Package ‘glmmrBase’

July 22, 2025

Type Package

Title Generalised Linear Mixed Models in R

Version 1.0.2

Date 2025-07-22

Description Specification, analysis, simulation, and fitting of generalised linear mixed models. Includes Markov Chain Monte Carlo Maximum likelihood and Laplace approximation model fitting for a range of models, non-linear fixed effect specifications, a wide range of flexible covariance functions that can be combined arbitrarily, robust and bias-corrected standard error estimation, power calculation, data simulation, and more. See <https://samuel-watson.github.io/glmmr-web/> for a detailed manual.

License GPL (>= 2)

Imports methods, Rcpp (>= 1.0.11), R6, rstan (>= 2.32.1), rstantools (>= 2.3.1.1)

LinkingTo Rcpp (>= 1.0.11), RcppEigen, SparseChol (>= 0.3.2), BH, RcppParallel (>= 5.0.1), rstan (>= 2.32.1), StanHeaders (>= 2.32.0)

RoxygenNote 7.3.2

NeedsCompilation yes

Author Sam Watson [aut, cre]

URL https://github.com/samuel-watson/glmmrBase

BugReports https://github.com/samuel-watson/glmmrBase/issues

Biarch true

Depends R (>= 3.5.0), Matrix (>= 1.3-1)

SystemRequirements GNU make

Encoding UTF-8

Config/testthat/edition 3

LazyData true

Maintainer Sam Watson <S.I.Watson@bham.ac.uk>

Repository CRAN

Date/Publication 2025-07-22 11:40:02 UTC

Contents

glmmrBase-package *Generalised Linear Mixed Models in R*

Description

Specification, analysis, simulation, and fitting of generalised linear mixed models. Includes Markov Chain Monte Carlo Maximum likelihood and Laplace approximation model fitting for a range of models, non-linear fixed effect specifications, a wide range of flexible covariance functions that can be combined arbitrarily, robust and bias-corrected standard error estimation, power calculation, data simulation, and more. See <https://samuel-watson.github.io/glmmr-web/> for a detailed manual. `glmmrBase` provides functions for specifying, analysing, fitting, and simulating mixed models including linear, generalised linear, and models non-linear in fixed effects.

Differences between glmmrBase and lme4 and related packages.

glmmrBase is intended to be a broad package to support statistical work with generalised linear mixed models. While there are Laplace Approximation methods in the package, it does not intend to replace or supplant popular mixed model packages like **lme4**. Rather it provides broader functionality around simulation and analysis methods, and a range of model fitting algorithms not found in other mixed model packages. The key features are:

- Stochastic maximum likelihood methods. The most widely used methods for mixed model fitting are penalised quasi-likelihood, Laplace approximation, and Gaussian quadrature methods. These methods are widely available in other packages. We provide Markov Chain Monte Carlo (MCMC) Maximum Likelihood and Stochastic Approximation Expectation Maximisation algorithms for model fitting, with various features. These algorithms approximate the intractable GLMM likelihood using MCMC and so can provide an arbitrary level of precision. These methods may provide better maximum likelihood performance than other approximations in settings with high-dimensional or complex random effects, small sample sizes, or non-linear models.
- Flexible support for a wide range of covariance functions. The support for different covariance functions can be limited in other packages. For example, **lme4** only provides exchangeable random effects structures. We include multiple different functions that can be combined arbitrarily.
- We similarly use efficient linear algebra methods with the **Eigen** package along with **Stan** to provide MCMC sampling.
- Gaussian Process approximations. We include Hibert Space and Nearest Neighbour Gaussian Process approximations for high dimensional random effects.
- The `Model` class includes methods for power estimation, data simulation, MCMC sampling, and calculation of a wide range of matrices and values associated with the models.
- We include natively a range of small sample corrections to information matrices, including Kenward-Roger, Box, Satterthwaite, and others, which typically require add-on packages for **lme4**.
- The package provides a flexible class system for specifying mixed models that can be incorporated into other packages and settings. The linked package **glmmrOptim** provides optimal experimental design algorithms for mixed models.

- (New in version 0.9.1) The package includes functions to replicate the functionality of **lme4**, mcml_lmer and mcml_glmer, which will also accept **lme4** syntax.
- (New in version 0.10.1) The package also provides mixed quantile regression models estimated using the stochastic maximum likelihood algorithms described above. These models specify an asymmetric Laplace distribution for the likelihood and integrate with the other features of the package described above.

Package development

The package is still in development and there may still be bugs and errors. While we do not expect the general user interface to change there may be changes to the underlying library as well as new additions and functionality.

Author(s)

Sam Watson [aut, cre]

Maintainer: Sam Watson <S.I.Watson@bham.ac.uk>

`Beta` *Beta distribution declaration*

Description

Skeleton list to declare a Beta distribution in a 'Model' object

Usage

```
Beta(link = "logit")
```

Arguments

`link`	Name of link function. Only accepts 'logit' currently.

Value

A list with two elements naming the family and link function `coef.mcml` *Extracts fixed effect coefficients from a mcml object*

Description

Extracts the fitted fixed effect coefficients from an ‘mcml‘ object returned from a call of ‘MCML‘ or ‘LA‘ in the Model class.

Usage

```
## S3 method for class 'mcml'
coef(object, ...)
```

Arguments

`object`	An ‘mcml‘ model fit.
`...`	Further arguments passed from other methods

Value

A named vector.

`coef.Model` *Extracts coefficients from a Model object*

Description

Extracts the coefficients from a ‘Model‘ object.

Usage

```
## S3 method for class 'Model'
coef(object, ...)
```

Arguments

`object`	A ‘Model‘ object.
`...`	Further arguments passed from other methods

Value

Fixed effect and covariance parameters extracted from the model object.

`confint.mcml` *Fixed effect confidence intervals for a ‘mcml‘ object*

Description

Returns the computed confidence intervals for a ‘mcml‘ object.

Usage

```
## S3 method for class 'mcml'
confint(object, ...)
```

Arguments

`object`	A ‘mcml‘ object.
`...`	Further arguments passed from other methods

Value

A matrix (or vector) with columns giving lower and upper confidence limits for each parameter.

`Covariance` *R6 Class representing a covariance function and data*

Description

R6 Class representing a covariance function and data

R6 Class representing a covariance function and data

Details

For the generalised linear mixed model

$$Y \sim F(\mu, \sigma)$$
$$\mu = h^-1(X\beta + Z\gamma)$$
$$\gamma \sim MVN(0, D)$$

where h is the link function, this class defines Z and D. The covariance is defined by a covariance function, data, and parameters. A new instance can be generated with $new(). The class will generate the relevant matrices Z and D automatically. See glmmrBase for a detailed guide on model specification.

Intitialisation A covariance function is specified as an additive formula made up of components with structure `(1|f(j))`. The left side of the vertical bar specifies the covariates in the model that have a random effects structure. The right side of the vertical bar specify the covariance function 'f' for that term using variable named in the data 'j'. Covariance functions on the right side of the vertical bar are multiplied together, i.e. `(1|f(j)*g(t))`.

There are several common functions included for a named variable in data `x`. A non-exhaustive list (see glmmrBase for a full list): * `gr(x)`: Indicator function (1 parameter) * `fexp(x)`: Exponential function (2 parameters) * `ar(x)`: AR function (2 parameters) * `sqexp(x)`: Squared exponential (1 parameter) * `matern(x)`: Matern function (2 parameters) * `bessel(x)`: Modified Bessel function of the 2nd kind (1 parameter) For many 2 parameter functions, such as 'ar' and 'fexp', alternative one parameter versions are also available as 'ar0' and 'fexp0'. These function omit the variance parameter and so can be used in combination with 'gr' functions such as 'gr(j)*ar0(t)'.

Parameters are provided to the covariance function as a vector. The parameters in the vector for each function should be provided in the order the covariance functions are written are written. For example, * Formula: '~(1|gr(j))+(1|gr(j*t))'; parameters: 'c(0.05,0.01)' * Formula: '~(1|gr(j)*fexp0(t))'; parameters: 'c(0.05,0.5)'

Updating of parameters is automatic if using the 'update_parameters()' member function.

Using 'update_parameters()' is the preferred way of updating the parameters of the mean or covariance objects as opposed to direct assignment, e.g. 'self$parameters <- c(...)'. The function calls check functions to automatically update linked matrices with the new parameters.

Public fields

`data` Data frame with data required to build covariance

`formula` Covariance function formula.

`parameters` Model parameters specified in order of the functions in the formula.

`Z` Design matrix

`D` Covariance matrix of the random effects

Methods

Public methods:

- `Covariance$n()`
- `Covariance$new()`
- `Covariance$update_parameters()`
- `Covariance$print()`
- `Covariance$subset()`
- `Covariance$chol_D()`
- `Covariance$log_likelihood()`
- `Covariance$simulate_re()`
- `Covariance$sparse()`
- `Covariance$parameter_table()`
- `Covariance$nngp()`
- `Covariance$hsgp()`
- `Covariance$clone()`

Method `n()`: Return the size of the design

Usage:

```
Covariance$n()
```

Returns: Scalar

Method `new()`**:** Create a new Covariance object

Usage:

```
Covariance$new(formula, data = NULL, parameters = NULL)
```

Arguments:

`formula` Formula describing the covariance function. See Details

`data` (Optional) Data frame with data required for constructing the covariance.

`parameters` (Optional) Vector with parameter values for the functions in the model formula. See Details.

Returns: A Covariance object

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
df <- nelder(~(cl(5)*t(5)) > ind(5))
cov <- Covariance$new(formula = ~(1|gr(cl)*ar0(t)),
                      parameters = c(0.05,0.7),
                      data= df)
```

Method `update_parameters()`**:** Updates the covariance parameters

Usage:

```
Covariance$update_parameters(parameters)
```

Arguments:

`parameters` A vector of parameters for the covariance function(s). See Details.

Method `print()`**:** Show details of Covariance object

Usage:

```
Covariance$print()
```

Arguments:

`...` ignored

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
df <- nelder(~(cl(5)*t(5)) > ind(5))
Covariance$new(formula = ~(1|gr(cl)*ar0(t)),
                      parameters = c(0.05,0.8),
                      data= df)
```

Method `subset()`**:** Keep specified indices and removes the rest

Usage:

```
Covariance$subset(index)
```

Arguments:

`index` vector of indices to keep

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
df <- nelder(~(cl(10)*t(5)) > ind(10))
cov <- Covariance$new(formula = ~(1|gr(cl)*ar0(t)),
                      parameters = c(0.05,0.8),
                      data= df)
cov$subset(1:100)
```

Method `chol_D()`**:** Returns the Cholesky decomposition of the covariance matrix D

Usage:

```
Covariance$chol_D()
```

Returns: A matrix

Method `log_likelihood()`**:** The function returns the values of the multivariate Gaussian log likelihood with mean zero and covariance D for a given vector of random effect terms.

Usage:

```
Covariance$log_likelihood(u)
```

Arguments:

`u` Vector of random effects

Returns: Value of the log likelihood

Method `simulate_re()`**:** Simulates a set of random effects from the multivariate Gaussian distribution with mean zero and covariance D.

Usage:

