Package 'glmmrBase'

September 14, 2023

Type Package

Title Generalised Linear Mixed Models in R

Version 0.4.6

Date 2023-09-11

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, digest, Rcpp (>= 1.0.7), R6

LinkingTo Rcpp (>= 0.12.0), RcppEigen, SparseChol (>= 0.2.1), BH, RcppParallel (>= 5.0.1), rminqa (>= 0.2.2)

RoxygenNote 7.2.3

NeedsCompilation yes

Author Sam Watson [aut, cre]

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

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

Suggests testthat (>= 3.1.0)

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 2023-09-14 10:00:02 UTC

R topics documented:

`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.

Details

The DESCRIPTION file:

Package:	glmmrBase
Type:	Package
Title:	Generalised Linear Mixed Models in R
Version:	0.4.6
Date:	2023-09-11
Authors@R:	c(person("Sam", "Watson", email = "S.I.Watson@bham.ac.uk", role = c("aut", "cre")))
Description:	Specification, analysis, simulation, and fitting of generalised linear mixed models. Includes
License:	GPL (>= 2)
Imports:	methods, digest, Rcpp (>= 1.0.7), R6

LinkingTo:	Rcpp (>= 0.12.0), RcppEigen, SparseChol (>= 0.2.1), BH, RcppParallel (>= 5.0.1), rminqa
RoxygenNote:	7.2.3
NeedsCompilation:	yes
Author:	Sam Watson [aut, cre]
URL:	https://github.com/samuel-watson/glmmrBase
BugReports:	https://github.com/samuel-watson/glmmrBase/issues
Suggests:	testthat (>= 3.1.0)
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>
ExperimentalWindowsRuntime:	ucrt
Archs:	x64

Index of help topics:

https://github.com/samuel-watson/glmmrBase/blob/master/README.md

Author(s)

Sam Watson [aut, cre]

Maintainer: NA

`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

`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$parameters <- c(...)'. The function calls check functions to automatically update linked matrices with the new parameters. If using direct assignment, call 'self$check()' afterwards.

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$check()`
- `Covariance$update_parameters()`
- `Covariance$print()`
- `Covariance$subset()`
- `Covariance$get_chol_D()`
- `Covariance$log_likelihood()`
- `Covariance$simulate_re()`
- `Covariance$sparse()`
- `Covariance$parameter_table()`
- `Covariance$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, verbose = TRUE)
```

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.

`verbose` Logical whether to provide detailed output.

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 `check()`**:** Check if anything has changed and update matrices if so.

Usage:

```
Covariance$check(verbose = TRUE)
```

Arguments:

`verbose` Logical whether to report if any changes detected.

Returns: NULL

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
df <- nelder(~(cl(5)*t(5)) > ind(5))
cov <- Covariance$new(formula = ~(1|gr(cl)*ar0(t)),
                      parameters = c(0.03,0.8),
                      data= df)
cov$parameters <- c(0.25,0.1)
cov$check(verbose=FALSE)
```

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

Usage:

```
Covariance$get_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)
```

Arguments:

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

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 `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$check`
## ------------------------------------------------
```

```
df <- nelder(~(cl(5)*t(5)) > ind(5))
cov <- Covariance$new(formula = ~(1|gr(cl)*ar0(t)),
                      parameters = c(0.03,0.8),
                      data= df)
cov$parameters <- c(0.25,0.1)
cov$check(verbose=FALSE)

## ------------------------------------------------
## 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

`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'))
```

---

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)
```

Arguments

family	family object

Value

list with filename and type

MeanFunction *R6 Class representing a mean function/linear predictor*

Description

R6 Class representing a mean function/linear predictor

R6 Class representing a mean function/linear predictor

Details

For the generalised linear mixed model

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

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$parameters <- c(...)'. The function calls check functions to automatically update linked matrices with the new parameters. If using direct assignment, call 'self$check()' afterwards.

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$n()`
- `MeanFunction$check()`
- `MeanFunction$new()`
- `MeanFunction$print()`
- `MeanFunction$update_parameters()`
- `MeanFunction$colnames()`
- `MeanFunction$subset_rows()`
- `MeanFunction$subset_cols()`
- `MeanFunction$linear_predictor()`

- 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 `check()`**:** Checks if any changes have been made and updates

Checks if any changes have been made and updates, usually called automatically.

Usage:

```
MeanFunction$check(verbose = TRUE)
```

Arguments:

`verbose` Logical whether to report if any changes detected.

