(sampleSize = 100, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# the plot function shows 2 plots and 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)

# KS test for correct distribution within and between groups
# group specified as formula (recommended)
testCategorical(simulationOutput, ~group)

# group specified as variable in your environment
testCategorical(simulationOutput, testData$group)

# Dispersion test - for details see ?testDispersion
testDispersion(simulationOutput) # tests under and overdispersion

# Outlier test (number of observations outside simulation envelope)
# Use type = "bootstrap" for exact values, see ?testOutliers
testOutliers(simulationOutput, type = "binomial")

# 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, either created via simulateResiduals for supported models or by createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.

Details

Deprecated, switch your code to using the testResiduals function

Author(s)

Florian Hartig

`testSpatialAutocorrelation` *Test for distance-based spatial (or similar type) autocorrelation*

Description

This function performs a Moran's I test for distance-based spatial (or similar type) autocorrelation on the calculated quantile residuals.

Usage

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

Arguments

`simulationOutput`	an object of class DHARMa, either created via simulateResiduals for supported models or via createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.
`x`	the x coordinate, in the same order as the data points. Must be specified unless distMat is provided.
`y`	the y coordinate, in the same order as the data points. Must be specified unless distMat is provided.
`distMat`	optional distance matrix. Must be provided as class "matrix/array" (class "dist" is not supported). If distMat is not provided, euclidean distances based on x and y will be calculated. 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`	if T, and if x and y is provided, plot the output (see details).

Details

The function performs Moran.I test from the package ape on the DHARMa residuals. If a distance matrix (distMat) is provided, calculations will be based on this distance matrix, and x,y coordinates will only be used for the plotting (if provided). If distMat is not provided, the function will calculate the euclidean distances between x,y coordinates, and test Moran.I based on these distances.

If plot = T, a plot will be produced showing each residual at its x,y position, colored according to the residual value. Residuals with 0.5 are colored white, everything below 0.5 is colored increasingly red, everything above 0.5 is colored increasingly blue.

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 be 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

Standard DHARMa simulations from models with (temporal / spatial / phylogenetic) conditional autoregressive terms will still have the respective temporal / spatial / phylogenetic correlation in the DHARMa residuals, unless the package you are using is modelling the autoregressive terms as explicit REs and is able to simulate conditional on the fitted REs. This has two consequences:

1. If you check the residuals for such a model, they will still show significant autocorrelation, even if the model fully accounts for this structure.
2. Because the DHARMa residuals for such a model are not statistically independent anymore, other tests (e.g. dispersion, uniformity) may have inflated type I error, i.e. you will have a higher likelihood of spurious residual problems.

There are three (non-exclusive) routes to address these issues when working with spatial / temporal / phylogenetic / other autoregressive models:

1. Simulate conditional on the fitted CAR structures (see conditional simulations in the help of simulateResiduals).
2. Rotate simulations prior to residual calculations (see parameter rotation in simulateResiduals).
3. Use custom tests / plots that explicitly compare the correlation structure in the simulated data to the correlation structure in the observed data.

Author(s)

Florian Hartig

See Also

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

Examples

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

# Standard use
# specify x,y as formulas (recommended)
testSpatialAutocorrelation(res, x = ~x, y = ~y)

# or as variables in your environment
testSpatialAutocorrelation(res, x =  testData$x, y = testData$y)

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

# You could add a spatial variogram via
# library(gstat)
# dat = data.frame(res = residuals(res), x = testData$x, y = testData$y)
# coordinates(dat) = ~x+y
# vario = variogram(res~1, data = dat, alpha=c(0,45,90,135))
# plot(vario, ylim = c(-1,1))

# 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 autocorrelation on the new group

# modifying x, y, so that we have the same location per group
# just for completeness
testData$x = as.numeric(testData$group)
testData$y = as.numeric(testData$group)