```
Covariance$simulate_re()
```

Returns: A vector of random effect values

Method `sparse()`**:** If this function is called then sparse matrix methods will be used for calculations involving D

Usage:

```
Covariance$sparse(sparse = TRUE, amd = TRUE)
```

Arguments:

`sparse` Logical. Whether to use sparse methods (TRUE) or not (FALSE)

`amd` Logical indicating whether to use and Approximate Minimum Degree algorithm to calculate an efficient permutation matrix so that the Cholesky decomposition of PAP^T is calculated rather than A.

Returns: None. Called for effects.

Method `parameter_table()`**:** Returns a table showing which parameters are members of which covariance function term.

Usage:

```
Covariance$parameter_table()
```

Returns: A data frame

Method `nngp()`**:** Reports or sets the parameters for the nearest neighbour Gaussian process

Usage:

```
Covariance$nngp(nn = NULL)
```

Arguments:

`nn` Integer. Number of nearest neighbours. Optional - leave as NULL to return details of the NNGP instead.

Returns: If ‘nn‘ is NULL then the function will either return FALSE if not using a Nearest neighbour approximation, or TRUE and the number of nearest neighbours, otherwise it will return nothing.

Method `hsgp()`**:** Reports or sets the parameters for the Hilbert Space Gaussian process

Usage:

```
Covariance$hsgp(m = NULL, L = NULL)
```

Arguments:

`m` Integer or vector of integers. Number of basis functions per dimension. If only a single number is provided and there is more than one dimension the same number will be applied to all dimensions.

`L` Decimal. The boundary extension.

Returns: If ‘m‘ and ‘L‘ are NULL then the function will either return FALSE if not using a Hilbert space approximation, or TRUE and the number of bases functions and boundary value, otherwise it will return nothing.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Covariance$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `Covariance$new`
## ------------------------------------------------


df <- nelder(~(cl(5)*t(5)) > ind(5))
cov <- Covariance$new(formula = ~(1|gr(cl)*ar0(t)),
                      parameters = c(0.05,0.7),
                      data= df)
```

```
## ------------------------------------------------
## Method `Covariance$print`
## ------------------------------------------------


df <- nelder(~(cl(5)*t(5)) > ind(5))
Covariance$new(formula = ~(1|gr(cl)*ar0(t)),
                      parameters = c(0.05,0.8),
                      data= df)

## ------------------------------------------------
## Method `Covariance$subset`
## ------------------------------------------------


df <- nelder(~(cl(10)*t(5)) > ind(10))
cov <- Covariance$new(formula = ~(1|gr(cl)*ar0(t)),
                      parameters = c(0.05,0.8),
                      data= df)
cov$subset(1:100)
```

`cross_df` *Generate crossed block structure*

Description

Generate a data frame with crossed rows from two other data frames

Usage

```
cross_df(df1, df2)
```

Arguments

`df1`	data frame
`df2`	data frame

Details

For two data frames ‘df1‘ and ‘df2‘, the function will return another data frame that crosses them, which has rows with every unique combination of the input data frames

Value

data frame

Examples

```
cross_df(data.frame(t=1:4),data.frame(cl=1:3))
```

`cycles` *Generates all the orderings of a*

Description

Given input a, returns a length(a)^2 vector by cycling through the values of a

Usage

```
cycles(a)
```

Arguments

`a`	vector

Value

vector

`family.mcml` *Extracts the family from a ‘mcml‘ object.*

Description

Extracts the family from a ‘mcml‘ object.

Usage

```
## S3 method for class 'mcml'
family(object, ...)
```

Arguments

`object`	A ‘mcml‘ object.
`...`	Further arguments passed from other methods

Value

A family object.

`family.Model` *Extracts the family from a ‘Model‘ object. This information can also be accessed directly from the Model as ‘Model$family‘*

Description

Extracts the family from a ‘Model‘ object.

Usage

```
## S3 method for class 'Model'
family(object, ...)
```

Arguments

`object`	A ‘Model‘ object.
`...`	Further arguments passed from other methods

Value

A family object.

`fitted.mcml` *Fitted values from a ‘mcml‘ object*

Description

Fitted values should not be generated directly from an ‘mcml‘ object, rather fitted values should be generated using the original ‘Model‘. A message is printed to the user.

Usage

```
## S3 method for class 'mcml'
fitted(object, ...)
```

Arguments

`object`	A ‘mcml‘ object.
`...`	Further arguments passed from other methods

Value

Nothing, called for effects, unless ‘override‘ is TRUE, when it will return a vector of fitted values.

`fitted.Model` *Extract or generate fitted values from a ‘Model‘ object*

Description

Return fitted values. Does not account for the random effects. This function is a wrapper for ‘Model$fitted()‘, which also provides a variety of additional options for generating fitted values from mixed models. For simulated values based on resampling random effects, see also ‘Model$sim_data()‘. To predict the values including random effects at a new location see also ‘Model$predict()‘.

Usage

```
## S3 method for class 'Model'
fitted(object, ...)
```

Arguments

`object`	A ‘Model‘ object.
`...`	Further arguments passed from other methods

Value

Fitted values

`fixed.effects` *Extracts the fixed effect estimates*

Description

Extracts the fixed effect estimates from an mcml object returned from call of ‘MCML‘ or ‘LA‘ in the Model class.

Usage

```
fixed.effects(object)
```

Arguments

`object`	An ‘mcml‘ model fit.

Value

A named, numeric vector of fixed-effects estimates.

formula.mcml — *Extracts the formula from a 'mcml' object.*

Description

Extracts the formula from a 'mcml' object. Separate formulae are specified for the fixed and random effects in the model, either of which can be returned. The complete formula is available from the generating 'Model' object as 'Model$formula' or 'formula(Model)'

Usage

```
## S3 method for class 'mcml'
formula(x, ...)
```

Arguments

`x`	A 'mcml' object.
`...`	Further arguments passed from other methods

Value

A formula object.

formula.Model — *Extracts the formula from a 'Model' object*

Description

Extracts the formula from a 'Model' object. This information can also be accessed directly from the Model as 'Model$formula'

Usage

```
## S3 method for class 'Model'
formula(x, ...)
```

Arguments

`x`	A 'Model' object.
`...`	Further arguments passed from other methods

Value

A formula object.

hessian_from_formula *Automatic differentiation of formulae*

Description

Exposes the automatic differentiator. Allows for calculation of Jacobian and Hessian matrices of formulae in terms of specified parameters. Formula specification is as a string. Data items are automatically multiplied by a parameter unless enclosed in parentheses.

Usage

```
hessian_from_formula(form_, data_, colnames_, parameters_)
```

Arguments

`form_`	String. Formula to differentiate specified in terms of data items and parameters. Any string not identifying a function or a data item names in ‘colnames‘ is assumed to be a parameter.
`data_`	Matrix. A matrix including the data. Rows represent observations. The number of columns should match the number of items in ‘colnames_‘
`colnames_`	Vector of strings. The names of the columns of ‘data_‘, used to match data named in the formula.
`parameters_`	Vector of doubles. The values of the parameters at which to calculate the derivatives. The parameters should be in the same order they appear in the formula.

Value

A list including the jacobian and hessian matrices.

Examples

```
# obtain the Jacobian and Hessian of the log-binomial model log-likelihood.
# The model is of data from an intervention and control group
# with n1 and n0 participants, respectively, with y1 and y0 the number of events in each group.
# The mean is exp(alpha) in the control
# group and exp(alpha + beta) in the intervention group, so that beta is the log relative risk.
hessian_from_formula(
  form_ = "(y1)*(a+b)+((n1)-(y1))*log((1-exp(a+b)))+(y0)*a+((n0)-(y0))*log((1-exp(a)))",
  data_ = matrix(c(10,100,20,100), nrow = 1),
  colnames_ = c("y1","n1","y0","n0"),
  parameters_ = c(log(0.1),log(0.5)))
```

lme4_to_glmmr *Map lme4 formula to glmmrBase formula*

Description

Returns a formula that can be used for glmmrBase Models from an lme4 input.

Usage

```
lme4_to_glmmr(formula, cnames)
```

Arguments

`formula`	A lme4 style formula
`cnames`	The column names of the data to be used. These are used to check if the specified clustering variables are in the data.

Details

The package lme4 uses a syntax to specify random effects as ‘(1|x)‘ where ‘x‘ is the grouping variable. This function will modify such a formula, including those with nesting and crossing operators ‘/‘ and ‘:‘ into the glmmrBase syntax using the ‘gr()‘ function. Not typically required by the user as it is used internally in the ‘mcml_lmer‘ and ‘mcml_glmer‘ functions.

Value

A formula.

Examples

```
df <- data.frame(cl = 1:3, t = 4:6)
f1 <- lme4_to_glmmr(y ~ x + (1|cl/t),colnames(df))
```

logLik.mcml *Extracts the log-likelihood from an mcml object*

Description

Extracts the final log-likelihood value from an mcml object returned from call of ‘MCML‘ or ‘LA‘ in the Model class. The fitting algorithm estimates the fixed effects, random effects, and covariance parameters all separately. The log-likelihood is separable in the fixed and covariance parameters, so one can return the log-likelihood for either component, or the overall log-likelihood.

Usage

```
## S3 method for class 'mcml'
logLik(object, fixed = TRUE, covariance = TRUE, ...)
```

Arguments

`object`	An ‘mcml‘ model fit.
`fixed`	Logical whether to include the log-likelihood value from the fixed effects.
`covariance`	Logical whether to include the log-likelihood value from the covariance parameters.
`...`	Further arguments passed from other methods

Value

An object of class ‘logLik‘. If both ‘fixed‘ and ‘covariance‘ are FALSE then it returns NA.

`logLik.Model` *Extracts the log-likelihood from an mcml object*

Description

Extracts the log-likelihood value from an ‘Model‘ object. If no data ‘y‘ are specified then it returns NA.

Usage

```
## S3 method for class 'Model'
logLik(object, ...)
```

Arguments

`object`	An ‘Model‘ object.
`...`	Further arguments passed from other methods

Value

An object of class ‘logLik‘. If both ‘fixed‘ and ‘covariance‘ are FALSE then it returns NA.

`match_rows` *Generate matrix mapping between data frames*

Description

For a data frames ‘x‘ and ‘target‘, the function will return a matrix mapping the rows of ‘x‘ to those of ‘target‘.