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$new(formula = ~ int ,
                        data=df,
                        parameters = c(-1,1)
                        )
mf1$parameters <- c(0,0)
mf1$check()
```

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 attemps 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 `subset_cols()`**:** Keeps a subset of the columns of X

All indices not in the provided vector of column numbers will be removed from the fixed effects design matrix X.

Usage:

```
MeanFunction$subset_cols(index)
```

Arguments:

index Columns of X 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_cols(1:2)
```

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

Returns the linear predictor, X * beta

Usage:

```
MeanFunction$linear_predictor()
```

Returns: A vector

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$check`
## ------------------------------------------------
```

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

## ------------------------------------------------
## 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)
```

```
## ------------------------------------------------
## Method `MeanFunction$subset_cols`
## ------------------------------------------------


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_cols(1:2)
```

`Model` *A GLMM Model*

Description

A GLMM Model

A GLMM Model

Details

A generalised linear mixed model and a range of associated functions

A detailed vingette for this package is available online`doi:10.48550/arXiv.2303.12657`. Briefly, for the generalised linear mixed model

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

where h is the link function. The class provides access to all of the matrices above and associated calculations and functions including model fitting, power analysis, and various relevant decompositions. The object is an R6 class and so can serve as a parent class for extended functionality.

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.

The class also includes model fitting with Markov Chain Monte Carlo Maximum Likelihood implementing the algorithms described by McCulloch (1997), and fast model fitting using Laplace approximation. Functions for returning related values such as the log gradient, log probability, and other matrices are also available.

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. If using direct assignment, call `self$check()` afterwards.

MCMCML Fits generalised linear mixed models using one of three algorithms: Markov Chain Newton Raphson (MCNR), Markov Chain Expectation Maximisation (MCEM), or Maximum simulated likelihood (MSL). All the algorithms are described by McCullagh (1997). For each iteration of the algorithm the unobserved random effect terms (γ) are simulated using Markov Chain Monte Carlo (MCMC) methods (we use Hamiltonian Monte Carlo through Stan), and then these values are conditioned on in the subsequent steps to estimate the covariance parameters and the mean function parameters (β). For all the algorithms, the covariance parameter estimates are updated using an expectation maximisation step. For the mean function parameters you can either use a Newton Raphson step (MCNR) or an expectation maximisation step (MCEM). A simulated likelihood step can be added at the end of either MCNR or MCEM, which uses an importance sampling technique to refine the parameter estimates.

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.

Options for the MCMC sampler are set by changing the values in `self$mcmc_options`.

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.

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 algorithm. 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.
- `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$predict()`
- `Model$new()`
- `Model$print()`
- `Model$n()`
- `Model$subset_rows()`
- `Model$subset_cols()`
- `Model$sim_data()`
- `Model$check()`
- `Model$update_parameters()`
- `Model$information_matrix()`
- `Model$sandwich()`
- `Model$kenward_roger()`
- `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$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 `sim_data()`. To predict the values at a new location see `predict()`.

Usage:

```
Model$fitted(type = "link", X, u)
```

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

Returns: A Matrix class object containing the predicted values

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

Usage:

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

Arguments:

`formula` An optional model formula containing fixed and random effect terms. If not specified, then seaparate formulae need to be provided to the covariance and mean arguments below.

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

`mean` Either a MeanFunction object, or an equivalent list of arguments that can be passed to `MeanFunction` to create a new object. 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. This argument is optional if the family is provided either via a `MeanFunction` or `MeanFunction` objects, or as members of the list of arguments to `mean`. Current accepts binomial, gaussian, Gamma, poisson, and Beta.

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

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

`weights` A vector of weights. Optional..

`trials` 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).

`verbose` Logical indicating whether to provide detailed output

Returns: A new Model class object

Examples:

```
\dontshow{
setParallel(FALSE) # for the CRAN check
}
```

```
#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()
)

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

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

# or as
des <- Model$new(
  formula = ~ int + (1|gr(ind)),
  covariance = list(parameters = c(0.05)),
  mean = list(parameters = 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(covariance = list( formula = ~(1|ar0(t)*fexp(x,y))),
                        mean = list(formula = ~ 1),
                        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 `subset_cols()`**:** Subsets the columns of the design

Removes the specified columns from the linked mean function object's X matrix.

Usage:

```
Model$subset_cols(index)
```

Arguments:

`index` Integer or vector of integers specifying the indexes of the columns 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(
  covariance = list(
    formula = ~ (1|gr(cl)*ar0(t)),
```

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

Method `check()`**:** Checks for any changes in linked objects and updates.

Checks for any changes in any object and updates all linked objects if any are detected. Generally called automatically and not required by the user.

