Package 'spaMM'

October 9, 2026

Type Package

Title Mixed-Effect Models, with or without Spatial Random Effects

Encoding UTF-8

Version 4.7.17

Date 2026-10-09

Maintainer François Rousset <francois.rousset@umontpellier.fr>

Imports methods, stats, graphics, Matrix (>= 1.5.0), proxy, Rcpp (>= 0.12.10), nlme, nloptr, minqa, pbapply, cli, gmp (>= 0.6.0), ROI, boot, geometry (>= 0.4.0), numDeriv, backports, reformulas, matrixStats

LinkingTo Rcpp, RcppEigen (>= 0.3.3.5.0)

Depends R (>= 3.5.0)

Suggests MASS, maps, testthat, rcdd, foreach, future, future.apply, RANN, Infusion (>= 1.3.0), IsoriX (>= 0.8.1), blackbox (>= 1.1.25), RSpectra, ROI.plugin.glpk, lme4, rsae, multilevel, agridat, fmesher, knitr, rmarkdown, tweedie, tweedieDistr

Enhances multcomp, RLRsim, lmerTest, emmeans

NeedsCompilation yes

SystemRequirements GNU Scientific Library (GSL)

Description

Inference based on models with or without spatially-correlated random effects, multivariate responses, or non-Gaussian random effects (e.g., Beta). Variation in residual variance (heteroscedasticity) can itself be represented by a mixed-effect model. Both classical geostatistical models (Rousset and Ferdy 2014 <doi:10.1111/ecog.00566>), and Markov random field models on irregular grids (as considered in the 'INLA' package, <https://www.r-inla.org>), can be fitted, with distinct computational procedures exploiting the sparse matrix representations for the latter case and other autoregressive models. Laplace approximations are used for likelihood or restricted likelihood. Penalized quasi-likelihood and other variants discussed in the h-likelihood literature (Lee and Nelder 2001 <doi:10.1093/biomet/88.4.987>) are also implemented.

License CeCILL-2

URL https://gitlab.in2p3.fr/francois.rousset/spamm-ref

RoxygenNote 7.1.2

ByteCompile true

VignetteBuilder knitr

Author François Rousset [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0003-4670-0371>),
Jean-Baptiste Ferdy [aut, cph],
Alexandre Courtiol [aut] (ORCID:
<https://orcid.org/0000-0003-0637-2959>)

Repository CRAN

Date/Publication 2026-10-09 02:20:03 UTC

Contents

adjlg . . . 5
AIC . . . 6
algebra . . . 10
aliases . . . 12
anova . . . 13
arabidopsis . . . 16
ARp . . . 17
as_LMLT . . . 19
autoregressive . . . 20
betabin . . . 22
beta_resp . . . 23
blackcap . . . 25
CauchyCorr . . . 26
clinics . . . 27
COMPoisson . . . 28
composite-ranef . . . 30
confint.HLfit . . . 34
control.HLfit . . . 36
convergence . . . 37
corMatern . . . 38
corrFamily . . . 40
corrFamily-definition . . . 45
corrFamily-design . . . 46
corrHLfit . . . 49
corrMatrix . . . 52
corr_family . . . 55
covStruct . . . 56
diallel . . . 57
div_info . . . 60
DoF . . . 61
dofuture . . . 62
dopar . . . 64
drop1.HLfit . . . 67

dyad . . . 69
eval_replicate . . . 71
extractors . . . 72
extreme_eig . . . 75
fitme . . . 76
fitmv . . . 79
fixed . . . 83
fixedLRT . . . 85
fix_predVar . . . 88
freight . . . 89
get_cPredVar . . . 90
get_inits_from_fit . . . 92
get_matrix . . . 93
get_ranPars . . . 95
get_RLRsim_args . . . 97
gof . . . 99
good-practice . . . 100
Gryphon . . . 102
hatvalues.HLfit . . . 104
HLCor . . . 106
HLfit . . . 109
how . . . 113
inits . . . 114
inner-vs-outer . . . 115
inverse.Gamma . . . 117
is_separated . . . 118
Leuca . . . 119
lev2bool . . . 120
llm.fit . . . 121
Loaloa . . . 123
LRT . . . 125
make_scaled_dist . . . 129
mapMM . . . 131
MaternCorr . . . 136
MaternIMRFa . . . 138
mat_sqrt . . . 140
method . . . 141
MSFDR . . . 143
multIMRF . . . 144
multinomial . . . 148
mv . . . 151
negbin . . . 153
negbin1 . . . 154
numInfo . . . 156
optimBounds . . . 158
options . . . 160
pedigree . . . 162
phi-resid.model . . . 164

plot.HLfit . . . 166
plot_effects . . . 168
pois4mlogit . . . 172
Poisson . . . 176
post-fit . . . 178
predict . . . 179
predVar . . . 184
pseudoR2 . . . 187
random-effects . . . 189
rankinfo . . . 191
register_cF . . . 192
remove_fixef . . . 193
resid.model . . . 194
residuals.HLfit . . . 196
residVar . . . 198
salamander . . . 200
scotlip . . . 202
seaMask . . . 203
seeds . . . 204
setNbThreads . . . 205
simulate.HLfit . . . 206
spaMM . . . 210
spaMM-conventions . . . 214
spaMM.colors . . . 215
spaMM.filled.contour . . . 216
spaMM_boot . . . 218
spaMM_glm.fit . . . 222
stripHLfit . . . 225
summary.HLfit . . . 226
transffit . . . 227
tweedie . . . 229
update.HLfit . . . 232
vcov . . . 234
verbose . . . 237
wafers . . . 238
welding . . . 239
WinterWheat . . . 240
wrap_parallel . . . 241
X.GCA . . . 241
X2X . . . 243
ZAXlist . . . 246

Index **248**

adjlg *Simulated data set for testing sparse-precision code*

Description

This is used in tests/test-adjacency-long.R

Usage

```
data("adjlg")
```

Format

Includes an adjacency matrix adjlgMat. and a data frame adjlg with 5474 observations on the following 8 variables.

ID a factor with levels 1 to 1000

months a numeric vector

GENDER a character vector

AGE a numeric vector

X1 a numeric vector

X2 a numeric vector

month a numeric vector

BUY a numeric vector

Source

The simulation code shown below is derived from an example produced by Jeroen van den Ochtend. Following a change incorporated in spaMM version 3.8.0, that implied stricter checks of the input matrix, it appeared that the precision matrix generated by this example had inappropriate (repeated) dimnames. This example was then updated to reproduce past fitting results with a correctly formatted matrix. Note that changing the names of an adjacency matrix (as below) is generally unwise as it generally changes the statistical model because these names are matched whenever possible to levels of the grouping factor in the data.

The code was also modified to compensate for changes in R's default random number generator.

Examples

```
data(adjlg)
## See further usage in tests/test-adjacency-long.R
## Not run:
# as produced by:
  library(data.table) ## Included data produced using version 1.10.4.3
  library(igraph) ## Included data produced using version 1.2.1

  rsample <- function(N=100, ## size of implied adjacency matrix
```

```
                    month_max=10,seed) {
    if (is.integer(seed)) set.seed(seed)
    dt <- data.table(ID=factor(1:N))
    dt$months <- sample(1:month_max,N,replace=T) ## # of liens for each level of ID
    dt$GENDER <- sample(c("MALE","FEMALE"),N,replace=TRUE)
    dt$AGE <- sample(18:99,N,replace=T)
    dt$X1 <- sample(1000:9900,N,replace=T)
    dt$X2 <-  runif(N)

    dt <- dt[, c(.SD, month=data.table(seq(from=1, to=months, by = 1))), by = ID]
    dt[,BUY := 0]
    dt[month.V1==months,BUY := sample(c(0,1),1),by=ID]
    setnames(dt,"month.V1","month")

    #### create adjacency matrix
    Network <- data.table(OUT=sample(dt$ID,N*month_max*4/10))
    Network$IN <- sample(dt$ID,N*month_max*4/10)
    Network <- Network[IN != OUT]
    Network <- unique(Network)
    g <- graph.data.frame(Network,directed=F)
    g <- add_vertices(g,sum(!unique(dt$ID) %in% V(g)),
             name=unique(dt[!dt$ID %in% V(g),list(ID)])) # => improper names
    Network <- as_adjacency_matrix(g,sparse = TRUE,type="both")
    colnames(Network) <- rownames(Network) <- seq(nrow(Network)) # post-v3.8.0 names
    return(list(data=dt,adjMatrix=Network))
  }

  RNGkind("Mersenne-Twister", "Inversion", "Rounding"  )
  set.seed(123)
  adjlg_sam <- rsample(N=1000,seed=NULL)
  RNGkind("Mersenne-Twister", "Inversion", "Rejection"  )
  #
  adjlg <- as.data.frame(adjlg_sam$data)
  adjlgMat <- adjlg_sam$adjMatrix

## End(Not run)
```

AIC *Extractors for information criteria such as AIC*

Description

get_any_IC computes model selection/information criteria such as AIC. See Details for more information about these criteria. The other extractors AIC and extractAIC are methods for HLfit objects of generic functions defined in other packages: AIC is equivalent to get_any_IC (for a single fitted-model object), and extractAIC returns the marginal AIC and the number of degrees of freedom for the fixed effects.

Usage

```
get_any_IC(object, nsim=0L, ..., verbose=interactive(),
           also_cAIC=TRUE, short.names=NULL)
## S3 method for class 'HLfit'
AIC(object, ..., nsim=0L, k, verbose=interactive(),
                    also_cAIC=TRUE, short.names=NULL)
## S3 method for class 'HLfit'
extractAIC(fit, scale=0, k=2L, ..., verbose=FALSE)
```

Arguments

`object`, `fit`	A object of class `HLfit`, as returned by the fitting functions in `spaMM`.
`scale`	numeric: if scale is 0, AIC is computed; if scale is positive, Mallows' C_p for linear models (neither generalized nor mixed) is computed using `scale` as given value of residual variance (as in `stats::extractAIC`. No check of the model structure is performed, so if the model is not a linear model, a value is also returned but it may not make sense.
`k`	Numeric specifying the weight of the degrees of freedom part in the AIC formula (as in `stats::extractAIC`).
`verbose`	Whether to print the model selection criteria or not.
`also_cAIC`	Whether to include the plug-in estimate of conditional AIC in the result (its computation may be slow).
`nsim`	Controls whether to include the bootstrap estimate of conditional AIC (see Details) in the result. If positive, `nsim` gives the number of bootstrap replicates.
`short.names`	NULL, or boolean; controls whether the return value uses short names (`mAIC`, etc., as shown by screen output if `verbose` is TRUE), or the descriptive names (`"marginal AIC:"`, etc.) also shown in the screen output. Short names are more appropriate for programming but descriptive names may be needed for back-compatibility. The default (NULL) ensures back-compatibility by using descriptive names unless the bootstrap estimate of conditional AIC is reported.
`...`	For `AIC.HLfit`: may include more fitted-model objects, consistently with the generic. For this and the other functions: other arguments that may be needed by some method. For example, if `nsim` is positive, a `seed` argument may be passed to `simulate`, and the other "`...`" may be used to control the optional parallel execution of the bootstrap computations (by providing arguments to `dopar`).

Details

The AIC is a measure (by Kullback-Leibler directed distance, up to an additive constant) of quality of prediction of new data by a fitted model. Comparing information criteria may be viewed as a fast alternative to a comparison of the predictive accuracy of different models by cross-validation. Further procedures for model choice may also be useful (e.g. Williams, 1970; Lewis et al. 2010).

The **conditional AIC** (Vaida and Blanchard 2005) applies the AIC concept to new realizations of a mixed model, conditional on the realized values of the random effects. Lee et al. (2006) and Ha et al (2007) defined a corrected AIC [i.e., AIC(D*) in their eq. 7] which is here interpreted as the conditional AIC.

Such Kullback-Leibler relative distances cannot generally be evaluated exactly and various estimates have been discussed. `get_any_IC` computes, optionally prints, and returns invisibly one or more of the following quantities:
* Akaike's classical AIC (**marginal AIC**, `mAIC`, i.e., minus twice the marginal log-likelihood plus twice the number of fitted parameters);
* a plug-in estimate (`cAIC`) and/or a bootstrap estimate (`b_cAIC`) of the conditional AIC;
* a focussed AIC for dispersion parameters (**dispersion AIC**, `dAIC`).

For the **conditional AIC**, Vaida and Blanchard's plug-in estimator involves the conditional likelihood, and degrees of freedom for (i) estimated residual error parameters and (ii) the overall linear predictor characterized by the **Effective degrees of freedom** already discussed by previous authors including Lee and Nelder (1996), which gave a plug-in estimator (p_D) for it in HGLMs. By default, the plug-in estimate of both the conditional AIC and of $n - p_D$ (`GoFdf`, where n is the length of the response vector) are returned by `get_any_IC`. But these are biased estimates of conditional AIC and effective df, and an alternative procedure is available for GLM response families if a non-default positive `nsim` value is used. In that case, the conditional AIC is estimated by a bootstrap version of Saefken et al. (2014)'s equation 2.5; this involves refitting the model to each bootstrap samples, so it may take time, and a full cross-validation procedure might as well be considered for model selection.

The dispersion AIC has been defined from restricted likelihood by Ha et al (2007; eq.10). The present implementation will use restricted likelihood only if made available by an REML fit, otherwise marginal likelihood is used.

Value

`get_any_IC`, a numeric vector whose possible elements are described in the Details, and whose names are controlled by the `short.names` argument. Note that the bootstrap computation actually makes sense and works also for fixed-effect models (although it is not clear how useful it is in that case). The return value will still refer to its results as conditional AIC.

For `AIC`, If just one fit object is provided, the same return value as for `get_any_IC`. If multiple objects are provided, a data.frame built from such vectors, with rows corresponding to the objects.

For `extractAIC`, a numeric vector of length 2, with first and second elements giving

`* edf`	the degree of freedom of the fixed-effect terms of the model for the fitted model `fit`.
`* AIC`	the (marginal) Akaike Information Criterion for `fit`.

Likelihood is broadly defined up to a constant, which opens the way for inconsistency between different likelihood and AIC computations. In **spaMM**, likelihood is nothing else than the probability or probability density of the data as function of model parameters. No constant is ever added, in contrast to `stats::extractAIC` output, so there are discrepancies with the latter function (see Examples).

References

Ha, I. D., Lee, Y. and MacKenzie, G. (2007) Model selection for multi-component frailty models. Statistics in Medicine 26: 4790-4807.

Lee Y. and Nelder. J. A. 1996. Hierarchical generalized linear models (with discussion). J. R. Statist. Soc. B, 58: 619-678.

Lewis, F., Butler, A. and Gilbert, L. (2011), A unified approach to model selection using the likelihood ratio test. Methods in Ecology and Evolution, 2: 155-162. doi:10.1111/j.2041210X.2010.00063.x

Saefken B., Kneib T., van Waveren C.-S., Greven S. (2014) A unifying approach to the estimation of the conditional Akaike information in generalized linear mixed models. Electron. J. Statist. 8, 201-225.

Vaida, F., and Blanchard, S. (2005) Conditional Akaike information for mixed-effects models. Biometrika 92, 351-370.

Williams D.A. (1970) Discrimination between regression models to determine the pattern of enzyme synthesis in synchronous cell cultures. Biometrics 26: 23-32.

See Also

DoF, the extractor for the number of fitted parameters of a model.

Examples

```
data("wafers")
m1 <- fitme(y ~ X1+X2+X3+X1*X3+X2*X3+I(X2^2)+(1|batch), data=wafers,
            family=Gamma(log))

get_any_IC(m1)
# => The plug-in estimate is stored in the 'm1' object
#    as a result of the previous computation, and is now returned even by:
get_any_IC(m1, also_cAIC=FALSE)

if (spaMM.getOption("example_maxtime")>4) {
 get_any_IC(m1, nsim=100L, seed=123) # provides bootstrap estimate of cAIC.
 # (parallelisation options could be used, e.g. nb_cores=detectCores(logical=FALSE)-1L)
}

extractAIC(m1)

## Not run:
# Checking (in)consistency with glm example from help("stats::extractAIC"):
utils::example(glm) # => provides 'glm.D93' fit object
logLik(glm.D93) # logL= -23.38066 (df=5)
dataf <- data.frame(counts=counts,outcome=outcome, treatment=treatment)
extractAIC(fitme(counts ~ outcome + treatment, family = poisson(), data=dataf))
# => 56.76132 = -2 logL + 2* df
extractAIC(glm.D93) # 56.76132 too
#
# But for LM:
lm.D93 <- lm(counts ~ outcome + treatment, data=dataf)
logLik(lm.D93) # logL=-22.78576 (df=6)
extractAIC(fitme(counts ~ outcome + treatment, data=dataf)) # 57.5715 = -2 logL + 2* df
extractAIC(lm.D93) # 30.03062

### Inconsistency also apparent in drop1 output for :
# Toy data from McCullagh & Nelder (1989, pp. 300-2), as in 'glm' doc:
clotting <- data.frame(
    u = c(5,10,15,20,30,40,60,80,100),
```

```
    lot1 = c(118,58,42,35,27,25,21,19,18),
    lot2 = c(69,35,26,21,18,16,13,12,12))
#
drop1( fitme(lot1 ~ log(u), data = clotting), test = "F") # agains reports marginal AIC
# => this  may differ strongly from those returned by drop1( < glm() fit > ),
# but the latter are not even consistent with those from drop1( < lm() fit > )
# for linear models. Compare
drop1( lm(lot1 ~ log(u), data = clotting), test = "F") # consistent with drop1.HLfit()
drop1( glm(lot1 ~ log(u), data = clotting), test = "F") # inconsistent

## Discrepancies in drop1 output with Gamma() family:

gglm <- glm(lot1 ~ 1, data = clotting, family=Gamma())
logLik(gglm) # -40.34633 (df=2)

spgglm <-  fitme(lot1 ~ 1, data = clotting, family=Gamma())
logLik(spgglm)                      # -40.33777 (slight difference:
#   see help("method") for difference in estimation method between glm() and fitme()).
# Yet this does not explain the following:

drop1(  fitme(lot1 ~ log(u), data = clotting, family=Gamma()), test = "F")
# => second AIC is 84.676 as expected from above logLik(spgglm).
drop1( glm(lot1 ~ log(u), data = clotting, family=Gamma()), test = "F")
# => second AIC is 1465.27, quite different from -2*logLik(gglm) + 2*df


## End(Not run)
```

`algebra` *Control of matrix-algebraic methods*

Description

Autocorrelated gaussian random effects can be specified in terms of their covariance matrix, or in terms of the precision matrix (i.e. inverse covariance matrix). In a pre-processing step, **spaMM** may assess whether such precision matrices are sparse but the correlation matrix is dense, and if so, it may use "sparse-precision" algorithms efficient for this case. If the precision matrix does not appear sufficiently sparser than the correlation matrix, correlation matrices are used, and they can themselves be sparse or dense, with distinct algebraic methods used in each case.

For example, when the model includes a corrMatrix term specified by a covariance matrix, the precision matrix may be computed to assess its sparseness. The Example below illustrates a case where detecting sparsity of the precision matrix allows a faster fit. However, such a comparison of correlation and precision matrices takes time and is not performed for all types of random-effect structures. Instead, some fast heuristics may be used (see Details).

The default selection of methods may not always be optimal, and may be overcome on a fit-by-fit basis, or more globally. Indeed, the sparse-precision algorithms are often faster than dense-correlation algorithms even when the precision matrix is dense. Yet, the dense-correlation algorithms will be used by default in this case, as they are numerically more stable.

Per-fit control is provided by using the `control.HLfit` argument of the fitting function (see Details for global control). In particular one can use either `control.HLfit=list(sparse_precision=<TRUE|FALSE>)` or
`control.HLfit=list(algebra=<"spprec"|"spcorr"|"decorr">)`
with the obvious expected effects. Fit-specific controls by `control.HLfit` override global ones.

Such control may be useful when you already know that the precision matrix is sparse (as spaMM may even kindly remind you of, see Example below). In that case, it is also efficient to specify the precision matrix directly (see Example in `Gryphon`), as spaMM then assumes that sparse-precision methods are better without checking the correlation matrix.

Such control may also be useful when the correlation matrix is nearly singular so that computation of its inverse fails. This may occur if the model is poorly specified, but also occurs sometimes for valid correlation models because inversion of large matrices though Cholesky methods is not numerically accurate enough. In the latter case, you may be directed to this documentation by an error message, and specifying `sparse_precision=FALSE` may be useful.

Details

Currently the sparse-precision methods are selected by default in two cases (with possible exceptions indicated by specific messages): (1) for models including `IMRF` random effects; and (2) when the `corrMatrix` (or `covStruct`) syntax is used to provide a fixed precision matrix. In other cases (including models with other autoregressive terms such as adjacency or AR1), sparse-precision methods may or may not be selected, depending on heuristic rules based on the likely structure of the design matrix for random effects.

Algebraic methods can be controlled globally over all fits by using
`> spaMM.options(sparse_precision=<TRUE|FALSE>)`
and, among the correlation-based methods,
`> spaMM.options(QRmethod=<"sparse"|"dense">)`
to select `"spcorr"` vs. `"decorr"` methods.
`> spaMM.options(dec2spp=<FALSE|TRUE>)`
controls whether to use algorithms suited for random effects with sparse precision matrices in many cases where the precision matrix is not particularly sparse, and where algorithms suited for random effects with dense correlation matrices would otherwise be used (as can be selected on a fit-by-fit basis by `algebra="spprec"`). Default is `FALSE`; when this is set to `TRUE`, other conditions (subject to future changes) are checked before using switching to sparse-precision algorithms. Practice show that sparse-precision methods are actually efficient even when the precision matrix is not sparse. This is partly due to them using Cholesky instead of QR matrix factorizations. QR is numerically more accurate, which may matter at least in challenging cases with near-singular correlation matrices, but otherwise `dec2spp=TRUE` is an interesting option.

See Also

`how` for extracting information about the selected `algebra` from a fit. `pedigree` for context to the Examples.

Examples

```
if (spaMM.getOption("example_maxtime")>6) {
 data("Gryphon")
```

```
gry_df <- fitme(BWT ~ 1 + corrMatrix(1|ID), corrMatrix = Gryphon_A,
                data = Gryphon_df, method = "REML")
how(gry_df)

# => Note the message about 'Choosing matrix methods...'.
# Using control.HLfit=list(algebra="spprec") would indeed
# save the time used to select this method.

# Conversely, using a correlation-based method would be a waste of time:

gry_dn <- fitme(BWT ~ 1 + corrMatrix(1|ID), corrMatrix = Gryphon_A,
              data = Gryphon_df, method = "REML",
              control.HLfit=list(sparse_precision=FALSE))
how(gry_dn) # forced dense-correlation methods, which is slower here.
}
```

---

aliases *Variable aliases for multivariate-response fits*

Description

Consider the `toy_iris` dataset defined in the Examples, noting in particular the lines of code

```
u <- rnorm(10,sd=1)
id_y <- gl(10,5)
id_b <- rep(seq(10),5)
```

and the subsequent use of `u[id_y]` and `u[id_b]` in the expectations of Poisson draws. Suppose that we fit to a multivariate-response model with two Poisson responses to it. A fit by

```
fitmv(submodels = list(
    list(yellow ~ 1+(1|id_y), family = poisson()),
    list(blue ~ 1+(1|id_b), family = poisson())),
    data = toy_iris)
```

does not match the data-generating algorithm, because the fit of the `(1|id_y)` and `(1|id_b)` random effects does not take into account that the latent values of these random effects are sampled from a single vector `u` of 10 latent values. One should rather fit a single random-effect variance and produce a single predicted vector for `u`, rather than two distinct vectors of predicted values.

Conversely, a fit such as

```
fitmv(submodels = list(
    list(yellow ~ 1+(1|id_y), family = poisson()),
    list(blue ~ 1+(1|id_y), family = poisson())),
    data = toy_iris)
```

would fit a single random-effect variance and produce a single predicted vector, but the factor indices are incorrect in the second submodel.

We need a syntax such that a single variance and a single vector are fitted (as when the random-effect terms are identical in the two formulas), but where the factor indices are effectively `id_y` and `id_b` in the two submodels. The syntax of the `proper` fit in the Examples achieves this effect. The random effect is specified as `(1|id)` where `id` is *not* an actual factor in the `data` but is an alias for the variable `id_y` in the first formula and `id_b` in the second one. This interpretation of the `(1|id)` term is specified by the argument `aliases=list(id=c("id_y","id_b"))`.

See Also

`vignette("Multinomial-logit", package = "spaMM")` for more examples.

Examples

```
# Toy data
set.seed(123)
ssize <- 50L
u <- rnorm(10,sd=1)
id_y <- gl(10,5)
id_b <- rep(seq(10),5)
yellow <- rpois(ssize, lambda=exp(1+u[id_y]))
blue <- rpois(ssize, lambda=exp(1+u[id_b]))
toy_iris <- data.frame(
  id_y=id_y, id_b=id_b, yellow=yellow, blue=blue
)

# Fit
proper <- fitmv(submodels = list(
    list(yellow ~ 1+(1|id), family = poisson()),
    list(blue ~ 1+(1|id), family = poisson())),
    data = toy_iris, aliases=list(id=c("id_y","id_b")))

ranef(proper) # single vector of 10 values

# Rows 1-50 and 51-100 of Z matrix show
# the distinct design of the two submodels:
get_matrix(proper,"ZA")
```

`anova` *ANOVA tables (and likelihood ratio tests).*

Description

The `anova` method for fit objects from **spaMM** has two uses: if a single fit object is provided, ANOVA tables may be returned, with specific procedures for univariate-response LMs, GLMs and LMMs (see Details). Alternatively, if a second fit object is provided (`object2` argument), `anova` performs as an alias for `LRT`.

Usage

```
## S3 method for class 'HLfit'
anova(object, object2, type = "2", method="", ...)
```

Arguments

`object`	Fit object returned by a **spaMM** fitting function.
`object2`	Optional second model fit to be compared to the first (their order does not matter, except in non-standard cases where the second model is taken as the null one and a message is issued).
`type`	ANOVA type for LMMs, as interpreted by **lmerTest**. Note that the default (single-term deletion ANOVA) differs from that of **lmerTest**.
`method`	Only non-default value is `"t.Chisq"` which forces evaluation of a table of chi-squared tests for each fixed-effect term, using the classical "Wald" test (see Details). Further methods are available through the ... for specific classes of models.
`...`	Further arguments, passed to `spaMM_boot` (e.g., for parallelization) in the case of LRTs (i.e., when `object2` is provided). For ANOVA tables, arguments of functions `anova.lm` `anova.glm`, and `as_LMLT`, respectively for LMs, GLMs and LMMs, may be handled (e.g. the `test` argument for `anova.glm`).

Details

The ANOVA-table functionality has been included in **spaMM** mainly to provide access to F tests (including, for LMMs, the "Satterthwaite method" as developed by Fai and Cornelius, 1996), using pre-existing procedures as template or backend for expediency and familiarity:

1. ANOVA tables **for LMs and GLMs** have been conceived to replicate the functionality, output format and details of base R `anova`, and therefore replicate some of their limitations, e.g., they only perform sequential analysis ("type 1") in the same way as `anova.lm` and `anova.glm`. However, a difference occurs for Gamma GLMs, because the dispersion estimates for Gamma GLMs differ between `stats::glm` and **spaMM** fits (see Details in `method`). Therefore, F tests and Mallows' Cp differ too; results from **spaMM** REML fits being closer than ML fits to those from `glm()` fits;

2. **For LMMs**, ANOVA tables are provided by interfacing `lmerTest::anova` (with non-default `type`). This procedure should handle all types of LMMs that can be fitted by **spaMM**; yet, the displayed information should be inspected to check that some fitted random-effect parameters are not ignored when computing information for the Satterthwaite method.

3. For fitted models that do not lay within previous categories, such as **GLMMs**, models with a **residual-dispersion** submodel, and **multivariate-response** models, a table of tests for single-term deletions using the classical "Wald" chi-squared test based on coefficient values and their conditional standard error estimates will be returned. LRTs (moreover, with bootstrap correction) are more reliable than such tests and, as calling them requires a second model to be explicitly specified, they may also help users thinking about the hypothesis they are testing.

Value

The return format is that of the function called (`lmerTest::anova` for LMMs) or emulated (for LMs or GLMs). For GLMs, by default no test is reported, as has been the default for `anova.glm` before R 4.4.0.

References

Fai AH, Cornelius PL (1996). Approximate F-tests of multiple degree of freedom hypotheses in generalised least squares analyses of unbalanced split-plot experiments. Journal of Statistical Computation and Simulation, 54(4), 363-378. doi:10.1080/00949659608811740

See Also

`LRT`,
`as_LMLT` for the interface to `lmerTest::anova`,
and `summary.HLfit(.,details=list(<true|"Wald">))` for reporting the p-value for each t-statistic in the summary table for fixed effects, either by Student's t distribution, or by the approximation of t^2 distribution by the Chi-squared distribution ("Wald's test").

Examples

```
data("wafers")
## Gamma GLMM with log link
m1 <- HLfit(y ~X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch),family=Gamma(log),
          resid.model = ~ X3+I(X3^2) ,data=wafers,method="ML")
m2 <- update(m1,formula.= ~ . -I(X2^2))
#
anova(m1,m2) # LRT

## 'anova' (Wald chi-squared tests...) for GLMM or model with a 'resid.model'
anova(m1)

## ANOVA table for GLM
# Gamma example, from McCullagh & Nelder (1989, pp. 300-2), as in 'glm' doc:
clotting <- data.frame(
    u = c(5,10,15,20,30,40,60,80,100),
    lot1 = c(118,58,42,35,27,25,21,19,18),
    lot2 = c(69,35,26,21,18,16,13,12,12))
spglm <- fitme(lot1 ~ log(u), data = clotting, family = Gamma, method="REML")
anova(spglm, test = "F")
anova(spglm, test = "Cp")
anova(spglm, test = "Chisq")
anova(spglm, test = "Rao")

## ANOVA table for LMM
if(requireNamespace("lmerTest", quietly=TRUE)) {
  lmmfit <- fitme(y ~X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch),data=wafers)
  print(anova(lmmfit)) # => Satterthwaite method, here giving p-values
                       #   quite close to traditional t-tests given by:
  summary(lmmfit, details=list(p_value=TRUE))
}
```

`arabidopsis` *Arabidopsis genetic and climatic data*

Description

For 948 "accessions" from European Arabidopsis thaliana populations, this data set merges the genotypic information at four single nucleotide polymorphisms (SNP) putatively involved in adaptation to climate (Fournier-Level et al, 2011, Table 1), with 13 climatic variables from Hancock et al. (2011).

Usage

```
data("arabidopsis")
```

Format

The data frame includes 948 observations on the following variables:

pos1046738, pos5510910, pos6235221, pos8132698 Genotypes at four SNP loci

LAT latitude

LONG longitude

seasonal, tempWarmest, tempColdest, preciWettest, preciDriest, preciCV, PAR_SPRING,

growingL, conseqCold, conseqFrFree, RelHumidSp, dayLSp, aridity Thirteen climatic variables. See Hancock et al. (2011) for details about these variables.

Details

The response is binary so `method="PQL/L"` seems warranted (see Rousset and Ferdy, 2014).

Source

The data were retrieved from `http://bergelson.uchicago.edu/regmap-data/climate-genome-scan` on 22 February 2013 (they may no longer be available from there).