Usage

```
match_rows(x, target, by)
```

Arguments

`x`	data.frame
`target`	data.frame to map to
`by`	vector of strings naming columns in ‘x‘ and ‘target‘

Details

‘x‘ is a data frame with n rows and ‘target‘ a data frame with m rows. This function will return a n times m matrix that maps the rows of ‘x‘ to those of ‘target‘ based on the values in the columns specified by the argument ‘by‘

Value

A matrix with nrow(x) rows and nrow(target) columns

Examples

```
df <- nelder(~(cl(10)*t(5)) > ind(10))
df_unique <- df[!duplicated(df[,c('cl','t')]),]
match_rows(df,df_unique,c('cl','t'))
```

`mcml_glmer` *lme4 style generlized linear mixed model*

Description

A wrapper for Model stochastic maximum likelihood model fitting replicating lme4’s syntax

Usage

```
mcml_glmer(
  formula,
  data,
  family,
  start = NULL,
  offset = NULL,
  verbose = 1L,
  iter.warmup = 100,
  iter.sampling = 50,
  weights = NULL,
  ...
)
```

Arguments

`formula`	A two-sided linear formula object including both the fixed and random effects specifications, see Details.
`data`	A data frame containing the variables named in ‘formula‘.
`family`	A family object expressing the distribution and link function of the model, see family.
`start`	Optional. A vector of starting values for the fixed effects.
`offset`	Optional. A vector of offset values.
`verbose`	Integer, controls the level of detail printed to the console, either 0 (no output), 1 (main output), or 2 (detailed output)
`iter.warmup`	The number of warmup iterations for the MCMC sampling step of each iteration.
`iter.sampling`	The number of sampling iterations for the MCMC sampling step of each iteration.
`weights`	Optional. A vector of observation level weights to apply to the model fit.
`...`	additional arguments passed to ‘Model$MCML()‘

Details

This function aims to replicate the syntax of lme4’s ‘lmer‘ command. The specified formula can be the standard lme4 syntax, or alternatively a glmmrBase style formula can also be used to allow for the wider range of covariance function specifications. For example both ‘y~x+(1|cl/t)‘ and ‘y~x+(1|gr(cl))+(1|gr(cl)*ar1(t))‘ would be valid formulae.

Value

A ‘mcml‘ model fit object.

Examples

```
#create a data frame describing a cross-sectional parallel cluster
data(Salamanders, package = "glmmrBase")
## Not run:
glm0 <- mcml_glmer(mating~fpop:mpop-1+(1|mnum)+(1|fnum),
  data=Salamanders,family=binomial(),reml=FALSE)
glm1 <- mcml_glmer(mating~fpop:mpop-1+(1|mnum)+(1|fnum),
  data =Salamanders, family=binomial(),reml=TRUE)

## End(Not run)
```

`mcml_lmer` *lme4 style linear mixed model*

Description

A wrapper for Model stochastic maximum likelihood model fitting replicating lme4's syntax

Usage

```
mcml_lmer(
  formula,
  data,
  start = NULL,
  offset = NULL,
  verbose = 1L,
  iter.warmup = 100,
  iter.sampling = 50,
  weights = NULL,
  ...
)
```

Arguments

`formula`	A two-sided linear formula object including both the fixed and random effects specifications, see Details.
`data`	A data frame containing the variables named in 'formula'.
`start`	Optional. A vector of starting values for the fixed effects.
`offset`	Optional. A vector of offset values.
`verbose`	Integer, controls the level of detail printed to the console, either 0 (no output), 1 (main output), or 2 (detailed output)
`iter.warmup`	The number of warmup iterations for the MCMC sampling step of each iteration.
`iter.sampling`	The number of sampling iterations for the MCMC sampling step of each iteration.
`weights`	Optional. A vector of observation level weights to apply to the model fit.
`...`	additional arguments passed to 'Model$MCML()'

Details

This function aims to replicate the syntax of lme4's 'lmer' command. The specified formula can be the standard lme4 syntax, or alternatively a glmmrBase style formula can also be used to allow for the wider range of covariance function specifications. For example both 'y~x+(1|cl/t)' and 'y~x+(1|gr(cl))+(1|gr(cl)*ar1(t))' would be valid formulae.

Value

A 'mcml' model fit object.

Examples

```
#create a data frame describing a cross-sectional parallel cluster
#randomised trial
df <- nelder(~(cl(10)*t(5)) > ind(10))
df$int <- 0
df[df$cl > 5, 'int'] <- 1
# simulate data using the Model class
df$y <- Model$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)) + (1|gr(cl,t)),
  data = df,
  family = stats::gaussian()
)$sim_data()
## Not run:
fit <- mcml_lmer(y ~ factor(t) + int - 1 + (1|cl/t), data = df)

## End(Not run)
```

---

mcnr_family — *Returns the file name and type for MCNR function*

Description

Returns the file name and type for MCNR function

Usage

```
mcnr_family(family, cmdstan)
```

Arguments

family	family object
cmdstan	Logical indicating whether cmdstan is being used and the function will return the filename

Value

list with filename and type

`MeanFunction` *For the generalised linear mixed model*

Description

For the generalised linear mixed model

For the generalised linear mixed model

Details

$$Y \sim F(\mu, \sigma)$$

$$\mu = h^-1(X\beta + Z\gamma)$$

$$\gamma \sim MVN(0, D)$$

this class defines the fixed effects design matrix X. The mean function is defined by a model formula, data, and parameters. A new instance can be generated with $new(). The class will generate the relevant matrix X automatically. See glmmrBase for a detailed guide on model specification.

Specification of the mean function follows standard model formulae in R. For example for a stepped-wedge cluster trial model, a typical mean model is $E(y_{ijt}|\delta) = \beta_0 + \tau_t + \beta_1 d_{jt} + z_{ijt}\delta$ where τ_t are fixed effects for each time period. The formula specification for this would be '~ factor(t) + int' where 'int' is the name of the variable indicating the treatment.

One can also include non-linear functions of variables in the mean function, and name the parameters. The resulting X matrix is then a matrix of first-order partial derivatives. For example, one can specify '~ int + b_1*exp(b_2*x)'.

Using 'update_parameters()' is the preferred way of updating the parameters of the mean or covariance objects as opposed to direct assignment, e.g. 'self$parameters <- c(...)'. The function calls check functions to automatically update linked matrices with the new parameters.

Public fields

`formula` model formula for the fixed effects

`data` Data frame with data required to build X

`parameters` A vector of parameter values for β used for simulating data and calculating covariance matrix of observations for non-linear models.

`offset` An optional vector specifying the offset values

`X` the fixed effects design matrix

Methods

Public methods:

- MeanFunction$n()
- MeanFunction$new()
- MeanFunction$print()
- MeanFunction$update_parameters()
- MeanFunction$colnames()
- MeanFunction$subset_rows()
- MeanFunction$linear_predictor()
- MeanFunction$any_nonlinear()
- MeanFunction$clone()

Method `n()`**:** Returns the number of observations

Usage:

```
MeanFunction$n()
```

Arguments:

`...` ignored

Returns: The number of observations in the model

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
df <- nelder(~(cl(4)*t(5)) > ind(5))
df$int <- 0
df[df$cl <= 2, 'int'] <- 1
mf1 <- MeanFunction$new(formula = ~ int ,
                        data=df,
                        parameters = c(-1,1)
                        )
mf1$n()
```

Method `new()`**:** Create a new MeanFunction object

Usage:

```
MeanFunction$new(
  formula,
  data,
  parameters = NULL,
  offset = NULL,
  verbose = FALSE
)
```

Arguments:

`formula` A formula object that describes the mean function, see Details

`data` (Optional) A data frame containing the covariates in the model, named in the model formula `parameters` (Optional) A vector with the values of the parameters β to use in data simulation and covariance calculations. If the parameters are not specified then they are initialised to 0.

`offset` A vector of offset values (optional)

`verbose` Logical indicating whether to report detailed output

Returns: A MeanFunction object

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
df <- nelder(~(cl(4)*t(5)) > ind(5))
df$int <- 0
df[df$cl <= 2, 'int'] <- 1
mf1 <- MeanFunction$new(formula = ~ int ,
                        data=df,
                        parameters = c(-1,1),
                        )
```

Method `print()`**:** Prints details about the object

Usage:

```
MeanFunction$print()
```

Arguments:

`...` ignored

Method `update_parameters()`**:** Updates the model parameters

Usage:

```
MeanFunction$update_parameters(parameters)
```

Arguments:

`parameters` A vector of parameters for the mean function.

`verbose` Logical indicating whether to provide more detailed feedback

Method `colnames()`**:** Returns or replaces the column names of the data in the object

Usage:

```
MeanFunction$colnames(names = NULL)
```

Arguments:

`names` If NULL then the function prints the column names, if a vector of names, then it attempts to replace the current column names of the data

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
df <- nelder(~(cl(4)*t(5)) > ind(5))
df$int <- 0
df[df$cl <= 5, 'int'] <- 1
```

```
mf1 <- MeanFunction$new(formula = ~ int ,
                        data=df,
                        parameters = c(-1,1)
                        )
mf1$colnames(c("cluster","time","individual","treatment"))
mf1$colnames()
```

Method `subset_rows()`**:** Keeps a subset of the data and removes the rest

All indices not in the provided vector of row numbers will be removed from both the data and fixed effects design matrix X.

Usage:

```
MeanFunction$subset_rows(index)
```

Arguments:

`index` Rows of the data to keep

Returns: NULL

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
df <- nelder(~(cl(4)*t(5)) > ind(5))
df$int <- 0
df[df$cl <= 5, 'int'] <- 1
mf1 <- MeanFunction$new(formula = ~ int ,
                        data=df,
                        parameters = c(-1,1)
                        )
mf1$subset_rows(1:20)
```

Method `linear_predictor()`**:** Returns the linear predictor

Returns the linear predictor, X * beta

Usage:

```
MeanFunction$linear_predictor()
```

Returns: A vector

Method `any_nonlinear()`**:** Returns a logical indicating whether the mean function contains non-linear functions of model parameters. Mainly used internally.

Usage:

```
MeanFunction$any_nonlinear()
```

Returns: None. Called for effects

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MeanFunction$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `MeanFunction$n`
## ------------------------------------------------


df <- nelder(~(cl(4)*t(5)) > ind(5))
df$int <- 0
df[df$cl <= 2, 'int'] <- 1
mf1 <- MeanFunction$new(formula = ~ int ,
                        data=df,
                        parameters = c(-1,1)
                        )
mf1$n()

## ------------------------------------------------
## Method `MeanFunction$new`
## ------------------------------------------------


df <- nelder(~(cl(4)*t(5)) > ind(5))
df$int <- 0
df[df$cl <= 2, 'int'] <- 1
mf1 <- MeanFunction$new(formula = ~ int ,
                        data=df,
                        parameters = c(-1,1),
                        )

## ------------------------------------------------
## Method `MeanFunction$colnames`
## ------------------------------------------------


df <- nelder(~(cl(4)*t(5)) > ind(5))
df$int <- 0
df[df$cl <= 5, 'int'] <- 1
mf1 <- MeanFunction$new(formula = ~ int ,
                        data=df,
                        parameters = c(-1,1)
                        )
mf1$colnames(c("cluster","time","individual","treatment"))
mf1$colnames()

## ------------------------------------------------
## Method `MeanFunction$subset_rows`
## ------------------------------------------------


df <- nelder(~(cl(4)*t(5)) > ind(5))
df$int <- 0
df[df$cl <= 5, 'int'] <- 1
mf1 <- MeanFunction$new(formula = ~ int ,
```