# calculating x, y positions per group
groupLocations = aggregate(testData[, 6:7], list(testData$group), mean)

# calculating residuals per group
res2 = recalculateResiduals(res, group = ~group)

# running the spatial test on grouped residuals
testSpatialAutocorrelation(res2, groupLocations$x, groupLocations$y)

# careful 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 the 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)

# Re-simulating REs (unconditional) means spatial pattern remains
# because residuals are still clustered

res = simulateResiduals(fittedModel, simulateREs = "unconditional")
testSpatialAutocorrelation(res, x = ~x, y = ~y)

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

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

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

res = simulateResiduals(fittedModel, simulateREs = "conditional")
testSpatialAutocorrelation(res, x = ~x, y = ~y)
```

`testTemporalAutocorrelation` *Test for temporal autocorrelation*

Description

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

Usage

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

Arguments

`simulationOutput`	an object of class DHARMa, either created via simulateResiduals for supported models or by createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.
`time`	time variable in the same order as the datapoints and with the same length as the residuals in simulationOutput. If specified as formula, e.g. time = ~time, NAs are handled automatically (recommended).
`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 DW test was originally 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 DW test on those.

Testing for temporal autocorrelation requires unique time values - if you have several observations per time value, either use recalculateResiduals function to aggregate residuals per time step (note that the aggregated residuals are ordered by the sorted group levels, so time must be provided in the same order, e.g. sort(unique(time))), 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 testTemporalAutocorrelation.

Note

Standard DHARMa simulations from models with (temporal / spatial / phylogenetic) conditional autoregressive terms will still have the respective temporal / spatial / phylogenetic correlation in the DHARMa residuals, unless the package you are using is modelling the autoregressive terms as explicit REs and is able to simulate conditional on the fitted REs. This has two consequences:

1. If you check the residuals for such a model, they will still show significant autocorrelation, even if the model fully accounts for this structure.
2. Because the DHARMa residuals for such a model are not statistically independent anymore, other tests (e.g. dispersion, uniformity) may have inflated type I error, i.e. you will have a higher likelihood of spurious residual problems.

There are three (non-exclusive) routes to address these issues when working with spatial / temporal / other autoregressive models:

1. Simulate conditional on the fitted CAR structures (see conditional simulations in the help of simulateResiduals).
2. Rotate simulations prior to residual calculations (see parameter rotation in simulateResiduals).
3. Use custom tests / plots that explicitly compare the correlation structure in the simulated data to the correlation structure in the observed data.

Author(s)

Florian Hartig

See Also

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

Examples

```
testData = createData(sampleSize = 40, family = gaussian(),
                      randomEffectVariance = 0)
fittedModel <- lm(observedResponse ~ Environment1, data = testData)
res = simulateResiduals(fittedModel)

# Standard use
# specify time as formula (recommended)
testTemporalAutocorrelation(res, time =  ~time)

# or as variable in your environment
testTemporalAutocorrelation(res, time =  testData$time)

# If you have several observations per time step, e.g.
# because you have several locations, you will have to
# aggregate

timeSeries1 = createData(sampleSize = 40, family = gaussian(),
                         randomEffectVariance = 0)
timeSeries1$location = 1
timeSeries2 = createData(sampleSize = 40, family = gaussian(),
                         randomEffectVariance = 0)
timeSeries2$location = 2
testData = rbind(timeSeries1, timeSeries2)

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

# Will not work because several residuals per time
# testTemporalAutocorrelation(res, time = testData$time)

# aggregating residuals by time
res = recalculateResiduals(res, group = ~time)
# the aggregated residuals are sorted by time, so time must be sorted as well
testTemporalAutocorrelation(res, time = sort(unique(testData$time)))

# testing only subgroup location 1, could do same with loc 2
res = recalculateResiduals(res, sel = testData$location == 1)
testTemporalAutocorrelation(res, time = testData$time[testData$location == 1])

# example to demonstrate problems with strong temporal correlations and
# how to possibly remove them by rotating residuals
# note that if your model allows to condition on estimated REs, this may
```

```
# be preferable!