References

Fournier-Level A, Korte A., Cooper M. D., Nordborg M., Schmitt J., Wilczek AM (2011). A map of local adaptation in Arabidopsis thaliana. Science 334: 86-89.

Hancock, A. M., Brachi, B., Faure, N., Horton, M. W., Jarymowycz, L. B., Sperone, F. G., Toomajian, C., Roux, F., and Bergelson, J. 2011. Adaptation to climate across the Arabidopsis thaliana genome, Science 334: 83-86.

Rousset F., Ferdy, J.-B. (2014) Testing environmental and genetic effects in the presence of spatial autocorrelation. Ecography, 37: 781-790. doi:10.1111/ecog.00566

Examples

```
data("arabidopsis")
if (spaMM.getOption("example_maxtime")>2.5) {
  fitme(cbind(pos1046738,1-pos1046738)~seasonal+Matern(1|LAT+LONG),
        fixed=list(rho=0.119278,nu=0.236990,lambda=8.599),
        family=binomial(),method="PQL/L",data=arabidopsis)
}
## The above 'fixed' values are deduced from the following fit:
if (spaMM.getOption("example_maxtime")>46) {
  SNPfit <- fitme(cbind(pos1046738,1-pos1046738)~seasonal+Matern(1|LAT+LONG),
              verbose=c(TRACE=TRUE),
              family=binomial(),method="PQL/L",data=arabidopsis)
  summary(SNPfit) # p_v=-125.0392
}
```

ARp

Random effect with AR(p) (autoregressive of order p) or ARMA(p,q) structure.

Description

These times-series correlation models can be declared as correlation models for random effect. The AR(p) model is here parametrized by the **partial** correlation coefficients of the levels of the random effect, {U_t}, corr(U_s,U_t|U_(s+1),...,U_(t-1)), with valid values in the hypercube]-1,,1[^p (Barndorff-Nielsen and Schou, 1973). In the autoregressive-moving average ARMA(p,q) model, the AR part is parametrized in the same way. AR parameters are named "p1", "p2"..., and MA parameters are named "q1", "q2"... .

Implementation of the AR(p) model uses the sparsity of the inverse covariance matrix. In the ARMA(p,q) model, neither the covariance nor its inverse are sparse, so fits are expected to be more time- and memory-consuming.

Usage

```
# corrFamily constructors:
ARp(p=1L, fixed=NULL, corr=TRUE, tpar=1/(1+seq(p)))
ARMA(p=1L, q=1L, fixed=NULL, tpar=c(1/(1+seq_len(p)),1/(1+seq_len(q))))
```

Arguments

p	Integer: order of the autoregressive process.
q	Integer: order of the moving-average process.
tpar	Numeric vector: template values of the **partial coefficient coefficients** of the autoregressive process, and the traditional coefficients of the moving-average processe, in this order. The tpar vector must always have full length, even when some parameters are fixed.
fixed	NULL or numeric vector, to fix the parameters of this model.
corr	For development purposes, better ignored in normal use.

Value

The ARp and ARMA functions return a corrFamily descriptor, hence a list including element Cf, a function returning, for given ARMA or AR parameters, the correlation matrix for ARMA, or its **inverse** for ARp.

The fitted correlation matrix can be extracted from a fit object, as for any autocorrelated random effect, by Corr(<fit object>)[[<random-effect index>]].

References

Barndorff-Nielsen 0. and Schou G., 1973 On the parametrization of autoregressive models by partial autocorrelations. J. Multivariate Analysis 3: 408-419. doi:10.1016/0047259X(73)900304

Examples

```
if (spaMM.getOption("example_maxtime")>2) {
ts <- data.frame(lh=lh,time=seq(48)) ## using 'lh' data from 'stats' package

## Default 'tpar' => AR1 model
#
(ARpfit <-  fitme(lh ~ 1 + ARp(1|time), data=ts, method="REML"))
#
## which is equivalent to
#
(AR1fit <- fitme(lh ~ 1 +AR1(1|time), data=ts, method="REML"))

## AR(3) model
#
(AR3fit <- fitme(lh ~ 1 + ARp(1|time, p=3), data=ts, method="REML"))

## Same but with fixed 2-lag partial autocorrelation
#
(AR3fix <- fitme(lh ~ 1 + ARp(1|time, p=3, fixed=c(p2=0)), data=ts, method="REML"))
#
# The fit should be statistically equivalent to
#
(AR3_fix <- fitme(lh ~ 1 + ARp(1|time, p=3), data=ts, method="REML",
                  fixed=list(corrPars=list("1"=c(p2=0)))))
#
# with subtle differences in the structure of the fit objects:
#
get_ranPars(AR3fix)$corrPars     # p2 was not a parameter of the model
get_ranPars(AR3_fix)$corrPars    # p2 was a fixed parameter of the model
#
# get_fittefPars() expectedly ignores 'p2' whichever way it was fixed.


## Same as 'AR3fix' but with an additional MA(1) component
#
(ARMAfit <- fitme(lh ~ 1 + ARMA(1|time, p=3, q=1, fixed=c(p2=0)),
                  data=ts, method="REML"))
}
```

as_LMLT *Conversion to input for procedures from lmerTest package*

Description

The lmerTest::contest function, drop1 and anova methods implement a number of tests for linear mixed models, e.g. using effective degrees of freeedom based on (a generalization of) Satterthwaite's method. These tests can be performed using **spaMM** fits through the conversion of the fit object, by the as_LMLT function, to an ad-hoc format acceptable as input to **lmerTest**'s internal procedures. The separately documented drop1.HLfit and (optionally) anova.HLfit methods, when called on a single LMM fit object, perform the conversion by as_LMLT and call drop1 or anova methods defined by **lmerTest**.

Only the tests using **lmerTest**'s default method ddf="Satterthwaite" are formally supported, as the converted object do not have the required format for the other methods. Only LMMs are handled by **lmerTest**, and residual-dispersion models are not yet handled by the conversion. However, the conversion extends **lmerTest**'s functionality by handling all random-effect parameters handled by numInfo, therefore including (e.g.) spatial-correlation parameters not handled by **lme4**.

Usage

```
as_LMLT(fitobject, nuisance=NULL, verbose=TRUE, transf=TRUE, check_deriv=NULL, ...)
```

Arguments

fitobject	Object of class HLfit resulting from the fit of a linear mixed model (LMM).
nuisance	A list of fitted values of parameters that affect the distribution of the test of fixed effects, in the format of the fixed argument of the fitme function. If NULL (default), then the list is constructed from the fitted values of the random-effect parameters and of phi (residual dispersion parameter). The nuisance argument is better ingored unless the extractor he construct the default value fails in some way.
verbose	boolean: controls printing of the message that shows the unlisted value of the nuisance list.
transf	boolean: whether to evaluate numerical derivatives on a transformed parameter scale, or not (may affect numerical precision).
check_deriv	See same-named argument of numInfo
...	Other arguments that may be needed by some method (currently ignored).

Value

The value is returned invisibly. It is an S4 object of class "LMLT" with slots matching those required in objects of S4 class "lmerModLmerTest" when used by package **lmerTest** with ddf="Satterthwaite" (many additional slots of a formal "lmerModLmerTest" object are missing). The additional nuisance slot contains the nuisance list.

References

Alexandra Kuznetsova, Per B. Brockhoff and Rune H. B. Christensen (2017) lmerTest Package: Tests in Linear Mixed Effects Models. Journal of Statistical Software, 82(13), 1–26. doi:10.18637/jss.v082.i13

Examples

```
## Reproducing an example from the doc of lmerTest::contest.lmerModLmerTest,
#    using a spaMM fit as input.
## Not run:
  data("sleepstudy", package="lme4")

  ## The fit:
  spfit <- fitme(Reaction ~ Days + I(Days^2) + (1|Subject) + (0+Days|Subject),
             sleepstudy, method="REML")

  ## Conversion:
  spfit_lmlt <-  as_LMLT(spfit)

  ## Functions from package lmerTest can then be called on this object:
  lmerTest::contest(spfit_lmlt, L=diag(3)[2:3, ])   # Test of 'Days + I(Days^2)'.
  #
  anova(spfit_lmlt, type="1")         # : using lmerTest:::anova.lmerModLmerTest()
  drop1(spfit_lmlt)                   # : using lmerTest:::drop1.lmerModLmerTest()


## End(Not run)
```

`autoregressive` *Fitting autoregressive models*

Description

Diverse autoregressive (AR) models are implemented in spaMM. This documentation describe the `adjacency` model (a conditional AR, i.e., CAR), and the `AR1` model for time series. Other documentation deals with more or less distantly related models: `ARp` for more general AR(p) and ARMA(p,q) models for time series, and `IMRF` and `MaternIMRFa` for mesh-based approximations of geostatistical models.

An AR1 random effect is specified as `AR1(1|<grouping factor>)`. It describes correlations between realizations of the random effect for (typically) successive time-steps by a correlation ϕ, denoted `ARphi` in function calls. Nested AR1 effects can be specified by a nested grouping factor, as in `AR1(1|<time index> %in% <nesting factor>)`.

A CAR random effect is specified as `adjacency(1|<grouping factor>)`. The correlations among levels of the random effect form a matrix $(\mathbf{I}-\rho$ `adjMatrix`$)^{-1}$, in terms of an `adjMatrix` matrix which must be provided, and of the scalar ρ, denoted `rho` in function calls. The rows and columns of `adjMatrix` must have names matching those of levels of the random effect **or else** are assumed to match a sequence, from 1 to the number of columns, of values of the geographic location index specifying the spatial random effect. For example, if the model formula is `y ~ adjacency(1|geo.loc)` and `<data>$geo.loc` is 2,4,3,1,... the first row/column of the matrix refers to `geo.loc`=1, i.e. to the fourth row of the data.

Details

Efficient algorithms for CAR models have been widely discussed in particular in the econometric literature (e.g., LeSage and Pace 2009), but these models are not necessarily recommended for irregular lattices (see Wall, 2004 and Martellosio, 2012 for some insights on the implications of autoregressive models).

In **CAR** models, the covariance matrix of random effects **u** can be described as $\lambda(\mathbf{I}-\rho\,\mathbf{W})^{-1}$ where **W** is the (symmetric) adjacency matrix. `HLCor` uses the spectral decomposition of the adjacency matrix, written as **W=VDV'** where **D** is a diagonal matrix of eigenvalues d_i. The covariance of **V'u** is $\lambda(\mathbf{I}-\rho\,\mathbf{D})^{-1}$, which is a diagonal matrix with elements λ_i=$\lambda/(1-\rho d_i)$. Hence $1/\lambda_i$ is in the linear predictor form α+βd_i.

This can be used to fit λ and ρ efficiently by "inner" iterations (see `inner-vs-outer` for this concept). A call to `corrHLfit` with the additional argument `init.HLfit=list(rho=0)` should be equivalent in speed and result to the `HLCor` call. This is fast for small datasets (as in the example below) but more generic maximization algorithms may be preferable for large ones. It is suggested to use `fitme` generally unless one has a large number of small data sets to analyze. A call to `fitme` or `corrHLfit` without that `init.HLfit` value does not use the spectral decomposition. It performs numerical maximization of the likelihood (or restricted likelihood) as function of the correlation parameter ρ. The choice of fitting function may slightly impact the results. The ML fits by `corrHLfit` and `HLCor` should be practically equivalent. The REML fits should slightly differ from each other (see Details in `inner-vs-outer`).

If `HLCor` is used, the results are reported as the coefficients α (`(Intercept)`) and β (`adjd`) of the predictor for $1/\lambda_i$, in addition to the resulting values of ρ and of the common λ factor.

Different fits may also differ in using or not algorithms that exploit the sparsity of the precision matrix of the autoregressive random effect. By default, spaMM tends to select sparse-precision algorithms for large datasets and large (i.e. many-level) random effects (details are complex). However, for **AR1** models, the dimension of the implied precision matrix is determined by the extreme values of grouping factor (typically interpreted as a time index), as all intermediate values must be considered. Then, the correlation-based algorithms may be more efficient if only a few levels are present in the data, as only a small correlation matrix is required in that case.

References

LeSage, J., Pace, R.K. (2009) Introduction to Spatial Econometrics. Chapman & Hall/CRC.

Martellosio, F. (2012) The correlation structure of spatial autoregressions, Econometric Theory 28, 1373-1391.

Wall M.M. (2004) A close look at the spatial structure implied by the CAR and SAR models: Journal of Statistical Planning and Inference 121: 311-324.

Examples

```
##### AR1 random effect:
ts <- data.frame(lh=lh,time=seq(48)) ## using 'lh' data from stats package
fitme(lh ~ 1 +AR1(1|time), data=ts, method="REML")
# With fixed parameters:
# HLCor(lh ~ 1 +AR1(1|time), data=ts, ranPars=list(ARphi=0.5,lambda=0.25,phi=0.001))

##### CAR random effect:
```

```
data("scotlip")
# CAR by Laplace with 'outer' estimation of rho
if (spaMM.getOption("example_maxtime")>0.8) {
  fitme(cases ~ I(prop.ag/10)+adjacency(1|gridcode)+offset(log(expec)),
          adjMatrix=Nmatrix, family=poisson(), data=scotlip)
}

# CAR by Laplace with 'inner' estimation of rho
HLCor(cases ~ I(prop.ag/10)+adjacency(1|gridcode)+offset(log(expec)),
          adjMatrix=Nmatrix, family=poisson(), data=scotlip, method="ML")
```

`betabin` *Beta-binomial family object*

Description

Returns a `family` object for beta-binomial models. The precision parameter `prec` of this family is a positive value such that the variance of the beta-distributed latent variable given its mean μ is $\mu(1-\mu)/(1+$`prec`$)$. `prec` is thus the same precision parameter as for the beta family (see `beta_resp`. The variance of the beta-binomial sample of size n is response is $\mu(1-\mu)n(n+$`prec`$)/(1+$`prec`$)$.

The family described by this object is not a GLM family, so it should not be used as a `glm` argument unless one additionally uses `method="llm.fit"` (see Examples). But it is still characterized by the same concepts as GLMs: a linear predictor, a link function, and the given distribution of residual variation.

A **fixed-effect** residual-dispersion model can be fitted, using the `resid.model` argument, which is used to specify the form of the logarithm of the precision parameter (see Examples). Thus the variance of the latent beta-distributed variable becomes $\mu(1-\mu)/(1+$`exp(<specified linear expression>)`$)$.

Usage

```
betabin(prec, link = "logit")
```

Arguments

`prec`	Scalar (or left unspecified): precision parameter of the beta distribution. The "outer-optimizing" functions, such as `fitme` (see `inner-vs-outer`) will estimate this parameter if called with `family=betabin()` (no given `prec` value). In other usages of this family, `prec` must be specified.
`link`	logit, probit, cloglog or cauchit link, specified by any of the available ways for GLM links (name, character string, one-element character vector, or object of class `link-glm` as returned by `make.link`).

Details

Prior weights are meaningful for this family and handled as a factor of the precision parameter of the latent beta-distributed variable: the variance of the latent variable become $\mu(1-\mu)/(1+$`prec*<prior weights>`$)$. However, this feature is experimental and may be removed in the future. The fitting function's `resid.model` argument may be preferred to obtain the same effect, by specifying an `offset(log(<prior weights>))` in its formula (given the log link used in that model). As usual in **spaMM**, the offset(.) argument should be a vector and any variable necessary for evaluating it should be in the `data`.

Value

A list, formally of class `c("LLF", "family")`. See `LL-family` for details about the structure and usage of such objects.

Examples

```
data(clinics)
fitme(cbind(npos,nneg)~1+(1|clinic), family=betabin(), data=clinics)

# Compatibility with glm(., method="llm.fit"):
glm(cbind(npos,nneg)~1, family=betabin(2), data=clinics,
    method="llm.fit")
# (trying to use default method=stats::glm.fit in this case will fail).
```

---

beta_resp *Beta-distribution family object*

Description

Returns a `family` object for beta-response models. The precision parameter `prec` of this family is a positive value such that the variance of the response given its mean μ is $\mu(1-\mu)/(1+$`prec`$)$. `prec` is thus the precision parameter ϕ of Ferrari & Cribari-Neto (2004) and of the **betareg** package (Cribari-Neto & Zeileis 2010).

The family described by this object is not a GLM family, so it should not be used as a `glm` argument unless one additionally uses `method="llm.fit"` (see Examples for `negbin1` which is equivalent in this respect). But it is still characterized by the same concepts as GLMs: a linear predictor, a link function, and the given distribution of residual variation.

A **fixed-effect** residual-dispersion model can be fitted, using the `resid.model` argument, which is used to specify the form of the logarithm of the precision parameter (see Examples). Thus the variance of the response become $\mu(1-\mu)/(1+$`exp(<specified linear expression>)`$)$.

Usage

```
beta_resp(prec, link = "logit")
```

Arguments

prec	Scalar (or left unspecified): precision parameter of the beta distribution. The "outer-optimizing" functions, such as fitme (see inner-vs-outer) will estimate this parameter if called with family=beta_prec() (no given prec value). In other usages of this family, prec must be specified.
link	logit, probit, cloglog or cauchit link, specified by any of the available ways for GLM links (name, character string, one-element character vector, or object of class link-glm as returned by make.link).

Details

Prior weights are meaningful for this family and handled as a factor of the precision parameter (as for GLM families) hence here not as a divisor of the variance (in contrast to GLM families): the variance of the response become $\mu(1-\mu)/(1+$prec*<prior weights>). However, this feature is experimental and may be removed in the future. The fitting function's resid.model argument may be preferred to obtain the same effect, by specifying an offset(log(<prior weights>)) in its formula (given the log link used in that model). As usual in **spaMM**, the offset(.) argument should be a vector and any variable necessary for evaluating it should be in the data.

Value

A list, formally of class c("LLF", "family"). See LL-family for details about the structure and usage of such objects.

References

Cribari-Neto, F., & Zeileis, A. (2010). Beta Regression in R. Journal of Statistical Software, 34(2), 1-24. doi:10.18637/jss.v034.i02

Ferrari SLP, Cribari-Neto F (2004). "Beta Regression for Modelling Rates and Proportions." Journal of Applied Statistics, 31(7), 799-815.

See Also

Further examples in LL-family.

Examples

```
 set.seed(123)
beta_dat <- data.frame(y=runif(100),grp=sample(2,100,replace = TRUE), x_het=runif(100))

fitme(y ~1+(1|grp), family=beta_resp(), data= beta_dat)
## same logL, halved 'prec' when prior weights=2 are used:
# fitme(y ~1+(1|grp), family=beta_resp(), data= beta_dat, prior.weights=rep(2,100))

## With model for residual dispersion:
# fitme(y ~1+(1|grp), family=beta_resp(), data= beta_dat, resid.model= ~ x_het)
```

blackcap *Genetic polymorphism in relation to migration in the blackcap*

Description

This data set is extracted from a study of genetic polymorphisms potentially associated to migration behaviour in the blackcap (Sylvia atricapilla). Across different populations in Europe and Africa, the average migration behaviour was found to correlate with average allele size (dependent on the number of repeats of a small DNA motif) at the locus ADCYAP1, encoding a neuropeptide. This data set is quite small and ill-suited for separating random-effect variance from residual variance. The likelihood surface for the Matérn model actually has local maxima.

Usage

```
data("blackcap")
```

Format

The data frame includes 14 observations on the following variables:

latitude latitude, indeed.

longitude longitude, indeed.

migStatus migration status as determined by Mueller et al, from 0 (resident populations) to 2.5 (long-distance migratory populations)

means Mean allele sizes in each population

pos Numerical index for the populations

Details

Migration status was coded as : pure resident populations as '0', resident populations with some migratory restlessness as '0.5', partial migratory populations as '1', completely migratory populations migrating short-distances as '1.5', intermediate-distance migratory populations as '2' and distinct long-distance migratory populations as '2.5'.

Source

Data from Mueller et al. (2011), including supplementary material now available from doi:10.1098/rspb.2010.2567.

References

Mueller, J. C., Pulido, F., and Kempenaers, B. 2011. Identification of a gene associated with avian migratory behaviour, Proc. Roy. Soc. (Lond.) B 278, 2848-2856.

Examples

```
## see 'fitme', 'corrHLfit' and 'fixedLRT' for examples involving these data
```

CauchyCorr *Cauchy correlation function and Cauchy formula term*

Description

The Cauchy family of correlation functions is useful to describe spatial processes with power-law decrease of correlation at long distance. It is valid for Euclidean distances in spaces of any dimension, and for great-circle distances on spheres of any dimension. It has a scale parameter (`rho`, as in the Matérn correlation function), a `shape` (or "smoothness", Gneiting 2013) parameter, and a `long`-memory dependence (or, more abstractly, "shape"; Gneiting 2013) parameter (Gneiting and Schlater 2004). The present implementation also accepts a `Nugget` parameter. The family can be invoked in two ways. First, the `CauchyCorr` function evaluates correlations, using distances as input. Second, a term of the form `Cauchy(1|<...>)` in a `formula` specifies a random effect with Cauchy correlation function, using coordinates found in a data frame as input. In the latter case, the correlations between realizations of the random effect for any two observations in the data will be the value of the Cauchy function at the scaled distance between coordinates specified in <...>, using "+" as separator (e.g., `Cauchy(1|longitude+latitude)`). A syntax of the form `Cauchy(1|longitude+latitude %in% grp)` can be used to specify a Cauchy random effect with independent realizations for each level of the grouping variable `grp`.

Usage

```
## Default S3 method:
CauchyCorr(d, rho=1, shape, longdep, Nugget=NULL)
# Cauchy(1|...)
```

Arguments

`d`	Euclidean or great-circle distance
`rho`	The scaling factor for distance, a real >0.
`shape`	The shape (smoothness) parameter, a real 0<.<=2 for Euclidean distances and 0<.<=1 for great-circle distances. Smoothness increases, and fractal dimension decreases, with increasing shape (the fractal dimension of realizations in spaces of dimension d being d+1-shape/2).
`longdep`	The long-memory dependence parameter, a real >0. It gives the exponent of the asymptotic decrease of correlation with distance: the **smaller** `longdep` is, the longer the dependence.
`Nugget`	(Following the jargon of Kriging) a parameter describing a discontinuous decrease in correlation at zero distance. Correlation will always be 1 at $d = 0$, and from which it immediately drops to (1-Nugget). Defaults to zero.
`...`	Names of coordinates, using "+" as separator (e.g., `Matern(1|longitude+latitude)`. The coordinates are numeric values found in the `data` data frame provided to the fitting function. No additional declaration of groups, factors, or other specific formatting is required.

Details

The correlation at distance $d > 0$ is

$$(1 - \text{Nugget})(1 + (\rho d)^{\text{shape}})^{-\text{longdep/shape}}$$

Value

Scalar/vector/matrix depending on input.

References

Gneiting, T. and Schlater M. (2004) Stochastic models that separate fractal dimension and the Hurst effect. SIAM Rev. 46: 269–282.

Gneiting T. (2013) Strictly and non-strictly positive definite functions on spheres. Bernoulli 19: 1327-1349.

Examples

```
data("blackcap")
fitme(migStatus ~ means+ Cauchy(1|longitude+latitude),
      data=blackcap,
      #  fixed=list(longdep=0.5,shape=0.5,rho=0.05)
     )
## The Cauchy family can be used in Euclidean spaces of any dimension:
set.seed(123)
randpts <- matrix(rnorm(20),nrow=5)
distMatrix <- as.matrix(proxy::dist(randpts))
CauchyCorr(distMatrix,rho=0.1,shape=1,longdep=10)

# See ?MaternCorr for examples of syntaxes for group-specific random effects,
#  also handled by Cauchy().
```

---

clinics *Toy dataset for binomial response*

Description

A small data set used by Booth & Hobert (1998).

Usage

```
data("clinics")
```

Format

A data frame with 16 observations on the following 4 variables.

`npos` a numeric vector

`nneg` a numeric vector

`treatment` a numeric vector

`clinic` a numeric vector

References

Booth, J.G., Hobert, J.P. (1998) Standard errors of prediction in generalized linear mixed models. J. Am. Stat. Assoc. 93: 262-272.

Examples

```
data(clinics)
## Not run:
# The dataset was built as follows
npos <- c(11,16,14,2,6,1,1,4,10,22,7,1,0,0,1,6)
ntot <- c(36,20,19,16,17,11,5,6,37,32,19,17,12,10,9,7)
treatment <- c(rep(1,8),rep(0,8))
clinic <-c(seq(8),seq(8))
clinics <- data.frame(npos=npos,nneg=ntot-npos,treatment=treatment,clinic=clinic)

## End(Not run)
```

COMPoisson *Conway-Maxwell-Poisson (COM-Poisson) GLM family*

Description

The COM-Poisson family is a generalization of the Poisson family which can describe over-dispersed as well as under-dispersed count data. It is indexed by a parameter `nu` that quantifies such dispersion. For `nu`>1, the distribution is under-dispersed relative to the Poisson distribution with same mean. It includes the Poisson, geometric and Bernoulli as special (or limit) cases (see Details). Several links are allowed for this family, corresponding to different versions of the COMPoisson described in the literature (e.g., Sellers & Shmueli 2010; Huang 2017).

Following established notation, the COM-Poisson dispersion parameter is named `nu` or ν, but it is distinct from the parameter $\nu = 1/\phi$ considered elsewhere in this package and in standard accounts of GLMs: the latter ν is a multiplicative factor of the derivative of the log-likelihood with respect to the GLM "canonical parameter", while the COM-Poisson ν affects the log-likelihood differently (which is why this parameter cannot be fitted by `glm`).

The COM-Poisson family is here implemented as a `family` object, so that it can be fitted by `glm` (but for fixed `nu` only, for the reason just given; see Examples), beyond its primary intended use with **spaMM**'s fitting functions.

Usage

```
COMPoisson(nu, link = "loglambda")
```

Arguments

link	GLM link function. The default is the canonical link "loglambda" (see Details), but other links are allowed (currently log, sqrt or identity links as commonly handled for the Poisson family).
nu	Under-dispersion parameter. The "outer-optimizing" functions, such as fitme (see inner-vs-outer) will estimate this parameter if called with family=COMPoisson() (no given nu value). In other usages of this family, nu must be specified. COMPoisson(nu=1) is the Poisson family.

Details

The ith term of the distribution can be written q_i/Z where $q_i = \lambda^i/(i!)^\nu$ and $Z = \sum_{(i=0)}^{\infty} q_i$, for $\lambda = \lambda(\mu)$ implied by its inverse relationship, the expectation formula $\mu = \mu(\lambda) = \sum_{(i=0)}^{\infty} i q_i(\lambda)/Z$. The case nu=0 is the geometric distribution with parameter λ; nu=1 is the Poisson distribution with mean λ; and the limit as nu -> ∞ is the Bernoulli distribution with expectation $\lambda/(1+\lambda)$.

From this definition, this is an exponential family model with canonical parameters $\log(\lambda)$ and ν. When the linear predictor η specifies $\log(\lambda(\mu))$, the canonical link is used (e.g., Sellers & Shmueli 2010). This link is here nicknamed "loglambda" and does not have a known expression in terms of elementary functions. To obtain μ as the link inverse of the linear predictor η, one then first computes $\lambda = e^\eta$ and then $\mu(\lambda)$ by the expectation formula. For other links (Huang 2017), one directly computes μ by the link inverse (e.g., $\mu = e^\eta$ for link "log"), and then one may solve for $\lambda = \lambda(\mu)$ to obtain other features of the distribution.

The relationships between λ and μ or other moments of the distribution involve infinite summations. These sums can be easily approximated by a finite number of terms for large nu but not when nu approaches zero. For this reason, the code may fail to fit distributions with nu approaching 0 (strong residual over-dispersion). The case nu=0 (the geometric distribution) is fitted by an ad hoc algorithm devoid of such problems. Otherwise, spaMM truncates the sum, and uses numerical integrals to approximate missing terms (which slows down the fitting operation). In addition, it applies an ad hoc continuity correction to ensure continuity of the result in nu=1 (Poisson case). These corrections affect numerical results for the case of residual overdispersion but are negligible for the case of residual underdispersion. Alternatively, spaMM uses Gaunt et al.'s (2017) approximations when the condition defined by spaMM.getOption("CMP_asympto_cond") is satisfied. All approximations reduces the accuracy of computations, in a way that can impede the extended Levenberg-Marquardt algorithm sometimes needed by spaMM.

The name COMP_nu should be used to set initial values or bounds on nu in control arguments of the fitting functions (e.g., fitme(.,init=list(COMP_nu=1))). Fixed values should be set by the family argument (COMPoisson(nu=.)).

Value

A family object.

References

Gaunt, Robert E. and Iyengar, Satish and Olde Daalhuis, Adri B. and Simsek, Burcin. (2017) An asymptotic expansion for the normalizing constant of the Conway–Maxwell–Poisson distribution. Ann Inst Stat Math doi:10.1007/s1046301706296.

Huang, Alan (2017) Mean-parametrized Conway-Maxwell-Poisson regression models for dispersed counts. Stat. Modelling doi:10.1177/1471082X17697749

G. Shmueli, T. P. Minka, J. B. Kadane, S. Borle and P. Boatwright (2005) A useful distribution for fitting discrete data: revival of the Conway-Maxwell-Poisson distribution. Appl. Statist. 54: 127-142.

Sellers KF, Shmueli G (2010) A Flexible Regression Model for Count Data. Ann. Appl. Stat. 4: 943–961

Examples

```
if (spaMM.getOption("example_maxtime")>0.9) {
  # Fitting COMPoisson model with estimated nu parameter:
  #
  data("freight") ## example from Sellers & Shmueli, Ann. Appl. Stat. 4: 943-961 (2010)
  fitme(broken ~ transfers, data=freight, family = COMPoisson())
  fitme(broken ~ transfers, data=freight, family = COMPoisson(link="log"))

  # glm(), HLCor() and HLfit() handle spaMM::COMPoisson() with fixed overdispersion:
  #
  glm(broken ~ transfers, data=freight, family = COMPoisson(nu=10))
  HLfit(broken ~ transfers+(1|id), data=freight, family = COMPoisson(nu=10),method="ML")

  # Equivalence of poisson() and COMPoisson(nu=1):
  #
  COMPglm <- glm(broken ~ transfers, data=freight, family = poisson())
  coef(COMPglm)
  logLik(COMPglm)
  COMPglm <- glm(broken ~ transfers, data=freight, family = COMPoisson(nu=1))
  coef(COMPglm)
  logLik(COMPglm)
  HLfit(broken ~ transfers, data=freight, family = COMPoisson(nu=1))

}
```

`composite-ranef` *Composite random effects*

Description

An example of a composite random effect is `corrMatrix(sex|pair)`. It combines features of a random-coefficient model `(sex|pair)` and of an autocorrelated random effect `corrMatrix(.|.)`. The random-coefficient model is characterized by a $2 * 2$ covariance matrix $\mathbf{C}$ for the random effects $u_{1,pair}$ and $u_{2,pair}$ both affecting each of the two sexes for each `pair`, and the `corrMatrix`

random effect assumes that elements of each of the two vectors $u_i = (u_{i,pair})$ for pair=1,...,P are correlated according to a given $P * P$ correlation matrix **A**. Then the composite random effect is defined as the one with $2P*2P$ covariance matrix kronecker(**C**,**A**).