```
                        data=df,
                        parameters = c(-1,1)
                        )
mf1$subset_rows(1:20)
```

Model *A GLMM Model*

Description

A GLMM Model

A GLMM Model

Details

A generalised linear mixed model

See glmmrBase-package for a more in-depth guide.

The generalised linear mixed model is:

$$Y \sim F(\mu, \sigma)$$
$$\mu = h^{-}1(X\beta + Zu)$$
$$u \sim MVN(0, D)$$

where F is a distribution with scale parameter

$$\sigma$$

, h is a link function, X are the fixed effects with parameters

$$\beta$$

, Z is the random effect design matrix with multivariate Gaussian distributed effects u. The class provides access to all of the elements of the model above and associated calculations and functions including model fitting, power analysis, and various relevant matrices, including information matrices and related corrections. The object is an R6 class and so can serve as a parent class for extended functionality.

The currently supported families (links) are Gaussian (identity, log), Binomial (logit, log, probit, identity), Poisson (log, identity), Gamma (logit, identity, inverse), and Beta (logit).

This class provides model fitting functionality with a variety of stochastic maximum likelihood algorithms with and without restricted maximum likelihood corrections. A fast Laplace approximation is also included. Small sample corrections are also provided including Kenward-Roger and Satterthwaite corrections.

Many calculations use the covariance matrix of the observations, such as the information matrix, which is used in power calculations and other functions. For non-Gaussian models, the class uses the first-order approximation proposed by Breslow and Clayton (1993) based on the marginal quasi-likelihood:

$$\Sigma = W^{-1} + ZDZ^T$$

where W is a diagonal matrix with the GLM iterated weights for each observation equal to, for individual i $\left(\frac{(\partial h^{-1}(\eta_i))}{\partial \eta_i}\right)^2 Var(y|u)$ (see Table 2.1 in McCullagh and Nelder (1989)). The modification proposed by Zegers et al to the linear predictor to improve the accuracy of approximations based on the marginal quasilikelihood is also available, see `use_attenuation()`.

See glmmrBase for a detailed guide on model specification. A detailed vingette for this package is also available online`doi:10.48550/arXiv.2303.12657`.

Attenuation For calculations such as the information matrix, the first-order approximation to the covariance matrix proposed by Breslow and Clayton (1993), described above, is used. The approximation is based on the marginal quasilikelihood. Zegers, Liang, and Albert (1988) suggest that a better approximation to the marginal mean is achieved by "attenuating" the linear predictor. Setting `use` equal to TRUE uses this adjustment for calculations using the covariance matrix for non-linear models.

Calls the respective print methods of the linked covariance and mean function objects.

The matrices X and Z both have n rows, where n is the number of observations in the model/design.

Using `update_parameters()` is the preferred way of updating the parameters of the mean or covariance objects as opposed to direct assignment, e.g. `self$covariance$parameters <- c(...)`. The function calls check functions to automatically update linked matrices with the new parameters.

Monte Carlo maximum likelihood Fits generalised linear mixed models using one of several algorithms: Markov Chain Newton Raphson (MCNR), Markov Chain Expectation Maximisation (MCEM), or stochastic approximation expectation maximisation (SAEM) with or without Polyak-Ruppert averaging. MCNR and MCEM are described by McCulloch (1997) `doi:10.1080/01621459.1997.10473613`. For each iteration of the algorithms the unobserved random effect terms (γ) are simulated using Markov Chain Monte Carlo (MCMC) methods, and then these values are conditioned on in the subsequent steps to estimate the covariance parameters and the mean function parameters (β). SAEM uses a Robbins-Munroe approach to approximating the likelihood and requires fewer MCMC samples and may have lower Monte Carlo error, see Jank (2006)`doi:10.1198/106186006X157469`. The option `alpha` determines the rate at which succesive iterations "forget" the past and must be between 0.5 and 1. Higher values will result in lower Monte Carlo error but slower convergence. The options `mcem.adapt` and `mcnr.adapt` will modify the number of MCMC samples during each step of model fitting using the suggested values in Caffo, Jank, and Jones (2006)`doi:10.1111/j.1467-9868.2005.00499.x` as the estimates converge.

The accuracy of the algorithm depends on the user specified tolerance. For higher levels of tolerance, larger numbers of MCMC samples are likely need to sufficiently reduce Monte Carlo error. However, the SAEM approach does overcome reduce the required samples, especially with R-P averaging. As such a lower number (20-50) samples per iteration is normally sufficient to get convergence.

There are several stopping rules for the algorithm. Either the algorithm will terminate when successive parameter estimates are all within a specified tolerance of each other (`conv.criterion = 1`), or when there is a high probability that the estimated log-likelihood has not been improved. This latter criterion can be applied to either the overall log-likelihood (`conv.criterion = 2`), the likelihood just for the fixed effects (`conv.criterion = 3`), or both the likelihoods for the fixed effects and covariance parameters (`conv.criterion = 4`; default).

Options for the MCMC sampler are set by changing the values in `self$mcmc_options`. The information printed to the console during model fitting can be controlled with the `self$set_trace()` function.

To provide weights for the model fitting, store them in self$weights. To set the number of trials for binomial models, set self$trials.

Laplace approximation Fits generalised linear mixed models using Laplace approximation to the log likelihood. For non-Gaussian models the covariance matrix is approximated using the first order approximation based on the marginal quasilikelihood proposed by Breslow and Clayton (1993). The marginal mean in this approximation can be further adjusted following the proposal of Zeger et al (1988), use the member function `use_attenuated()` in this class, see Model. To provide weights for the model fitting, store them in self$weights. To set the number of trials for binomial models, set self$trials. To control the information printed to the console during model fitting use the `self$set_trace()` function.

Public fields

`covariance` A Covariance object defining the random effects covariance.

`mean` A MeanFunction object, defining the mean function for the model, including the data and covariate design matrix X.

`family` One of the family function used in R's glm functions. See family for details

`weights` A vector indicting the weights for the observations.

`trials` For binomial family models, the number of trials for each observation. The default is 1 (bernoulli).

`formula` The formula for the model. May be empty if separate formulae are specified for the mean and covariance components.

`var_par` Scale parameter required for some distributions (Gaussian, Gamma, Beta).

`mcmc_options` There are five options for MCMC sampling that are specified in this list:

- `warmup` The number of warmup iterations. Note that if using the internal HMC sampler, this only applies to the first iteration of the MCML algorithm, as the values from the previous iteration are carried over.
- `samps` The number of MCMC samples drawn in the MCML algorithms. For smaller tolerance values larger numbers of samples are required. For the internal HMC sampler, larger numbers of samples are generally required than if using Stan since the samples generally exhibit higher autocorrealtion, especially for more complex covariance structures. For SAEM a small number is recommended as all samples are stored and used from every iteration.
- `lambda` (Only relevant for the internal HMC sampler) Value of the trajectory length of the leapfrog integrator in Hamiltonian Monte Carlo (equal to number of steps times the step length). Larger values result in lower correlation in samples, but require larger numbers of steps and so is slower. Smaller numbers are likely required for non-linear GLMMs.
- `refresh` How frequently to print to console MCMC progress if displaying verbose output.
- `maxsteps` (Only relevant for the internal HMC sampler) Integer. The maximum number of steps of the leapfrom integrator

Methods

Public methods:

- Model$use_attenuation()
- Model$fitted()
- Model$residuals()
- Model$predict()
- Model$new()
- Model$print()
- Model$n()
- Model$subset_rows()
- Model$sim_data()
- Model$update_parameters()
- Model$information_matrix()
- Model$sandwich()
- Model$small_sample_correction()
- Model$box()
- Model$power()
- Model$w_matrix()
- Model$dh_deta()
- Model$Sigma()
- Model$MCML()
- Model$LA()
- Model$sparse()
- Model$mcmc_sample()
- Model$gradient()
- Model$partial_sigma()
- Model$u()
- Model$log_likelihood()
- Model$calculator_instructions()
- Model$marginal()
- Model$update_y()
- Model$set_trace()
- Model$clone()

Method `use_attenuation()`**:** Sets the model to use or not use "attenuation" when calculating the first-order approximation to the covariance matrix.

Usage:

```
Model$use_attenuation(use)
```

Arguments:

`use` Logical indicating whether to use "attenuation".

Returns: None. Used for effects.

Method `fitted()`**:** Return fitted values. Does not account for the random effects. For simulated values based on resampling random effects, see also `sim_data()`. To predict the values including random effects at a new location see also `predict()`.

Usage:

```
Model$fitted(type = "link", X, u, sample = FALSE, sample_n = 100)
```

Arguments:

`type` One of either "`link`" for values on the scale of the link function, or "`response`" for values on the scale of the response

`X` (Optional) Fixed effects matrix to generate fitted values

`u` (Optional) Random effects values at which to generate fitted values

`sample` Logical. If TRUE then the parameters will be re-sampled from their sampling distribution. Currently only works with existing X matrix and not user supplied matrix X and this will also ignore any provided random effects.

`sample_n` Integer. If sample is TRUE, then this is the number of samples.

Returns: Fitted values as either a vector or matrix depending on the number of samples

Method `residuals()`**:** Generates the residuals for the model

Generates one of several types of residual for the model. If conditional = TRUE then the residuals include the random effects, otherwise only the fixed effects are included. For type, there are raw, pearson, and standardized residuals. For conditional residuals a matrix is returned with each column corresponding to a sample of the random effects.

Usage:

```
Model$residuals(type = "standardized", conditional = TRUE)
```

Arguments:

`type` Either "standardized", "raw" or "pearson"

`conditional` Logical indicating whether to condition on the random effects (TRUE) or not (FALSE)

Returns: A matrix with either one column is conditional is false, or with number of columns corresponding to the number of MCMC samples.

Method `predict()`**:** Generate predictions at new values

Generates predicted values using a new data set to specify covariance values and values for the variables that define the covariance function. The function will return a list with the linear predictor, conditional distribution of the new random effects term conditional on the current estimates of the random effects, and some simulated values of the random effects if requested.

Usage:

```
Model$predict(newdata, offset = rep(0, nrow(newdata)), m = 0)
```

Arguments:

`newdata` A data frame specifying the new data at which to generate predictions

`offset` Optional vector of offset values for the new data

`m` Number of samples of the random effects to draw

Returns: A list with the linear predictor, parameters (mean and covariance matrices) for the conditional distribution of the random effects, and any random effect samples.

Method `new()`**:** Create a new Model object. Typically, a model is generated from a formula and data. However, it can also be generated from a previous model fit.

Usage:

```
Model$new(
  formula,
  covariance,
  mean,
  data = NULL,
  family = NULL,
  var_par = NULL,
  offset = NULL,
  weights = NULL,
  trials = NULL,
  model_fit = NULL
)
```

Arguments:

`formula` A model formula containing fixed and random effect terms. The formula can be one way (e.g. `~ x + (1|gr(cl))`) or two-way (e.g. `y ~ x + (1|gr(cl))`). One way formulae will generate a valid model enabling data simulation, matrix calculation, and power, etc. Outcome data can be passed directly to model fitting functions, or updated later using member function `update_y()`. For binomial models, either the syntax `cbind(y, n-y)` can be used for outcomes, or just `y` and the number of trials passed to the argument `trials` described below.

`covariance` (Optional) Either a Covariance object, an equivalent list of arguments that can be passed to `Covariance` to create a new object, or a vector of parameter values. At a minimum the list must specify a formula. If parameters are not included then they are initialised to 0.5.

`mean` (Optional) Either a MeanFunction object, an equivalent list of arguments that can be passed to `MeanFunction` to create a new object, or a vector of parameter values. At a minimum the list must specify a formula. If parameters are not included then they are initialised to 0.

`data` A data frame with the data required for the mean function and covariance objects. This argument can be ignored if data are provided to the covariance or mean arguments either via `Covariance` and `MeanFunction` object, or as a member of the list of arguments to both `covariance` and `mean`.

`family` A family object expressing the distribution and link function of the model, see family. Currently accepts binomial, gaussian, Gamma, poisson, Beta, and Quantile.

`var_par` (Optional) Scale parameter required for some distributions, including Gaussian. Default is NULL.

`offset` (Optional) A vector of offset values. Optional - could be provided to the argument to mean instead.

`weights` (Optional) A vector of weights.

`trials` (Optional) For binomial family models, the number of trials for each observation. If it is not set, then it will default to 1 (a bernoulli model).

`model_fit` (optional) A `mcml` model fit resulting from a call to `MCML` or `LA`

Returns: A new Model class object

Examples:

```
\dontshow{
setParallel(FALSE)
}
# For more examples, see the examples for MCML.