## Not run:

  set.seed(123)

  # Gaussian error

  # Create AR data with 5 observations per time point
  n <- 100
  x <- MASS::mvrnorm(mu = rep(0,n),
                     Sigma = .9 ^ as.matrix(dist(1:n)) )
  y <- rep(x, each = 5) + 0.2 * rnorm(5*n)
  times <- factor(rep(1:n, each = 5), levels=1:n)
  levels(times)
  group <- factor(rep(1,n*5))
  dat0 <- data.frame(y,times,group)

  # fit model / would be similar for nlme::gls and similar models
  model = glmmTMB(y ~ ar1(times + 0 | group), data=dat0)

  # Note that standard residuals still show problems because of autocorrelation
  res <- simulateResiduals(model)
  plot(res)

  # The reason is that most (if not all) autoregressive models treat the
  # autocorrelated error as random, i.e. the autocorrelated error structure
  # is not used for making predictions. If you then make predictions based
  # on the fixed effects and calculate residuals, the autocorrelation in the
  # residuals remains. We can see this if we again calculate the auto-
  # correlation test

  res2 <- recalculateResiduals(res, group=dat0$times)
  testTemporalAutocorrelation(res2, time = 1:length(res2$scaledResiduals))

  # so, how can we check then if the current model correctly removed the
  # autocorrelation?

  # Option 1: rotate the residuals in the direction of the autocorrelation
  # to make the independent. Note that this only works perfectly for gls
  # type models as nonlinear link function make the residuals covariance
  # different from a multivariate normal distribution

  # this can be either done by extracting estimated AR1 covariance
  cov <- VarCorr(model)
  cov <- cov$cond$group # extract covariance matrix of REs

  # grouped according to times, rotated with estimated Cov - how all fine!
  res3 <- recalculateResiduals(res, group=dat0$times, rotation=cov)
  plot(res3)
  testTemporalAutocorrelation(res3, time = 1:length(res2$scaledResiduals))

  # alternatively, you can let DHARMa estimate the covariance from the
```

```
# simulations

res4 <- recalculateResiduals(res, group=dat0$times, rotation="estimated")
plot(res4)
testTemporalAutocorrelation(res4, time = 1:length(res2$scaledResiduals))

# Alternatively, in glmmTMB, we can condition on the estimated correlated
# residuals. Unfortunately, in this case, we will have to do simulations by
# hand as glmmTMB does not allow to simulate conditional on a fitted
# correlation structure

# re.form = NULL creates predictions conditional on the fitted temporally
# autocorreated REs
pred = predict(model, re.form = NULL)

# now we simulate data, conditional on the autocorrelation part, with the
# uncorrelated residual error
simulations = sapply(1:250, function(i) rnorm(length(pred),
                                              pred,
                                              summary(model)$sigma))
res5 = createDHARMa(simulations, dat0$y, pred)
plot(res5)

res5b <- recalculateResiduals(res5, group=dat0$times)
testTemporalAutocorrelation(res5b, time = 1:length(res5b$scaledResiduals))

# Poisson error
# note that for GLMMs, covariances will be estimated at the scale of the
# linear predictor, while residual covariance will be at the responses scale
# and thus further distorted by the link. Thus, for GLMMs with a nonlinear
# link, there will be no exact rotation for a given covariance structure

set.seed(123)

# Create AR data with 5 observations per time point
n <- 100
x <- MASS::mvrnorm(mu = rep(0,n),
                   Sigma = .9 ^ as.matrix(dist(1:n)) )
y <- rpois(n = n*5, lambda = exp(rep(x, each = 5)))
times <- factor(rep(1:n, each = 5), levels=1:n)
levels(times)
group <- factor(rep(1,n*5))
dat0 <- data.frame(y,times,group)

# fit model
model = glmmTMB(y ~ ar1(times + 0 | group), data=dat0, family = poisson)

res <- simulateResiduals(model)

# grouped according to times, unrotated
res2 <- recalculateResiduals(res, group=dat0$times)
testTemporalAutocorrelation(res2, time = 1:length(res2$scaledResiduals))
```