The definition of composite random effects through the kronecker product may be motivated and understood in light of a quantitative-genetics application (see help("Gryphon") for an example). In this context the two response variables are typically two individual traits. Each trait is affected by two sets of genes, the effect of each set being represented by a gaussian random effect (u_1 or u_2). The effect of genetic relatedness on the correlation of random effects u_i,ID among individuals ID within each set i of genes is described by the corrMatrix **A**. The effects on the two traits for each individual are interpreted as different linear combinations of these two random effects (the coefficients of these linear combinations determining the **C** matrix). Under these assumptions the correlation matrix of the responses (in order (trait, individual)=(1,1)...(1,ID)... (2,1)...(2,ID)...) is indeed kronecker(**C**,**A**).

Composite random effects are not restricted to corrMatrix terms, and can also be fitted for multivariate-response models. For example, Matern(mv(1,2)|longitude+latitude) terms can be fitted, in which case the correlation model is still defined through the Kronecker product, where **A** will be a (fitted) Matérn correlation matrix, and **C** will be the correlation matrix of the fitted random-coefficient model for the mv virtual factor for multivariate response.

Implementation and tests of composite random effects is work in progress, with the following ones having been tested: corrMatrix, Matern, Cauchy, adjacency, IMRF, AR1, and to a lesser extent MaternIMRFa. Fits of other composite terms may fail. Even if they succeed, not all post-fit procedures may be operational: in particular, prediction (and then, simulation) with newdata may not yet work. Further, as for random-coefficient terms in univariate-response models, some components of the computed prediction variance depend on a poorly characterized approximation, yielding different results for different fitting algorithms (see Details in predVar).

The summary of the model provides fitted parameters for **A** if this matrix derives from a parametric correlation model (e.g., Matern), and a description of the **C** matrix where it is viewed as a covariance matrix, parametrized by its variances and its correlation coefficient(s). In a standard random-coefficient model these variances would be those of the correlated random effects (see summary.HLfit). In the composite random-effect model this is not necessarily so as the variance of the correlated random effects also depend on the variances implied by the **A** matrix, which are not necessarily 1 if **A** is a covariance matrix rather than simply a correlation matrix.

A <prefix>(<LHS>|<RHS>) term is *not* a composite random effect when the LHS in boolean, a factor from boolean, or "0+numeric". See the Matérn examples, and the corrMatrix "<LHS> is 0+numeric" example, below.

Examples

```
if (spaMM.getOption("example_maxtime")>1.8) {

########################
#### corrMatrix examples
########################

## Toy data preparation

data("blackcap")
toy <- blackcap
```

```
toy$ID <- gl(7,2)
grp <- rep(1:2,7)
toy$migStatus <- toy$migStatus +(grp==2)
toy$loc <- rownames(toy) # to use as levels matching the corrMatrix dimnames

toy$grp <- factor(grp)
toy$bool <- toy$grp==1L
toy$boolfac <- factor(toy$bool)
toy$num <- seq(from=1, to=2, length.out=14)

## Build a toy corrMatrix as perturbation of identity matrix:
n_rhs <- 14L
eps <- 0.1
set.seed(123)
rcov <- ((1-eps)*diag(n_rhs)+eps*rWishart(1,n_rhs,diag(n_rhs)/n_rhs)[,,1])
# eigen(rcov)$values
colnames(rcov) <- rownames(rcov) <- toy$loc # DON'T FORGET NAMES


##### Illustrating the different LHS types

### <LHS> is logical (TRUE/FALSE) => No induced random-coefficient C matrix;
#   corrMatrix affects only responses for which <LHS> is TRUE:
#
(fit1 <- fitme(migStatus ~ bool + corrMatrix(bool|loc), data=toy, corrMatrix=rcov))
#
# Matrix::image(get_ZALMatrix(fit1))


### <RHS> is a factor built from a logical => same a 'logical' case above:
#
(fit2 <- fitme(migStatus ~ boolfac + corrMatrix(boolfac|loc), data=toy, corrMatrix=rcov))
#
# Matrix::image(get_ZALMatrix(fit2))


### <RHS> is a factor not built from a logical:
# (grp|.) and (0+grp|.) lead to equivalent fits of the same composite model,
#   but contrasts are not used in the second case and the C matrices differ,
#   as for standard random-coefficient models.
#
(fit1 <- fitme(migStatus ~ grp +  corrMatrix(grp|loc), data=toy, corrMatrix=rcov))
(fit2 <- fitme(migStatus ~ grp +  corrMatrix(0+grp|loc), data=toy, corrMatrix=rcov))
#
# => same fits, but different internal structures:
Matrix::image(fit1$ZAlist[[1]]) # (contrasts used)
Matrix::image(fit2$ZAlist[[1]]) # (contrasts not used)
# Also compare ranef(fit1) versus ranef(fit2)
#
#
## One can fix the C matrix, as for standard random-coefficient terms
#
(fit1 <- fitme(migStatus ~ grp +  corrMatrix(0+grp|loc),data=toy, corrMatrix=rcov,
```

```
               fixed=list(ranCoefs=list("1"=c(1,0.5,1)))))
#
# same result without contrasts hence different 'ranCoefs':
#
(fit2 <- fitme(migStatus ~ grp +  corrMatrix(grp|loc), data=toy, corrMatrix=rcov,
               fixed=list(ranCoefs=list("1"=c(1,-0.5,1)))))


### <LHS> is numeric (but not '0+numeric'):
# composite model with C being 2*2 for Intercept and numeric variable
#
(fitme(migStatus ~ num +  corrMatrix(num|loc), data=toy, corrMatrix=rcov))

### <LHS> is 0+numeric: no random-coefficient C matrix
#  as the Intercept is removed, but the correlated random effects
#  arising from the corrMatrix are multiplied by sqrt(<numeric variable>)
#
(fitme(migStatus ~ num +  corrMatrix(0+num|loc), data=toy, corrMatrix=rcov))


### <LHS> for multivariate response (see help("Gryphon") for more typical example)
## More toy data preparation for multivariate response
ch <- chol(rcov)
set.seed(123)
v1 <- tcrossprod(ch,t(rnorm(14,sd=1)))
v2 <- tcrossprod(ch,t(rnorm(14,sd=1)))
toy$status <- 2*v1+v2
toy$status2 <- 2*v1-v2

## Fit:
fitmv(submodels=list(mod1=list(status ~ 1+ corrMatrix(0+mv(1,2)|loc)),
                     mod2=list(status2 ~ 1+ corrMatrix(0+mv(1,2)|loc))),
      data=toy, corrMatrix=rcov)

##################################################
#### Matern examples: sex-dependent random effects
##################################################

if (spaMM.getOption("example_maxtime")>2) {
data(Leuca)
subLeuca <- Leuca[c(1:10,79:88),] # subset for faster examples

# The random effects in the following examples are composite because 'sex' is not
# boolean nor factor from boolean. If 'Matern(factor(female)|x+y)' were used, the effect
# would be the same 'Matern(female|x)', fitting

  fitme(fec_div ~ 1 + Matern(sex|x+y),data=subLeuca)  # => builds a random effect
# correlated across sexes, from 2 independent realizations u_1 and u_2 of 20 values
# (for the 20 locations in the data). In a (sex|x) term the 20 values would be
# independent from each other within each u_i. In the Matern(sex|x+y) such 20 values
# are autocorrelated within each u_i.

# For pedagogic purposes, one can also fit
```

```
  fitme(fec_div ~ 1 + Matern(sex|x + y %in% sex),data=subLeuca)
# which again builds a random effect from 2 independent realizations
# u_1 and u_2, but each u_i now contains two realizations u_i1 and u_i2 of 10 values,
# autocorrelated within each u_ij following the Matern model,
# but independent between u_i1 and u_i2. As a result, the overall random effect in each sex,
# v_m or v_f, is a weighted sum of two sex-specific Matern random effect,
# so that v_m and v_f are independent from each other.
}

}
```

`confint.HLfit` *Confidence intervals*

Description

This function interfaces two procedures: a profile confidence interval procedure implemented for fixed-effects coefficients only; and a parametric bootstrap procedure that can be used to provide confidence interval for any parameter, whether a canonical parameter of the model or any function of one or several such parameters. The bootstrap is performed if the `parm` argument is a function or a quoted expression or if the `boot_args` argument is a list. The profile confidence interval is computed if neither of these conditions is true. In that case `parm` must be the name(s) of some **fixed-effect** coefficient, and the (`p_v` approximation of the) profile likelihood ratio for the given parameter is used to define the interval, where the profiling is over all other fitted parameters, including other fixed-effects coefficients, as well as variances of random effects and spatial correlations if these were fitted.

Of related interest, see `numInfo` which evaluates numerically the information matrix for given sets of canonical model parameters, from which asymptotic confidence intervals can be deduced.

Usage

```
## S3 method for class 'HLfit'
confint(object, parm, level=0.95, verbose=TRUE,
                          boot_args=NULL, format="default", ...)
```

Arguments

`object`	An object of class `HLfit`, as returned by the fitting functions in `spaMM`. This includes objects also inheriting from class `pois4mlogit`.
`parm`	character vector, integer vector, or function, or a quoted expression. If **character**, the name(s) of parameter(s) to be fitted; if **integer**, their position in the `fixef(object)` vector. Valid names are those of this vector. If a **function**, it must return a (vector of) parameter estimate(s), from a fit object as single argument. If a **quoted expression**, it must likewise extract parameter estimate(s) from a fit object; this expression must refer to the fitted object as 'hlfit' (see Examples).
`level`	The coverage of the interval.

verbose	whether to print the interval or not. As the function returns its more extensive results invisibly, this printing is the only visible output. In some cases, this argument may also control the verbosity of fits performed by confint to find the bound of the intervals: see verbose for details.
boot_args	NULL or a list of arguments passed to functions spaMM_boot and boot.ci. It must contain element nsim (for spaMM_boot). The type argument of boot.ci can only be given as element ci_type, to avoid conflict with the type argument of spaMM_boot.
format	Only effective non-default value is "stats" to return results in the format of the stats::confint result (see Value).
...	Additional arguments, which may be ignored and have no currently suggested use, apart from any exception described in the Details (notably progress for pois4mlogit fits).

Details

The ... arguments may have an effect (1) in the basic LRT procedure, only when the input object inherits from class pois4mlogit (being a fit by the so-named function). In this case the ... are useful to pass a progress value down to the pois4mlogit function; (2) in the bootstrap procedure, where the ... are ultimately passed to the fitting procedure applied to each bootstrap sample (but there is no suggested use of this feature).

Value

The format of the value varies, but in all cases distinguished below, one or more tables are included, as a table attribute, in the format of the stats::confint result, to facilitate consistent extraction of results. By default confint returns invisibly the full values described below, but if format="stats", only the table attribute is returned.

If a profile CI has been computed for a single parameter, a list is returned including the confidence interval as shown by verbose=TRUE, and the fits lowerfit and upperfit giving the profile fits at the confidence bounds. This list bears the table attribute.

If a profile CI has been computed for several parameters, a structured list, named according to the parameter names, of such single-parameter results is returned, and a single table attribute for all parameters is attached to the upper level of the list.

If a bootstrap was performed, for a single parameter the result of the boot.ci call is returned, to which a table attribute is added. This table is now a list of tables for the different bootstrap CI types (default being normal, percent, and basic), each such table in the format of the stats::confint results. For several parameters, a named list of boot.ci results is returned, its names being the parameter names, and the table attribute is attached to the upper level of the list.

The boot.ci return value for each parameter includes the call to boot.ci. This call is typically shown including a long t vector, which makes a bulky display. spaMM hacks the printing to abbreviate long ts.

See Also

numInfo for information matrix.

Examples

```
data("wafers")
wfit <- HLfit(y ~X1+(1|batch), family=Gamma(log), data=wafers, method="ML")
confint(wfit,"X1")  # profile CI
if (spaMM.getOption("example_maxtime")>30) {

   # bootstrap CI induced by 'boot_args':
   confint(wfit,names(fixef(wfit)), boot_args=list(nsim=99, seed=123))

   # bootstrap CI induced by 'parm' being a function:
   confint(wfit,parm=function(v) fixef(v),
           boot_args=list(nb_cores=10, nsim=199, seed=123))

   # Same effect if 'parm' is a quoted expression in terms of 'hlfit':
   confint(wfit,parm=quote(fixef(hlfit)),
           boot_args=list(nb_cores=10, nsim=199, seed=123))

   # CI for the variance of the random effect:
   ci <- confint(wfit,parm=function(fit){get_ranPars(fit)$lambda[1]},
        boot_args=list(nb_cores=10, nsim=199, seed=123))
   # The distribution of bootstrap replicates:
   plot(ecdf(ci$call$t))
   # We may be far from ideal condition for accuracy of bootstrap intervals;
   # for variances, a log transformation may sometimes help, but not here.

   # Passing arguments to child processes, as documented in help("spaMM_boot"):
   set.seed(123)
   rvar <- runif(nrow(wafers))
   wrfit <- fitme(y ~X1+(1|batch), family=Gamma(log), data=wafers, fixed=list(phi=rvar))
   confint(wrfit, parm = "(Intercept)", boot_args = list(nsim = 100, nb_cores = 2,
           fit_env = list(rvar=rvar)))

}
```

`control.HLfit` *Control parameters of the HLfit fitting algorithm*

Description

A list of parameters controlling the `HLfit` fitting algorithm (called by all fitting functions in **spaMM**). It can mostly be ignored in routine use.

Usage

```
# <fitting function>(., control.HLfit=list(...))
```

Details

Possible controls are:

- `algebra`, `sparse_precision`: see `algebra`.
- `max.iter`: the number of iterations of the iterative algorithm for joint estimation of dispersion parameters and of coefficients of the linear predictor. Defaults to 200. This is typically much more than necessary, unless there is little information to separately estimate λ and ϕ parameters.
- `conv.threshold` and `spaMM_tol`: `spaMM_tol` is a list of tolerance values, with elements `Xtol_rel` and `Xtol_abs` that define thresholds for relative and absolute changes in parameter values in iterative algorithms (used in tests of the form "d(param)< Xtol_rel * param + Xtol_abs", so that `Xtol_abs` is operative only for small parameter values). `conv.threshold` is the older way to control `Xtol_rel`. Default values are given by `spaMM.getOption("spaMM_tol")`.
- `min_phi`: NULL or a positive numeric value, being the minimum value of the residual dispersion parameter ϕ, when "inner-estimated" (`inner-vs-outer` for this concept). Values approaching zero may induce numerical errors in various steps of the fitting algorithm. For Gaussian- or Gamma-family models, ϕ values as low as `1e-12` are usually not a source of concern; but setting `min_phi` to higher values has been needed at least for fits with an experimental implementation of the `tweedie` family.
- `break_conv_logL`: a boolean specifying whether the iterative algorithm should terminate when log-likelihood appears to have converged (roughly, when its relative variation over on iteration is lower than 1e-8). Default is FALSE (convergence is then assessed on the parameter estimates rather than on log-likelihood).
- `iter.mean.dispFix`: the number of iterations of the iterative algorithm for coefficients of the linear predictor, if no dispersion parameters are estimated by the iterative algorithm. Defaults to 200 except for Gamma(log)-family models.
- `iter.mean.dispVar`: the number of iterations of the iterative algorithm for coefficients of the linear predictor, if some dispersion parameter(s) is estimated by the iterative algorithm. Defaults to 50 except for Gamma(log)-family models.

`resid.family` was a previously documented control (before version 2.6.40), and might still operate as previously documented, but should not be used in new code.

`convergence` *Assessing convergence for fitted models*

Description

spaMM fits can produce various convergence warnings or messages.

Messages referring to convergence issues in initialization can generally be ignored but may help to diagnose other apparent problems, if any.

Warnings referring to `.calc_dispGammaGLM` (for residual-dispersion fits) can generally be ignored when they show a small criterion (say <1e-5) but may otherwise suggest that the final fit did not optimize its objective.

Messages about slow convergence and drawing users to this documentation do not necessarily mean the fit is incorrect. Rather, they suggest that another fitting strategy could be tried. Keep in mind that several parameters (notably the dispersion parameters: the variance of random effects and the

residual variance parameter, if any) can be estimated either by the iterative algorithms, or by generic optimization methods. In my experience, slow convergence happens in certain cases where a large random-effect variance is considered by the algorithm used.

How to know which algorithm has been selected for each parameter? `fitme(., verbose=c(TRACE=TRUE))` shows successive values of the variables estimated by optimization (See Examples; if no value appears, then all are estimated by iterative methods). The first lines of the summary of a fit object should tell which variances are estimated by the "outer" method.

If the iterative algorithm is being used, then it is worth trying to use the generic optimization methods. In particular, if you used `HLfit`, try using `fitme`; if you already use `fitme`, try to enforce generic optimization of the random-effect variance(s) (see `inits`). Conversely, if generic optimization is being used, the maximum lambda value could be controlled (say, `upper=list(lambda=c(10,NA))`), or the iterative algorithm can be called (see `inits` again).

For the largest datasets, it may be worth comparing the speed of the `"spcorr"` and `"spprec"` choices of the `algebra` control, in case spaMM has not selected the most appropriate by default. However, this will not be useful for geostatistical models with many spatial locations.

If nothing else works and you are not afraid of long fitting times, you may try `try control.HLfit=list(LevenbergM=TRUE)` or increasing `control.HLfit$max.iter.mean`.

Examples

```
# See help("inits") for examples of control by initial values.
```

`corMatern` *Matern Correlation Structure as a corSpatial object*

Description

This implements the Matérn correlation structure (see `Matern`) for use with `lme` or `glmmPQL`. Usage is as for others `corSpatial` objects such as `corGaus` or `corExp`, except that the Matérn family has an additional parameter. This function was defined for comparing results obtained with `corrHLfit` to those produced by `lme` and `glmmmPQL`. There are problems in fitting (G)LMMs in the latter way, so it is not a recommended practice.

Usage

```
corMatern(value = c(1, 0.5), form = ~1, nugget = FALSE, nuScaled = FALSE,
          metric = c("euclidean", "maximum", "manhattan"), fixed = FALSE)
```

Arguments

`value`	An optional vector of parameter values, with serves as initial values or as fixed values depending on the `fixed` argument. It has either two or three elements, depending on the `nugget` argument. If `nugget` is `FALSE`, `value` should have two elements, corresponding to the "range" and the "smoothness" ν of the Matérn correlation structure. If `value` has zero length, the default is a range of 90% of the minimum distance and

a smoothness of 0.5 (exponential correlation). **Warning**: the range parameter used in `corSpatial` objects is the inverse of the scale parameter used in `MaternCorr` and thus they have opposite meaning despite both being denoted ρ elsewhere in this package or in `nlme` literature.

If `nugget` is `TRUE`, meaning that a nugget effect is present, `value` can contain two or three elements, the first two as above, the third being the "nugget effect" (one minus the correlation between two observations taken arbitrarily close together). If `value` has length zero or two, the nugget defaults to 0.1. The range and smoothness must be greater than zero and the nugget must be between zero and one.

`form` (Pasted from corSpatial) a one sided formula of the form `~ S1+...+Sp`, or `~ S1+...+Sp | g`, specifying spatial covariates `S1` through `Sp` and, optionally, a grouping factor `g`. When a grouping factor is present in `form`, the correlation structure is assumed to apply only to observations within the same grouping level; observations with different grouping levels are assumed to be uncorrelated. Defaults to `~ 1`, which corresponds to using the order of the observations in the data as a covariate, and no groups.

`nugget` an optional logical value indicating whether a nugget effect is present. Defaults to `FALSE`.

`nuScaled` If `nuScaled` is set to `TRUE` the "range" parameter ρ is divided by $2\sqrt{\nu}$. With this option and for large values of ν, `corMatern` reproduces the calculation of `corGaus`. Defaults to `FALSE`, in which case the function compares to `corGaus` with range parameter $2(\sqrt{\nu})\rho$ when ν is large.

`metric` (Pasted from corSpatial) an optional character string specifying the distance metric to be used. The currently available options are `"euclidean"` for the root sum-of-squares of distances; `"maximum"` for the maximum difference; and `"manhattan"` for the sum of the absolute differences. Partial matching of arguments is used, so only the first three characters need to be provided. Defaults to `"euclidean"`.

`fixed` an optional logical value indicating whether the coefficients should be allowed to vary in the optimization, or kept fixed at their initial value. Defaults to `FALSE`, in which case the coefficients are allowed to vary.

Details

This function is a constructor for the `corMatern` class, representing a Matérn spatial correlation structure. See `MaternCorr` for details on the Matérn family.

Value

an object of class `corMatern`, also inheriting from class `corSpatial`, representing a Matérn spatial correlation structure.

Note

The R and C code for the methods for `corMatern` objects builds on code for `corSpatial` objects, by D.M. Bates, J.C. Pinheiro and S. DebRoy, in a circa-2012 version of nlme.

References

Mixed-Effects Models in S and S-PLUS, José C. Pinheiro and Douglas M. Bates, Statistics and Computing Series, Springer-Verlag, New York, NY, 2000.

See Also

glmmPQL, lme

Examples

```
## LMM
data("blackcap")
blackcapD <- cbind(blackcap,dummy=1) ## obscure, isn't it?
## With method= 'ML' in lme, The correlated random effect is described
##  as a correlated residual error and no extra residual variance is fitted:
nlme::lme(fixed = migStatus ~ means, data = blackcapD, random = ~ 1 | dummy,
    correlation = corMatern(form = ~ longitude+latitude  | dummy),
    method = "ML", control=nlme::lmeControl(sing.tol=1e-20))

## Binomial GLMM
if (spaMM.getOption("example_maxtime")>32) {
 data("Loaloa")
 LoaloaD <- cbind(Loaloa,dummy=1)
 MASS::glmmPQL(fixed =cbind(npos,ntot-npos)~elev1+elev2+elev3+elev4+maxNDVI1+seNDVI,
        data = LoaloaD, random = ~ 1 | dummy,family=binomial,
        correlation = corMatern(form = ~ longitude+latitude | dummy))
}
```

---

corrFamily *Using corrFamily constructors and descriptors.*

Description

One can declare and fit correlated random effects belonging to a user-defined correlation (or covariance) model (i.e., a parametric family of correlation matrices, although degenerate case with no parameter are also possible). This documentation is a first introduction to this feature. It is experimental in the sense that its design has been formalized only from a limited number of corrFamily examples, and that the documentation is not mature. Implementing prediction for random-effects defined in this way may be tricky. A distinct documentation corrFamily-design provides more information for the efficient design of new correlation models to be fitted in this way.

A simple example of random-effect model implemented in this way is the autoregressive model of order p (AR(p) in the literature; specifically documented elsewhere, see ARp). It can be used as a formula term like other autocorrelated random-effects predefined in **spaMM**, to be fitted by `fitme` or `fitmv`:

```
fitme(lh ~ 1 + ARp(1|time, p=3),  # <= declaration of random effect
  < data and other possible arguments >)
```

User-defined correlation models should be registered for this simple syntax to work (see Details for an alternative syntax):

```
myARp <- ARp                    # 'myARP' is thus a user-defined model
register_cF("myARp")        # Register it so that the next call works
fitme(lh ~ 1 + myARp(1|time, p=3),
  < data and other possible arguments >)
```

The `ARp` object here copied in `myARp` is a function (the *corrFamily constructor*) which returns a `list` (the *corrFamily descriptor*) which contains the necessary information to fit a random effect with an AR(p) correlation. The `p` argument in the `myARp(1|time, p=3)` term enforces evaluation of `myARp(p=3)`, producing the descriptor for the AR(3) model. The structure of this descriptor is

```
List of 5
 $ Cf            :function (parvec)
  ..- < with some attribute >
 $ tpar          : num [1:3] 0.5 0.333 0.25
 $ type          : chr "precision"
 $ initialize    :function (Zmatrix, ...)
  ..- < with some attribute >
 $ fixed         : NULL
 $ calc_moreargs:function (corrfamily, ...)
  ..- < with some attribute >
 $ levels_type        : chr "time_series"
 $ calc_corr_from_dist:function (ranFix, char_rd, distmat, ...)
  ..- < with some attribute >
 < and possibly other elements >
```

The meaning of these elements and some additional ones is explained below.

Only `Cf` and `tpar` are necessary elements of a corrFamily object. If one designs a new descriptor where some other elements are absent, **spaMM** will try to provide plausible defaults for these elements. Further, if the descriptor does not provide parameter names (as the names of `tpar`, or in some more cryptic way), default names `"p1"`, `"p2"`... will be provided.

Usage

```
## corrFamily descriptor provided as a list of the form
#
# list(Cf=<.>, tpar=<.>, ...)

## corrFamily constructor: any function that returns
#    a valid corrFamily descriptor
#
# function(tpar=<.>, fixed=<.>, ...) # typical but not mandatory arguments

## There is a distinct documentation page for 'register_cF'.
```

Arguments

Elements* of the corrFamily *descriptor:

`Cf`	(required): function returning the correlation matrix (or covariance matrix, or their inverse), given its first argument, a parameter vector.
`tpar`	(required): a feasible argument of `Cf`. `tpar` is **not** an initial **nor** a fixed value.
`type`	optional, but required if the return value of `Cf` is an inverse correlation matrix rather than a correlation matrix, in which case one should specify `type="precision"`.
`fixed`	optional: fixed values for some correlation parameters, provided as a named vector with names consistent with those to be used for `tpar`. This is conceived to achieve the same statistical fit as by using the `fixed` argument of `fitme`, although the structure of the result of the fit differs in some subtle ways whether parameters are fixed through the descriptor or through the fitting function (see Examples in `ARp`).
`calc_moreargs`	optional: a function returning a list with possible elements `init`, `lower` and `upper` for control of estimation (and possibly other elements for other purposes). If the descriptor does not provide this function, a default `calc_moreargs` will be provided, implementing unbounded optimization.
`initialize`	optional: a function evaluating variables that may be needed repeatedly by `Cf` or `Af`.
`Af`	This function should be defined if the correlation model requires an **A** matrix (the middle term in the case the design matrix of a random effect term is described by a triple matrix product **ZAL** as described in `random-effects`). Examples can be found in the descriptors returned by the `ranGCA` and `MaternIMRFa` constructors.
`levels_type`	In the above example its value `"time_series"` informs **spaMM** that levels of the random effect should be considered for all integer values within the range of the `time` variable, not only for levels present in the data. If this element is not provided by the constructor, **spaMM** will internally assume a `levels_type` suitable for geostatistical models. Further level types may be defined in the future.
`calc_corr_from_dist`, `make_new_corr_lists`	Functions possibly needed for prediction (see Details).
`need_Cnn`	optional: a boolean; default is TRUE. Controls prediction computations (see Details).
`public`	An environment where some variables can be saved, typically by the `initialize` expression, for inspection at user level and for re-use. See `diallel` for an example.

***fitting-function* arguments**:

`lower`, `upper`, `init` and `fixed` optimization controls can be used to control optimization of continuous parameters as for other random-effect parameters. They are specified as numeric vectors, themselves being element of the `corrPars` list (see Examples in `corrFamily-design`). Parameter names (consistent with those to be used for the `tpar` argument) may be required to disambiguate incomplete vectors (e.g., to specify only its second element). Apart from `fixed` ones, any of the

values not specified through the fitting-function arguments will be sought in the return value of the `calc_moreargs` function, if provided in the descriptor. If the `lower` or `upper` information is missing there, it must be provided throught the fitting-function call. If the `init` information is missing, a default value will be deduced from the bounds. The `init` specification is thus always optional while the bounds specification is optional only if the descriptor provides default values.

Arguments* of the corrFamily *constructor

These may be ad libitum, as design rules are defined only for the returned descriptor. However, arguments `tpar`, `fixed`, and `public` of predefined constructors, such as `ARp`, are designed to match the above respective elements of the descriptor.

Details

*** Constructor elements for prediction:**

For prediction of autocorrelated random effects, one must first assess whether levels of the random effect not represented in the fit are possible in new data (corresponding to new spatial locations in geostatistical models, or new time steps in time series). In that case `need_Cnn` must be TRUE (Interpolated MRFs do not require this as all required random-effect levels are determined by the IMRF `mesh` argument rather than by the fitted data or new data).

Next, for autocorrelated random effects where `need_Cnn` is TRUE, a `make_new_corr_lists` function must be provided, except when a `calc_corr_from_dist` function is instead provided (which may be sufficient for models that construct the correlation from a spatial distance matrix). When `need_Cnn` is FALSE, a `make_new_corr_lists` function may still be needed.

The Examples section provides a simple example of such design, and the source code of the `ARp` or `ARMA` constructors provide further examples. They show that the `make_new_corr_lists` function may assign matrices or vectors as elements of several lists contained in a `newLv_env` environment. A matrix is assigned in the `cov_newLv_oldv_list`, specifying correlations between "new" levels of the random effect (implied by the new data) and "old" levels (those already included in the design matrix of the random effect for the fit). If `need_Cnn` is TRUE, a second matrix may be assigned in the `cov_newLv_newLv_list`, specifying correlation between "new" levels, and the diagonal of this matrix is assigned in the `diag_cov_newLv_newLv_list`. The overall structure of the code (the conditions where these assignments are performed, and the list indices), should be conserved.

When calling `simulate(.,` `newdata=<non-NULL>,` `type="marginal"`, a fourth matrix may be useful, assigned into a `L_newLv_newLv_list`, specifying the matrix root (as a `tcrossprod` factor) of the correlation matrix stored in `cov_newLv_newLv_list`. The relevant **spaMM** procedure will however try to compute it on the fly when it has not been provided by the `make_new_corr_lists` function.

*** Fitting a `corrFamily` without a constructor:**

It is possible to use an unregistered corrFamily, as follows:

```
AR3 <- ARp(p=3)          # Generate descriptor of AR model of order 3

fitme(lh ~ 1 + corrFamily(1|time),  # <= declaration of random effect
  covStruct=list(
    corrFamily= AR3     # <= declaration of its correlation structure
  ),
  < data and other possible arguments >)
```

Here the fit only uses a descriptor list, not a constructor function. This descriptor is here provided to the fitting function as an element of the covStruct argument (using the general syntax of this argument), and in the model formula the corresponding random effect is declared as a term of the form
corrFamily(1|<grouping factor>).

This syntax is more complex than the one using a registered constructor, but it might be useful for development purposes (one only has to code the descriptor, not the constructor function). However, it is not general; in particular, using registered constructors may be required to obtain correct results when fitting multivariate-response models by fitmv.

See Also

See ARp, diallel, and MaternIMRFa for basic examples of using a predefined corrFamily descriptor, and corrFamily-design for more geeky stuff including examples of implementing simple new correlation families.