#create a data frame describing a cross-sectional parallel cluster
#randomised trial
df <- nelder(~(cl(10)*t(5)) > ind(10))
df$int <- 0
df[df$cl > 5, 'int'] <- 1
mod <- Model$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)) + (1|gr(cl,t)),
  data = df,
  family = stats::gaussian()
)

# We can also include the outcome data in the model initialisation.
# For example, simulating data and creating a new object:
df$y <- mod$sim_data()

mod <- Model$new(
  formula = y ~ factor(t) + int - 1 + (1|gr(cl)) + (1|gr(cl,t)),
  data = df,
  family = stats::gaussian()
)

# Here we will specify a cohort study
df <- nelder(~ind(20) * t(6))
df$int <- 0
df[df$t > 3, 'int'] <- 1

des <- Model$new(
  formula = ~ int + (1|gr(ind)),
  data = df,
  family = stats::poisson()
)

# or with parameter values specified

des <- Model$new(
  formula = ~ int + (1|gr(ind)),
  covariance = c(0.05),
  mean = c(1,0.5),
  data = df,
  family = stats::poisson()
)
```

```
#an example of a spatial grid with two time points

df <- nelder(~ (x(10)*y(10))*t(2))
spt_design <- Model$new(formula = ~ 1 + (1|ar0(t)*fexp(x,y)),
                        data = df,
                        family = stats::gaussian())
```

Method `print()`**:** Print method for `Model` class

Usage:

`Model$print()`

Arguments:

`...` ignored

Method `n()`**:** Returns the number of observations in the model

Usage:

`Model$n(...)`

Arguments:

`...` ignored

Method `subset_rows()`**:** Subsets the design keeping specified observations only

Given a vector of row indices, the corresponding rows will be kept and the other rows will be removed from the mean function and covariance

Usage:

`Model$subset_rows(index)`

Arguments:

`index` Integer or vector integers listing the rows to keep

Returns: The function updates the object and nothing is returned.

Method `sim_data()`**:** Generates a realisation of the design

Generates a single vector of outcome data based upon the specified GLMM design.

Usage:

`Model$sim_data(type = "y")`

Arguments:

`type` Either 'y' to return just the outcome data, 'data' to return a data frame with the simulated outcome data alongside the model data, or 'all', which will return a list with simulated outcomes y, matrices X and Z, parameters beta, and the values of the simulated random effects.

Returns: Either a vector, a data frame, or a list

Examples:

```
df <- nelder(~(cl(10)*t(5)) > ind(10))
df$int <- 0
df[df$cl > 5, 'int'] <- 1
\dontshow{
```

```
setParallel(FALSE) # for the CRAN check
}
des <- Model$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)*ar0(t)),
  covariance = c(0.05,0.8),
  mean = c(rep(0,5),0.6),
  data = df,
  family = stats::binomial()
)
ysim <- des$sim_data()
```

Method `update_parameters()`: Updates the parameters of the mean function and/or the covariance function

Usage:

```
Model$update_parameters(mean.pars = NULL, cov.pars = NULL, var.par = NULL)
```

Arguments:

`mean.pars` (Optional) Vector of new mean function parameters

`cov.pars` (Optional) Vector of new covariance function(s) parameters

`var.par` (Optional) A scalar value for var_par

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
df <- nelder(~(cl(10)*t(5)) > ind(10))
df$int <- 0
df[df$cl > 5, 'int'] <- 1
des <- Model$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)*ar0(t)),
  data = df,
  family = stats::binomial()
)
des$update_parameters(cov.pars = c(0.1,0.9))
```

Method `information_matrix()`: Generates the information matrix of the mixed model GLS estimator (X'S^-1X). The inverse of this matrix is an estimator for the variance-covariance matrix of the fixed effect parameters. For various small sample corrections see `small_sample_correction()` and `box()`. For models with non-linear functions of fixed effect parameters, a correction to the Hessian matrix is required, which is automatically calculated or optionally returned or disabled.

Usage:

```
Model$information_matrix(include.re = FALSE, theta = FALSE, oim = FALSE)
```

Arguments:

`include.re` logical indicating whether to return the information matrix including the random effects components (TRUE), or the mixed model information matrix for beta only (FALSE).

`theta` Logical. If TRUE the function will return the variance-coviariance matrix for the covariance parameters and ignore the first argument. Otherwise, the fixed effect parameter information matrix is returned.

`oim` Logical. If TRUE, returns the observed information matrix for both beta and theta, disregarding other arguments to the function.

Returns: A matrix

Method `sandwich()`: Returns the robust sandwich variance-covariance matrix for the fixed effect parameters

Usage:

```
Model$sandwich()
```

Returns: A PxP matrix

Method `small_sample_correction()`: Returns a small sample correction. The option "KR" returns the Kenward-Roger bias-corrected variance-covariance matrix for the fixed effect parameters and degrees of freedom. Option "KR2" returns an improved correction given in Kenward & Roger (2009) doi:j.csda.2008.12.013. Note, that the corrected/improved version is invariant under reparameterisation of the covariance, and it will also make no difference if the covariance is linear in parameters. Exchangeable covariance structures in this package (i.e. `gr()`) are parameterised in terms of the variance rather than standard deviation, so the results will be unaffected. Option "sat" returns the "Satterthwaite" correction, which only includes corrected degrees of freedom, along with the GLS standard errors.

Usage:

```
Model$small_sample_correction(type, oim = FALSE)
```

Arguments:

`type` Either "KR", "KR2", or "sat", see description.

`oim` Logical. If TRUE use the observed information matrix, otherwise use the expected information matrix

Returns: A PxP matrix

Method `box()`: Returns the inferential statistics (F-stat, p-value) for a modified Box correction doi:10.1002/sim.4072 for Gaussian-identity models.

Usage:

```
Model$box(y)
```

Arguments:

`y` Optional. If provided, will update the vector of outcome data. Otherwise it will use the data from the previous model fit.

Returns: A data frame.

Method `power()`: Estimates the power of the design described by the model using the square root of the relevant element of the GLS variance matrix:

$$(X^T\Sigma^{-1}X)^{-1}$$

Note that this is equivalent to using the "design effect" for many models.

Usage:

```
Model$power(alpha = 0.05, two.sided = TRUE, alternative = "pos")
```

Arguments:

`alpha` Numeric between zero and one indicating the type I error rate. Default of 0.05.

`two.sided` Logical indicating whether to use a two sided test

`alternative` For a one-sided test whether the alternative hypothesis is that the parameter is positive "pos" or negative "neg"

Returns: A data frame describing the parameters, their values, expected standard errors and estimated power.

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
df <- nelder(~(cl(10)*t(5)) > ind(10))
df$int <- 0
df[df$cl > 5, 'int'] <- 1
des <- Model$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)) + (1|gr(cl,t)),
  covariance = c(0.05,0.1),
  mean = c(rep(0,5),0.6),
  data = df,
  family = stats::gaussian(),
  var_par = 1
)
des$power() #power of 0.90 for the int parameter
```

Method `w_matrix()`**:** Returns the diagonal of the matrix W used to calculate the covariance matrix approximation

Usage:

```
Model$w_matrix()
```

Returns: A vector with values of the glm iterated weights

Method `dh_deta()`**:** Returns the derivative of the link function with respect to the linear preditor

Usage:

```
Model$dh_deta()
```

Returns: A vector

Method `Sigma()`**:** Returns the (approximate) covariance matrix of y

Returns the covariance matrix Sigma. For non-linear models this is an approximation. See Details.

Usage:

```
Model$Sigma(inverse = FALSE)
```

Arguments:

`inverse` Logical indicating whether to provide the covariance matrix or its inverse

Returns: A matrix.

Method `MCML()`**:** Stochastic Maximum Likelihood model fitting

Usage:

```
Model$MCML(
  y = NULL,
  method = "saem",
  tol = 0.01,
  max.iter = 50,
  se = "gls",
  oim = FALSE,
  reml = TRUE,
  mcmc.pkg = "rstan",
  se.theta = TRUE,
  algo = 2,
  lower.bound = NULL,
  upper.bound = NULL,
  lower.bound.theta = NULL,
  upper.bound.theta = NULL,
  alpha = 0.8,
  convergence.prob = 0.95,
  pr.average = FALSE,
  conv.criterion = 2,
  skip.theta = FALSE
)
```

Arguments:

`y` Optional. A numeric vector of outcome data. If this is not provided then either the outcome must have been specified when initialising the Model object, or the outcome data has been updated using member function `update_y()`

`method` The MCML algorithm to use, either `mcem` or `mcnr`, or `saem` see Details. Default is `saem`. `mcem.adapt` and `mcnr.adapt` will use adaptive MCMC sample sizes starting small and increasing to the the maximum value specified in `mcmc_options$sampling`, which results in faster convergence. `saem` uses a stochastic approximation expectation maximisation algorithm. MCMC samples are kept from all iterations and so a smaller number of samples are needed per iteration.

`tol` Numeric value, tolerance of the MCML algorithm, the maximum difference in parameter estimates between iterations at which to stop the algorithm. If two values are provided then different tolerances will be applied to the fixed effect and covariance parameters.

`max.iter` Integer. The maximum number of iterations of the MCML algorithm.

`se` String. Type of standard error and/or inferential statistics to return. Options are "gls" for GLS standard errors (the default), "robust" for robust standard errors, "kr" for original Kenward-Roger bias corrected standard errors, "kr2" for the improved Kenward-Roger correction, "sat" for Satterthwaite degrees of freedom correction (this is the same degrees of freedom correction as Kenward-Roger, but with GLS standard errors), "box" to use a modified Box correction (does not return confidence intervals), "bw" to use GLS standard errors with a between-within correction to the degrees of freedom, "bwrobust" to use robust standard errors with between-within correction to the degrees of freedom.

`oim` Logical. If TRUE use the observed information matrix, otherwise use the expected information matrix for standard error and related calculations.

`reml` Logical. Whether to use a restricted maximum likelihood correction for fitting the covariance parameters `mcmc.pkg` String. Either `cmdstan` for cmdstan (requires the package `cmdstanr`), `rstan` to use rstan sampler, or `hmc` to use a cruder Hamiltonian Monte Carlo sampler. cmdstan is recommended as it has by far the best number of effective samples per unit time. cmdstanr will compile the MCMC programs to the library folder the first time they are run, so may not currently be an option for some users.

`se.theta` Logical. Whether to calculate the standard errors for the covariance parameters. This step is a slow part of the calculation, so can be disabled if required in larger models. Has no effect for Kenward-Roger standard errors.

`algo` Integer. 1 = L-BFGS for beta and BOBYQA for theta, 2 = BOBYQA for both, 3 = L-BFGS for both (default). The L-BFGS algorithm may perform poorly with some covariance structures, in this case select 1 or 2, or apply an upper bound.

`lower.bound` Optional. Vector of lower bounds for the fixed effect parameters. To apply bounds use MCEM.

`upper.bound` Optional. Vector of upper bounds for the fixed effect parameters. To apply bounds use MCEM.

`lower.bound.theta` Optional. Vector of lower bounds for the covariance parameters (default is 0; negative values will cause an error)

`upper.bound.theta` Optional. Vector of upper bounds for the covariance parameters.

`alpha` If using SAEM then this parameter controls the step size. On each iteration i the step size is (1/alpha)^i, default is 0.8. Values around 0.5 will result in lower bias but slower convergence, values closer to 1 will result in higher convergence but potentially higher error.

`convergence.prob` Numeric value in (0,1) indicating the probability of convergence if using convergence criteria 2, 3, or 4.

`pr.average` Logical indicating whether to use Polyak-Ruppert averaging if using the SAEM algorithm (default is TRUE)

`conv.criterion` Integer. The convergence criterion for the algorithm. 1 = the maximum difference between parameter estimates between iterations as defined by `tol`, 2 = The probability of improvement in the overall log-likelihood is less than 1 - `convergence.prob` 3 = The probability of improvement in the log-likelihood for the fixed effects is less than 1 - `convergence.prob` 4 = The probabilities of improvement in the log-likelihood the fixed effects and covariance parameters are both less than 1 - `convergence.prob`

`skip.theta` Logical. If TRUE then the covariance parameter estimation step is skipped. This option is mainly used for testing, but may be useful if covariance parameters are known.

Returns: A `mcml` object

Examples:

```
\dontrun{
# Simulated trial data example
data(SimTrial,package = "glmmrBase")
model <- Model$new(
  formula = y ~ int + factor(t) - 1 + (1|gr(cl)*ar1(t)),
  data = SimTrial,
  family = gaussian()
)
glm3 <- model$MCML()
```

```
# Salamanders data example
data(Salamanders,package="glmmrBase")
model <- Model$new(
  mating~fpop:mpop-1+(1|gr(mnum))+(1|gr(fnum)),
  data = Salamanders,
  family = binomial()
)

# we will try MCEM with 500 MCMC iterations
model$mcmc_options$samps <- 500
# view the grouping structure
glm2 <- model$MCML(method = "mcem")

# Example using simulated data
#create example data with six clusters, five time periods, and five people per cluster-period
df <- nelder(~(cl(6)*t(5)) > ind(5))
# parallel trial design intervention indicator
df$int <- 0
df[df$cl > 3, 'int'] <- 1
# specify parameter values in the call for the data simulation below
des <- Model$new(
  formula= ~ factor(t) + int - 1 +(1|gr(cl)*ar0(t)),
  covariance = c(0.05,0.7),
  mean = c(rep(0,5),0.2),
  data = df,
  family = gaussian()
)
ysim <- des$sim_data() # simulate some data from the model
fit1 <- des$MCML(y = ysim) # Default model fitting with SAEM-PR
# use MCEM instead and stop when parameter values are within 1e-2 on successive iterations
fit2 <- des$MCML(y = ysim, method="mcem",tol=1e-2,conv.criterion = 1)

# Non-linear model fitting example using the example provided by nlmer in lme4
data(Orange, package = "lme4")

# the lme4 example:
startvec <- c(Asym = 200, xmid = 725, scal = 350)
(nm1 <- lme4::nlmer(circumference ~ SSlogis(age, Asym, xmid, scal) ~ Asym|Tree,
              Orange, start = startvec))

Orange <- as.data.frame(Orange)
Orange$Tree <- as.numeric(Orange$Tree)

# Here we can specify the model as a function.

model <- Model$new(
  circumference ~ Asym/(1 + exp((xmid - (age))/scal)) - 1 + (Asym|gr(Tree)),
  data = Orange,
```

```
  family = gaussian(),
  mean = c(200,725,350),
  covariance = c(500),
  var_par = 50
)

# for this example, we will use MCEM with adaptive MCMC sample sizes

model$mcmc_options$samps <- 1000
nfit <- model$MCML(method = "mcem.adapt")

summary(nfit)
summary(nm1)


}
```

Method `LA()`**:** Maximum Likelihood model fitting with Laplace Approximation

Usage:

```
Model$LA(
  y = NULL,
  start,
  method = "nr",
  se = "gls",
  oim = FALSE,
  reml = TRUE,
  max.iter = 40,
  tol = 1e-04,
  se.theta = TRUE,
  algo = 2,
  lower.bound = NULL,
  upper.bound = NULL,
  lower.bound.theta = NULL,
  upper.bound.theta = NULL
)
```

Arguments:

`y` Optional. A numeric vector of outcome data. If this is not provided then either the outcome must have been specified when initialising the Model object, or the outcome data has been updated using member function `update_y()`

`start` Optional. A numeric vector indicating starting values for the model parameters.

`method` String. Either "nloptim" for non-linear optimisation, or "nr" for Newton-Raphson (default) algorithm

`se` String. Type of standard error and/or inferential statistics to return. Options are "gls" for GLS standard errors (the default), "robust" for robust standard errors, "kr" for original Kenward-Roger bias corrected standard errors, "kr2" for the improved Kenward-Roger correction, "sat" for Satterthwaite degrees of freedom correction (this is the same degrees of freedom correction as Kenward-Roger, but with GLS standard errors)"box" to use a modified Box correction (does not return confidence intervals), "bw" to use GLS standard errors with a between-within correction to the degrees of freedom, "bwrobust" to use robust standard errors with between-within correction to the degrees of freedom. Note that Kenward-Roger assumes REML estimates, which are not currently provided by this function.

`oim` Logical. If TRUE use the observed information matrix, otherwise use the expected information matrix for standard error and related calculations.

`reml` Logical. Whether to use a restricted maximum likelihood correction for fitting the covariance parameters

`max.iter` Maximum number of algorithm iterations, default 20.

`tol` Maximum difference between successive iterations at which to terminate the algorithm

`se.theta` Logical. Whether to calculate the standard errors for the covariance parameters. This step is a slow part of the calculation, so can be disabled if required in larger models. Has no effect for Kenward-Roger standard errors.

`algo` Integer. 1 = L-BFGS for beta-u and BOBYQA for theta (default), 2 = BOBYQA for both.

`lower.bound` Optional. Vector of lower bounds for the fixed effect parameters. To apply bounds use nloptim.

`upper.bound` Optional. Vector of upper bounds for the fixed effect parameters. To apply bounds use nloptim.

`lower.bound.theta` Optional. Vector of lower bounds for the covariance parameters.

`upper.bound.theta` Optional. Vector of upper bounds for the covariance parameters.

Returns: A `mcml` object

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
#create example data with six clusters, five time periods, and five people per cluster-period
df <- nelder(~(cl(6)*t(5)) > ind(5))
# parallel trial design intervention indicator
df$int <- 0
df[df$cl > 3, 'int'] <- 1
# specify parameter values in the call for the data simulation below
des <- Model$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)*ar0(t)),
  covariance = c(0.05,0.7),
  mean = c(rep(0,5),-0.2),
  data = df,
  family = stats::binomial()
)
ysim <- des$sim_data() # simulate some data from the model
fit1 <- des$LA(y = ysim)
```

Method `sparse()`**:** Set whether to use sparse matrix methods for model calculations and fitting. By default the model does not use sparse matrix methods.

Usage:

```
Model$sparse(sparse = TRUE, amd = TRUE)
```

Arguments:

`sparse` Logical indicating whether to use sparse matrix methods

`amd` Logical indicating whether to use and Approximate Minimum Degree algorithm to calculate an efficient permutation matrix so that the Cholesky decomposition of PAP^T is calculated rather than A.

Returns: None, called for effects

Method `mcmc_sample()`**:** Generate an MCMC sample of the random effects

Usage:

```
Model$mcmc_sample(mcmc.pkg = "rstan")
```

Arguments:

`mcmc.pkg` String. Either `cmdstan` for cmdstan (requires the package `cmdstanr`), `rstan` to use rstan sampler, or `hmc` to use a cruder Hamiltonian Monte Carlo sampler. cmdstan is recommended as it has by far the best number of effective samples per unit time. cmdstanr will compile the MCMC programs to the library folder the first time they are run, so may not currently be an option for some users.

Returns: A matrix of samples of the random effects

Method `gradient()`**:** The gradient of the log-likelihood with respect to either the random effects or the model parameters. The random effects are on the N(0,I) scale, i.e. scaled by the Cholesky decomposition of the matrix D. To obtain the random effects from the last model fit, see member function `$u`

Usage:

```
Model$gradient(y, u, beta = FALSE)
```

Arguments:

`y` (optional) Vector of outcome data, if not specified then data must have been set in another function.

`u` (optional) Vector of random effects scaled by the Cholesky decomposition of D

`beta` Logical. Whether the log gradient for the random effects (FALSE) or for the linear predictor parameters (TRUE)

Returns: A vector of the gradient

Method `partial_sigma()`**:** The partial derivatives of the covariance matrix Sigma with respect to the covariance parameters. The function returns a list in order: Sigma, first order derivatives, second order derivatives. The second order derivatives are ordered as the lower-triangular matrix in column major order. Letting 'd(i)' mean the first-order partial derivative with respect to parameter i, and d2(i,j) mean the second order derivative with respect to parameters i and j, then if there were three covariance parameters the order of the output would be: (sigma, d(1), d(2), d(3), d2(1,1), d2(1,2), d2(1,3), d2(2,2), d2(2,3), d2(3,3)).

Usage:

```
Model$partial_sigma()
```

Returns: A list of matrices, see description for contents of the list.

Method `u()`**:** Returns the sample of random effects from the last model fit, or updates the samples for the model.

Usage:

```
Model$u(scaled = TRUE, u)
```

Arguments:

`scaled` Logical indicating whether the samples are on the N(0,I) scale (`scaled=FALSE`) or N(0,D) scale (`scaled=TRUE`)

`u` (optional) Matrix of random effect samples. If provided then the internal samples are replaced with these values. These samples should be N(0,I).

Returns: A matrix of random effect samples

Method `log_likelihood()`**:** The log likelihood for the GLMM. The random effects can be left unspecified. If no random effects are provided, and there was a previous model fit with the same data `y` then the random effects will be taken from that model. If there was no previous model fit then the random effects are assumed to be all zero.

Usage:

```
Model$log_likelihood(y, u)
```

Arguments:

`y` A vector of outcome data

`u` An optional matrix of random effect samples. This can be a single column.

Returns: The log-likelihood of the model parameters

Method `calculator_instructions()`**:** Prints the internal instructions and data used to calculate the linear predictor. Internally the class uses a reverse polish notation to store and calculate different functions, including user-specified non-linear mean functions. This function will print all the steps. Mainly used for debugging and determining how the class has interpreted non-linear model specifications.

Usage:

```
Model$calculator_instructions()
```

Returns: None. Called for effects.