Usage:

```
Model$check(verbose = TRUE)
```

Arguments:

`verbose` Logical indicating whether to report if any updates are made, defaults to TRUE

Returns: Linked objects are updated by nothing is returned

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$new(
  covariance = list(
    formula = ~ (1|gr(cl)*ar0(t)),
    parameters = c(0.05,0.8)),
  mean = list(
    formula = ~ factor(t) + int - 1,
    parameters = c(rep(0,5),0.6)),
  data = df,
  family = stats::binomial()
)
des$check() #does nothing
des$covariance$parameters <- c(0.1,0.9)
des$check() #updates
des$mean$parameters <- c(rnorm(5),0.1)
des$check() #updates
```

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

`verbose` Logical indicating whether to provide more detailed feedback

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$new(
  covariance = list(
    formula = ~ (1|gr(cl)*ar0(t))),
  mean = list(
    formula = ~ factor(t) + int - 1),
  data = df,
  family = stats::binomial()
)
des$update_parameters(cov.pars = c(0.1,0.9))
```

Method `information_matrix()`**:** Generates the information matrix of the GLS estimator

Usage:

```
Model$information_matrix(include.re = FALSE)
```

Arguments:

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

Returns: A PxP matrix

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

Usage:

```
Model$sandwich()
```

Returns: A PxP matrix

Method `kenward_roger()`**:** Returns the bias-corrected variance-covariance matrix for the fixed effect parameters.

Usage:

```
Model$kenward_roger()
```

Returns: A PxP matrix

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$new(
  covariance = list(
    formula = ~ (1|gr(cl)) + (1|gr(cl,t)),
    parameters = c(0.05,0.1)),
  mean = list(
    formula = ~ factor(t) + int - 1,
    parameters = c(rep(0,5),0.6)),
  data = df,
  family = stats::gaussian(),
  var_par = 1
)
des$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()`**:** Markov Chain Monte Carlo Maximum Likelihood model fitting

Usage:

```
Model$MCML(
  y,
  method = "mcnr",
  sim.lik.step = FALSE,
  verbose = TRUE,
  tol = 0.01,
  max.iter = 30,
  se = "gls",
  sparse = FALSE,
  usestan = TRUE,
  se.theta = TRUE
)
```

Arguments:

`y` A numeric vector of outcome data

`method` The MCML algorithm to use, either `mcem` or `mcnr`, see Details. Default is `mcem`.

`sim.lik.step` Logical. Either TRUE (conduct a simulated likelihood step at the end of the algorithm), or FALSE (does not do this step), defaults to FALSE.

`verbose` Logical indicating whether to provide detailed output, defaults to TRUE.

`tol` Numeric value, tolerance of the MCML algorithm, the maximum difference in parameter estimates between iterations at which to stop the algorithm.

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

`se` String. Type of standard error to return. Options are "gls" for GLS standard errors (the default), "robust" for Huber robust sandwich estimator, "kr" for Kenward-Roger bias corrected standard errors, "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.

`sparse` Logical indicating whether to use sparse matrix methods

`usestan` Logical whether to use Stan (through the package `cmdstanr`) for the MCMC sampling. If FALSE then the internal Hamiltonian Monte Carlo sampler will be used instead. We recommend Stan over the internal sampler as it generally produces a larger number of effective samplers per unit time, especially for more complex covariance functions.

`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.

Returns: A `mcml` object

Examples:

```
\dontrun{
#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$new(
  formula= ~ factor(t) + int - 1 +(1|gr(cl)*ar0(t)),
  covariance = list(parameters = c(0.05,0.7)),
  mean = list(parameters = c(rep(0,5),0.2)),
  data = df,
  family = gaussian(),
  var_par = 1
)
ysim <- des$sim_data() # simulate some data from the model
fit1 <- des$MCML(y = ysim,method="mcnr",usestan=FALSE) # don't use Stan
#fits the models using Stan
fit2 <- des$MCML(y = ysim, method="mcnr")
 #adds a simulated likelihood step after the MCEM algorithm
fit3 <- des$MCML(y = ysim, sim.lik.step = TRUE)

 # we could use LA to find better starting values
fit4 <- des$LA(y=ysim)
# the fit parameter values are stored in the internal model class object
fit5 <- des$MCML(y = ysim, method="mcnr") # it should converge much more quickly
}
```

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

Usage:

```
Model$LA(
  y,
  start,
  method = "nr",
  verbose = FALSE,
  se = "gls",
  max.iter = 40,
  tol = 1e-04,
  se.theta = TRUE
)
```

Arguments:

`y` A numeric vector of outcome data

`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

`verbose` logical indicating whether to provide detailed algorithm feedback (default is TRUE).

`se` String. Type of standard error to return. Options are "gls" for GLS standard errors (the default), "robust" for Huber robust sandwich estimator, "kr" for Kenward-Roger bias corrected standard errors, "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.

`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.