Examples

```
## Not run:
### Minimal (with many features missing) reimplementation
#     of corrMatrix() terms as a corrFamily


corrMatrix_cF <- function(corrMatrix) {

  force(corrMatrix) # Makes it available in the environment of the functions next defined.
  oldZlevels <- NULL

  initialize <- function(Zmatrix, ...) {
   oldZlevels <<- colnames(Zmatrix) # Pass info about levels of the random effect in the data.
  }

  Cf <- function(newlevels=oldZlevels ) {
    if (length(newlevels)) {
      corrMatrix[newlevels,newlevels]
    } else corrMatrix[oldZlevels,oldZlevels] # for Cf(tpar=numeric(0L))
  }

  calc_moreargs <- function(corrfamily, ...) {
    list(init=c(),lower=c(),upper=c())
  }

  make_new_corr_lists <- function(newLv_env, which_mats, newZAlist, new_rd, ...) {
    newlevels <- colnames(newZAlist[[new_rd]])
   newLv_env$cov_newLv_oldv_list[[new_rd]] <- corrMatrix[newlevels,oldZlevels, drop=FALSE]
    if (which_mats$nn[new_rd]) {
    newLv_env$cov_newLv_newLv_list[[new_rd]] <- corrMatrix[newlevels,newlevels, drop=FALSE]
    } else {
      newLv_env$diag_cov_newLv_newLv_list[[new_rd]] <- rep(1,length(newlevels))
    }
  }
```

```
  list(Cf=Cf, tpar=numeric(0L), initialize=initialize, calc_moreargs=calc_moreargs,
       make_new_corr_lists=make_new_corr_lists,
       tag="corrMatrix_cF")
}

register_cF("corrMatrix_cF")

# usage:

data("blackcap")
MLcorMat <- MaternCorr(proxy::dist(blackcap[,c("latitude","longitude")]),
                       nu=0.6285603,rho=0.0544659)
corrmat <- proxy::as.matrix(MLcorMat, diag=1)

fitme(migStatus ~ means+ corrMatrix_cF(1|name, corrMatrix=corrmat),data=blackcap,
      corrMatrix=MLcorMat,method="ML")

unregister_cF("corrMatrix_cF") # Tidy things before leaving.


## End(Not run)
```

`corrFamily-definition` *corrFamily definition*

Description

Tentative formal rules for definition of corrFamily descriptors (work in progress). This is likely to repeat and extend information partially given in `corrFamily` and `corrFamily-design` documentations.

User-level rules (not relevant fo corrFamily descriptors internally modified during a fit):

tpar Should always be present. For trivial parameterless cases (e.g. `ranGCA`), it should be `numeric(0L)`, not `NULL`.

Cf function; should always be present. For trivial uncorrelated random effects (e.g. `ranGCA`, where only the `Af` function carries the information for the model), it should return an identity matrix, not `NULL`, with row names to be matched to the column names of the **Z** matrix for the random effect.

calc_moreargs optional function. If present, it should have formal arguments including at least `corrfamily` and ....

Af function; optional. If present, it should have row names to be matched to the column names of the **Z** matrix for the random effect, and also needs column names if it is to be matched with the row names of a correlation matrix (or its inverse).

initialize Optional function. If present, should have formal arguments including at least `Zmatrix` and ....

In predefined corrFamily constructors, variables created by `initialize` for use by `Cf` or `Af` should be declared (typically as `NULL`) in the body of the constructor, so that R CMD check does not complain.

public An environment. `initialize` may write into it. It might also read into it, for example read the result of a long previous computation by `initialize` during a previous fit, though this opens the door to various errors.

`corrFamily-design` *Designing new corrFamily descriptors for parametric correlation families*

Description

This documentation describe additional design features to be taken into account when defining a new `corrFamily` descriptor for a correlation model. Using such a descriptor will be more efficient than the equally general method, of maximizing an objective function of the correlation parameters that calls (say) `fitme()` on a model including a `corrMatrix` itself function of the correlation parameters. But this may still be inefficient if a few issues are ignored.

For elements of the corrFamily descriptor for basic cases:

Cf The function value should (1) be of constant class for all parameter values. For families of mathematically sparse matrices, the `CsparseMatrix` class is recommended (and more specifically the `dsCMatrix` class since the matrix is symmetric); (2) have row names that match the levels of the grouping factor (the nested random effect Example shows the code needed when this nested effect is defined from two variables).

tpar In order to favor the automatic selection of suitable algorithms, `tpar` should be chosen so that `Cf(tpar)` is **least** sparse (i.e., has the minimal number of elements equal to zero) in the correlation family, in terms of its sparsity and of the sparsity of its inverse. A `tpar` yielding an identity matrix is often a ***bad*** template as least sparse correlation matrices and their inverses are denser for most families except diagonal ones. For degerate corrFamily objects that describe a constant correlation model rather than a parametric family, use `tpar=numeric(0)`.

type Do not forget `type="precision"` it if the return value of `Cf` is an inverse correlation matrix rather than a correlation matrix, in which case one should specify .

calc_moreargs should have formal arguments including at least `corrfamily` and `...`. The source code of `ARp`, `ARMA` or `diallel` shows the expected structure of its return value.

For advanced features of the corrFamily descriptor:

Af `Af` has (minimally) three formal arguments `(newdata, term, ...)`. **spaMM** will call `Af` with distinct values of the `newdata` argument for the fit, and for predictions for new data. For the curious: the `term` argument that will be provided by **spaMM** to `Af` is the formula term for the random effect – an object of class `call`, as obtained e.g. by `( ~ 1+ corrFamily(1 | longitude + latitude))[[2]][[3]]` –, which will provide the names of the variables that need to be taken from the `newdata` to construct the matrix returned by `Af`.

Details

- **spaMM** will regularize invalid or nearly-singular correlation or covariance matrices internally if the correlation function has not done so already, but it it better to control this in the correlation function. The `regularize` convenience function is available for that purpose, but parametrizations that avoid the need for regularization are even better, since fitting models

with nearly-singular correlation matrices is prone to various difficulties (The Toeplitz example below is good to illustrate potential problems but is otherwise poor as it produces non-positive definite matrices; the ARp constructor illustrates a parametrization that avoids that problem).

- Users should make sure that any regularized matrix still belongs to the intended parametric family of matrices, and they should keep in mind that the **spaMM** output will show the input parameters of the unregularized matrix, not the parameters of the regularized one (e.g., in the Toeplitz example below, the fitted matrix is a regularized Toepliz matrix with slightly different coefficients than the input parameters).

And for efficiency,

- Let us repeat that the correlation function should return matrices of constant class, and in sparse format when the matrices are indeed mathematically sparse. For mathematically dense matrices (as in the Toeplitz example below), the `dsyMatrix` class may be suitable.
- Let us repeat that in order to favor the automatic selection of suitable algorithms, `tpar` should be chosen so that `Cf(tpar)` is **least** sparse in the correlation family. For matrices of `CsparseMatrix`, a check is implemented to catch wrong choices of `tpar`.
- For challenging problems (large data, many parameters...) it may pay to optimize a bit the correlation function. The Example of nested effects with heterogenous variance below illustrates a possible trick. In the same cases, It may also pay to try the alternative algebraic methods, by first comparing speed of the different methods (`control.HLfit=list(algebra=<"spprec"|"spcorr"|"decorr">)`) for given correlation parameter values, rather than to assume that **spaMM** will find the best method (even if it often does so).
- The corrFamily descriptor may optionally contain booleans `possiblyDenseCorr` and `sparsePrec` to help spaMM select the most appropriate matrix algebraic methods. `sparsePrec` should be set to TRUE if sparse-precision methods are expected to be efficient for fitting the random effect. `possiblyDenseCorr` should be set to FALSE if the correlation matrix is expected to be sparse, which means here that less than 15% of its elements are non-zero.

Examples

```
if (spaMM.getOption("example_maxtime")>2 &&
      requireNamespace("agridat", quietly = TRUE)) {

data("onofri.winterwheat", package="agridat")

##### Fitting a Toeplitz correlation model for temporal correlations #####

# A Toeplitz correlation matrix of dimension d*d has d-1 parameters
# (by symmetry, and with 1s on the main diagonal). These d-1 parameters
# can be fitted as follows:

Toepfn <- function(v) {
  toepmat <- Matrix::forceSymmetric(toeplitz(c(1,v))) # dsyMatrix
  # Many of the matrices in this family are not valid correlation matrices;
  #   the regularize() function is handy here:
  toepmat <- regularize(toepmat, maxcondnum=1e12)
  # And don't forget the rownames!
  rownames(toepmat) <- unique(onofri.winterwheat$year)
  toepmat
}
```

```
(Toepfit <- spaMM::fitme(
  yield ~ gen + corrFamily(1|year), data=onofri.winterwheat, method="REML",
  covStruct=list(corrFamily=list(Cf=Toepfn, tpar=rep(1e-4,6))),
         # (Note the gentle warning if one instead uses tpar=rep(0,6) here)
  lower=list(corrPars=list("1"=rep(-0.999,6))),
  upper=list(corrPars=list("1"=rep(0.999,6)))))

# The fitted matrix is (nearly) singular, and was regularized:

eigen(Corr(Toepfit)[[1]])$values

# which means that the returned likelihood may be inaccurate,
# and also that the actual matrix elements differ from input parameters:

Corr(Toepfit)[[1]][1,-1]

### The usual rules for specifying covStruct, 'lower', 'upper' and 'init' apply
# here when the corrFamily term is the second random-effect:

(Toep2 <- spaMM::fitme(
       yield ~ 1 + (1|gen) + corrFamily(1|year), data=onofri.winterwheat, method="REML",
        covStruct=list("1"=NULL, corrFamily=list(Cf=Toepfn, tpar=rep(1e-4,6))),
        , init=list(corrPars=list("2"=rep(0.1,6))),
        lower=list(corrPars=list("2"=rep(-0.999,6))),
        upper=list(corrPars=list("2"=rep(0.999,6)))))

##### Fitting one variance among years per each of 8 genotypes. #####

# First, note that this can be *more efficiently* fitted by another syntax:

### Fit as a constrained random-coefficient model:

# Diagonal matrix of NA's, represented as vector for its lower triangle:
ranCoefs_for_diag <- function(nlevels) {
  vec <- rep(0,nlevels*(nlevels+1L)/2L)
  vec[cumsum(c(1L,rev(seq(nlevels-1L)+1L)))] <- NA
  vec
}

(by_rC <- spaMM::fitme(yield ~ 1 + (0+gen|year), data=onofri.winterwheat, method="REML",
                       fixed=list(ranCoefs=list("1"=ranCoefs_for_diag(8)))))

### Fit as a corrFamily model:

gy_levels <- paste0(gl(8,1,length =56,labels=levels(onofri.winterwheat$gen)),":",
                        gl(7,8,labels=unique(onofri.winterwheat$year)))

# A log scale is often suitable for variances, hence is used below;

# a correct but crude implementation of the model is
diagf <- function(logvar) {
  corr_map <- kronecker(Matrix::.symDiagonal(n=7),diag(x=exp(logvar)))
```

```
  rownames(corr_map) <- gy_levels
  corr_map
}

# but we can minimize matrix operations as follows:

corr_map <- Matrix::.symDiagonal(n=8,x=seq(8))
rownames(corr_map) <- unique(onofri.winterwheat$gen)

diagf <- function(logvar) {
  corr_map@x <- exp(logvar)[corr_map@x]
  corr_map
}                 # (and this returns a dsCMatrix)

(by_cF <- spaMM::fitme(
        yield ~ 1 + corrFamily(1|gen %in% year), data=onofri.winterwheat, method="REML",
        covStruct=list(corrFamily = list(Cf=diagf, tpar=rep(1,8))),
        fixed=list(lambda=1),           # Don't forget to fix this redundant parameter!
        # init=list(corrPars=list("1"=rep(log(0.1),8))), # 'init' optional
        lower=list(corrPars=list("1"=rep(log(1e-6),8))), # 'lower' and 'upper' required
        upper=list(corrPars=list("1"=rep(log(1e6),8)))))

# => The 'gen' effect is nested in the 'year' effect and this must be specified in the
# right-hand side of corrFamily(1|gen %in% year) so that the design matrix 'Z' for the
# random effects to have the correct structure. And then, as for other correlation
# structures (say Matern) it should be necessary to specify only the correlation matrix
# for a given year, as done above. Should this fail, it is also possible to specify the
# correlation matrix over years, as done below. spaMM will automatically detect, from
# its size matching the number of columns of Z, that it must be the matrix over years.

corr_map <- Matrix::forceSymmetric(kronecker(Matrix::.symDiagonal(n=7),diag(x=seq(8))))
rownames(corr_map) <- gy_levels

diagf <- function(logvar) {
  corr_map@x <- exp(logvar)[corr_map@x]
  corr_map
}                 # (and this returns a dsCMatrix)

(by_cF <- spaMM::fitme(
  yield ~ 1 + corrFamily(1|gen %in% year), data=onofri.winterwheat, method="REML",
  covStruct=list(corrFamily = list(Cf=diagf, tpar=rep(1,8))),
  fixed=list(lambda=1),           # Don't forget to fix this redundant parameter!
  # init=list(corrPars=list("1"=rep(log(0.1),8))), # 'init' optional
  lower=list(corrPars=list("1"=rep(log(1e-6),8))), # 'lower' and 'upper' required
  upper=list(corrPars=list("1"=rep(log(1e6),8)))))

exp(get_ranPars(by_cF)$corrPars[[1]]) # fitted variances

}
```

corrHLfit *Fits a mixed model, typically a spatial GLMM.*

Description

This was the first function for fitting all spatial models in **spaMM**, and is still fully functional, but it is recommended to use `fitme` which has different defaults and generally selects more efficient fitting methods, and will handle all classes of models that **spaMM** can fit, including non-spatial ones. `corrHLfit` performs the joint estimation of correlation parameters, fixed effect and dispersion parameters.

Usage

```
corrHLfit(formula, data, init.corrHLfit = list(), init.HLfit = list(),
          ranFix, fixed=list(), lower = list(),  upper = list(),
          objective = NULL, resid.model = ~1,
          control.dist = list(), control.corrHLfit = list(),
          processed = NULL, family = gaussian(), method="REML",
          nb_cores = NULL, weights.form = NULL, ...)
```

Arguments

`formula`	Either a linear model `formula` (as handled by various fitting functions) or a `predictor`, i.e. a formula with attributes (see `Predictor` and examples below). See Details in `spaMM` for allowed terms in the formula.
`data`	A data frame containing the variables in the response and the model formula.
`init.corrHLfit`	An optional list of initial values for correlation and/or dispersion parameters, e.g. `list(rho=1,nu=1,lambda=1,phi=1)` where `rho` and `nu` are parameters of the Matérn family (see `Matern`), and `lambda` and `phi` are dispersion parameters (see Details in `spaMM` for the meaning of these parameters). All are optional, but giving values for a dispersion parameter changes the ways it is estimated (see Details). `rho` may be a vector (see `make_scaled_dist`) and, in that case, it is possible that some or all of its elements are `NA`, for which `corrHLfit` substitutes automatically determined values.
`init.HLfit`	See identically named `HLfit` argument.
`fixed, ranFix`	A list similar to `init.corrHLfit`, but specifying fixed values of the parameters not estimated. `ranFix` is the old argument, maintained for back compatibility; `fixed` is the new argument, uniform across **spaMM** fitting functions. See `ranFix` for further information.
`lower`	An optional (sub)list of values of the parameters specified through `init.corrHLfit`, in the same format as `init.corrHLfit`, used as lower values in calls to `optim`. See Details for default values.
`upper`	Same as `lower`, but for upper values.
`objective`	For development purpose, not documented (this had a distinct use in the first version of spaMM, but has been deprecated as such).
`resid.model`	See identically named `HLfit` argument.

control.dist See control.dist in HLCor

control.corrHLfit

This may be used control the optimizer. See spaMM.options

for default values.

processed For programming purposes, not documented.

family Either a family or a multi value.

method Character: the fitting method to be used, such as "ML", "REML" or "PQL/L". "REML" is the default. Other possible values of HLfit's method argument are handled.

weights.form Specification of prior weights by a one-sided formula: use weights.form = ~ pw instead of prior.weights = pw. The effect will be the same except that such an argument, known to evaluate to an object of class "formula", is suitable to enforce safe programming practices (see good-practice).

nb_cores **Not yet operative**, only for development purposes. Number of cores to use for parallel computations.

... Optional arguments passed to HLCor, HLfit or mat_sqrt, for example the distMatrix argument of HLCor, or the verbose argument of HLfit. Arguments that do not fit within these functions are detected and a warning is issued. In a corrHLfit call, the verbose vector of booleans may include a TRACE=TRUE element, in which case information is displayed for each set of correlation and dispersion parameter values considered by the optimiser (see verbose for further information, mostly useless except for development purposes).

Details

For approximations of likelihood, see method. For the possible structures of random effects, see random-effects,

By default corrHLfit will estimate correlation parameters by maximizing the objective value returned by HLCor calls wherein the dispersion parameters are estimated jointly with fixed effects for given correlation parameters. If dispersion parameters are specified in init.corrHLfit, they will also be estimated by maximizing the objective value, and HLCor calls will not estimate them jointly with fixed effects. This means that in general the fixed effect estimates may vary depending on init.corrHLfit when any form of REML correction is applied (see inner-vs-outer for further elaboration).

Correctly using corrHLfit for likelihood ratio tests of fixed effects may then be tricky. It is safe to perform full ML fits of all parameters (using method="ML") for such tests (see Examples). The higher level function fixedLRT is a safe interface for likelihood ratio tests using some form of REML estimation in corrHLfit.

attr(<fitted object>,"optimInfo")$lower and ...$upper gives the lower and upper bounds for optimization of correlation parameters. These are the default values if the user did not provide explicit values. For the adjacency model, the default values are the inverse of the maximum and minimum eigenvalues of the adjMatrix. For the Matérn model, the default values are not so easily summarized: they are intended to cover the range of values for which there is statistical information to distinguish among them.

Value

The return value of an HLCor call, with additional attributes. The HLCor call is evaluated at the estimated correlation parameter values. These values are included in the return object as its $corrPars member. The attributes added by corrHLfit include the original call of the function (which can be retrived by getCall(<fitted object>), and information about the optimization call within corrHLfit.

See Also

See more examples on data set Loaloa, to compare fit times by corrHLfit and fitme. See fixedLRT for likelihood ratio tests.

Examples

```
# Example with an adjacency matrix (autoregressive model):
if (spaMM.getOption("example_maxtime")>0.7) {
  corrHLfit(cases~I(prop.ag/10) +adjacency(1|gridcode)+offset(log(expec)),
          adjMatrix=Nmatrix,family=poisson(),data=scotlip,method="ML")
}

#### Examples with Matern correlations
## A likelihood ratio test based on the ML fits of a full and of a null model.
if (spaMM.getOption("example_maxtime")>1.4) {
 data("blackcap")
 (fullfit <- corrHLfit(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap,
                    method="ML") )
 (nullfit <- corrHLfit(migStatus ~ 1 + Matern(1|longitude+latitude),data=blackcap,
                    method="ML",init.corrHLfit=list(phi=1e-6)))
 ## p-value:
 1-pchisq(2*(logLik(fullfit)-logLik(nullfit)),df=1)
}
```

---

corrMatrix *Using the* corrMatrix *and* distmatrix *arguments*

Description

corrMatrix is a formal argument of HLCor, also handled by higher-level fitting functions such as fitme, which can be used if the model formula contains a term of the form corrMatrix(1|...). It describes a correlation matrix, possibly as a dist half-matrix object. The covStruct argument can be used for the same purpose and is much more general, in particular allowing to specify several correlation matrices.

A co*variance* matrix (whose diagonal elements are not necessarily 1) can also be passed through this argument, but then its format must be a full matrix, not a dist object (as dist objects are interpreted as correlation matrices by filling the diagonal with 1's).

The way the rows and columns of the matrix are matched to the rows of the data depends on the nature of the grouping term ... in corrMatrix(1|...), and on the dimnames of the matrix, both of which should be carefully controlled (see Examples).

corrMatrix specify a fixed correlation matrix. It can be used to fit parametric correlation models not already implemented in **spaMM**, by wrapping the call of the fitting function with given correlation matrix in an objective function called by a general optimizer such as optim. However, for the same purpose, it may be more efficient to use a corrFamily "constructor".

For geospatial models, the alternative distMatrix argument can also be used: a spatial correlation matrix will be recomputed from it and from spatial correlation parameters, fixed or not. This allows in principle to use distance matrices not derived from geographical coordinates. But not all matrices obtained by, say, applying the Matern correlation function to an arbitrary distance matrix are valid correlation matrices.

Details

The simplest case is illustrated in the first examples below: the grouping term is identical to a single variable which is present in the data, whose levels match the rownames of the corrMatrix. As illustrated, the order of the data does not matter in that case, because the factor levels are used to match the data rows to the appropriate row and columns of the corrMatrix. The corrMatrix may even contain rows (and columns) in excess of the levels of the grouping term, in which case these rows are ignored.

These convenient properties no longer hold when the grouping term is not a single variable from the data (see nofactor fit in Examples), or when its levels do not correspond to row names of the matrix. In these cases, (1) no attempt is made to match the data rows to the row and column names of the corrMatrix (such attempt could succeed only if the user had given names to the matrix matching those that the called function could create from the information in the data, in which case the user should find easier to specify a single variable that can be matched); (2) the order of data and corrMatrix matter: internally, a single factor variable is constructed from all levels of the variables in the grouping term (i.e., from all levels of latitude and longitude, in the nofactor fit), with levels 1,2,3... that are matched to rows 1,2,3... of the corrMatrix. Thus the first row of the data is always associated to the first row of the matrix; (3) further, the dimension of the matrix must match the number of levels implied by the grouping term. For example, one might consider the case of 14 response values but of correlations between only 7 levels of a random effect, with two responses for each level. Then the matrix must be of dimension 7x7.

Examples

```
# These examples demonstrate correct use of the corrMatrix argument
# by using it to fit a Matern spatial model and comparing the results
# to the result using the alternative basic Matern(1|.) syntax

# The first series of examples uses a simplified syntax that works
# only because each geographical location is represented only once
# in the data. This condition is relaxed afterwards.

data("blackcap")

# Fit to be reproduced by the different syntaxes:
fitme(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap,
      fixed=list(nu=0.6285603,rho=0.0544659))

# Here we manually reconstruct the correlation matrix of this fit:
MLcorMat <- MaternCorr(proxy::dist(blackcap[,c("longitude","latitude")]),
```

```
                        nu=0.6285603,rho=0.0544659)
# (Crucially, rownames of 'blackcap' define the dimnames of 'MLcorMat')

# We create a factor ordered as geographical locations are ordered
# in the data, hence matching (in this simplified example)
# their order in the correlation matrix:
blackcap$name <- as.factor(rownames(blackcap))

fitme(migStatus ~ means+ corrMatrix(1|name),data=blackcap,
      corrMatrix=MLcorMat)

# Correct results after permutation of the matrix:
# (here the 'dist' object has to be converted to 'matrix' to allow permutation).
perm <- sample(14)
pmat <- proxy::as.matrix(MLcorMat, diag=1)[perm,perm]
# Crucially, info about the permutation is provided
# by the dimnames of 'pmat' which allow a correct match
# to levels of the 'name' variable.

fitme(migStatus ~ means+ corrMatrix(1|name),data=blackcap,
      corrMatrix=pmat)

# It is possible, but more risky, not to use a factor whose levels
# are specifically defined to match the dimnames of the matrix.
# Note the message if we don't use such a factor:
(nofactor  <- fitme(migStatus ~ means+ corrMatrix(1|longitude+latitude),
                    data=blackcap, corrMatrix=MLcorMat))

#### Case with several samples in each location: ####
if (spaMM.getOption("example_maxtime")>0.7 &&
    requireNamespace("IsoriX", quietly = TRUE)) {
# [the fits are quite fast, but requireNamespace("IsoriX"...)
#  is too slow on CRAN servers].

data("GNIPDataDE", package = "IsoriX")

## Fit to be reproduced by the different syntaxes:
fitme(source_value ~ 1 + Matern(1|long+lat), data=GNIPDataDE,
      fixed=list(nu=0.75, rho=0.2))

## Reconstruct the distance matrix of this fit:
dat <- GNIPDataDE
longlat  <- paste0(dat$long,":",dat$lat)
# Use unique() to get unique geographical coordinates *and* unique names:
distmat <- as.matrix(dist(unique(dat[,c("long","lat")])))
dimnames(distmat)  <- list(unique(longlat), unique(longlat))
# Define matching factor:
dat$ID <- factor(longlat)

## Various possible syntaxes for the fit (last one more general):
# corrMatrix(1|.) term + general 'covStruct' argument (see help("covStruct"))
fitme(source_value ~ 1 + corrMatrix(1|ID), data=dat,
      covStruct=list(corrMatrix=MaternCorr(distmat,rho = 0.2, nu=0.75)))
```

```
# corrMatrix(1|.) term + more ad hoc 'corrMatrix' argument
fitme(source_value ~ 1 + corrMatrix(1|ID), data=dat,
      corrMatrix=MaternCorr(distmat,rho = 0.2, nu=0.75))

# Matern(1|.) term + even more ad hoc 'distMatrix' argument
fitme(source_value ~ 1 + Matern(1|ID), data=dat,
      distMatrix=distmat,  fixed=list(rho = 0.2, nu=0.75))

# 'distMatrix' in covStruct=list(distMatrix=distmat) would not work,
# but a general syntax allowing multiple distance matrices is:

fitme(source_value ~ 1 + Matern(1|ID), data=dat,
      distMatrix=list("1"=distmat), # more than one element might be specified
      # covStruct=list(...), can be used if required for other random effects
      fixed=list(corrPars=list("1"=list(rho = 0.2, nu=0.75)))
     )
}
```

---

corr_family corr_family *objects*

Description

corr_family objects provide a convenient way to implement correlation models handled by spaMM, analogous to family objects. These objects are undocumented (but there are documentation pages for each of the models implemented).

Usage

```
# Matern(...)           # see help(Matern)
# Cauchy(...)           # see help(Cauchy)
# corrMatrix(...)       # see help(corrMatrix)
# AR1(...)              # see help(AR1)
# adjacency(...)        # see help(adjacency)
# IMRF(...)             # see help(IMRF)
## S3 method for class 'corr_family'
print(x,...)
```

Arguments

x	corr_family object.
...	arguments that may be needed by some corr_family object or some print method.

covStruct *Specifying correlation structures*

Description

covStruct is a formal argument of HLCor, also handled by higher level fitting functions such as fitme and corrHLfit, that allows one to specify the correlation structure for different types of random effects. It is an alternative to other ad hoc formal arguments such as corrMatrix or adjMatrix. It replaces the long-deprecated function Predictor(...) which has served as an interface for specifying the design matrices for random effects in early versions of **spaMM**.

The main use of covStruct is to specify the correlation matrix of levels of a given random effect term. Assuming that the design matrix of each random effect term follows the structure **ZAL** described in random-effects, it is thus an indirect way of specifying a "square root" **L** of the correlation matrix. The optional **A** factor can also be given by the optional "AMatrices" attribute of covStruct.

The correlation structure of a random effect can be specified directly as a correlation matrix, or by its inverse (a *precision* matrix **Q** such that the covariance matrix is $\lambda\mathbf{Q}^{-1}$), or by an adjacency matrix. covStruct is a list of matrices with names specifying the type of information provided: covStruct=list(corrMatrix=<some correlation (or covariance) matrix>) or covStruct=list(adjMatrix=<some adjacency matrix>) or covStruct=list(precision=<some precision matrix>).

When several matrices are provided, these names can be repeated, and NULL list members may be necessary, e.g.
covStruct=list(corrMatrix=<.>,"2"=NULL,corrMatrix=<.>))
when correlations matrices are required only for the first and third random effect.

The function as_precision can be used to perform the conversion from correlation information to precision factor in a controlled way (but using a numerical inversion that may not always be efficient). Fitting functions may also perform such a conversion automatically. **Do not run** as_precision on a matrix which is already the inverse of a correlation or covariance matrix, as the result will no longer be a precision matrix.

"AMatrices" is a list of matrices. The names of elements of the list does not matter, but the *i*th A matrix, and its row names, should match the *i*th **Z** matrix, and its column names. This implies that NULL list members may be necessary, as for the covStruct list.

Usage

```
as_precision(corrMatrix, condnum=1e12)
```

Arguments

corrMatrix	Correlation matrix, specified as matrix or as dist object.
condnum	Numeric: when a standard Cholesky factorization fails, the matrix is regularized so that the regularized matrix has this condition number (in version 3.10.0 this correction has been implemented more exactly than in previous versions).

Details

covStruct can also be specified as a list with an optional "types" attribute, e.g. structure(list(<some matrix>,types="corrMatrix")).

Value

as_precision returns a list with additional class precision and with single element a symmetric matrix of class dsCMatrix.

See Also

Gryphon and pedigree for a type of applications where declaring a precision matrix is useful.

Examples

```
## Not run:
data("blackcap")
# a 'dist' object can be used to specify a corrMatrix:
MLdistMat <- MaternCorr(proxy::dist(blackcap[,c("latitude","longitude")]),
                        nu=0.6285603,rho=0.0544659) # a 'dist' object!
blackcap$name <- as.factor(rownames(blackcap))
fitme(migStatus ~ means + corrMatrix(1|name), data=blackcap,
      corrMatrix=MLdistMat)

#### Same result using precision matrix,
(by_sp <- fitme(migStatus ~ means + corrMatrix(1|name), data=blackcap,
      covStruct=list(precision=as_precision(MLdistMat))))
# This typically induced the use of a distinct fitting algorithm:
how(by_sp) # "...using sparse-precision method..."

# Less controlled specification of precision matrix,
# but essentially same result again:
as_mat <- proxy::as.matrix(MLdistMat, diag=1)
prec_mat <- solve(as_mat) ## precision factor matrix
fitme(migStatus ~ means + corrMatrix(1|name), data=blackcap,
      covStruct=list(precision=prec_mat))
# !!! but do *not* use list(precision=as.precision(prec_mat)) !!!

# Since no correlation parameter is estimated,
# HLcor(., method="ML")  is here equivalent to fitme()

## End(Not run)
```

---

diallel *Random-effect structures for symmetric or antisymmetric dyadic interactions*

Description

ranGCA and diallel are random-effect structures designed to represent the effet of symmetric interactions between pairs of individuals (order of individuals in the pair does not matter), while antisym represents anti-symmetric interactions (the effect of reciprocal ordered pairs on the outcome are opposite, as in the so-called Bradley-Terry models). These random-effect structures all account for multiple membership, i.e., the fact that the same individual may act as the first or the second individual among different pairs, or even within one pair if this makes sense).

More formally, the outcome of an interaction between a pair i, j of agents is subject to a symmetric overall random effect v_{ij} when the effect "on" individual i (or viewed from the perspective of individual i) equals the effect on j: $v_{ij} = v_{ji}$. This may result from the additive effect of individual random effects v_i and v_j: $v_{ij} = v_i + v_j$, but also from non-additive effects $v_{ij} = v_i + v_j + a_{ij}$ if the interaction term a_{ij} is itself symmetric ($a_{ij} = a_{ji}$). ranGCA and diallel effects represent such symmetric effects, additive or non-additive respectively, in a model formula (see Details for the semantic origin of these names and how they can be changed). Conversely, antisymmetry is characterized by $v_{ij} = v_i - v_j = -v_{ji}$ and is represented by the antisym formula term.

If individual-level random effects of the form (1|ID1)+ (1|ID2) were included in the model formula instead of ranGCA(1|ID1+ID2) for symmetric additive interactions, this would result in different variances being fitted for each random effect (breaking the assumption of symmetry), and the value of the random effect would differ for an individual whether it appears as a level of the first random effect or of the second (which is also inconsistent with the idea that the random effect represents a property of the individual).

When ranGCA or antisym random effects are fitted, the individual effects are inferred. By contrast, when a diallel random effect is fitted, an autocorrelated random effect v_{ij} is inferred for each **unordered** pair (no individual effect is inferred), with correlation ρ between levels for pairs sharing one individual. This correlation parameter is fitted and is constrained by $\rho < 0.5$ (see Details). ranGCA is equivalent to the case $\rho = 0.5$. diallel fits can be slow for large data if the correlation matrix is large, as this matrix can have a fair proportion of nonzero elements. There may also be identifiability issues for variance parameters: in a LMM as shown in the examples, there will be three parameters for the random variation (phi, lambda and rho) but only two can be estimated if only one observation is made for each dyad.