Method `marginal()`**:** Calculates the marginal effect of variable `x`. There are several options for marginal effect and several types of conditioning or averaging. The type of marginal effect can be the derivative of the mean with respect to `x` (`dydx`), the expected difference E(y|x=a)-E(y|x=b) (`diff`), or the expected log ratio log(E(y|x=a)/E(y|x=b)) (`ratio`). Other fixed effect variables can be set at specific values (`at`), set at their mean values (`atmeans`), or averaged over (`average`). Averaging over a fixed effects variable here means using all observed values of the variable in the relevant calculation. The random effects can similarly be set at their estimated value (`re="estimated"`), set to zero (`re="zero"`), set to a specific value (`re="at"`), or averaged over (`re="average"`). Estimates of the expected values over the random effects are generated using MCMC samples. MCMC samples are generated either through MCML model fitting or using `mcmc_sample`. In the absence of samples `average` and `estimated` will produce the same result. The standard errors are calculated using the delta method with one of several options for the variance matrix of the fixed effect parameters. Several of the arguments require the names of the variables as given to the model object. Most variables are as specified in the formula, factor variables are specified as the name of the `variable_value`, e.g. `t_1`. To see the names of the stored parameters and data variables see the member function `names()`.

Usage:

```
Model$marginal(
  x,
  type,
  re,
  se,
  at = c(),
  atmeans = c(),
  average = c(),
  xvals = c(1, 0),
  atvals = c(),
  revals = c(),
  oim = FALSE
)
```

Arguments:

`x` String. Name of the variable to calculate the marginal effect for.

`type` String. Either `dydx` for derivative, `diff` for difference, or `ratio` for log ratio. See description.

`re` String. Either `estimated` to condition on estimated values, `zero` to set to zero, `at` to provide specific values, or `average` to average over the random effects.

`se` String. Type of standard error to use, either `GLS` for the GLS standard errors, `KR` for Kenward-Roger estimated standard errors, `KR2` for the improved Kenward-Roger correction (see `small_sample_correction()`), or `robust` to use a robust sandwich estimator.

`at` Optional. A vector of strings naming the fixed effects for which a specified value is given.

`atmeans` Optional. A vector of strings naming the fixed effects that will be set at their mean value.

`average` Optional. A vector of strings naming the fixed effects which will be averaged over.

`xvals` Optional. A vector specifying the values of `a` and `b` for `diff` and `ratio`. The default is (1,0).

`atvals` Optional. A vector specifying the values of fixed effects specified in `at` (in the same order).

`revals` Optional. If `re="at"` then this argument provides a vector of values for the random effects.

`oim` Logical. If TRUE use the observed information matrix, otherwise use the expected information matrix for standard error and related calculations.

Returns: A named vector with elements `margin` specifying the point estimate and `se` giving the standard error.

Method `update_y()`**:** Updates the outcome data y

Some functions require outcome data, which is by default set to all zero if no model fitting function has been run. This function can update the interval y data.

Usage:

```
Model$update_y(y)
```

Arguments:

`y` Vector of outcome data

Returns: None. Called for effects

Method `set_trace()`**:** Controls the information printed to the console for other functions.

Usage:

```
Model$set_trace(trace)
```

Arguments:

`trace` Integer, either 0 = no information, 1 = some information, 2 = all information

Returns: None. Called for effects.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Model$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Breslow, N. E., Clayton, D. G. (1993). Approximate Inference in Generalized Linear Mixed Models. Journal of the American Statistical Association<, 88(421), 9–25. doi:10.1080/01621459.1993.10594284

McCullagh P, Nelder JA (1989). Generalized linear models, 2nd Edition. Routledge.

McCulloch CE (1997). "Maximum Likelihood Algorithms for Generalized Linear Mixed Models." Journal of the American statistical Association, 92(437), 162–170.doi:10.2307/2291460

Zeger, S. L., Liang, K.-Y., Albert, P. S. (1988). Models for Longitudinal Data: A Generalized Estimating Equation Approach. Biometrics, 44(4), 1049.doi:10.2307/2531734

See Also

nelder, MeanFunction, Covariance

Model, Covariance, MeanFunction

Model, Covariance, MeanFunction

Examples

```
## ------------------------------------------------
## Method `Model$new`
## ------------------------------------------------


# For more examples, see the examples for MCML.

#create a data frame describing a cross-sectional parallel cluster
#randomised trial
df <- nelder(~(cl(10)*t(5)) > ind(10))
df$int <- 0
df[df$cl > 5, 'int'] <- 1
```

```
mod <- Model$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)) + (1|gr(cl,t)),
  data = df,
  family = stats::gaussian()
)

# We can also include the outcome data in the model initialisation.
# For example, simulating data and creating a new object:
df$y <- mod$sim_data()

mod <- Model$new(
  formula = y ~ factor(t) + int - 1 + (1|gr(cl)) + (1|gr(cl,t)),
  data = df,
  family = stats::gaussian()
)

# Here we will specify a cohort study
df <- nelder(~ind(20) * t(6))
df$int <- 0
df[df$t > 3, 'int'] <- 1

des <- Model$new(
  formula = ~ int + (1|gr(ind)),
  data = df,
  family = stats::poisson()
)

# or with parameter values specified

des <- Model$new(
  formula = ~ int + (1|gr(ind)),
  covariance = c(0.05),
  mean = c(1,0.5),
  data = df,
  family = stats::poisson()
)

#an example of a spatial grid with two time points

df <- nelder(~ (x(10)*y(10))*t(2))
spt_design <- Model$new(formula = ~ 1 + (1|ar0(t)*fexp(x,y)),
                        data = df,
                        family = stats::gaussian())

## ------------------------------------------------
## Method `Model$sim_data`
## ------------------------------------------------

df <- nelder(~(cl(10)*t(5)) > ind(10))
df$int <- 0
df[df$cl > 5, 'int'] <- 1

des <- Model$new(
```

```
  formula = ~ factor(t) + int - 1 + (1|gr(cl)*ar0(t)),
  covariance = c(0.05,0.8),
  mean = c(rep(0,5),0.6),
  data = df,
  family = stats::binomial()
)
ysim <- des$sim_data()

## ------------------------------------------------
## Method `Model$update_parameters`
## ------------------------------------------------


df <- nelder(~(cl(10)*t(5)) > ind(10))
df$int <- 0
df[df$cl > 5, 'int'] <- 1
des <- Model$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)*ar0(t)),
  data = df,
  family = stats::binomial()
)
des$update_parameters(cov.pars = c(0.1,0.9))

## ------------------------------------------------
## Method `Model$power`
## ------------------------------------------------


df <- nelder(~(cl(10)*t(5)) > ind(10))
df$int <- 0
df[df$cl > 5, 'int'] <- 1
des <- Model$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)) + (1|gr(cl,t)),
  covariance = c(0.05,0.1),
  mean = c(rep(0,5),0.6),
  data = df,
  family = stats::gaussian(),
  var_par = 1
)
des$power() #power of 0.90 for the int parameter

## ------------------------------------------------
## Method `Model$MCML`
## ------------------------------------------------

## Not run:
# Simulated trial data example
data(SimTrial,package = "glmmrBase")
model <- Model$new(
  formula = y ~ int + factor(t) - 1 + (1|gr(cl)*ar1(t)),
  data = SimTrial,
  family = gaussian()
)
```

```
glm3 <- model$MCML()

# Salamanders data example
data(Salamanders,package="glmmrBase")
model <- Model$new(
  mating~fpop:mpop-1+(1|gr(mnum))+(1|gr(fnum)),
  data = Salamanders,
  family = binomial()
)

# we will try MCEM with 500 MCMC iterations
model$mcmc_options$samps <- 500
# view the grouping structure
glm2 <- model$MCML(method = "mcem")

# Example using simulated data
#create example data with six clusters, five time periods, and five people per cluster-period
df <- nelder(~(cl(6)*t(5)) > ind(5))
# parallel trial design intervention indicator
df$int <- 0
df[df$cl > 3, 'int'] <- 1
# specify parameter values in the call for the data simulation below
des <- Model$new(
  formula= ~ factor(t) + int - 1 +(1|gr(cl)*ar0(t)),
  covariance = c(0.05,0.7),
  mean = c(rep(0,5),0.2),
  data = df,
  family = gaussian()
)
ysim <- des$sim_data() # simulate some data from the model
fit1 <- des$MCML(y = ysim) # Default model fitting with SAEM-PR
# use MCEM instead and stop when parameter values are within 1e-2 on successive iterations
fit2 <- des$MCML(y = ysim, method="mcem",tol=1e-2,conv.criterion = 1)

# Non-linear model fitting example using the example provided by nlmer in lme4
data(Orange, package = "lme4")

# the lme4 example:
startvec <- c(Asym = 200, xmid = 725, scal = 350)
(nm1 <- lme4::nlmer(circumference ~ SSlogis(age, Asym, xmid, scal) ~ Asym|Tree,
              Orange, start = startvec))

Orange <- as.data.frame(Orange)
Orange$Tree <- as.numeric(Orange$Tree)

# Here we can specify the model as a function.

model <- Model$new(
  circumference ~ Asym/(1 + exp((xmid - (age))/scal)) - 1 + (Asym|gr(Tree)),
  data = Orange,
  family = gaussian(),
  mean = c(200,725,350),
  covariance = c(500),
```

```
  var_par = 50
)

# for this example, we will use MCEM with adaptive MCMC sample sizes

model$mcmc_options$samps <- 1000
nfit <- model$MCML(method = "mcem.adapt")

summary(nfit)
summary(nm1)



## End(Not run)

## ------------------------------------------------
## Method `Model$LA`
## ------------------------------------------------


#create example data with six clusters, five time periods, and five people per cluster-period
df <- nelder(~(cl(6)*t(5)) > ind(5))
# parallel trial design intervention indicator
df$int <- 0
df[df$cl > 3, 'int'] <- 1
# specify parameter values in the call for the data simulation below
des <- Model$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)*ar0(t)),
  covariance = c(0.05,0.7),
  mean = c(rep(0,5),-0.2),
  data = df,
  family = stats::binomial()
)
ysim <- des$sim_data() # simulate some data from the model
fit1 <- des$LA(y = ysim)
```

`nelder` *Generates a block experimental structure using Nelder's formula*

Description

Generates a data frame expressing a block experimental structure using Nelder's formula

Usage

```
nelder(formula)
```

Arguments

`formula`	A model formula. See details

Details

Nelder (1965) suggested a simple notation that could express a large variety of different blocked designs. The function ‘nelder()‘ that generates a data frame of a design using the notation. There are two operations:

‘>‘ (or $\rightarrow$ in Nelder’s notation) indicates "clustered in".

‘*‘ (or $\times$ in Nelder’s notation) indicates a crossing that generates all combinations of two factors.

The implementation of this notation includes a string indicating the name of the variable and a number for the number of levels, such as ‘abc(12)‘. So for example ‘~cl(4) > ind(5)‘ means in each of five levels of ‘cl‘ there are five levels of ‘ind‘, and the individuals are different between clusters. The formula ‘~cl(4) * t(3)‘ indicates that each of the four levels of ‘cl‘ are observed for each of the three levels of ‘t‘. Brackets are used to indicate the order of evaluation. Some specific examples:

‘~person(5) * time(10)‘: A cohort study with five people, all observed in each of ten periods ‘time‘

‘~(cl(4) * t(3)) > ind(5)‘: A repeated-measures cluster study with four clusters (labelled ‘cl‘), each observed in each time period ‘t‘ with cross-sectional sampling and five indviduals (labelled ‘ind‘) in each cluster-period.