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$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)*ar0(t)),
  covariance = list( parameters = c(0.05,0.7)),
  mean = list(parameters = c(rep(0,5),-0.2)),
  data = df,
  family = stats::binomial()
)
ysim <- des$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)
```

Arguments:

`sparse` Logical indicating whether to use sparse matrix methods

Returns: None, called for effects

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

Usage:

```
Model$mcmc_sample(y, usestan = TRUE, verbose = TRUE)
```

Arguments:

`y` Numeric vector of outcome data

`usestan` Logical whether to use Stan (through the package `cmdstanr`) for the MCMC sampling. If FALSE then the internal Hamiltonian Monte Carlo sampler will be used instead. We recommend Stan over the internal sampler as it generally produces a larger number of effective samplers per unit time, especially for more complex covariance functions.

`verbose` Logical indicating whether to provide detailed output to the console

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` Vector of outcome data

`u` 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

Usage:

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

Arguments:

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

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 `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`
## ------------------------------------------------


#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()
)

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

des <- Model$new(
  covariance = list(
    formula = ~ (1|gr(ind)),
```

```
    parameters = c(0.05)),
  mean = list(
    formula = ~ int,
    parameters = c(1,0.5)),
  data = df,
  family = stats::poisson())

# or as
des <- Model$new(
  formula = ~ int + (1|gr(ind)),
  covariance = list(parameters = c(0.05)),
  mean = list(parameters = 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(covariance = list( formula = ~(1|ar0(t)*fexp(x,y))),
                        mean = list(formula = ~ 1),
                        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$new(
  covariance = list(
    formula = ~ (1|gr(cl)*ar0(t)),
    parameters = c(0.05,0.8)),
  mean = list(
    formula = ~ factor(t) + int - 1,
    parameters = c(rep(0,5),0.6)),
  data = df,
  family = stats::binomial()
)
ysim <- des$sim_data()

## ------------------------------------------------
## Method `Model$check`
## ------------------------------------------------


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

```
  covariance = list(
    formula = ~ (1|gr(cl)*ar0(t)),
    parameters = c(0.05,0.8)),
  mean = list(
    formula = ~ factor(t) + int - 1,
    parameters = c(rep(0,5),0.6)),
  data = df,
  family = stats::binomial()
)
des$check() #does nothing
des$covariance$parameters <- c(0.1,0.9)
des$check() #updates
des$mean$parameters <- c(rnorm(5),0.1)
des$check() #updates

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


df <- nelder(~(cl(10)*t(5)) > ind(10))
df$int <- 0
df[df$cl > 5, 'int'] <- 1
des <- Model$new(
  covariance = list(
    formula = ~ (1|gr(cl)*ar0(t))),
  mean = list(
    formula = ~ factor(t) + int - 1),
  data = df,
  family = stats::binomial()
)
des$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$new(
  covariance = list(
    formula = ~ (1|gr(cl)) + (1|gr(cl,t)),
    parameters = c(0.05,0.1)),
  mean = list(
    formula = ~ factor(t) + int - 1,
    parameters = c(rep(0,5),0.6)),
  data = df,
  family = stats::gaussian(),
  var_par = 1
)
des$power() #power of 0.90 for the int parameter
```

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

## Not run:
#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$new(
  formula= ~ factor(t) + int - 1 +(1|gr(cl)*ar0(t)),
  covariance = list(parameters = c(0.05,0.7)),
  mean = list(parameters = c(rep(0,5),0.2)),
  data = df,
  family = gaussian(),
  var_par = 1
)
ysim <- des$sim_data() # simulate some data from the model
fit1 <- des$MCML(y = ysim,method="mcnr",usestan=FALSE) # don't use Stan
#fits the models using Stan
fit2 <- des$MCML(y = ysim, method="mcnr")
 #adds a simulated likelihood step after the MCEM algorithm
fit3 <- des$MCML(y = ysim, sim.lik.step = TRUE)

 # we could use LA to find better starting values
fit4 <- des$LA(y=ysim)
# the fit parameter values are stored in the internal model class object
fit5 <- des$MCML(y = ysim, method="mcnr") # it should converge much more quickly

## 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$new(
  formula = ~ factor(t) + int - 1 + (1|gr(cl)*ar0(t)),
  covariance = list( parameters = c(0.05,0.7)),
  mean = list(parameters = c(rep(0,5),-0.2)),
  data = df,
  family = stats::binomial()
)
ysim <- des$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))
```

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)
```

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

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. TBC!!

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

---

`yexample312a` *Data for first example in Section 3.12 of JSS paper*

Description

Data for first example in Section 3.12 of JSS paper

`yexample312b` *Data for second example in Section 3.12 of JSS paper*

Description

Data for second example in Section 3.12 of JSS paper

`yexample312c` *Data for third example in Section 3.12 of JSS paper*

Description

Data for third example in Section 3.12 of JSS paper

`ytest1` *Data for model tests*

Description

Data for model tests

Index