Usage

```
## formula terms:

# ranGCA(1| <.> + <.>)
# antisym(1| <.> + <.>)
# diallel(1|  <.> + <.>, tpar, fixed = NULL, public = NULL)

## where the <.> are two factor identifiers, ** whose levels
## must be identical when representing the same individual **

## corrFamily constructors:
ranGCA() # no argument
antisym() # no argument
diallel(tpar, fixed = NULL, public = NULL)
```

Arguments

tpar	Numeric: template value of the correlation coefficient for pairs sharing one individual.
fixed	NULL or fixed value of the correlation coefficient.
public	NULL, or an environment. When an empty environment is provided, a template CorNA for the correlation matrix (with NA's in place of ρ) will be copied therein, for inspection at user level.

Details

Although the symmetric random-effect structures may be used in many different contexts (including social network analysis, or "round robin" experiments in psychology; another possibly relevant literature keyword here is "multi membership"), their present names refer to the semantics established for diallel experiments (e.g., Lynch & Walsh, 1998, p. 611), because it is not easy to find a more general yet intuitive semantics. If the names ranGCA and diallel sound inappropriate for your context of application, you can declare and use an alternative name for them, taking advantage of the fact that they are random-effect structures defined through corrFamily constructors, which are functions named as the formula term. For example, symAdd(1|ID1+ID2) can be used in a model formula after the following two steps:

```
# Define the 'symAdd' corrFamily constructor (a function) by copy:
symAdd <- ranGCA
# Associate the 'symAdd' function to 'symAdd' formula terms:
register_cF("symAdd")
```

In diallel experiments, one analyzes the phenotypes of offspring from multiple crosses among which the mother in a cross can be the father in another, so this is an example of multiple-membership. The additive genetic effects of each parent's genotypes are described as "general combining abilities" (GCA). In case of non-additive effect, the half-sib covariance is not half the full-sib covariance and this is represented by the interaction a_{ij} described as "specific combining abilities" (SCA). The sum of GCA and SCA defines a synthetic random effect "received" by the offspring, with distinct levels for each unordered parental pair, and with correlation ρ between effects received by half-sibs (one shared parent). ρ corresponds to var(GCA)/[2*var(GCA)+var(SCA)] and is necessarily ≤ 0.5.

See the X.GCA documentation for similar constructs for fixed effects.

Value

The functions return corrFamily descriptors whose general format is described in corrFamily. The ones produced by ranGCA and antisym are atypical in that only their Af element is non-trivial.

References

Lynch, M., Walsh, B. (1998) Genetics and analysis of quantitative traits. Sinauer, Sunderland, Mass.

See Also

X.GCA for similar constructs for fixed effects; mmfn for asymmetric multi-membership models.

Examples

```
#### Simulate dyadic data

set.seed(123)
nind <- 10       # Beware data grow as O(nind^2)
x <- runif(nind^2)
id12 <- expand.grid(id1=seq(nind),id2=seq(nind))
id1 <- id12$id1
id2 <- id12$id2
u <-  rnorm(nind,mean = 0, sd=0.5)

## additive individual effects:
y <-  0.1 + 1*x + u[id1] +  u[id2] + rnorm(nind^2,sd=0.2)

## Same with non-additive individual effects:
dist.u <- abs(u[id1] -  u[id2])
z <- 0.1 + 1*x + dist.u + rnorm(nind^2,sd=0.2)

## anti-smmetric individual effects:
t <-  0.1 + 1*x + u[id1] - u[id2] + rnorm(nind^2,sd=0.2)

dyaddf <- data.frame(x=x, y=y, z=z, t=t, id1=id1,id2=id2)
# : note that this contains two rows per dyad, which avoids identifiability issues.

# Enforce that interactions are between distinct individuals (not essential for the fit):
dyaddf <- dyaddf[- seq.int(1L,nind^2,nind+1L),]


# Fits:

(addfit <- fitme(y ~x +ranGCA(1|id1+id2), data=dyaddf))
#
# practically equivalent to:
#
(fitme(y ~x +diallel(1|id1+id2, fixed=0.49999), data=dyaddf))

(antifit <- fitme(t ~x +antisym(1|id1+id2), data=dyaddf))

(distfit <- fitme(z ~x +diallel(1|id1+id2), data=dyaddf))
```

---

div_info *Information about numerical problems*

Description

This experimental function displays information about parameter values for which some numerical problems have occurred. Some warnings suggest its use.

Numerical problems may occur if the information matrix (for the augmented linear model used in the iteratively reweighted least-squares algorithm) is nearly singular. **spaMM** may try to check

whether such singularity occurs when this algorithm has not converged. But itself may be slow so it is not performed systematically for large matrices. `spaMM.options(diagnose_conv=<integer>)` may be used to control the maximum size of matrices for which the check is performed.

When "outer" generic optimization is performed, information is reported about the range of parameter values for which problems occurred, (see Value). The fit object `divinfo` element may also contain more informative tables of parameter values. This information is currently missing for "inner"-estimated parameters (see `inner-vs-outer` for this concept).

Usage

```
div_info(object, ...)
```

Arguments

`object`	An object of class `HLfit`, as returned by the fitting functions in `spaMM`.
`...`	Currently not used

Value

Used mainly for the side effects (printed output) but returns invisibly either a single parameter vector (if a single numerical problem occurred) or a matrix of parameter ranges, or NULL if there is no problem to report.

Examples

```
# Tragically ;-), no simple example of numerical problems
# that can be diagnosed by div_info() is currently available.
```

DoF *Degrees of freedom extractor*

Description

This extracts the number of degrees of freedom for a model, in the usual sense for likelihood-ratio tests: a count of number of fitted parameters, distinguishing different classes of parameters (see Value).

Usage

```
DoF(object)
```

Arguments

`object`	A fitted-model object, of class `"HLfit"`.

Details

The output distinguishes counts of random-effect vs residual-dispersion parameters, following the conceptual distinction between effects that induce correlations between different levels of the resonse vs. observation-level effects. However, a residual-dispersion component can be declared as a random effect, so that the counts for logically equivalent models may differ according to the way a model was declared. For example if residual dispersion for an LLM is declared as an observation-level random effect while `phi` is fixed, the `p_lambda` component will include 1 df for what would otherwise be accounted by the `p_rdisp` component. A more involved case where the same contrast happens is when a negative-binomial model (with a residual-dispersion `shape` parameter) is declared as a Poisson-gamma mixture model (with a varaince parameter for the Gamma-distributed individual-level random effect).

Value

A vector with possible elements `p_fixef`, `p_lambda`, `p_corrPars` and `p_rdisp` for, respectively, the number of fixed-effect coefficients of the main-response model, the number of random-effect variance parameters, the number of random-effect correlation parameters, and the number of residual dispersion parameters (the latter being itself, for a mixed-effect residual-dispersion model, the sum of such components).

See Also

`df.residual.HLfit`; `get_any_IC` for extracting effective degrees of freedom considered in the model-selection literature; `as_LMLT` for access to the effective degrees of freedom considered in Satterthwaite's test and its extentions.

`dofuture` *Interface for parallel computations*

Description

interface to apply some function `fn` in parallel on columns of a matrix. It is not logically restricted to mixed-effect applications, hence it can be used more widely. Depending on the `nb_cores` argument, parallel or serial computation is performed, calling the `future.apply::future_apply` function. A socket cluster is used by default for parallel computations, but a fork cluster can be requested on linux and alike operating systems by using argument `cluster_args=list(type="FORK")`.

Usage

```
dofuture(newresp, fn, nb_cores=NULL, fit_env, control=list(),
      cluster_args=NULL, debug.=FALSE, iseed=NULL,
      showpbar="ignored", pretest_cores="ignored",
      ... )
```

Arguments

newresp	A matrix on whose columns fn will be applied (e.g., as used internally in **spaMM**, the return value of a simulate.HLfit() call); or an integer, then converted to a trivial matrix matrix(seq(newresp),ncol=newresp,nrow=1).
fn	Function whose first argument is named y. The function will be applied for y taken to be each column of newresp.
nb_cores	Integer. Number of cores to use for parallel computations. If >1, a cluster of nb_cores nodes is used. Otherwise, no parallel computation is performed.
fit_env	(for socket clusters only:) An environment, or a list, containing variables to be exported on the nodes of the cluster (by parallel::clusterExport).
control	A list. The only effective control is .combine="rbind" (mimicking the foreach syntax used in the alternative interface dopar).
cluster_args	A list of arguments passed to parallel::makeCluster or parallel::makeForkCluster. E.g., outfile="log.txt" may be useful to collect output from the nodes, and type="FORK" to force a fork cluster on linux(-alikes).
debug.	(for socket clusters only:) For debugging purposes. Effect, if any, is to be defined by the fn as provided by the user.
iseed	Integer, or NULL. If an integer, it is used to initialize "L'Ecuyer-CMRG" random-number generator (iseed argument of clusterSetRNGStream), with identical effect across different models of parallelisation. If iseed is NULL, the seed is not controlled.
showpbar, pretest_cores	Currently ignored; for consistency with dopar formal arguments.
...	Further arguments to be passed (unevaluated) to future.apply (and then possibly to fn).

Value

The result of calling future.apply. If the progressr package is loaded, a side-effect of dofuture is to show a progress bar with character 'S' or 'P' or 'F' depending on parallelisation status (serial/socket/fork).

See Also

dopar for an alternative implementation of (essentially) the same functionalities, and wrap_parallel for its differences from dofuture.

Examples

```
## Not run:
if (requireNamespace("future.apply", quietly = TRUE)) {

 # Useless function, but requiring some argument beyond the first
 foo <- function(y, somearg, ...) {
   if ( is.null(somearg) || TRUE ) length(y)
 }
```

```
 # Whether FORK can be used depends on OS and whether Rstudio is used:
   dofuture(matrix(1,ncol=4,nrow=3), foo, fit_env=list(), somearg=NULL,
     nb_cores=2, cluster_args=list(type="FORK"))
}

## End(Not run)
```

---

dopar *Interface for parallel computations*

Description

dopar and combinepar are interfaces primarily designed to apply some function fn in parallel on columns of a matrix, although other uses are possible. Depending on the nb_cores argument, parallel or serial computation is performed. A socket cluster is used by default for parallel computations, but a fork cluster can be requested on linux and alike operating systems by using argument cluster_args=list(type="FORK").

dopar has been designed to provide by default a progress bar in all evaluations contexts. A drawback is that different procedures are called depending e.g. on the type of cluster, with different possible controls. In particular, foreach is called in some cases but not others, so non-trivial values of its .combine control are not always enforced. The alternative interface combinepar will always use foreach, and will still try to provide by default a progress bar but may fail to do so in some cases (see Details).

Usage

```
dopar(newresp, fn, nb_cores = NULL, fit_env,
      control = list(),
      cluster_args = NULL, debug. = FALSE, iseed = NULL,
      showpbar = eval(spaMM.getOption("barstyle")),
      pretest_cores =NULL, ...)
combinepar(newresp, fn, nb_cores = NULL, cluster=NULL, fit_env,
      control = list(),
      cluster_args = NULL, debug. = FALSE, iseed = NULL,
      showpbar = eval(spaMM.getOption("barstyle")),
      pretest_cores =NULL, ...)
```

Arguments

newresp	A matrix on whose columns fn will be applied (e.g., as used internally in **spaMM**, the return value of a simulate.HLfit() call); or an integer, then converted to a trivial matrix matrix(seq(newresp),ncol=newresp,nrow=1).
fn	Function whose first argument is named y. The function will be applied for y taken to be each column of newresp.

`nb_cores` Integer. Number of cores to use for parallel computations. If >1 (and no cluster is provided by the `cluster` argument), a cluster of `nb_cores` nodes is created, used, and stopped on completion of the computation. Otherwise, no parallel computation is performed.

`cluster` (for `combinepar` only): a cluster object (as returned by `parallel::makeCluster` or `parallel::makeForkCluster`). If this is used, the `nb_cores` and `cluster_args` arguments are ignored. The cluster is not stopped on completion of the computation

`fit_env` (for socket clusters only:) An environment, or a list, containing variables to be exported on the nodes of the cluster (by `parallel::clusterExport`); e.g., `list(bar=bar)` to pass object `bar` to each node. The argument `control(.errorhandling = "pass")`, below, is useful to find out missing variables.

`control` A list following the `foreach` control syntax, even if `foreach` is not used. There are limitations when `dopar` (but not `combinepar`) is used, in all but the first case below:

1. for socket clusters, with `doSNOW` attached, `foreach` is called with default arguments including `i = 1:ncol(newresp), .inorder = TRUE, .errorhandling = "remove", .packages = "spaMM"`, and further arguments taken from the present function's `control` argument, which may also be used to override the defaults. For example, `.errorhandling = "pass"` is useful to get error messages from the nodes, and therefore **strongly recommended** when first experimenting with this function.
2. for socket clusters, with `doSNOW` **not** attached, `dopar` calls `pbapply` instead of `foreach` but `control$.packages` is still handled. The result is still in the format returned in the first case, i.e. by `foreach`, taking the `control` argument into account. `pbapply` arguments may be passed through the ... argument.
3. if a fork cluster is used, `dopar` calls `mclapply` instead of `foreach`. `control$mc.silent` can be used to control the `mc.silent` argument of `mclapply`.
4. (if `nb_cores=1` `dopar` calls `mclapply`).

`cluster_args` A list of arguments passed to `parallel::makeCluster`. E.g., `outfile="log.txt"` may be useful to collect output from the nodes, and `type="FORK"` to force a fork cluster on linux(-alikes).

`debug.` (for socket clusters only:) For debugging purposes. Effect, if any, is to be defined by the `fn` as provided by the user.

`iseed` (all parallel contexts:) Integer, or NULL. If an integer, it is used as the `iseed` argument of `clusterSetRNGStream` to initialize `"L'Ecuyer-CMRG"` random-number generator (see Details). If `iseed` is `NULL`, the default generator is selected on each node, where its seed is not controlled.

`showpbar` (for socket clusters only:) Controls display of progress bar. See `barstyle` option for details.

`pretest_cores` (for socket clusters only:) A function to run on the cores before running `fn`. It may be used to check that all arguments of the `fn` can be evaluated in the cores' environments (the internal function `.pretest_fn_on_cores` provides an example).

... Further arguments to be passed (unevaluated) to fn, if not caught on the way by pbapply (which means that different results may in principle be obtained depending on the mode of parallelisation, which is the kind of design issues that combinepar aims to resolve by always calling foreach).

Details

Control of random numbers through the "L'Ecuyer-CMRG" generator and the iseed argument is not sufficient for consistent results when the doSNOW parallel backend is used, so if you really need such control in a fn using random numbers, do not use doSNOW. Yet, it is fine to use doSNOW for bootstrap procedures in spaMM, because the fitting functions do not use random numbers: only sample simulation uses them, and it is not performed in parallel.

combinepar calls foreach::%dopar% which assumes that a cluster has been declared using a suitable backend such as doSNOW, doFuture or doParallel. If only the latter is available, no progress bar is displayed. A method to render a bar when doParallel is used can be found on the Web, but that bar is not a valid progress bar as it is displayed only after all the processes have been run.

Value

The result of calling the foreach, pbapply or mclapply interface, dependent on the control argument and possibly on the parallelisation backend used. Ideally, the default format should be same whatever the interface and backend used, and the default when foreach is used is essentially as defined by calling
foreach(.final=if(! is.list(v[[1]])) {do.call(cbind,v)} else v)
when a list v of results (one result for each child process) is given to .final. Then, if each child process itself returns a list, cbind is applied.

A side-effect of dopar is to show a progress bar whose character informs about the type of parallelisation performed: a "F" or default "=" character for fork clusters, a "P" for parallel computation via foreach and doSNOW, a "p" for parallel computation via foreach and doFuture or via pbapply, and "s" for serial computation foreach and doParallel or via pbapply.

See Also

dofuture is yet another interface with (essentially) the same functionalities as dopar. See the documentation of the wrap_parallel option for its differences from dopar.

See spaMM_boot and spaMM2boot for more specialized interfaces for parametric bootstrap computations. By default, these functions themselves call dopar.

Examples

```
## See source code of spaMM_boot()

## Not run:
# Useless function, but requiring some argument beyond the first
foo <- function(y, somearg, ...) {
  if ( is.null(somearg) || TRUE ) length(y)
}

# Whether FORK can be used depends on OS and whether Rstudio is used:
```

```
dopar(matrix(1,ncol=4,nrow=3), foo, fit_env=list(), somearg=NULL,
  nb_cores=2, cluster_args=list(type="FORK"))

## End(Not run)
```

`drop1.HLfit` *Drop all possible single fixed-effect terms from a model*

Description

Drop single terms from the model. The `drop1` method for **spaMM** fit objects is conceived to replicate the functionality, output format and details of pre-existing methods for similar models. Results for LMs and GLMs should replicate base R `drop1` results, with some exceptions:
* somewhat stricter default check of non-default `scope` argument;
* Because the dispersion estimates for Gamma GLMs differ between `stats::glm` and **spaMM** fits (see Details in `method`), some tests may differ too; results from **spaMM** REML fits being closer than ML fits to those from `glm()` fits;
* AIC values reported in tables are always the marginal AIC as computed by `AIC.HLfit`, while `drop1.glm` may report confusing (to me, at least) values (see `AIC.HLfit`) for reasons that seem to go beyond differences in dispersion estimates.

For LMMs, ANOVA tables are provided by interfacing `lmerTest::anova` (with non-default `type`).

For other classes of models, a table of likelihood ratio tests is returned, each test resulting from a call to `LRT`.

Usage

```
## S3 method for class 'HLfit'
drop1(object, scope, method="", check_marg=NULL, check_time=60, ...)
```

Arguments

`object`	Fit object returned by a **spaMM** fitting function.
`scope`	Default "scope" (terms to be tested, specified as a formula, see Examples) is determined by applying `stats::drop.scope` on fixed-effect terms. Non-default scope can be specified a formula giving the terms to be considered for dropping. It is also possible to specify them as a character vector, but then one has to make sure that the elements are consistent with term labels produced by `terms`, as inconsistent elements will be ignored.
`method`	Only non-default value is `"LRT"` which forces evaluation of a likelihood ratio tests by `LRT`, instead of specific methods for specific classes of models.
`check_marg`	NULL or boolean: whether effects should be checked for marginality. By default, this check is performed when a non-default scope is specified, and then no test is reported for terms that do not satisfy the marginality condition. If `check_marg=FALSE`, marginality is not checked and tests are always performed.

check_time	numeric: whether to output some information when the execution time of drop1 may be of the order of the time specified by check_time, or more. This is checked only when random effect are present. Such output can thus be suppressed by check_time=Inf.
...	Further arguments passed from or to methods, or to LRT.

Details

As for the ANOVA-table functionality, it has been included here mainly to provide access to F tests (including, for LMMs, the "Satterthwaite method", using pre-existing procedures as template or backend for expediency and familiarity. The procedures for specific classes of models have various limitations, e.g., none of them handle models with variable dispersion parameter. For classes of models not well handled by these procedures (by design or due to the experimental nature of the recent implementation), method="LRT" can still be applied (and will be applied by default for GLMMs).

Value

The return format is that of the function called (lmerTest::drop1 for LMMs), or emulated (base drop1 methods for LMs or GLMs), or is a data frame whose rows are each the result of calling LRT.

See Also

as_LMLT for the interface to lmerTest::drop1.

Examples

```
data("wafers")
#### GLM

wfit <- fitme(y ~ X1+X2+X1*X3+X2*X3+I(X2^2),  family=Gamma(log), data=wafers)
drop1(wfit, test = "F")
drop1(wfit, test = "F", scope= ~ X1 +  X1 * X3 )  # note the message!

#### LMM
if(requireNamespace("lmerTest", quietly=TRUE)) {
  lmmfit <- fitme(y ~X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch),data=wafers)
  drop1(lmmfit) # => Satterthwaite method here giving p-values quite close to
                #    traditional t-tests given by:
  summary(lmmfit, details=list(p_value=TRUE))
}

#### GLMM
wfit <- fitme(y ~ X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch), family=Gamma(log),
              rand.family=inverse.Gamma(log), resid.model = ~ X3+I(X3^2) , data=wafers)
drop1(wfit)
drop1(wfit, scope= ~ X1 +  X1 * X3 )  # note the message!
```

dyad *Correlated random effects in dyadic interactions*

Description

mmfn is a function to be used for specifying random effects in asymmetric multi-membership interactions. This feature is experimental and has only been tested for some dyadic interactions.

The simplest case is that of dyadic interactions between one focal individual and its partner in each dyad. The models declared in this way include correlation parameter(s) for the correlation of effects expressed by a given individual in different possible roles (such as focal vs. partner).

At the same time, correlation of effects expressed among individuals in the same role can also be specified. For example, the random effects may represent genetic influences on focal and partner effects. The correlation between effects expressed in the focal role may depend on the relatedness between individuals, and likewise for effects expressed in the partner role. This can be specified by a random-effect term corrMatrix(<LHS> | <RHS>) where:

* the LHS expression includes an expression of the form mm(<id1>, <id2>), <id1> and <id2> being two factors in the data; and the RHS expression is of the form mmfn(<id1>, <id2>). A dummy variable can be used instead of the mm(<id1>, <id2>) expression (see Examples), the factors to be used being determined from the RHS expression;

* The LHS expression is interpreted as a random-coefficient specification, meaning that distinct random effects $u_{\texttt{<id1>}(i)}$,$v_{\texttt{<id2>}(i)}$,... affect the ith response, and that correlations between the effects are fitted by default, as for other random-coefficient terms. The elements $u_{\texttt{<id1>}(i)}$,$v_{\texttt{<id2>}(j)}$,... are correlated when <id1>(i) and <id2>(j) are identical, rather than when $i = j$. This is illustrated in the Examples, where the two factors represents roles of individuals as focal individuals and as their mothers, so distinct but correlated random effects u_k,v_k, affecting different levels of the response ($i \neq j$), are assigned to a same individual k in these two roles;

* a corrMatrix specifies the correlations of random effects among individuals in either role (correlations as focal, or correlations as partner). The whole corrMatrix(<LHS> | <RHS>) expression thus describes a composite-ranef controlled by two correlation models, one for within-role correlations, the other for among-role correlations.

If the LHS is a more complex expression than only the mm(...) term (as in the Example with interaction with sex), different random effects vectors are assigned to each term implied by the LHS (as for random-coefficient terms specified by other syntaxes), and the corrMatrix specifies the correlations between the effects within each vector;

* the RHS of the form mmfn(...) specifies the factors containing the levels to be matched to rows and columns of the corrMatrix, in the same way as other forms of RHS serve to identify rows and columns of the correlation matrix of correlated random effects.

PAIRfn is an alias for mmfn (its name suggests that it handles only dyadic interactions, but the mmfn aims to be more general).

The syntax using a dummy variable assumes that this variable is present in the input data, as the numeric constant 1 (see Examples). By default, this dummy variable is named PAIR, though this can be changed by the VAR argument of mmfn or PAIRfn.

Usage

```
mmfn(..., VAR = "PAIR", only.vars=TRUE)
PAIRfn(..., VAR = "PAIR", only.vars=TRUE)

## formula term:
# corrMatrix(  < LHS using mm(<.>, <.>) >| PAIRfn(<.>, <.>) )
# corrMatrix(  < LHS using dummy variable >| PAIRfn(<.>, <.>) )
```

Arguments

...	factors present in the data provided for the fit, identifying the individuals in the dyadic or multi-membership interaction.
VAR	character: name of dummy variable possibly used in LHS of random-effect term.
only.vars	For programming purposes, not documented.

See Also

diallel for (anti-)symmetric dyadic interactions.

Examples

```
if (spaMM.getOption("example_maxtime")>1) {

data("Gryphon")
fitme(BWT ~ 1+ corrMatrix(mm(ID,mother) | PAIRfn(ID,mother)),
      data=Gryphon_df, corrMatrix=Gryphon_A)

# : same as using a PAIR dummy variable as follows:
#
# Gryphon_df$PAIR <- 1 # Do not use values other than 1.
# fitme(BWT ~ 1+ corrMatrix(PAIR | PAIRfn(ID,mother)),
#       data=Gryphon_df, corrMatrix=Gryphon_A)

if (spaMM.getOption("example_maxtime")>15) {
  # Distinct random effects for each role, and each sex of the focal:
  #
  fitme(BWT ~ 1+ corrMatrix(mm(ID,mother)*sex | PAIRfn(ID,mother)),
        data=Gryphon_df, corrMatrix=Gryphon_A)
  #
  # This is controlled as other rancom-coefficient terms. E.g.,
  #   fixed=list(ranCoefs=list("1"=c(NA,0,0,0,NA,0,0,NA,0,NA)))
  # could be used to fit only the variances.

}
}
```

eval_replicate *Evaluating bootstrap replicates*

Description

eval_replicate is the default simuland function applied to simulated bootstrap samples by likelihood-ratio testing functions (fixedLRT, LRT, anove.HLfit). This documentation presents the requirements and possible features of this function and of possible user-defined alternatives.

An alternative function spaMM:::.eval_replicate2 is also provided. It is slower, as it refits the models compared with different initial values for random-effect parameters, which is useful in some difficult cases where initial values matter. The eval_replicate function may also refit the "full" models with different initial values when the logLik of the refitted full model is substantially lower than that of the refitted null model. "Substantially" means that a tolerance of 1e-04 is applied to account for inaccuracies of numerical maximization.

Usage

```
eval_replicate(y)
```

Arguments

y	a response vector on which a previously fitted model may be refitted.

Details

likelihood-ratio testing functions have a debug. argument whose effect depends on the simuland function. The default behaviour is thus defined by eval_replicate, as: if debug.=TRUE, upon error in the fitting procedures, dump.frames will be called, in which case **a dump file will be written on disk**; and a **list** with debugging information will be returned (so that, say, pbapply will not return a matrix). This behaviour may change in later versions, so non-default debug. values should not be used in reproducible code. In serial computation, debug.=2 may induce a stop; this should not happen in parallel computation because the calling functions check against debug.==2.

Essential information such as the originally fitted models is passed to the function not as arguments but through its environment, which is controlled by the calling functions (see the eval_replicate source code to know which are these arguments). Users should thus not assume that they can control their own simuland function's environment as this environment will be altered.

Advanced users can define their own simuland function. The eval_replicate source code provides a template showing how to use the function's environment. The Example below illustrates another approach augmenting eval_replicate. A further example is provided in the file tests/testthat/test-LRT-boot.R, using ... to pass additional arguments beyond response values.

Value

A vector of the form c(full=logLik(<refitted full model>),null=logLik(<refitted null model>); or possibly in debugging contexts, a list with the same elements each with some additional information provided as attribute.

See Also

Calling functions `fixedLRT`, `LRT`.

Examples

```
## Not run:
# Simple wrapper enhancing the default 'simuland'
#  with a call to some obscure option, and dealing with
#  the need to pass the environment assigned to 'simuland'
eval_with_opt <- function(y) {
  spaMM.options(some_obscure_option="some_obscure_value")
  eval_rep <- spaMM:::.eval_replicate
  environment(eval_rep) <- parent.env(environment()) # passing the environment
  eval_rep(y)
}

## End(Not run)
```

`extractors` *Functions to extract various components of a fit.*

Description

Most extractors are methods of generic functions defined in base R (see Usage), for which the base documentation may be useful.

- `formula` extracts the model formula.
- `family` extracts the assumed distribution family for the response variable.
- `terms` extracts the formula, with attributes describing the **fixed-effect** terms.
- `nobs` returns the length of the response vector.
- `logLik` extracts the log-likelihood (exact or approximated).
- `fitted` extracts fitted values.
- `response` extracts the response (as a vector).
- `fixef` extracts the fixed effects coefficients, β.
- `coef` may not do anything useful yet.
- `ranef` extracts the predicted random effects, **Lv** (default since version 1.12.0), or optionally **u** (see `random-effects` for definitions). `print.ranef` controls their printing.
- `getDistMat` returns a distance matrix for a geostatistical (Matérn etc.) random effect.
- `df.residual` extracts residual degrees-of-freedom for fitted models (here number of observations minus number of parameters of the model except residual dispersion parameters).
- `weights` extracts prior weights (as defined by the fitting functions's `prior.weights` argument).
- `get_drawSizes` extracts binomial or multinomial draw sizes.
- `dev_resids` returns a vector of squared (unscaled) deviance residuals, prior-weights if the model had prior weights; and
- `deviance` returns the sum of squares of these deviance residuals. See `residuals.HLfit` for details about these concepts, and their relation with the different possible residuals extracted by the latter function.

Usage

```
## S3 method for class 'HLfit'
formula(x, which="hyper", ...)
## S3 method for class 'HLfit'
family(object, submodel=NULL, ...)
## S3 method for class 'HLfit'
terms(x, ...)
## S3 method for class 'HLfit'
nobs(object, ...)
## S3 method for class 'HLfit'
logLik(object, which, ...)
## S3 method for class 'HLfit'
fitted(object, ...)
## S3 method for class 'HLfit'
coef(object, ...)
## S3 method for class 'HLfit'
fixef(object, na.rm=NULL, keep_fixed = TRUE, ...)
## S3 method for class 'HLfit'
ranef(object, type = "correlated", ...)
## S3 method for class 'ranef'
print(x, max.print = 40L, ...)
## S3 method for class 'HLfit'
deviance(object, ...)
## S3 method for class 'HLfit'
df.residual(object, ...)
## S3 method for class 'HLfit'
weights(object, type, ...)
##
getDistMat(object, scaled=FALSE, which = 1L)
response(object,...)
dev_resids(object,...)
get_drawSizes(object,p4m="P")
```

Arguments

object	An object of class HLfit, as returned by the fitting functions in spaMM.
type	For ranef, use type="correlated" (default) to display the correlated random effects (**Lv**), whether in a spatial model, or a random- coefficient model. Use type="uncorrelated" to pretty-print the elements of the <object>$ranef vector (**u**). For weights, either "prior" or "working", with the same meaning as for weights.glm: respectively the prior weights, or the weights used in the final iteration of the IRLS algorithm.
which	* For logLik, the name of the element of the APHLs list to return (see Details for any further possibility). The default depends on the fitting method. In particular, if it was REML or one of its variants, the default is to return the log restricted likelihood (exact or approximated). * For getDistMat, an integer, to select a random effect from several for which

	a distance matrix may be constructed. * For `formula`, by default the model formula with non-expanded `multIMRF` random-effect terms is returned, while for `which=""` a formula with `multIMRF` terms expanded as `IMRF` terms is returned.
`na.rm`	Boolean: whether to include NA values for missing coefficients of rank-deficient model matrices. Default is to exclude them for mixed models and to include them for other ones. See Details for the underlying reason.
`keep_fixed`	Boolean: whether to include values of coefficients that were fixed (by element `beta` of argument `etaFix`) in the fit `object`.
`scaled`	If `FALSE`, the function ignores the scale parameter ρ and returns unscaled distance.
`x`	For `print.ranef`: the return value of `ranef.HLfit`.
`max.print`	Controls `options("max.print")` locally.
`submodel`	Integer: to extract the family for a given submodel in a multivariate-response fit.
`p4m`	Character: for fit objects of class `pois4mlogit`, this controls whether to return the multinomial draw size (when `p4m="M"`) or a vector of 1 as for Poisson fits ((when `p4m="P"`). Note that the default value may change in a later version, so an explicit value should be provided to ensure portability of code.
`...`	Other arguments that may be needed by some method.