‘~(cl(4) > ind(5)) * t(3)‘: A repeated-measures cluster cohort study with four clusters (labelled ‘cl‘) wth five individuals per cluster, and each cluster-individual combination is observed in each time period ‘t‘.

‘~((x(100) * y(100)) > hh(4)) * t(2)‘: A spatio-temporal grid of 100x100 and two time points, with 4 households per spatial grid cell.

Value

A list with the first member being the data frame

Examples

```
nelder(~(j(4) * t(5)) > i(5))
nelder(~person(5) * time(10))
```

`nest_df` *Generate nested block structure*

Description

Generate a data frame that nests one data frame in another

Usage

```
nest_df(df1, df2)
```

Arguments

`df1`	data frame
`df2`	data frame

Details

For two data frames ‘df1‘ and ‘df2‘, the function will return another data frame that nests ‘df2‘ in ‘df1‘. So each row of ‘df1‘ will be duplicated ‘nrow(df2)‘ times and matched with ‘df2‘. The values of each ‘df2‘ will be unique for each row of ‘df1‘

Value

data frame

Examples

```
nest_df(data.frame(t=1:4),data.frame(cl=1:3))
```

---

predict.mcml *Predict from a ‘mcml‘ object*

Description

Predictions cannot be generated directly from an ‘mcml‘ object, rather new predictions should be generated using the original ‘Model‘. A message is printed to the user.

Usage

```
## S3 method for class 'mcml'
predict(object, ...)
```

Arguments

object	A ‘mcml‘ object.
...	Further arguments passed from other methods

Value

Nothing. Called for effects.

`predict.Model` *Generate predictions at new values from a 'Model' object*

Description

Generates predicted values from a 'Model' object using a new data set to specify covariance values and values for the variables that define the covariance function. The function will return a list with the linear predictor, conditional distribution of the new random effects term conditional on the current estimates of the random effects, and some simulated values of the random effects if requested. Typically this functionality is accessed using 'Model$predict()', which this function provides a wrapper for.

Usage

```
## S3 method for class 'Model'
predict(object, newdata, offset = rep(0, nrow(newdata)), m = 0, ...)
```

Arguments

`object`	A 'Model' object.
`newdata`	A data frame specifying the new data at which to generate predictions
`offset`	Optional vector of offset values for the new data
`m`	Number of samples of the random effects to draw
`...`	Further arguments passed from other methods

Value

A list with the linear predictor, parameters (mean and covariance matrices) for the conditional distribution of the random effects, and any random effect samples.

`print.mcml` *Prints an mcml fit output*

Description

Print method for class "'mcml'"

Usage

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

Arguments

`x`	an object of class "‘mcml‘" as a result of a call to MCML, see Model
`...`	Further arguments passed from other methods

Details

‘print.mcml‘ tries to replicate the output of other regression functions, such as ‘lm‘ and ‘lmer‘ reporting parameters, standard errors, and z- and p- statistics. The z- and p- statistics should be interpreted cautiously however, as generalised linear miobjected models can suffer from severe small sample biases where the effective sample size relates more to the higher levels of clustering than individual observations.

Parameters ‘b‘ are the mean function beta parameters, parameters ‘cov‘ are the covariance function parameters in the same order as ‘$covariance$parameters‘, and parameters ‘d‘ are the estimated random effects.

Value

No return value, called for side effects.

`progress_bar` *Generates a progress bar*

Description

Prints a progress bar

Usage

```
progress_bar(i, n, len = 30)
```

Arguments

`i`	integer. The current iteration.
`n`	integer. The total number of interations
`len`	integer. Length of the progress a number of characters

Value

A character string

Examples

```
progress_bar(10,100)
```

`Quantile` *Family declaration to support quantile regression*

Description

Skeleton list to declare a quantile regression model in a ‘Model‘ object.

Usage

```
Quantile(link = "identity", scaled = FALSE, q = 0.5)
```

Arguments

`link`	Name of the link function - any of ‘identity‘, ‘log‘, ‘logit‘, ‘inverse‘, or ‘probit‘
`scaled`	Logical indicating whether to include a scale parameter. If FALSE then the scale parameter is one.
`q`	Scalar in [0,1] declaring the quantile of interest.

Value

A list with two elements naming the family and link function

`random.effects` *Extracts the random effect estimates*

Description

Extracts the random effect estimates or samples from an mcml object returned from call of ‘MCML‘ or ‘LA‘ in the Model class.

Usage

```
random.effects(object)
```

Arguments

`object`	An ‘mcml‘ model fit.

Value

A matrix of dimension (number of fixed effects) x (number of MCMC samples). For Laplace approximation, the number of "samples" equals one.

residuals.mcml — *Residuals method for a ‘mcml‘ object*

Description

Calling residuals on an ‘mcml‘ object directly is not recommended. This function will currently only generate marginal residuals. It will generate a new ‘Model‘ object internally, thus copying all the data, which is not ideal for larger models. The preferred method is to call residuals on either the ‘Model‘ object or using ‘Model$residuals()‘, both of which will also generate conditional residuals.

Usage

```
## S3 method for class 'mcml'
residuals(object, type, ...)
```

Arguments

`object`	A ‘mcml‘ object.
`type`	Either "standardized", "raw" or "pearson"
`...`	Further arguments passed from other methods

Value

A matrix with either one column is conditional is false, or with number of columns corresponding to the number of MCMC samples.

residuals.Model — *Extract residuals from a ‘Model‘ object*

Description

Return the residuals from a ‘Model‘ object. This function is a wrapper for ‘Model$residuals()‘. Generates one of several types of residual for the model. If conditional = TRUE then the residuals include the random effects, otherwise only the fixed effects are included. For type, there are raw, pearson, and standardized residuals. For conditional residuals a matrix is returned with each column corresponding to a sample of the random effects.

Usage

```
## S3 method for class 'Model'
residuals(object, type, conditional, ...)
```

Arguments

`object`	A 'Model' object.
`type`	Either "standardized", "raw" or "pearson"
`conditional`	Logical indicating whether to condition on the random effects (TRUE) or not (FALSE)
`...`	Further arguments passed from other methods

Value

A matrix with either one column is conditional is false, or with number of columns corresponding to the number of MCMC samples.

`Salamanders` *Salamanders data*

Description

Obtained from `uu <- url("http://www.math.mcmaster.ca/bolker/R/misc/salamander.txt") sdat <- read.table(uu,header=TRUE,colClasses=c(rep("factor",5),"numeric"))` See https://rpubs.com/bbolker/salamander for more information.

`setParallel` *Disable or enable parallelised computing*

Description

By default, the package will use multithreading for many calculations if OpenMP is available on the system. For multi-user systems this may not be desired, so parallel execution can be disabled with this function.

Usage

```
setParallel(parallel_, cores_ = 2L)
```

Arguments

`parallel_`	Logical indicating whether to use parallel computation (TRUE) or disable it (FALSE)
`cores_`	Number of cores for parallel execution

Value

None, called for effects

SimGeospat — *Simulated data from a geospatial study with continuous outcomes*

Description

Simulated data from a geospatial study with continuous outcomes

Examples

```
#Data were generated with the following code:
n <- 600
SimGeospat <- data.frame(x = runif(n,-1,1), y = runif(n,-1,1))

sim_model <- Model$new(
  formula = ~ (1|fexp(x,y)),
  data = SimGeospat,
  covariance = c(0.25,0.8),
  mean = c(0),
  family = gaussian()
)

SimGeospat$y <- sim_model$sim_data()
```

SimTrial — *Simulated data from a stepped-wedge cluster trial*

Description

Simulated data from a stepped-wedge cluster trial

Examples

```
#Data were generated with the following code:
SimTrial <- nelder(~ (cl(10)*t(7))>i(10))
SimTrial$int <- 0
SimTrial[SimTrial$t > SimTrial$cl,'int'] <- 1

model <- Model$new(
  formula = ~ int + factor(t) - 1 + (1|gr(cl)*ar1(t)),
  covariance = c(0.05,0.8),
  mean = rep(0,8),
  data = SimTrial,
  family = gaussian()
)

SimTrial$y <- model$sim_data()
```

summary.mcml — *Summarises an mcml fit output*

Description

Summary method for class "‘mcml‘"

Usage

```
## S3 method for class 'mcml'
summary(object, ...)
```

Arguments

`object`	an object of class "‘mcml‘" as a result of a call to MCML, see Model
`...`	Further arguments passed from other methods

Details

‘print.mcml‘ tries to replicate the output of other regression functions, such as ‘lm‘ and ‘lmer‘ reporting parameters, standard errors, and z- and p- statistics. The z- and p- statistics should be interpreted cautiously however, as generalised linear miobjected models can suffer from severe small sample biases where the effective sample size relates more to the higher levels of clustering than individual observations.

Parameters ‘b‘ are the mean function beta parameters, parameters ‘cov‘ are the covariance function parameters in the same order as ‘$covariance$parameters‘, and parameters ‘d‘ are the estimated random effects.

Value

A list with random effect names and a data frame with random effect mean and credible intervals

summary.Model — *Summarizes a ‘Model‘ object*

Description

Summarizes ‘Model‘ object.

Usage

```
## S3 method for class 'Model'
summary(object, max_n = 10, ...)
```

Arguments

`object`	An ‘Model‘ object.
`max_n`	Integer. The maximum number of rows to print.
`...`	Further arguments passed from other methods

Value

An object of class ‘logLik‘. If both ‘fixed‘ and ‘covariance‘ are FALSE then it returns NA.

`vcov.mcml` *Extract the Variance-Covariance matrix for a ‘mcml‘ object*

Description

Returns the calculated variance-covariance matrix for a ‘mcml‘ object. The generating Model object has several methods to calculate the variance-convaariance matrix. For the standard GLS information matrix see ‘Model$information_matrix()‘. Small sample corrections can be accessed directly from the generating Model using ‘Model$small_sample_correction()‘. The varaince-covariance matrix including the random effects can be accessed using ‘Model$information_matrix(include.re = TRUE)‘.

Usage

```
## S3 method for class 'mcml'
vcov(object, ...)
```

Arguments

`object`	A ‘mcml‘ object.
`...`	Further arguments passed from other methods

Value

A variance-covariance matrix.

`vcov.Model` *Calculate Variance-Covariance matrix for a ‘Model‘ object*

Description

Returns the variance-covariance matrix for a ‘Model‘ object. Specifically, this function will return the inverse GLS information matrix for the fixed effect parameters. Small sample corrections can be accessed directly from the Model using ‘Model$small_sample_correction()‘. The varaince-covariance matrix including the random effects can be accessed using ‘Model$information_matrix(include.re = TRUE)‘.

Usage

```
## S3 method for class 'Model'
vcov(object, ...)
```

Arguments

`object`	A ‘Model‘ object.
`...`	Further arguments passed from other methods

Value

A variance-covariance matrix.

Index