```
# grouped according to times, rotated with estimated Cov - problems remain
cov <- VarCorr(model)
cov <- cov$cond$group # extract covariance matrix of REs
res3 <- recalculateResiduals(res, group=dat0$times, rotation=cov)
testTemporalAutocorrelation(res3, time = 1:length(res2$scaledResiduals))

# grouped according to times, rotated with covariance estimated from residual
# simulations at the response scale
res4 <- recalculateResiduals(res, group=dat0$times, rotation="estimated")
testTemporalAutocorrelation(res4, time = 1:length(res2$scaledResiduals))


## End(Not run)
```

`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 = TRUE)
```

Arguments

`simulationOutput`	an object of class DHARMa, either created via simulateResiduals for supported models or by createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.
`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 stats::ks.test for details.
`plot`	if TRUE, plots calls plotQQunif as well.

Details

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

Author(s)

Florian Hartig

See Also

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

Examples

```
testData = createData(sampleSize = 100, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# the plot function shows 2 plots and 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)

# KS test for correct distribution within and between groups
# group specified as formula (recommended)
testCategorical(simulationOutput, ~group)

# group specified as variable in your environment
testCategorical(simulationOutput, testData$group)

# Dispersion test - for details see ?testDispersion
testDispersion(simulationOutput) # tests under and overdispersion

# Outlier test (number of observations outside simulation envelope)
# Use type = "bootstrap" for exact values, see ?testOutliers
testOutliers(simulationOutput, type = "binomial")

# 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, either created via simulateResiduals for supported models or by createDHARMa for simulations created outside DHARMa, or a supported model. Providing a supported model directly is discouraged, because simulation settings cannot be changed in this case.
`...`	further arguments to testGeneric.

Details

Zero-inflation means that the observed data contains more zeros than would be expected under the fitted model. Zero-inflation must always be accessed with respect to a particular model, so the mere fact that there are many zeros in the observed data is not an indication of 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.

The testZeroInflation function simulates new datasets from the fitted model and compares this null distribution (gray histogram in the plot) with the observed values (red line in the plot). Technically, it is a wrapper for testGeneric, with the summary argument set to function(x) sum(x == 0). The test statistic is the ratio of observed to simulated zeros. A value < 1 means that the observed data have fewer zeros than expected, a value > 1 means that they have more zeros than expected (aka zero inflation). By default, the function tests both sides, so it would also test for fewer zeros than expected.

Note

Zero-inflation can occur for a number of reasons other than an underlying data generating process corresponding to a ZIP model. Vice versa, it is very well possible that no zero-inflation will be observed when fitting models to data derived from a ZIP process. The latter is due to the fact that excess zeros can often be explained by other model parameters, such as the theta parameter in the negative binomial.

For this reason, results of the zero-inflation test should be interpreted as a residual pattern that can have many reasons, not as a decision criterion for whether or not to fit a ZIP model. To decide whether to add a ZIP term, I would advise relying on appropriate model selection techniques such as AIC, BIC, WAIC, Bayes factor, or LRT. Note that these tests are often not reliable in GLMMs

because it is difficult to determine the df spent by the different models. The simulateLRT function in DHARMa provides a nonparametric alternative to obtain p-values for LRTs in nested models with unknown df.

Author(s)

Florian Hartig

See Also

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

Examples

```
testData = createData(sampleSize = 100, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# the plot function shows 2 plots and 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)

# KS test for correct distribution within and between groups
# group specified as formula (recommended)
testCategorical(simulationOutput, ~group)

# group specified as variable in your environment
testCategorical(simulationOutput, testData$group)

# Dispersion test - for details see ?testDispersion
testDispersion(simulationOutput) # tests under and overdispersion

# Outlier test (number of observations outside simulation envelope)
# Use type = "bootstrap" for exact values, see ?testOutliers
testOutliers(simulationOutput, type = "binomial")

# 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 qnorm) is used, residuals that are 0/1 are mapped to -Inf / Inf. outlierValue allows to convert -Inf / Inf values to an optional min / max value.