Details

For rank-deficient model matrices, base R procedures `lm` and `glm` estimate coefficients for a rank-trimmed matrix and `coefficient()` returns a full-length vector completed with NA values for coefficients not estimated, while the **lme4** `fixef` method returns a trimmed vector. **spaMM** has long followed the base R convention for all models but this may impede use of some post-fit procedures initially conceived for **lme4** objects (such as **lmerTest** procedures for LMMs). So now `fixef.HLfit` trims the vector by default for mixed-effect models only. The default is thus to maximize consistency/compatibility with preexisting procedures despite their inconsistencies with each other.

Value

`formula` returns a `formula`, except a list of them from `fitmv()` output.

`terms` returns an object of class `c("terms", "formula")` which contains the *terms* representation of a symbolic model. See `terms.object` for its structure. `terms(<fitmv() result>)` returns a list of such terms.

Other return values are numeric (for `logLik`), vectors (most cases), matrices or dist objects (for `getDistMat`), or a family object (for `family`). `ranef` returns a list of vectors or matrices (the latter for random-coefficient terms).

References

McCullagh, P. and Nelder J. A. (1989) Generalized linear models. Second ed. Chapman & Hall: London.

See Also

See summary.HLfit whose return value include the tables of fixed-effects coefficients and random-effect variances displayed by the summary, residuals.HLfit to extract various residuals, residVar to extract residual variances or information about residual variance models, hatvalues to extract leverages, get_matrix to extract the model matrix and derived matrices, and vcov.HLfit to extract covariances matrices from a fit, get_RLRsim_args to extract arguments for (notably) tests of random effects in LMMs, optimBounds to extract optimization box constraints.

Examples

```
data("wafers")
m1 <- fitme(y ~ X1+X2+(1|batch), data=wafers)
fixef(m1)
ranef(m1)

data("blackcap")
fitobject <- fitme(migStatus ~ 1 + Matern(1|longitude+latitude),data=blackcap,
                         fixed=list(nu=4,rho=0.4,phi=0.05))
getDistMat(fitobject)
```

---

extreme_eig *Utilities for regularization of a matrix*

Description

regularize can be used to regularize (nearly-)singular correlation matrices. It may also be used to regularize covariance matrices but will not keep their diagonal constant. Use on other types of matrices may give nonsense. The regularization corrects the diagonal of matrices with high condition number so that the condition number of a corrected matrix is the maximum value specified by maxcondnum. For that purpose, it needs the extreme eigenvalues of the matrix, by default provided by the function extreme_eig. Calls functions from **RSpectra** if available, and falls back on base functions otherwise.

Usage

```
extreme_eig(M, symmetric, required = TRUE)
regularize(A, EEV=extreme_eig(A,symmetric=TRUE), maxcondnum=1e12)
```

Arguments

M	Square matrix. Sparse matrices of class d[s|g]CMatrix (and some others too) are handled (some vagueness, as if it fails for some matrix types, an alternative function shoudl be easy to define based on this one as template.
A	Square matrix as M, assumed symmetric.
symmetric	Whether the matrix is symmetric. Helpful to select efficient methods for this case if the matrix class does not implies its symmetry.

required	Whether the computation should be attempted independently of the size of the matrix.
EEV	Two extreme eigenvalue in the return format of extreme_eig
maxcondnum	Target condition number when regularization is performed

Value

extreme_eig returns a vector of length 2, the largest and the smallest eigenvalues in this order. regularize returns a matrix, possibly in sparse format.

Examples

```
H10 <- Matrix::Hilbert(10)
extreme_eig(H10,symmetric=TRUE) # ratio > 1e13
rH10 <- regularize(H10)
extreme_eig(rH10,symmetric=TRUE)  # ratio = 1e12
```

---

fitme *Fitting function for fixed- and mixed-effect models with GLM response.*

Description

This is a common interface for fitting most models that **spaMM** can fit, from linear models to mixed models with non-gaussian random effects, except multivariate-response models (see fitmv for the latter). By default, it uses ML rather than REML (differing in this respect from the other fitting functions corrHLfit, HLCor and HLfit). It tends to use "outer optimization", i.e., generic optimization methods for estimating all dispersion parameters, rather than the iterative methods implemented in HLfit (see Details).

Usage

```
fitme(formula, data, family = gaussian(), init = list(), fixed = list(),
      lower = list(), upper = list(), resid.model = ~1, init.HLfit = list(),
      control = list(), control.dist = list(), method = "ML",
      HLmethod = method, processed = NULL, nb_cores = NULL, objective = NULL,
      weights.form = NULL, ...)
# '...' may notably include arguments
#     'rand.family', 'control.HLfit', and 'verbose'
# (see HLfit() documentation for them),
#     'covStruct', 'corrMatrix', 'adjMatrix' and 'distMatrix'
# (see HLCor() documentation for them).
```

Arguments

formula	Either a linear model formula (as handled by various fitting functions) or a predictor, i.e. a formula with attributes (see Predictor and examples below). See Details in spaMM for allowed terms in the formula.

`data` A data frame containing the variables in the response and the model formula.

`family` Either a response `family` or a `multi` value.

`init` An optional list of initial values for correlation and/or dispersion parameters for distribution family. The general syntax is the same as for the `fixed` argument, and a simplified syntax may be used when it is not ambiguous. E.g., `list(rho=1,nu=1,lambda=1,phi=1)` can be used to fix the `rho` and `nu` parameters of the Matérn family (see `Matern`), as well as the dispersion parameters denoted `lambda` and `phi` in `spaMM` (see Examples below, and in the additional `inits` documentation). All initial values are optional, but giving values for a dispersion parameter changes the ways it is estimated (see Details and Examples). `rho` may be a vector (see `make_scaled_dist`) and, in that case, it is possible that some or all of its elements are `NA`, for which `fitme` substitutes automatically determined initial values.

`fixed` A list similar to `init`, but specifying fixed values of the parameters not estimated. See `fixed` for further information; and keep in mind that fixed fixed-effect coefficients can be passed as the `etaFix` argument as part of the '...'.

`lower, upper` optional (sub)lists of values of parameters, in the same format as `init`, used to control lower or upper values in calls to `nloptr` or similar generic numerical optimization functions. See `optimBounds` for further information.

`resid.model` See identically named `HLfit` argument.

`init.HLfit` See identically named `HLfit` argument.

`control.dist` See `control.dist` in `HLCor`

`method, HLmethod`

Character: the fitting method to be used, such as `"ML"`, `"REML"` or `"PQL/L"`. `"ML"` is the default, in contrast to `"REML"` for `HLfit`, `HLCor` and `corrHLfit`. Other possible values of `HLfit`'s `method` argument are handled. `method=c(<"ML" or "REML">,"exp")` can be distinctly useful for slow fits of models with `Gamma(log)` family (see `method`).

`weights.form` Specification of prior weights by a one-sided formula: use `weights.form = ~ pw` instead of `prior.weights = pw`. The effect will be the same except that such an argument, known to evaluate to an object of class `"formula"`, is suitable to enforce safe programming practices (see `good-practice`).

`control` A list of (rarely needed) control parameters, with possible elements:

- `refit`, a boolean, or a `list` of booleans with possible elements `phi`, `lambda` and `ranCoefs`. If either element is set to TRUE, then the corresponding parameters are refitted by the internal `HLfit` methods (see Details), unless these methods were already selected for such parameters in the main fitting step. If `refit` is a single boolean, it affects all parameters. By default no parameter is refitted.
- `optimizer`, the numerical optimizer, specified as a string and whose default is controlled by the global **spaMM** option `"optimizer"`. Possible values are `"nloptr"`, `"bobyqa"`, `"L-BFGS-B"` and `".safe_opt"`, whose meanings are detailed in the documentation for the `optimizer` argument of `spaMM.options`. Better left unchanged unless suspect fits are obtained.

- `nloptr`, itself a list of control parameters to be copied in the `opts` argument of `nloptr`. Default value is given by `spaMM.getOption('nloptr')` and possibly other global **spaMM** options. Better left unchanged unless you are ready to inspect source code.
- `bobyqa, optim`, lists of controls similar to `nloptr` but for methods `"bobyqa"` and `"L-BFGS-B"`, respectively.

`nb_cores` For development purpose, not documented.

`processed` For programming purpose, not documented.

`objective` For development purpose, not documented.

`...` Optional arguments, notably those listed in the comments of the Usage section above, passed to (or operating as if passed to) either `HLfit` or `HLCor` or `mat_sqrt`.

Details

For approximations of likelihood, see `method`. For the possible structures of random effects, see `random-effects`. For the different results that may be obtained by using the `init` argument to select "inner"- vs."outer"-estimation methods for dispersion parameters λ and ϕ, and the availability of standard errors for these estimates, see `inner-vs-outer`.

Value

The return value of an `HLCor` or an `HLfit` call, with additional attributes. The `HLCor` call is evaluated at the estimated correlation parameter values. These values are included in the return object as its `$corrPars` member. The attributes added by `fitme` include the original call of the function (which can be retrived by `getCall`(<fitted object>), and information about the optimization call within `fitme`.

Examples

```
## Examples with Matern correlations
## A likelihood ratio test based on the ML fits of a full and of a null model.
 data("blackcap")
 (fullfit <- fitme(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap) )
 (nullfit <- fitme(migStatus ~ 1 + Matern(1|longitude+latitude),data=blackcap))
 ## p-value:
 1-pchisq(2*(logLik(fullfit)-logLik(nullfit)),df=1)

# See help("spaMM")" for examples of conditional
# autoregressive model and of non-spatial models.
# see help("COMPoisson"), help("negbin"), help("Loaloa"), help("inner-vs-outer"),
# and many other documentation pages, for further examples.
```

fitmv *Fitting multivariate responses*

Description

This function extends the fitme function to fit a joint model for different responses (following possibly different response families) sharing some random-effects, including a new type of random effect defined to exhibit correlations across different responses (see mv). It is also possible to declare shared fixed-effect coefficients among different submodels, using the X2X argument. Only a few features available for analysis of univariate response may not yet work (see Details).

Usage

```
fitmv(submodels, data, fixed=NULL, init=list(), lower=list(), upper=list(),
      control=list(), control.dist = list(), method="ML", init.HLfit=list(),
      X2X=NULL, aliases=NULL, ...)
```

Arguments

submodels	A list of sublists each specifying a model for each univariate response. The names given to each submodel in the main list are currently ignored. The names and syntax of elements within each sublist are those of a fitme call. In most cases, each sublist should not contain arguments whose names are those of formal arguments of fitmv itself (with the possible exception for fixed). prior.weights (or better, weights.form), if any, should be specified as part of a submodel.
data	A data frame containing the variables in the response and the model formulas.
fixed	A list of fixed values of the parameters controlling random effects. The syntax is that of the same argument in fitme (the optional fixed argument in each sublist of submodels may also be used but this feature may be confusing). Fixed phi values must be specified as a list, e.g., fixed=list(phi=list("2"=0.1)) to set the value for the second submodel.
init, lower, upper	Lists of initial values or bounds. The syntax is that of the same arguments in fitme. In these lists, random effects should be indexed according to their order of appearance in the total model (see Details). Any init, lower, or upper in a sublist of submodels will be ignored.
control	A list of control parameters, with possible elements as described for fitme
control.dist	See control.dist in HLCor
method	Character: the fitting method to be used, such as "ML", "REML" or "PQL/L". "ML" is the default, as for fitme and in contrast to "REML" for the other fitting functions. Other possible values of HLfit's method argument are handled.
init.HLfit	See identically named HLfit argument.

X2X	NULL, or a matrix, or a call to genX2X function. This argument allows one to specify fixed-effect coefficients shared between submodels, as further detailed in the X2X documentation. The matrix must have column names, defining the names of the fixed-effect coefficients to be fitted.
aliases	A list; experimental feature whose usage is explained in a dedicated aliases documentation.
...	Optional arguments passed to (or operating as if passed to) HLCor, HLfit or mat_sqrt, for example control.HLfit or the covStruct, distMatrix, corrMatrix or adjMatrix arguments of HLCor.

Details

Matching random effects across submodels, and referring to them;
Random effects are recognized as identical across submodels by matching the formula terms. As shown in the Examples, if the two models formulas share the (1|clinic) term, this term is recognized as a single random effect shared between the two responses. But the (1|clinic) and (+1|clinic) terms are recognized as distinct random effects. In that case, the init argument init=list(lambda=c('1'=1,'2'=0.5)) is shown to refer to these by names 1,2... where the order is defined as the order of first appearance of the terms across the model formulas in the order of the submodels list. Alternatively, the syntax fixed=list(lambda=c('clinic.1'=0.5,'clinic'=1)) works: this syntax makes order of input irrelevant but assumes that the user guesses names correctly (these are typically the names that appear in the summary of lambda values from the fit object or, more programmatically,
names(<fit object>$lambda.object$print_namesTerms)). Finally, fixed values of parameters can **also** be specified through each sub-model, with indices referring to the order of random effects with each model.

The matching of random-effect terms occurs after expansion of multIMRF terms, if any. This may have subtle consequences if two multIMRF terms differ only by their number of levels, as some of the expanded IMRF terms are then shared.

Capacities and limitations:
Practically all features of models that can be fitted by fitme should be available: this includes all combinations of GLM response families, residual dispersion models, and all types of random-effect terms, whether autocorrelated or not. Among the arguments handled through the ..., covStruct, distMatrix, corrMatrix should be effective; control.HLfit$LevenbergM and verbose=c(TRACE=TRUE) will work but some other controls available in fitme may not. Usage of the REMLformula argument is restricted as it cannot be used to specify a non-standard REML correction (but the more useful keepInREML attribute for fixed fixed-effect coefficients is handled).

The multi family-like syntax for multinomial models should not be used, but fitmv could provide other means to model multinomial responses.

Most post-fit functions work, at least with default arguments. This includes point prediction and prediction variances calculations sensu lato, including with newdata; but also simulate, spaMM_boot, confint, anova, update_resp, and update. The re.form argument now works for predict and simulate. Bootstrap computation may require special care for models where simulation of one response variable may depend on draws of another one (see Hurdle model example in the "Gentle introdution" to **spaMM**, https://gitlab.in2p3.fr/francois.rousset/spamm-ref/-/blob/master/vignettePlus/spaMMintro.pdf).

Prediction functions try to handle most forms of missing information in newdata (including information missing for a residual-dispersion model when predcitiosn fro mit are needed: see Examples). As information may be missing for some submodels but not others, different numbers of predictions are then returned for different submodels. As for univariate-response models, predict will return point predictions as a single 1-column matrix, here concatenating the prediction results of the different submodels. The nobs attribute specifies how may values pertain to each submodel.

Some plotting functions may fail. update.formula fails (see update_formulas for details). terms returns a list, which is not usable by other base R functions. stats::step is a good example of resulting limitations, as it is currently unable to perform any sensible operation on fitmv output. spaMM::MSFDR which calls stats::step likewise fails. multcomp::glht fails.

A perhaps not entirely satisfying feature is that simulate by default stacks the results of simulating each submodel in a single vector. Some non-trivial reformatting may then be required to include such simulation results in a suitable newdata data frame with (say) sufficient information for prediction of all responses. The syntax
update_resp(<fit>, newresp = simulate(<fit>, ...), evaluate = FALSE)$data
may be particularly useful to reformat simulation results in this perspective.

Which arguments belong to submodels**?**:
Overall, arguments specifying individual submodels should go into submodels, while other arguments of fitmv should be those potentially affecting several submodels (notably, random-effect structures, lower, and upper) and fitting controls (such as init and init.HLfit). One rarely-used exception is REMLformula which controls the fitting method but should be specified through the submodels.

The function proceeds by first preprocessing all submodels independently, before merging the resulting information by matching random effects across submodels. The merging operation includes some checks of consistency across submodels, implying that redundant arguments may be needed across submodels (e.g. specifying twice a non-default rand.family for a random effect shared by two submodels).

Value

A (single) list with additional classes HLfit (as returned by other fitting functions in **spaMM**) and fitmv. It contains a families element describing the response families, instead of the family elements of fitted objects for univariate response (the family extractor works in both cases). The data element retains rows of the data for which some responses or some predictors are missing, in contrast to the univariate case.

See Also

See X2X for details, examples and an helper function facilitating the use of this argument. See further examples in mv (modelling correlated random effects over the different submodels), and residVar.

Examples

```
### Data preparation
data(clinics)
climv <- clinics
(fitClinics <- HLfit(cbind(npos,nneg)~treatment+(1|clinic),
                     family=binomial(),data=clinics))
```

```
set.seed(123)
climv$np2 <- simulate(fitClinics, type="residual")
#
### fits

#### Shared random effect
(mvfit <- fitmv(
  submodels=list(mod1=list(formula=cbind(npos,nneg)~treatment+(1|clinic),family=binomial()),
                 mod2=list(formula=np2~treatment+(1|clinic),
                           family=poisson(), fixed=list(lambda=c("1"=1)))),
   data=climv))

# Two univariate-response independent fits because random effect terms are distinct
# (note how two lambda values are set; same syntax for 'init' values):
(mvfitind <- fitmv(
  submodels=list(mod1=list(formula=cbind(npos,nneg)~treatment+(1|clinic),family=binomial()),
                 mod2=list(formula=np2~treatment+(+1|clinic),family=poisson())),
   data=climv, fixed=list(lambda=c('1'=1,'2'=0.5)))) # '1': (1|clinic); '2': (+1|clinic)

#### Specifying fixed (but not init) values in submodels is also possible (maybe not a good idea)
# (mvfitfix <- fitmv(
#   submodels=list(mod1=list(formula=cbind(npos,nneg)~treatment+(1|clinic),
#                       family=binomial(),fixed=list(lambda=c('1'=1))), # '1': (1|clinic)
#                  mod2=list(formula=np2~treatment+(+1|clinic),family=poisson(),
#                         fixed=list(lambda=c('1'=0.5)))),            # '2': (+1|clinic)
#   data=climv))

#### Prediction with a residual-dispersion model
set.seed(123)
beta_dat <- data.frame(y=runif(100),grp=sample(2,100,replace = TRUE), x_het=runif(100),
                       y2=runif(100))
(mvfit <- fitmv(list(list(y ~1+(1|grp), family=beta_resp(), resid.model = ~x_het),
                  list(y2 ~1+(1|grp), family=beta_resp())),
              data= beta_dat))

misspred <- beta_dat[1:3,]
misspred$x_het[1] <- NA # missing info for residual variance of first submodel

## => prediction missing when this info is needed:
#
length(predict(mvfit, newdata=misspred)) # 6 values: missing info not needed for point predictions
length(get_residVar(mvfit, newdata=misspred)) # 5 values
length(get_respVar(mvfit, newdata=misspred)) # 5 values
#  Missing info not needed for predVar (**as opposed to respVar**)
length(get_predVar(mvfit, newdata=misspred)) # 6 values
#
# Same logic for interval computations:
#
dim(attr(predict(mvfit, newdata=misspred, intervals="respVar"),"intervals")) # 5,2
dim(attr(predict(mvfit, newdata=misspred, intervals="predVar"),"intervals")) # 6,2
#
# Same logic for simulate():
#
```

```
length(simulate(mvfit, newdata=misspred)) # 5 as simulation requires residVar
```

`fixed` *Fixing some parameters*

Description

The fitting functions allow all parameters to be fixed rather than estimated:
* Fixed-effect coefficients can be set by way of the `etaFix` argument (linear predictor coefficients) for all fitting functions.
* Random-effect parameters and the `phi` parameter of the `gaussian`, `Gamma` and `tweedie` response families can be set for all fitting function by the `fixed` argument (see Details for an alternative argument with the same effect, but using `fixed` uniformly is simpler).
* The ad-hoc dispersion parameter of some response families (`COMPoisson`, `negbin1`, `negbin2`, `beta_resp`, `betabin` and possibly future ones) can be fixed using the ad-hoc argument of such families rather than by `fixed`.

Details

etaFix is a list with single documented element `beta`, which should be a vector of (a subset of) the coefficients (β) of the fixed effects, with names as shown in a fit without such given values. If REML is used to fit random effect parameters, then `etaFix` affects by default the REML correction for estimation of dispersion parameters, which depends only on which β coefficients are estimated rather than given. This default behaviour will be overridden whenever a non-null `REMLformula` is provided to the fitting functions (see Example). Alternatively, with a non-NULL `etaFix$beta`, REML can also be performed as if all β coefficients were estimated, by adding attribute `keepInREML=TRUE` to `etaFix$beta`; in that case the REML computation is by default that implied by the fixed effects in the full model formula, unless a non-default `REMLformula` is also used.

The `fixed` argument coexists in some functions with another argument that has the same effect: `ranFix` for `HLfit` and `corrHLfit`, and `ranPars` for `HLCor`. Do not use both one such argument and `fixed` in a call. This older diversity of names was confusing, but its logic was that `ranFix` allows one to fix parameters that `HLfit` and `corrHLfit` would otherwise estimate, while `ranPars` can be used to set correlation parameters that `HLCor` does not estimate but nevertheless requires (e.g., Matérn parameters).

Theses arguments for fixing random-effect parameters all have a common syntax. They is a list, with the following possible elements, whose nature is further detailed below:
* **phi** (variance of residual error, for gaussian and Gamma HGLMs),
* **lambda** (random-effect variances, except for random-coefficient terms),
* **ranCoefs** (random-coefficient parameters),
* **corrPars** (correlation parameters, when handled by the fitting function).
* Individual correlation parameters such as **rho, nu, Nugget, ARphi**... are also possible top-level elements of the list when there is no ambiguity as to which random effect these correlation parameters apply. This syntax was conceived when `spaMM` handled a single spatial random effect, and it is still convenient when applicable, but it should not be mixed with `corrPars` element usage.

phi may be a single value or a vector of the same length as the response vector (the number of rows in the `data`, once non-informative rows are removed).

lambda may be a single value (if there is a single random effect, or a vector allowing to specify unambiguously variance values for some random effect(s). It can thus take the form `lambda=c(NA,1)` or `lambda=c("2"=1)` (note the name) to assign a value only to the variance of the second of two random effects.

ranCoefs is a `list` of numeric vectors, each numeric vector specifying the variance and correlation parameters for a random-coefficient term. As for `lambda`, it may be incomplete, using names to specify the random effect to which the parameters apply. For example, to assign variances values 3 and 7, and correlation value -0.05, to the second random effect in a model formula, one can use `ranCoefs=list("2"=c(3,-0.05,7))` (note the name). The elements of each vector are variances and correlations, matching those of the printed summary of a fit. The order of these elements must be the order of the `lower.tri` of a covariance matrix, as shown e.g. by
`m2 <- matrix(NA, ncol=2,nrow=2); m2[lower.tri(m2,diag=TRUE)] <- seq(3); m2`.
`fitme` accepts partially fixed parameters for a random coefficient term, e.g.,
`ranCoefs=list("2"=c(NA,-0.05,NA))`, although this may not mix well with some obscure options, such as
`control=list(refit=list(ranCoefs=TRUE))` which will ignore the fixed values. Several examples in the package documentation illustrate the use of the convenience function `ranCoefs_for_diag` to fit only the variances. `GxE` further shows how to use partially-fixed `ranCoefs` to fit different variances for different levels of a factor.

corrPars is a list, and it may also be incomplete, using names to specify the affected random effect as shown for `lambda` and `ranCoefs`. For example, `fixed=list(corrPars=list("1"=list(nu=0.5)))` makes explicit that `nu=0.5` applies to the first (`"1"`) random effect in the model formula. Its elements may be the correlation parameters of the given random effect. For the Matérn model, these are the correlation parameters `rho` (scale parameter(s)), `nu` (smoothness parameter), and (optionally) `Nugget` (see `Matern`). The `rho` parameter can itself be a vector with different values for different geographic coordinates. For the `adjacency` model, the only correlation parameter is a scalar `rho` (see `adjacency`). For the `AR1` model, the only correlation parameter is a scalar `ARphi` (see `AR1`). Consult the documentation for other types of random effects, such as `Cauchy` or `IMRF`, for any information missing here.

See Also

Convenience functions for specifying constraints on random-coefficient terms: `ranCoefs_for_diag`, `lev2bool`.

Examples

```
## Not run:
data("wafers")
# Fixing random-coefficient parameters:
fitme(y~X1+(X2|batch), data=wafers, fixed=list(ranCoefs=list("1"=c(2760, -0.1, 1844))))
##  HLfit syntax for the same effect (except that REML is used here)
# HLfit(y~X1+(X2|batch), data=wafers, ranFix=list(ranCoefs=list("1"=c(2760, -0.1, 1844))))


### Fixing coefficients of the linear predictor:
#
```

```
## ML fit
#
fitme(y ~X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch), data=wafers, family=Gamma(log),
      etaFix=list(beta=c("(Intercept)"=5.61208)))
#
## REML fit
# Evaluation of restricted likelihood depends on which fixed effects are estimated,
# so simply fixing the coefficients to their REML estimates will not yield
# the same REML fits, as see by comparing the next two fits:
#
unconstr <- fitme(y ~X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch), data=wafers,
                  family=Gamma(log), method="REML")
#
# Second fit is different from 'unconstr' despite the same fixed-effects:
naive <- fitme(y ~X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch), data=wafers, family=Gamma(log),
               method="REML", etaFix=list(beta=fixef(unconstr)))
#
# Using REMLformula to obtain the same REML fit as the unconstrained one:
fitme(y ~X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch), data=wafers, family=Gamma(log),
      method="REML", etaFix=list(beta=fixef(unconstr)),
      REMLformula=y ~X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch))

data("Loaloa")
# Fixing some Matern correlation parameters, in fitme():
fitme(cbind(npos,ntot-npos) ~ elev1 +Matern(1|longitude+latitude),
             data=Loaloa,family=binomial(),fixed=list(nu=0.5,Nugget=2/7))
# Fixing all mandatory Matern correlation parameters, in HLCor():
HLCor(cbind(npos,ntot-npos) ~ elev1 + Matern(1|longitude+latitude),
             data=Loaloa,family=binomial(),ranPars=list(nu=0.5,rho=0.7))

## End(Not run)
```

`fixedLRT` *Likelihood ratio test of fixed effects.*

Description

`fixedLRT` performs a likelihood ratio (LR) test between two models, the "full" and the "null" models, currently differing only in their fixed effects. Parametric bootstrap p-values can be computed, either using the raw bootstrap distribution of the likelihood ratio, or a bootstrap estimate of the Bartlett correction of the LR statistic. This function differs from `LRT` in its arguments (model fits for `LRT`, versus all arguments required to fit the models for `fixedLRT`), and in the format of its return value.

Usage

```
fixedLRT(null.formula, formula, data, method, HLmethod = method,
         REMLformula = NULL, boot.repl=0, control="DEPRECATED",
         control.boot="DEPRECATED", fittingFunction, seed=NULL,
         resp_testfn = NULL, weights.form = NULL, ...)
```

Arguments

null.formula	Either a formula (as in glm) or a predictor (see Predictor) for the null model.
formula	Either a formula or a predictor for the full model.
data	A data frame containing the variables in the model.
method	A method to fit the full and null models. See method information about such methods. The two most meaningful values of method in fixedLRT calls are: 'ML' for an LRT based on ML fits (generally recommended); and 'PQL/L' for an LRT based on PQL/L fits (recommended for spatial binary data). Also feasible, but more tricky, and not really recommended (see Rousset and Ferdy, 2014), is 'REML'. This will perform an LRT based on two REML fits of the data, *both* of which use the same conditional (or "restricted") likelihood of residuals for estimating dispersion parameters λ and ϕ (see REMLformula argument). Further, REML will not be effective on a given dispersion parameter if a non-trivial init.corrHLfit value is provided for this parameter.
HLmethod	Kept for back-compatibility. Same as method, but may work only for fittingFunction=corrHLfit.
REMLformula	a formula specifying the fixed effects which design matrix is used in the REML correction for the estimation of dispersion parameters, if these are estimated by REML. This formula is by default that for the *full* model.
weights.form	Specification of prior weights by a one-sided formula: use weights.form = ~ pw instead of prior.weights = pw. The effect will be the same except that such an argument, known to evaluate to an object of class "formula", is suitable to enforce safe programming practices (see good-practice).
boot.repl	the number of bootstrap replicates.
control	Deprecated.
control.boot	Deprecated.
fittingFunction	Character string giving the function used to fit each model: either "corrHLfit" or "fitme". Default is "corrHLfit" for small data sets (fewer than 300 observations), and "fitme" otherwise, but this may change in future versions.
seed	Passed to simulate.HLfit
resp_testfn	See argument resp_testfn of spaMM_boot
...	Further arguments passed to or from other methods; presently, additional arguments passed to fitting functions.

Details

Comparison of REML fits is a priori not suitable for performing likelihood ratio tests. Nevertheless, it is possible to contrive them for testing purposes (Welham & Thompson 1997). This function generalizes some of Wehlam & Thompson's methods to GLMMs.

See Details in LRT for details of the bootstrap procedures.

Value

An object of class `fixedLRT`, actually a list with as-yet unstable format, but here with typical elements (depending on the options)

`fullfit`	the HLfit object for the full model;
`nullfit`	the HLfit object for the null model;
`LRTori`	A likelihood ratio chi-square statistic
`LRTprof`	Another likelihood ratio chi-square statistic, after a profiling step, if any.
`df`	the number of degrees of freedom of the test.
`trace.info`	Information on various steps of the computation.

and, if a bootstrap was performed, the additional elements described in `LRT`.

References

Rousset F., Ferdy, J.-B. (2014) Testing environmental and genetic effects in the presence of spatial autocorrelation. Ecography, 37: 781-790. doi:10.1111/ecog.00566

Welham, S. J., and Thompson, R. (1997) Likelihood ratio tests for fixed model terms using residual maximum likelihood, J. R. Stat. Soc. B 59, 701-714.

See Also

See `LRT` for simular tests with a different interface, and perhaps `as_LMLT` for access to a different testing approach for LMMs, implemented in `lmerTest::contest`.

Examples

```
if (spaMM.getOption("example_maxtime")>1.9) {
 data("blackcap")
 ## result comparable to the corrHLfit examples based on blackcap
 fixedLRT(null.formula=migStatus ~ 1 + Matern(1|longitude+latitude),
       formula=migStatus ~ means + Matern(1|longitude+latitude),
       method='ML',data=blackcap)
}
if (spaMM.getOption("example_maxtime")>156) {
 ## longer version with bootstrap
 fixedLRT(null.formula=migStatus ~ 1 + Matern(1|longitude+latitude),
       formula=migStatus ~ means + Matern(1|longitude+latitude),
       method='ML',data=blackcap, boot.repl=100, seed=123)
 }
```

fix_predVar *Prediction from models with nearly-singular covariance matrices*

Description

This explains how to handle a warning occurring in computation of prediction variance, where the user is directed here.

For **Matern or Cauchy** correlation models with vanishing scale factor for distances, a warning may be produced when `predict.HLfit` (or `get_predVar`, etc.) is called with non-NULL `newdata`, because a nearly-singular correlation matrix of the random effect is met. **To decide what to do** in that case, users should compare the values of `get_predVar(.)` and `get_predVar(., newdata=myfit$data)` (see Example below). In the absence of numerical inaccuracies, The two values should be identical, and in the presence of such inaccuracies, the more reliable value is the first one. In really poor cases, the second syntax may yield negative prediction variances. If users deem the inaccuracies too large, they should use `control=list(fix_predVar=TRUE)` in the next call to `predict.HLfit` (or `get_predVar`, etc.) as shown in the Example. The drawback of this control is that the computation may be slower, and might even exceed memory capacity for large problems (some matrix operations being performed with exact rational arithmetic, which is memory-consuming for large matrices). it is also still experimental, in the sense that I fear that bugs (`stop`) may occur. If the user instead chooses `control=list(fix_predVar=FALSE)`, the default standard floating-point arithmetic is used, but no warning is issued.

For `fix_predVar` left NULL (the default), standard floating-point arithmetic is also used. But in addition (with exceptions: see Details), the warning keeps being issued, and the (possibly costly) computation of the inverse of the correlation matrix is not stored in the fitted model object, hence is repeated for each new prediction variance computation. This is useful to remind users that something needs to be done, but for programming purposes where repeated warnings may be a nuisance, one can use `control=list(fix_predVar=NA)` which will issue a warning then perform as `control=list(fix_predVar=FALSE)`, i.e. store an approximate inverse so the warning is not issued again. Finally, `control=list(fix_predVar=NaN)` will remove the inverse of the correlation matrix from the fitted model object, and start afresh as if the control was NULL.

Details

Nearly-singular correlation matrices of random effects occur in several contexts. For random-slope models, it commonly occurs that the fitted correlation between the random effects for Intercept and slope is 1 or -1, in which case the correlation matrix between these random effects is singular. This led to quite inaccurate computations of prediction variances in spaMM prior to version 3.1.0, but this problem has been fixed.

`control=list(fix_predVar=NaN)` may be more appropriate than `control=list(fix_predVar=NULL)` when `predict.HLfit` is called through code that one cannot control. For this reason, spaMM provides another mode of control of the default. It will convert `control=list(fix_predVar=NULL)` to other values when the call stack has call names matching the patterns given by `spaMM.getOption("fix_predVar")` (as understood by `grep`). Thus if `spaMM.getOption("fix_predVar")$"NA"=="MSL|b` the default behaviour is that defined by `control=list(fix_predVar=NA)` when `predict.HLfit` is called through `Infusion::MSL` or `blackbox::bboptim`. FALSE or TRUE are handled in a similar way.

Examples

```
data("blackcap")
fitobject <- corrHLfit(migStatus ~ 1 + Matern(1|longitude+latitude),data=blackcap,
                       ranFix=list(nu=10,rho=0.001)) ## numerically singular C
get_predVar(fitobject,newdata=blackcap[6,])
## => warning => let us apply the recommended procedure:
get_predVar(fitobject)
get_predVar(fitobject,newdata=fitobject$data)
# Negative values again in the second case => easy decision:
get_predVar(fitobject,newdata=blackcap[1:6,],
            control=list(fix_predVar=TRUE)) # now it's accurate
            # and the accuracy control is stored in the object:
get_predVar(fitobject,newdata=blackcap[1:6,])
# Clean and start afresh:
get_predVar(fitobject,newdata=blackcap[1:6,],
            control=list(fix_predVar=NaN))
```

---

freight *Freight dataset*

Description

A set of data on airfreight breakage. Data are given on 10 air shipments, each carrying 1000 ampules of some substance. For each shipment, the number of ampules found broken upon arrival, and the number of times the shipments were transferred from one aircraft to another, are recorded.

Usage

```
data("freight")
```

Format

The data frame includes 10 observations on the following variables:

broken number of ampules found broken upon arrival.

transfers number of times the shipments were transferred from one aircraft to another.

id Shipment identifier.

Source

The data set is reported by Kutner et al. (2003) and used by Sellers & Shmueli (2010) to illustrate COMPoisson analyses.

References

Kutner MH, Nachtsheim CJ, Neter J, Li W (2005, p. 35). Applied Linear Regression Models, Fourth Edition. McGraw-Hill.

Sellers KF, Shmueli G (2010) A Flexible Regression Model for Count Data. Ann. Appl. Stat. 4: 943–961

Examples

```
## see ?COMPoisson for examples
```

`get_cPredVar` *Estimation of prediction variance with bootstrap correction*

Description

This function is similar to `get_predVar` except that is uses a bootstrap procedure to correct for bias in the evaluation of the prediction variance.

Usage

```
get_cPredVar(pred_object, newdata, nsim, seed, type = "residual",
             variances=NULL, nb_cores = NULL, fit_env = NULL,
             sim_object=pred_object)
```

Arguments

`pred_object`	an object of class `HLfit`, as returned by the fitting functions in `spaMM`.
`newdata`	passed to `predict.HLfit` (it thus represents a prediction design, not to be confused with the bootstrap samples)
`nsim`	passed to `simulate.HLfit`
`seed`	passed to `simulate.HLfit`
`type`	passed to `simulate.HLfit`
`variances`	NULL or list; `variances["cov"]` will be passed to `predict.HLfit` to control whether a covariance matrix is computed or not. Other elements are currently ignored.
`nb_cores`	integer: number of cores to use for parallel computation of bootstrap. The default is `spaMM.getOption("nb_cores")`, and 1 if the latter is NULL. `nb_cores=1` prevents the use of parallelisation procedures.
`fit_env`	For parallel computations: an environment containing objects to be passed to the cores. They should have the same name in `fit_env` as in the environment they are passed from.
`sim_object`	an object of class `HLfit`, passed to `simulate.HLfit` as its `object` argument. Simulating from this object must produce response values that can be used as replacement to those of the original fitted `pred_object`. In standard usage, `sim_object=pred_object` (the default).

Details

The result provided by `get_cPredVar` is similar to the CMSEP (Conditional Mean Standard Error of Prediction) introduced by Booth and Hobert (1998; "B&H"). This paper is known for pointing the importance of using conditional variances when they differ from unconditional ones. This is hard to miss in spatial models, where the relevant prediction variance typically depends on the variance of random effects conditional on the data. Thus, the alternative function `get_predVar` already accounts for this and returns a prediction variance that depends on a joint covariance of fixed-effect estimates and of random effects given the data.

B&H also used a conditional bootstrap procedure to correct for some bias. `get_cPredVar` implements a similar procedure, in contrast to `get_predVar`. Their conditional bootstrap procedure is not applicable for autocorrelated random effects, and parametric bootstrapping of the residuals of the fitted model (as implied by the default value of argument `type`) is used instead here. Apart from this difference, the returned value includes exactly the same terms as those discussed by B&H: their "naive estimate" ν_i and its bootstrap correction b_i, their correction β for uncertainty in fixed-effect coefficients, and their correction σ^2 for uncertainty in dispersion parameters.

This use of the bootstrap does not account for uncertainty in correlation parameters "outer-optimized" by `fitme` or `corrHLfit`, because the correlation parameters are fixed when the model is refitted on the bootstrap replicates. Even if it the correlation parameters were refitted, the full computation would not be sufficient to account for uncertainty in them. To account for uncertainty in correlation parameters, one should rather perform a parametric bootstrap of the full model (typically using `spaMM_boot(., type="residual")`), which may take much more time.

The "naive estimate" ν_i is not generally an estimate of anything uniquely defined by the model parameters: for correlated random effects, it depends on the "root" of the correlation matrix of the random effects, which is not unique. Thus ν_i is not unique, and may differ for example for equivalent fits by sparse-precision methods vs. other methods. Nevertheless, `attr(cpredvar,"info")$naive` does recover published values in the Examples below, as they involve no correlation matrix.

Value

A vector of prediction variances, with an attribute `info` which is an **environment** containing variables:

`SEs`	the standard errors of the estimates (which are those of the bootstrap replicates)
`bias`	the bias term
`maive`	B&H's "naive" ν_i

References

Booth, J.G., Hobert, J.P. (1998) Standard errors of prediction in generalized linear mixed models. J. Am. Stat. Assoc. 93: 262-272.

Examples

```
## Not run:
if(requireNamespace("rsae", quietly = TRUE)) {
  # LMM example from Booth & Hobert 1998 JASA
  data("landsat", package = "rsae")
 fitCorn <- fitme(HACorn ~ PixelsCorn + PixelsSoybeans + (1|CountyName),data=landsat[-33,])
```

```
  newXandZ <- unique(data.frame(PixelsCorn=landsat$MeanPixelsCorn,
                                PixelsSoybeans=landsat$MeanPixelsSoybeans,
                                CountyName=landsat$CountyName))
 (cpredvar <- get_cPredVar(fitCorn, newdata=newXandZ, nsim=200L, seed=123)) # serial computation
  (cpredvar <- get_cPredVar(fitCorn, newdata=newXandZ, nsim=200L, seed=123,
        nb_cores=parallel::detectCores(logical=FALSE)-1L,
        fit_env=list2env(list(newXandZ=newXandZ))))
}

# GLMM example from Booth & Hobert 1998 JASA
data(clinics)
fitClinics <- HLfit(cbind(npos,nneg)~treatment+(1|clinic),family=binomial(),data=clinics)
#
(get_cPredVar(fitClinics, newdata=clinics[1:8,], nsim=200L, seed=123))  # serial computation
(get_cPredVar(fitClinics, newdata=clinics[1:8,], nsim=200L, seed=123,
      nb_cores=parallel::detectCores(logical=FALSE)-1L,
      fit_env=list2env(list(clinics=clinics))))

## End(Not run)
```

`get_inits_from_fit` *Initiate a fit from another fit*

Description

`get_inits_from_fit` is an extractor of some fitted values from a fit in a convenient format to initiate a next fit.

Usage

```
get_inits_from_fit(from, template = NULL, to_fn = NULL, inner_lambdas=FALSE)
```

Arguments

`from`	Fit object (inheriting from class "`HLfit`") from which fitted values are taken.
`template`	Another fit object. Usage with a `template` fit object is suitable for refitting this object using fitted values from the `from` object as starting values.
`to_fn`	NULL or character: the name of the function to be used the next fit. If NULL, taken from `template` (if available), else from `from`. It is meaningful to provide a `to_fn` distinct from the function used to fit a `template`.
`inner_lambdas`	Boolean; Whether the output should include estimates of the dispersion parameters estimated by the iterative methods (see `inner-vs-outer`).

Value

A list with elements

`init`, `init.corrHLfit`
(depending on the fitting function) giving initial values for outer-optimization;

`init.HLfit` giving initial values for the iterative algorithms in `HLfit`. It is itself a list with possible elements:

`fixef` for the coefficients of the linear predictor, adjusted to the format of the coefficients of the linear predictor of the `template` object, if available;

`ranCoefs` random-coefficients parameters (if **not** outer-optimized).

Undocumented attributes may be attached, and some of the parameters may be transformed.

See Also

`get_ranPars` and `VarCorr`.

Examples

```
## Not run:
data("blackcap")
(corrhlfit <- corrHLfit(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap,
                        method="ML"))
inits <- get_inits_from_fit(corrhlfit, to_fn = "fitme")
(fitfit <- fitme(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap,
                 init=inits[["init"]]))
inits <- get_inits_from_fit(corrhlfit, template = fitfit)
fitme(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap,
      init=inits[["init"]])
# In these examples, inits$init.HLfit is useless
# as it is ignored when LMMs are fitted by fitme().

## End(Not run)
```

---

get_matrix *Extract matrices from a fit*

Description

`get_matrix` is a first attempt at a unified extractor of various matrices from a fit. All augmented matrices follow (Henderson's) block order (upper blocks: X,Z; lower blocks: 0,I). `get_ZALMatrix` returns the design matrix for the random effects v.

Usage

```
get_matrix(object, which="model.matrix", augmented=TRUE, ...)
get_ZALMatrix(object, force_bind=TRUE)
```

Arguments

`object` An object of class `HLfit`, as returned by the fitting functions in `spaMM`.

augmented — Boolean; whether to return a matrix for all model coefficients (augmented matrix for fixed-effects coefficients and random-effect predictions) or a matrix only for fixed effects. Not operative for all which values (currently only for which="left_ginv").

which — Which element to extract. For "model.matrix", the design matrix for fixed effects (similarly to stats::model.matrix); for "ZAL", the design matrix for random effects (same as get_ZALMatrix()), while "ZA" and "L" may return these two factors (detailed in random-effects); for "AugX", the (unweighted) augmented design matrix of the least-square problem; for "hat_matrix", the projection matrix that gives model predictions from the (augmented) response vector; for "left_ginv", the pseudo-inverse that gives the model coefficients from the (augmented) response vector. See Details for further definitions and options for functions of the augmented design matrix.

force_bind — Boolean; with the non-default value FALSE, the function may return an object of class ZAXlist, which is poorly documented and for development purposes only.

... — Other arguments that may be needed in some future versions of spaMM.

Details

(Given the pain that it is to write maths in R documentation files, readers are gently asked to be tolerant about any imperfections of the following).

Model coefficients estimates of a (weighted) linear model can be written as $\mathbf{(X'WX)^{-1}X'Wy}$ where $\mathbf{X}$ is the design matrix for fixed effects, $\mathbf{W}$ a diagonal weight matrix, and $\mathbf{y}$ the response vector. In a linear mixed model, the same expression holds in terms of Henderson's augmented design matrix, of an augmented (still diagonal) weight matrix, and of an augmented response vector. For GLMMs and hierarchical GLMs generally, the solution of each step of the iteratively reweighted least squares algorithm again has the same expression in terms of appropriately defined augmented matrices and vectors.

get_matrix returns, for given values of the which argument, the following matrices from the model fit:
"AugX": $\mathbf{X}$ where, here and below $\mathbf{X}$ is the augmented design matrix when relevant;
"wei_AugX": $\mathbf{WX}$;
"wAugX": $\sqrt{(\mathbf{W})}\mathbf{X}$;
"left_ginv": $\mathbf{(X'WX)^{-1}X'W}$ (the name stems from the fact that it is generalized inverse, denoted $\mathbf{X^-}$, since $\mathbf{XX^-X=X}$, and it is a left one, since $\mathbf{X^-X}$ is an identity matrix when $\mathbf{X}$ has full rank);
"hat_matrix": $\mathbf{XX^-=X\,(X'WX)^{-1}X'W}$;
"fixef_left_ginv": same as "left_ginv" but for the fixed-effect design matrix only (not to be confused with the corresponding block of "left_ginv");
"beta_v_cov": joint covariance matrix of estimates/predictions of fixed-effect coefficients and random effects (the $\mathbf{v}$ in random-effects);
"v_condcov": covariance matrix of predictions of random effects (the $\mathbf{v}$ in random-effects) given fixed-effect coefficients;
"X2X": returns the X2X matrix.

Value

A matrix, possibly in sparseMatrix format.

See Also

vcov for the variance-covariance matrix of the fixed-effects coefficients, and Corr for correlation matrices of random effects.

---

get_ranPars *Operations on lists of parameters*

Description

get_fittedPars returns estimated parameters.

get_ranPars returns various subsets of random-effect parameters (correlation or variance parameters), as controlled by its which argument. It is one of several extractors for fixed or estimated parameters of different classes of parameters, for which a quick guide is

get_ranPars: for random-effect parameters, excluding residual dispersion (with a subtlety for corrFamily models: see Details);
VarCorr: alternative extractor for random-effect (co)variance and optionally residual variance, in a data frame format;
residVar: for residual variance parameters, family dispersion parameters, or information about residual variance models;
get_residVar: alternative extractor of residual variances with different features inherited from get_predVar;
get_inits_from_fit: extracts estimated parameters from a fit, in a different format from get_fittedPars.

remove_from_parlist removes elements from a list of parameters, and from its type attribute.

Usage

```
get_fittedPars(object, partial_rC="rm", phiPars=TRUE)
get_ranPars(object, which=NULL, verbose=TRUE,
            lambda_names = "Group.Term", ...)
remove_from_parlist(parlist, removand=NULL, rm_names=names(unlist(removand)))
```

Arguments

object	An object of class HLfit, as returned by the fitting functions in spaMM.
partial_rC	Controls handling of partially-fixed random coefficients. The default as set by "rm" is to remove the fixed values as for other parameters. But alternative option "keep" will keep the fixed value, and NA will replace it by a NA.
phiPars	Boolean: whether to include the parameters of any residual-dispersion model for phi (se phi-resid.model) in the rdisPars element of the returned list.
which	NULL or character string. Use which="corrPars" to get the correlation parameters. Use which="lambda" to get variances. see Details for the meaning of this for heteroscedastic models, and Value for other possible which values.
...	Other arguments that may be needed by some method.

verbose	Boolean: Whether to print some notes.
parlist	A list of parameters. see Details.
removand	Optional. A list of parameters to be removed from parlist.
rm_names	Names of parameters to be removed from parlist. Mandatory if removand is not given.
lambda_names	By default the names of the lambda vector are built from the Group (RHS of random effect term of the for (LHS|RHS)) and Term (variable from LHS). By setting a non-default value of lambda_names the names will be integer indices of the random-effect term in the model formula (currently, for which="ranef_var" or NULL.

Details

For heteroscedastic random effects, such as conditional autoregressive models, the variance parameter "lambda" refers to a common scaling coefficient. For other random-effect models, "lambda" typically refers to the single variance parameter.

remove_from_parlist is designed to manipulate structured lists of parameters, such as a list with elements phi, lambda, and corrPars, the latter being itself a list structured as the return value of get_ranPars(.,which="corrPars"). parlist may have an attribute type, also with elements phi, lambda, and corrPars... If given, removand must have the same structure (but typically not all the elements of parlist); otherwise, rm_names must have elements which match names of unlist(names(parlist)).

If a corrFamily parameter is fixed through the formula term, as in ARp(1|time, p=3, fixed=c(p2=0)), the fixed parameter is not considered a model parameter and get_ranPars will not extract it from the object. However, the parameter will be extracted if it has been fixed through fitme's fixed argument rather than through the formula term (see example in ARp).

Value

get_fittedPars returns a list of model parameters, with possible elements: beta (fixed-effect coefficients); lambda, phi, ranCoefs and corrPars (same meaning as in fixed parameters); hyper, for multIMRF models; the residual-dispersion parameters beta_prec, NB_shape and COMP_nu when they are single scalars; and rdisPars for more complex residual-dispersion parameters. See the specific resid.model and phi-resid.model documentations for the rdisPars format, dependent on the nature of the residual-dispersion parameter being modelized. Use residVar(., which="fam_parm") to extract the vector of fitted values of the dispersion parameter.

get_ranPars(.,which="corrPars") returns a (possibly nested) list of correlation parameters (or NULL if there is no such parameter). Top-level elements correspond to the different random effects. The list has a "type" attribute having the same nested-list structure and describing whether and how the parameters where fitted: "fix" means they where fixed, not fitted; "var" means they were fitted by HLfit's specific algorithms; "outer" means they were fitted by a generic optimization method.

get_ranPars(.,which="lambda") returns a vector of variance values, one per random effect, including both fixed, "outer"- and "inner"-estimated ones (see inner-vs-outer). The variances of random-coefficients terms with correlation parameters are not included.

get_ranPars(.,which="outer_lambda") returns only "outer"-optimized variance parameters, ignoring those fitted by HLfit's specific algorithms.

`get_ranPars(.,which=NULL)` (the default) is not fully defined. It returns a list including the results of `which="lambda"` and `which="corrPars"`, but possibly other elements too.

`get_ranPars(.,which="fitted")` is designed to provide fitted parameters with respect to which an information matrix is to be calculated (using **numDeriv**. It excludes fixed values, and has no `type` attribute.

`get_ranPars(. which="ranef_var")` (experimental) returns a list with elements

`Var` same as `get_ranPars(.,which="lambda")`

`lembda_est` A vector of variance values, one for each level of each random effect

`outer` A vector or outer-optimized variance values, as returned by `get_ranPars(.,which="outer_lambda")`

... Other elements, subject to change in later versions.

`remove_from_parlist` returns a list of model parameters with given elements removed, and likewise for its (optional) `type` attribute. See Details for context of application.

See Also

See `get_fittedPars`, `VarCorr`, `residVar`, `get_residVar`, or `get_inits_from_fit` as described in the quick guide above.

Examples

```
data("wafers")
m1 <- HLfit(y ~X1+X2+(1|batch), resid.model = ~ 1, data=wafers, method="ML")
get_ranPars(m1,which="corrPars") # NULL since no correlated random effect

parlist1 <- list(lambda=1,phi=2,corrPars=list("1"=list(rho=3,nu=4),"2"=list(rho=5)))
parlist2 <- list(lambda=NA,corrPars=list("1"=list(rho=NA))) # values of elements do not matter
remove_from_parlist(parlist1,parlist2) ## same result as:
remove_from_parlist(parlist1,rm_names = names(unlist(parlist2)))
```

`get_RLRsim_args` *Extractors of arguments for functions from package RLRsim*

Description

`get_RLRsim_args` extracts a list of arguments suitable for a call to `RLRsim::RLRTSim()` or `RLRsim::LRTSim()`. These functions use an efficient simulation procedure to compute restricted or marginal likelihood ratio tests, respectively, comparing a fixed-effect model and a mixed-effect model with one random effect. They are notably used to test for the presence of one random effect, although the models compared by marginal likelihood (`LRTSim()`) may differ both in their random and in their fixed effects (as shown in the Example). The tests are exact for small samples (up to simulation error) for LMMs with no free parameters in the random effect (beyond the variance being tested), so not for time-series or spatial models with estimated correlation parameters. Heteroscedasticity of the residuals or of the random effect variance are also not taken into account by the simulation procedure (see Value field below for an hint why this is so).

`get_RLRTSim_args` is the older extractor, originally for `RLRsim::RLRTSim()` only, now handling also ML fits with a warning (though the possible absence of the `nullfit` argument will result in an error).

Usage

```
get_RLRsim_args(fullfit, nullfit, verbose=TRUE, REML=NA, ...)
get_RLRTSim_args(object, verbose=TRUE, ...)
```

Arguments

object, fullfit	An object of class HLfit, as returned by the fitting functions in spaMM, for the more complete model to be compared.
nullfit	Same for the less complete model; required only for (marginal) LR test, as opposed to restricted LR test.
verbose	NA or boolean; Whether to display some message or not.
REML	For programming purposes, not documented.
...	Additional arguments (currently not used).

Details

If the models compared do not differ in their fixed effects, under the null hypothesis there is a probability mass P for a zero likelihood ratio, and the distribution of p-values can be uniform only on the range (0,1-P). If the fixed effects differ (as handled by RLRsim::LRTSim()), this does not occur.

Value

A list of arguments for a call to RLRsim::RLRTSim() or RLRsim::LRTSim(). The main arguments are the design matrix for the fixed effects, and the **ZA** matrix and **L** detailed in random-effects (here represented by the Z and sqrt.Sigma elements). The models handled by the testing procedure are the ones that are sufficiently characterized by these two matrices. LRTSim additionally requires q, the difference in number of parameters of fixed effects between the models.

Note

The inconsistent capitalisation of 's' in the function names is consistent with the inconsistencies in the RLRsim package.

References

Crainiceanu, C. and Ruppert, D. (2004) Likelihood ratio tests in linear mixed models with one variance component, *Journal of the Royal Statistical Society: Series B*,**66**,165–185.

See Also

The bootstrap procedure in LRT is more general but slower. It appears to provide results quite similar to those of RLRsim when both are applicable.

Examples

```
## Not run:
## Derived from example in RLRsim::LRTSim
 set.seed(123)
 dat <- data.frame(g = rep(1:10, e = 10), x = (x<-rnorm(100)),
                   y = 0.1 * x + rnorm(100))
 m <- fitme(y ~ x + (1|g), data=dat)
 m0 <- fitme(y ~ 1, data=dat)
 (obs.LRT <- 2*(logLik(m)-logLik(m0)))
 args <- get_RLRsim_args(m,m0)
 sim.LRT <- do.call(RLRsim::LRTSim, args )
 (RLRpval <- (sum(sim.LRT >= obs.LRT) + 1) / (length(sim.LRT) + 1))
 ## comparable test using LRT():
 # (bootpval <- LRT(m,m0, boot.repl = 199L)$rawBootLRT$p_value)

## End(Not run)
```

`gof` *Goodness of fit test*

Description

Performs a test of goodness of fit (by default: see the `testfn` argument for possible alternatives). The only method implemented so far is based on randomized quantile residuals (Dunn & Smyth, 1996).

Usage

```
gof(object, method = "RQR", plot.=FALSE, testfn=stats::shapiro.test, ...)
```

Arguments

`object`	A fit object inheriting from class `"HLfit"`, as returned by **spaMM**'s main fitting functions.
`method`	Character string; a method of test of goodness of fit.
`plot.`	Boolean: whether to produce a plot (effectively `plot(object, res_type="RQR", which="mean")` for RQRs as function of predicted values).
`testfn`	A function with the following interface: it takes the vector of values whose normality is assessed as first argument; and it returns a `list` (to which the `RQR` element will be added, see Value). There is no particular constraint on the contents of the returned list. For example, `qqnorm` is a possible `testfn` although it does not return a test.
`...`	Further arguments passed to `testfn`.

Value

Returns a list made up of the return value of a call to the function provided by the `testfn` argument, with added element `RQR`, itself a list including the randomized residuals. The default `testfn`, `shapiro.test` (except for datasets of more than 5000 fitted response values), returns "a list with class `"htest"`" which is therefore the class of the default result.

References

Dunn, K. P., and Smyth, G. K. (1996). Randomized quantile residuals. Journal of Computational and Graphical Statistics 5, 1-10.

Examples

```
data("Orthodont", package = "nlme")
sp1 <- fitme(distance ~ age * Sex + (1 | Subject), data = Orthodont)
gof(sp1) #  classic toy example, but poor fit.
gof(sp1, testfn=ks.test, y="pnorm")
```

`good-practice` *Clear and trustworthy formulas and prior weights*

Description

Base fitting functions in R will seek variables in the environment where the `formula` was defined (i.e., typically in the global environment), if they are not in the `data`. This increases the memory size of fit objects (as the formula and attached environment are part of such objects). This also easily leads to errors (see example in the discussion of `update.HLfit`). Indeed Chambers (2008, p.221), after describing how the environment is defined, comments that "Where clear and trustworthy software is a priority, I would personally avoid such tricks. Ideally, all the variables in the model frame should come from an explicit, verifiable data source...". Fitting functions in **spaMM** try to adhere to such a principle, as they assume by default that all variables from the `formula` should be in the `data` argument (and then, **one never needs to specify "`data$`" in the `formula`.**. **spaMM** implements this by default by stripping the formula environment from any variable. It is also possible to assign a given environment to the formula, through the control `control.HLfit$formula_env`: see Examples.

The variables defining the `prior.weights` should also be in the `data`. However, the implementation of the `prior.weights` argument has limitations that can be overcome by using the more recently introduced `weights.formula` argument of **spaMM** fitting functions (see Examples, where this is also compared with `stats::lm`'s handling of its `weights` argument).

However, variables used in other arguments such as `ranFix` are looked up neither in the data nor in the formula environment, but in the calling environment as usual.

References

Chambers J.M. (2008) Software for data analysis: Programming with R. Springer-Verlag New York

Examples

```
#######  Controlling the formula environment

set.seed(123)
d2 <- data.frame(y = seq(10)/2+rnorm(5)[gl(5,2)], x1 = sample(10), grp=gl(5,2), seq10=seq(10))
# Using only variables in the data: basic usage
# HLfit(y ~ x1 + seq10+(1|grp), data = d2)
# is practically equivalent to
HLfit(y ~ x1 + seq10+(1|grp), data = d2,
      control.HLfit = list(formula_env=list2env(list(data=d2))))
#
# The 'formula_env' avoids the need for the 'seq10' variable:
HLfit(y ~ x1 + I(seq_len(nrow(data)))+(1|grp), data = d2,
      control.HLfit = list(formula_env=list2env(list(data=d2))))
#
# Internal implementation allows formula_env to be
#   in 'control' if 'control.HLfit' is absent:
fitme(y ~ x1 + I(seq_len(nrow(data)))+(1|grp), data = d2,
      control = list(formula_env=list2env(list(data=d2))))

####### Prior-weights misery

data("hills", package="MASS")

(fit <- lm(time ~ dist + climb, data = hills, weights=1/dist^2))
# same as
(fit <- fitme(time ~ dist + climb, data = hills, prior.weights=1/dist^2, method="REML"))

# possible calls:
(fit <- fitme(time ~ dist + climb, data = hills, prior.weights=quote(1/dist^2)))
(fit <- fitme(time ~ dist + climb, data = hills, prior.weights= 1/hills$dist^2))
(fit <- fitme(time ~ dist + climb, data = hills, weights.form= ~ 1/dist^2))
(fit <- fitme(time ~ dist + climb, data = hills, weights.form= ~ I(1/dist^2)))

# Also syntactically correct since 'dist' is found in the data:
(fit <- fitme(time ~ dist + climb, data = hills, weights.form= ~ rep(2,length(dist))))

#### Programming with prior weights:

## Different ways of passing prior weights to fitme() from another function:

wrap_as_form <- function(weights.form) {
  fitme(time ~ dist + climb, data = hills, weights.form=weights.form)
}

wrap_as_pw <- function(prior.weights) {
  fitme(time ~ dist + climb, data = hills, prior.weights=prior.weights)
}

wrap_as_dots <- function(...) {
  fitme(time ~ dist + climb, data = hills,...)
}
```

```
## Similarly for lm:

wrap_lm_as_dots <- function(...) {
  lm(time ~ dist + climb, data = hills, ...)
}

wrap_lm_as_arg <- function(weights) {
  lm(time ~ dist + climb, data = hills,weights=weights)
}

## Programming errors with stats::lm():

pw <- rep(1e-6,35) # or even NULL

(fit <- wrap_lm_as_arg(weights=pw)) # catches weights from global envir!
(fit <- lm(time ~ dist + climb, data = hills, weights=pw)) # idem!

(fit <- lm(time ~ dist + climb, data = hills,
           weights=hills$pw)) # fails silently - no $pw in 'hills'
(fit <- wrap_lm_as_dots(weights=hills$pw)) # idem!
(fit <- wrap_lm_as_arg(weights=hills$pw)) # idem!

## Safer spaMM results:

try(fit <- wrap_as_pw(prior.weights= pw)) # correctly catches problem
try(fit <- wrap_as_dots(prior.weights=hills$pw)) # correctly catches problem
(fit <- wrap_as_dots(prior.weights=1/dist^2)) # correct
(fit <- wrap_as_dots(prior.weights=quote(1/dist^2))) # correct

## But 'prior.weights' limitations:

try(fit <- wrap_as_pw(prior.weights= 1/hills$dist^2)) # fails (stop)
try(fit <- wrap_as_pw(prior.weights= 1/dist^2)) # fails (stop)
try(fit <- wrap_as_pw(prior.weights= quote(1/dist^2))) # fails (stop)

## Problems all solved by using 'weights.form':

try(fit <- wrap_as_form(weights.form= ~ pw)) # correctly catches problem
(fit <- wrap_as_form(weights.form= ~1/dist^2)) # correct
(fit <- wrap_as_form(weights.form= ~1/hills$dist^2)) # correct
(fit <- wrap_as_dots(weights.form= ~ 1/dist^2)) # correct

rm("pw")
```

Gryphon *Gryphon data*

Description

Loading these data loads three objects describing a mythical 'Gryphon' population used by Wilson et al. to illustrate mixed-effect modelling in quantitative genetics. These objects are a data frame `Gryphon_df` containing the model variables, a genetic relatedness matrix `Gryphon_A`, and another data frame `Gryphon_pedigree` containing pedigree information (which can be used by some packages to reconstruct the relatedness matrix).