Index

benchmarkRuntime, 4

createData, 5
createDHARMa, 7, *54*, *56–58*, *60*, *62*, *64*, *66*, *68–70*, *74*, *78*, *80*

DHARMa *(*DHARMa-package*)*, 3
DHARMa-package, 3

gap::qqunif, *35*
getData, 9
getFamily, 10
getFitted, *10*, 11, *13, 14*, *16–18*, *24*, *27*
getFittedResponse, 12, *56*
getFixedEffects, *10*, *12*, 14, *16–18*, *23, 24*, *27*
getObservedResponse, *10*, *12–14*, 15, *17, 18*, *23, 24*, *27*
getPearsonResiduals, 16
getPredictorNames, 18
getQuantile, *8*, 19, *41, 42*, *50–52*
getRandomState, 20
getRefit, *10*, *12*, *14*, *16–18*, 22, *24*, *27*
getResiduals, 23
getSimulations, *10*, *12–14*, *16–18*, *23, 24*, 25, *48*, *50, 51*

hist.DHARMa, 28
hurricanes, 29

outliers, 31, *52*

plot.DHARMa, *28*, 32, *35*, *40*, *49, 50*, *52*
plot.DHARMaBenchmark, 33, *46*
plotConventionalResiduals, 34
plotQQunif, *32, 33*, 34, *38*, *40*, *78*
plotResiduals, *28*, *32, 33*, *35*, 36, *39, 40*, *52*, *66, 67*
plotResidualsAll, 39
plotSimulatedResiduals, 40
print.DHARMa, 40, *52*

recalculateResiduals, 41, *52*, *57*, *61*, *71*, *74*
residuals.DHARMa, 43, *49*, *52*, *82*
runBenchmarks, *33, 34*, 45

simulateLRT, 47, *81*
simulateResiduals, *3*, *8*, *26*, *31, 32*, *39–43*, *48*, 49, *54*, *56*, *58*, *60–62*, *64–66*, *68–71*, *74*, *78*, *80*, *82*
stats::family(), *6*
stats::ks.test, *78*

testCategorical, *37*, 53, *54*, *57*, *59*, *62*, *65*, *67, 68*, *71*, *75*, *79*, *81*
testDispersion, *8*, *13*, *35*, *54*, 55, *57*, *59*, *62, 63*, *65*, *67, 68*, *71*, *75*, *79*, *81*
testGeneric, *54*, *56, 57*, 58, *59*, *62*, *65*, *67, 68*, *71*, *75*, *79–81*
testOutliers, *35*, *37, 38*, *54*, *57*, *59*, 60, *62*, *65*, *67, 68*, *71*, *75*, *79*, *81*
testOverdispersion, 62
testOverdispersionParametric, 63
testPDistribution, 63
testPhylogeneticAutocorrelation, *52*, 64
testQuantiles, *36–38*, *54*, *57*, *59*, *62*, *65*, 66, *67, 68*, *71*, *75*, *79*, *81*
testResiduals, *49*, *52*, *54*, *57*, *59*, *62*, *65*, *67, 68*, 68, *70, 71*, *75*, *79*, *81*
testSimulatedResiduals, 69
testSpatialAutocorrelation, *52*, *54*, *57*, *59*, *62*, *64, 65*, *67, 68*, 70, *71*, *75*, *79*, *81*
testTemporalAutocorrelation, *20*, *51, 52*, *54*, *57*, *59*, *62*, *65*, *67, 68*, *71*, 73, *75*, *79*, *81*
testUniformity, *35*, *52*, *54*, *57*, *59*, *62*, *65*, *67, 68*, *71*, *75*, 78, *79*, *81*
testZeroInflation, *54*, *57*, *59*, *62*, *65*, *67, 68*, *71*, *75*, *79*, 80, *81*
transformQuantiles, 82