Usage

```
data("Gryphon")
```

Format

`Gryphon_df` is

```
'data.frame': 1084 obs. of  6 variables:
 $ ID    : int  1029 1299 ...:               individual identifier
 $ sex   : Factor w/ 2 levels "1","2":      sex, indeed
 $ year  : Factor w/ 34 levels "968","970", ...: birth year
 $ mother: Factor w/ 429 levels "1","2",..:  individual's mother identifier
 $ BWT   : num  10.77 9.3  ...:              birth weight
 $ TARSUS: num  24.8 22.5 12 ...:            tarsus length
```

`Gryphon_A` is a genetic relatedness matrix, in sparse matrix format, for 1309 individuals.

`Gryphon_pedigree` is

```
'data.frame': 1309 obs. of  3 variables:
 $ ID  : int  1306 1304 ...: individual identifier
 $ Dam : int  NA NA ...:     individual's mother
 $ Sire: int  NA NA ...:     individual's father
```

References

Wilson AJ, et al. (2010) An ecologist's guide to the animal model. Journal of Animal Ecology 79(1): 13-26. doi:10.1111/j.13652656.2009.01639.x

Examples

```
#### Bivariate-response model used as example in Wilson et al. (2010):
# joint modelling of birth weight (BWT) and tarsus length (TARSUS).

# The relatedness matrix is specified as a 'corrMatrix'. The random
# effect 'corrMatrix(0+mv(1,2)|ID)' then represents genetic effects
# correlated over traits and individuals (see help("composite-ranef")).
# The ...(0+...) syntax avoids contrasts being used in the design
# matrix of the random effects, as it would not does make much sense
# to represent TARSUS as a contrast to BWT.

# The relatedness matrix will be specified through its inverse,
# using as_precision(), so that spaMM does not have to find out and
```

```
# inform the user that using the inverse is better (as is typically
# the case for relatedness matrices). But using as_precision() is
# not required. See help("algebra") for Details.

# The second random effect '(0+mv(1,2)|ID)' represents correlated
# environmental effects. Since measurements are not repeated within
# individuals, this effect also absorbs all residual variation. The
# residual variances 'phi' must then be fixed to some negligible values
# in order to avoid non-identifiability.

if (spaMM.getOption("example_maxtime")>5) {
  data("Gryphon")
  gry_prec <- as_precision(Gryphon_A)
  gry_GE <- fitmv(
    submodels=list(BWT ~ 1 + corrMatrix(0+mv(1,2)|ID)+(0+mv(1,2)|ID),
                   TARSUS ~ 1 + corrMatrix(0+mv(1,2)|ID)+(0+mv(1,2)|ID)),
    fixed=list(phi=c(1e-6,1e-6)),
    corrMatrix = gry_prec,
    data = Gryphon_df, method = "REML")

  # Estimates are practically identical to those reported for package
  # 'asreml' (https://www.vsni.co.uk/software/asreml-r)
  # according to Supplementary File 3 of Wilson et al., p.7:

  lambda_table <- summary(gry_GE, digits=5,verbose=FALSE)$lambda_table
  by_spaMM <- na.omit(unlist(lambda_table[,c("Var.","Cor.1")]))[1:6]
  by_asreml <- c(3.368449,12.346304,3.849875,17.646017,0.381463,0.401968)
  by_spaMM/by_asreml-1  # relative differences ~ O(1e-4)

}
```

`hatvalues.HLfit` *Hat values and leverages extractor*

Description

`hatvalues.HLfit` is a method for the generic function `hatvalues`:
• `hatvalues(.,type="projection")` will return "hat values", defined as diagonal elements of the projection matrix of the augmented linear model;
• `hatvalues(.,type="std")` will return generalized "leverages", which are standardizing valeus that can be used for estimation of dispersion parameters (see Details for the differences).

To get the full hat matrix, see `get_matrix(., which="hat_matrix")`.

Usage

```
## S3 method for class 'HLfit'
hatvalues(model, type = "projection", which = "resid", force=FALSE, ...)
```

Arguments

`model`	An object of class `HLfit`, as returned by the fitting functions in `spaMM`.
`type`	Character: `"projection"`, `"std"`, or more cryptic values not documented.
`which`	Character: `"resid"` for the traditional leverages of the observations, `"ranef"` for random-effect leverages, or `"both"` for both.
`force`	Boolean: to force recomputation of the leverages even if they are available in the object, for checking purposes.
`...`	For consistency with the generic.

Details

Leverages may have distinct meanings depending on context. The textbook version for linear models is that leverages (q_i) are the diagonal elements of a projection matrix ("hat matrix"), and that they may be used to standardize ("studentize") residuals as follows. If the residual variance ϕ is known, then the variance of each fitted residual $\hat{e}_i$ is $\phi(1-q_i)$. Standardized residuals, all with variance 1, are then $\hat{e}_i/\sqrt{(\phi(1-q_i))}$. This standardization of variance no longer holds exactly with estimated ϕ, but if one uses here an unbiased (REML) estimator of ϕ, the studentized residuals may still practically have a unit expected variance.

In mixed models, a generalized definition of the projection matrix is available, in terms of the matrix of the augmented linear model [its value $\mathbf{P}_{\mathrm{a}}$ can be retrieved by `get_matrix(., which="hat_matrix")`]. The "hat values", defined as the diagonal elements of $\mathbf{P}_{\mathrm{a}}$, are also defined for the random effects. However, in general the standardizing "leverages" $\mathbf{q}^*$ that can again be used to obtain estimates of ϕ and λ, by standardizing residuals (for ϕ), or by standardizing predicted random effects (for λ), are no longer simply the "hat values".

For illustration, the following simple differences hold between `"projection"` hat values and `"std"` values:
• `"std"` values are defined so that they are 0 for ML fits of simple linear model.
• In LMMs fitted by REML, `"std"` values are the same as `"projection"` values. For ML fits, these two forms of leverages differ, because `"std"` values are diagonal values of another matrix than $\mathbf{P}_{\mathrm{a}}$. Note that the `"projection"` hat values also differ between ML and REML fits because the fitted $\mathbf{P}_{\mathrm{a}}$ itself depends on $\hat{\phi}$ and $\hat{\lambda}$, which differ between ML and REML fits.
• In GLMMs (or HGLMs), the `"std"` values may differ from the `"projection"` ones even for REML, because they may include corrections for non-Gaussian response family, for non-Gaussian random effects, and for taking into account the variation of the GLM weights in the logdet(info.mat) derivatives. Which corrections are included depend on the precise method used to fit the model (e.g., EQL vs PQL vs REML).

Value

A list with separate components `resid` (leverages of the observations) and `ranef` if `which="both"`, and a vector otherwise.

References

Lee, Y., Nelder, J. A. and Pawitan, Y. (2006) Generalized linear models with random effects: unified analysis via h-likelihood. Chapman & Hall: London.

Examples

```
if (spaMM.getOption("example_maxtime")>0.8) {
data("Orthodont",package = "nlme")
rnge <- (107:108)

# all different:
#
hatvalues(rlfit <- fitme(distance ~ age+(age|Subject),
                         data = Orthodont, method="REML"))[rnge]
hatvalues(mlfit <- fitme(distance ~ age+(age|Subject),
                         data = Orthodont))[rnge]
hatvalues(mlfit,type="std")[rnge]
}
```

HLCor *Fits a (spatially) correlated mixed model, for given correlation parameters*

Description

A fitting function acting as a convenient interface for HLfit, constructing the correlation matrix of random effects from the arguments, then estimating fixed effects and dispersion parameters using HLfit. Various arguments are available to constrain the correlation structure, covStruct and distMatrix being the more general ones (for any number of random effects), and adjMatrix and corrMatrix being alternatives to covStruct for a single correlated random effect. The uniqueGeo argument is deprecated.

Usage

```
HLCor(formula, data, family = gaussian(), fixed=NULL, ranPars, distMatrix,
      adjMatrix, corrMatrix, covStruct=NULL,
      method = "REML", verbose = c(inner=FALSE),
      control.dist = list(), weights.form = NULL, ...)
```

Arguments

formula	A predictor, i.e. a formula with attributes (see Predictor), or possibly simply a simple formula if an offset is not required.
fixed, ranPars	A list of given values for correlation parameters (some of which are mandatory), and possibly also dispersion parameters (optional, but passed to HLfit if present). ranPars is the old argument, maintained for back compatibility; fixed is the new argument, uniform across **spaMM** fitting functions. See fixed for further information about these two parameters.
data	The data frame to be analyzed.
family	A family object describing the distribution of the response variable. See HLfit for further information.

`distMatrix` This argument allows estimation of Matern or Cauchy correlation parameters to be combined with use of an ad hoc distance matrix. When there is a single spatial random effect, it may be a **distance matrix** between geographic locations, internally used as argument to `MaternCorr` or `CauchyCorr`. It then overrides the (by default, Euclidean) distance matrix that would otherwise be deduced from the variables in a `Matern(.))` or `Cauchy(.)` term.

More generally, it may be a **list of such matrices**. The list format may be necessary when there are several Matern/Cauchy terms, to avoid that all of them are affected by the same `distMatrix`. `NULL` list elements may be necessary, e.g. `distMatrix=list("1"=NULL,"2"=<.>))` when a matrix is specified only for the second random effect.

`adjMatrix` An single adjacency matrix, used if a random effect of the form `y ~ adjacency(1|<location index>)` is present. See `adjacency` for further details. If adjacency matrices are needed for several random effects, use `covStruct`.

`corrMatrix` A matrix **C** used if a random effect term of the form `corrMatrix(1|<stuff>)` is present. This allows to analyze non-spatial model by giving for example a matrix of genetic correlations. Each row corresponds to levels of a variable <stuff>. The covariance matrix of the random effects for each level is then $\lambda\mathbf{C}$, where as usual λ denotes a variance factor for the random effects (if **C** is a correlation matrix, then λ is the variance, but other cases are possible). See `corrMatrix` for further details. If matrices are needed for several random effects, use the `covStruct` argument.

`covStruct` An interface for specifying correlation structures for different types of random effect (`corrMatrix` or `adjacency`). See `covStruct` for details.

`method` Character: the fitting method to be used, such as `"ML"`, `"REML"` or `"PQL/L"`. `"REML"` is the default. Other possible values of `HLfit`'s `method` argument are handled.

`weights.form` Specification of prior weights by a one-sided formula: use `weights.form = ~ pw` instead of `prior.weights = pw`. The effect will be the same except that such an argument, known to evaluate to an object of class `"formula"`, is suitable to enforce safe programming practices (see `good-practice`).

`verbose` A vector of booleans. `inner` controls various diagnostic (possibly messy) messages about the iterations. This should be distinguished from the `TRACE` element, meaningful in `fitme` or `corrHLfit` calls.

`control.dist` A list of arguments that control the computation of the distance argument of the correlation functions. Possible elements are

rho.mapping a set of indices controlling which elements of the `rho` scale vector scales which dimension(s) of the space in which (spatial) correlation matrices of random effects are computed. See same argument in `make_scaled_dist` for details and examples.

dist.method `method` argument of `proxy::dist` function (by default, `"Euclidean"`, but see `make_scaled_dist` for other distances such as spherical ones.)

`...` Further arguments passed to `HLfit` or to `mat_sqrt`.

Details

For approximations of likelihood, see method. For the possible structures of random effects, see random-effects, but note that HLCor cannot adjust parameters of correlation models (with the exception of conditional autoregressive ones). Any such parameter must be specified by the ranPars argument. More generally, the correlation matrix for random effects can be specified by various combinations of formula terms and other arguments (see Examples):

Basic Matérn model Matern(1|<...>), using the spatial coordinates in <...>. This will construct a correlation matrix according to the Matérn correlation function (see MaternCorr);

Basic Cauchy model Cauchy(1|<...>), as for Matern (see CauchyCorr);

Same models with given distance matrix as provided by distMatrix (see Examples);

Given correlation matrix corrMatrix(1|<...>) with corrMatrix argument. See corrMatrix for further details.

CAR model with given adjacency matrix adjacency(1|<...>) with adjMatrix. See adjacency for further details;

AR1 model AR1(1|<...>) See AR1 for further details.

Value

The return value of an HLfit call, with the following additional attributes:

HLCorcall	the HLCor call
info.uniqueGeo	Unique geographic locations.

See Also

Additional example using an adjacency term in autoregressive;

Additional examples using corrMatrix or distMatrix argument in corrMatrix;

MaternCorr, which may be used to specify corrMatrix values.

Examples

```
# Example with an adjacency matrix (autoregressive model):
# see 'adjacency' documentation page

#### Matern correlation using only the Matern() syntax
data("blackcap")
(fitM <- HLCor(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap,
      method="ML", ranPars=list(nu=0.6285603,rho=0.0544659)))

#### Using the 'distMatrix' argument
data("blackcap")
#
# Build distance matrix (here equivalent to the default one for a Matern() term)
MLdistMat <- as.matrix(proxy::dist(blackcap[,c("latitude","longitude")]))
#
(fitD <- HLCor(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap,
      distMatrix=MLdistMat, method="ML", ranPars=list(nu=0.6285603,rho=0.0544659)))
```

```
# : result here must be equivalent to the one without the distMatrix.
diff(c(logLik(fitM),logLik(fitD)))
```

HLfit *Fit mixed models with given correlation matrix*

Description

Fits GL(M)Ms as well as some hierarchical generalized linear models (HGLM; Lee and Nelder 2001). It may be called on its own but is now better seen as a backend for the main fitting function `fitme` (or `fitmv` for multivariate-response models),

On its own, `HLfit` fits both fixed effects parameters, and dispersion parameters i.e. the variance of the random effects (full covariance for random-coefficient models), and the variance of the residual error. The linear predictor is of the standard form `offset+ X beta + Z b`, where X is the design matrix of fixed effects and Z is a design matrix of random effects (typically an incidence matrix with 0s and 1s, but not necessarily). The residual dispersion may follow a "structured-dispersion model" modeling heteroscedasticity.

Models are fitted by an iterative algorithm alternating estimation of fixed effects and of dispersion parameters. Fitting structured-dispersion model is performed by a form of fit of corrected residuals (Smyth et al. 2001). These corrected residuals are also used to estimate a single ϕ value. However, evaluation of the corrected residuals can be slow in particular for large datasets. Then, for model without structured dispersion, `fitme` should be faster.

Usage

```
HLfit(formula, data, family = gaussian(), rand.family = gaussian(),
      resid.model = ~1, REMLformula = NULL, verbose = c(inner = FALSE),
      HLmethod = "HL(1,1)", method="REML", control.HLfit = list(),
      control.glm = list(), init.HLfit = list(), fixed=list(), ranFix,
      etaFix = list(), prior.weights = NULL, weights.form = NULL,
      processed = NULL)
```

Arguments

`formula`	A `formula`; or a `predictor`, i.e. a formula with attributes created by `Predictor`, if design matrices for random effects have to be provided. See Details in `spaMM` for allowed terms in the formula (except spatial ones).
`data`	A data frame containing the variables named in the model formula.
`family`	A `family` object describing the distribution of the response variable. See Details in `spaMM` for handled families.
`rand.family`	A `family` object describing the distribution of the random effect, or a `list` of family objects for different random effects (see Examples). Possible options are `gaussian()`, `Gamma(log)`, `Gamma(identity)`, `Beta(logit)`, `inverse.Gamma(-1/mu)`, and `inverse.Gamma(log)`: in these expressions of form family(link), the family gives the distribution of a random effect u and the link gives `v` as function of u. See `random-effects` for further description. If there are several random effects and only one family is given, this family holds for all random effects.

`resid.model` Used to specify a model for the dispersion parameter of the mean-response distribution family. See the `resid.model` documentation, and the more specific `phi-resid.model` one for the phi parameter of gaussian and Gamma response families.

`REMLformula` A model `formula` that controls the estimation of dispersion parameters and the computation of restricted likelihood (`p_bv`), where the conditioning inherent in REML is defined by a model different from the predictor `formula`. A simple example (useless in practice) of its effect is to replicate an ML fit by specifying `method="REML"` and an `REMLformula` with no fixed effect. The latter implies that no conditioning is performed and that `p_bv` equals the marginal likelihood (or its approximation), `p_v`. One of the examples in `update.HLfit` shows how `REMLformula` can be useful, but otherwise this argument may never be needed for standard REML or ML fits. For non-standard likelihood ratio tests using `REMLformula`, see `fixedLRT`.

`verbose` A vector of booleans or integers. The `inner` element controls various diagnostic messages (possibly messy) about the iterations. This should be distinguished from the `TRACE` element, meaningful (and generally more useful) when `verbose` is passed through a higher-level function such as `fitme`, where `verbose(TRACE=TRUE)` allows information to be displayed for each set of correlation and dispersion parameter values considered by the optimiser. The `phifit` element controls messages about the progress of `phi-resid.model` fits (see the latter documentation). See `verbose` for further information, mostly useless except for development purposes.

`method` Character: the fitting method. allowed values include `"REML"`, `"ML"`, `"EQL-"` and `"EQL+"` for all models, and `"PQL"` (=`"REPQL"`) and `"PQL/L"` for GLMMs only. `method=c(<"ML" or "REML">,"exp")` can be distinctly useful for slow fits of models with `Gamma(log)` family. See (see `method`) for details, and further possible values for those curious to experiment. REML can be viewed as a form of conditional inference, and non-standard conditionings can be called by using a non-standard `REMLformula`.

The default is REML (standard REML for LMMs, an extended definition for other models) but **is overridden** when called by `fitme` or `fitmv`, where **the default is ML**.

`HLmethod` Alternative specification of `method`. It is useless to specify `HLmethod` when `method` is specified. The default value `"HL(1,1)"` means the same as `method="REML"`, but more accurately relates to definitions of approximations of likelihood in the h-likelihood literature.

`control.HLfit` A list of parameters controlling the fitting algorithms, which should mostly be ignored in routine use. See `control.HLfit` for possible controls.

`control.glm` List of parameters controlling calls to `glm`-"like" fits, passed to `glm.control`; e.g.
`control.glm=list(maxit=100)`. See `glm.control` for further details. `glm`-"like" fits (possibly using `llm.fit` for non-GLM families) may be performed as part of mixed-effect model fitting procedures, in particular to provide initial values for the different estimated parameters.

`init.HLfit` A list of initial values for the iterative algorithm, with possible elements of the list are `fixef` for fixed effect estimates (beta), `v_h` for random effects vector

	v in the linear predictor, `lambda` for the parameter determining the variance of random effects u as drawn from the `rand.family` distribution, and `phi` for the residual variance. However, this argument can be ignored in routine use.
`fixed`, `ranFix`	A list of fixed values of random effect parameters. `ranFix` is the old argument, maintained for back compatibility; `fixed` is the new argument, uniform across **spaMM** fitting functions. See `ranFix` for further information.
`etaFix`	A list of given values of the coefficients of the linear predictor. See `etaFix` for further information.
`prior.weights`	An optional vector of prior weights as in `glm`. This fits the data to a probability model with residual variance parameter given as `phi/prior.weights` instead of the canonical parameter `phi` of the distribution family for the response variable, and all further outputs are defined to be consistent with this (see section IV in Details).
`weights.form`	Specification of prior weights by a one-sided formula: use `weights.form = ~ pw` instead of `prior.weights = pw`. The effect will be the same except that such an argument, known to evaluate to an object of class `"formula"`, is suitable to enforce safe programming practices (see `good-practice`).
`processed`	A list of preprocessed arguments, for programming purposes only.

Details

I. Approximations of likelihood: see `method`.

II. The standard errors reported may sometimes be misleading. For each set of parameters among β, λ, and ϕ parameters these are computed assuming that the other parameters are known without error. This is why they are labelled `Cond. SE` (conditional standard error). This is most uninformative in the unusual case where λ and ϕ are not separately estimable parameters. Further, the SEs for λ and ϕ are rough approximations as discussed in particular by Smyth et al. (2001; V_1 method).

III. prior weights. This controls the likelihood analysis of heteroscedastic models. In particular, changing the weights by a constant factor f should, and will, yield a fit with unchanged likelihood and (Intercept) estimates of `phi` also increased by f (except if a non-trivial `resid.formula` with log link is used). This is consistent with what `glm` does, but other packages may not follow this logic (whatever their documentation may say: check by yourself by changing the weights by a constant factor). Further, post-fit functions (in particular those extracting various forms of residuals) may be inconsistent in their handling of prior weights.

Value

An object of class `HLfit`, which is a list with many elements, not all of which are documented.

Various extractor functions are available (see `extractors`, `vcov`, `get_fittedPars`, `get_matrix`, and so on). They should be used as far as possible as they should be backward-compatible from version 2.0.0 onwards, while the structure of the return object may still evolve. The following information may be useful for extracting further elements of the object.

Elements include **descriptors of the fit**:

`eta`	Fitted values on the linear scale (including the predicted random effects). `predict(.,type="link")` can be used as a formal extractor;

`fv`	Fitted values (μ =<inverse-link>(η)) of the response variable. `fitted(.)` or `predict(.)` can be used as formal extractors;
`fixef`	The fixed effects coefficients, β (returned by the `fixef` function);
`v_h`	The random effects on the linear scale, v, with atttribute the random effects u (returned by `ranef(*,type="uncorrelated")`;
`phi`	The residual variance ϕ. See `residVar` for one extractor;
`phi.object`	A possibly more complex object describing ϕ (see `residVar` again);
`lambda`	The random-effect (u) variance(s) λ in compact form;
`lambda.object`	A possibly more complex object describing λ (see `get_ranPars(.,which="lambda"))` and `VarCorr` extractors);
`ranef_info`	environment where information about the structure of random effects is stored (see `Corr`);
`corrPars`	Agglomerates information on correlation parameters, either fixed, or estimated ((see `get_ranPars(.,which="corrPars")`));
`APHLs`	A list whose elements are various likelihood components, including conditional likelihood, h-likelihood, and the Laplace approximations: the (approximate) marginal **likelihood** `p_v` and the (approximate) **restricted likelihood** `p_bv` (the latter two available through the `logLik` function). See the extractor function `get_any_IC` for information criteria ("AIC") and effective degrees of freedom;

The covariance matrix of β estimates is not included as such, but can be extracted by `vcov`.

Information about the input is contained in output elements named as arguments of the fitting function calls (`data,family,resid.family,ranFix,prior.weights`), with the following notable exceptions or modifications:

`predictor`	The `formula`, possibly reformatted (returned by the `formula` extractor);
`resid.predictor`	Analogous to `predictor`, for the residual variance (see `residVar(., which="formula")`);
`rand.families`	corresponding to the `rand.family` input;

Further miscellaneous diagnostics and descriptors of model structure:

`X.pv`	The design matrix for fixed effects (returned by the `model.matrix` extractor);
`ZAlist`, `strucList`	Two lists of matrices, respectively the design matrices "**Z**", and the "**L**" matrices, for the different random-effect terms. The extractor `get_ZALMatrix` can be used to reconstruct a single "**ZL**" matrix for all terms.
`BinomialDen`	(binomial data only) the binomial denominators;
`y`	the response vector; for binomial data, the frequency response.
`models`	Additional information on model structure for η, λ and ϕ;
`HL`	A set of indices that characterize the approximations used for likelihood;
`lev_phi`, `lev_lambda`	Leverages (see `hatvalues` extractor);

dfs	list (possibly structured): some information about degrees of freedom for different components of the model. But its details may be difficult to interpret and the DoF extractor should be used;
how	A list containing the information properly extracted by the how function;
warnings	A list of warnings for events that may have occurred during the fit.

Finally, the object includes programming tools: call, spaMM.version, fit_time and an environment envir that may contain whatever may be needed in some post-fit operations..

References

Lee, Y., Nelder, J. A. (2001) Hierarchical generalised linear models: A synthesis of generalised linear models, random-effect models and structured dispersions. Biometrika 88, 987-1006.

Smyth GK, Huele AF, Verbyla AP (2001). Exact and approximate REML for heteroscedastic regression. Statistical Modelling 1, 161-175.

See Also

HLCor for estimation with given spatial correlation parameters; corrHLfit for joint estimation with spatial correlation parameters; fitme as a **preferred alternative to all these functions**.

Examples

```
data("wafers")
## Gamma GLMM with log link
HLfit(y ~ X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch), family=Gamma(log),
          resid.model = ~ X3+I(X3^2) ,data=wafers)
## Gamma - inverseGamma HGLM with log link
HLfit(y ~ X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch), family=Gamma(log),
          rand.family=inverse.Gamma(log),
          resid.model = ~ X3+I(X3^2) , data=wafers)
```

---

how *Extract information about how an object was obtained*

Description

how is defined as a generic with currently only one non-default method, for objects of class HLfit. This method provide information about how such a fit was obtained.

Usage

```
how(object, ...)
## S3 method for class 'HLfit'
how(object, devel=FALSE, verbose=TRUE, format=print, ...)
## S3 method for class 'HLfitlist'
how(object, devel=FALSE, verbose=TRUE, format=print, ...)
```

Arguments

object	Any R object.
devel	Boolean; Whether to provide additional cryptic information. For development purposes, not further documented.
verbose	Boolean; Whether to print information about the input object.
format	wrapper for printing format. E.g., cat(cli::col_yellow(s),"\n") could be used instead of the default.
...	Other arguments that may be needed by some method.

Value

A list, returned invisibly, whose elements are not further described here, some being slightly cryptic or subject to future changes. However, how(.)$fit_time is a clean way of getting the fit time, and element algebra stores a value that can be used as the algebra element in the control.HLfit argument of the fitting functions. If verbose is TRUE, the function prints a message presenting some of the information stored in the returned list.

See Also

algebra

Examples

```
foo <- HLfit(y~x, data=data.frame(x=runif(3), y=runif(3)), method="ML", ranFix=list(phi=1))
how(foo)
```

---

inits *Controlling optimization strategy through initial values*

Description

Several parameters (notably the dispersion parameters: the variance of random effects and the residual variance parameter, if any) can be estimated either by iterative algorithms, or by generic optimization methods (see inner-vs-outer). Initial values provide control of the selection of algorithms. For example, if two random effects are fitted by fitme, then
init=list(lambda=c(NA,NaN)) enforces generic optimization for the first variance and iterative algorithms for the second.
init=list(lambda=c(0.1,NaN)) has the same effect and additionally provides control of the initial value for optimization (whereas init.HLfit=list(lambda=c(NA,0.1)) will provide control of the initial value for iterations).

How to know which algorithm has been selected for each parameter? fitme(., verbose=c(TRACE=TRUE)) shows successive values of the variables estimated by optimization (See Examples; if no value appears, then all are estimated by iterative methods). The first lines of the summary of a fit object should tell which variances are estimated by the "outer" method.

corrHLfit, which uses inner estimation by default, can be forced to perform outer optimization. Its control is more limited, as NAs and NaNs are not allowed. Instead, only numeric values as in init=list(lambda=0.1) are allowed.

Examples

```
## Not run:
air <- data.frame(passengers = as.numeric(AirPassengers),
                  year_z = scale(rep(1949:1960, each = 12)),
                  month = factor(rep(1:12, 12)))
air$time <- 1:nrow(air)
# Use verbose to find that lambda is estimated by optimization
fitme(passengers ~ month * year_z + AR1(1|time), data = air,
      verbose=c(TRACE=TRUE))
# Use init to enforce iterative algorithm for lambda estimation:
fitme(passengers ~ month * year_z + AR1(1|time), data = air,
      verbose=c(TRACE=TRUE), init=list(lambda=NaN))
# (but then it may be better to enforce it also for phi: init=list(lambda=NaN, phi=NaN))
#
# Use init to enforce generic optimization for lambda estimation,
#   and control initial value:
fitme(passengers ~ month * year_z + AR1(1|time), data = air,
      verbose=c(TRACE=TRUE), init=list(lambda=0.1))

# See help("multinomial") for more examples of control by initial values.

## End(Not run)
```

`inner-vs-outer` *Inner iterations versus outer optimization*

Description

For estimation of `phi` and `lambda`, two approaches are implemented: generic optimization functions to maximize the log-likelihood (or restricted log-likelihood in REML case), and iterative algorithms described by Lee et al. (2006). This distinction is extended to random-coefficient parameters (`ranCoefs`) and to the correlation parameter of `adjacency` models in **spaMM**, but other correlation parameters of random effects are estimated only by generic optimization. Fixed-effect coefficients (β) are always estimated by iterative methods. The iterative method is well suited for estimating parameters of heteroscedastic (or "structured-dispersion") models, but it then requires computations which can be slow for large datasets.

The `HLfit` and `HLCor` functions only use the iterative algorithms. Other fitting functions can call both approaches, internally using `HLfit` and `HLCor` to perform any iterative estimation. For this reason, all fitting functions, **except** `HLfit` and `HLCor`, are called **"outer" optimization functions** in this documentation, possibly calling **"inner" iterative algorithms**. Likewise, parameters may be **"outer"**- or **"inner"**-estimated.

`fitme` by default tries to select the fastest method when both can be applied. Therefore, `fitme` tends to use generic generic optimization, rather than iterative algorithms, for large datasets, unless there is a non-trivial `resid.model`. `corrHLfit` tends to use iterative methods as much as possible.

Compared to generic optimization, the iterative algorithms provide more information, such as the "cond. SE" of the dispersion parameter estimates (about which see warning in Details of `HLfit`). To force the evaluation of such information after an outer-optimization by an outer-optimizing function,

use the control$refit argument (see Examples in fitme documentation). Alternatively (and possibly of limited use), one can force inner-optimization of lambda for a given random effect, or of phi, by setting it to NaN in init (see further Example using 'blackcap' data). The same syntax may be used for phi.

Details

The results of REML fits of non-gaussian mixed models with both inner- and outer-estimated parameters may (generally slightly) differ according to which parameters are inner-estimated, when the fits include both some dispersion estimate(s) (say $\hat{\lambda}$), which maximize restricted likelihood for given fixed effects, and the fixed effects estimates ($\hat{\beta}$, which maximize marginal likelihood for given $\hat{\lambda}$). Fit results then depend whether λ is inner- or outer-estimated, if $\hat{\lambda}$ depends on $\hat{\beta}$ and conversely. The latter condition may have been ignored on the grounds of asymptotic arguments, but is nevertheless observed.

Explicit values of the init argument of the fitting functions may be used to control the use of inner iterations, and may be used to ensure better consistency of results over time if selection of methods changes changes over successive versions of spaMM: use NaN as noted above to enforce inner-optimization, and NA or a numeric value to enforce outer-optimization.

References

Lee, Y., Nelder, J. A. and Pawitan, Y. (2006). Generalized linear models with random effects: unified analysis via h-likelihood. Chapman & Hall: London.

Examples

```
###  Contrasting different optimization methods:

## Toy data:
set.seed(123)
gr <- data.frame(y=rgamma(100,shape=9/2,scale=2/3))
# => Gamma deviates with mean mu=3 and variance=2,
#  ie. phi= var/mu^2= 2/9

## Fits

# Here fitme uses HLfit methods which provide cond. SE for phi by default:
fitme(y~1,data=gr,family=Gamma(log))

# To force outer optimization of phi, use the init argument:
fitme(y~1,data=gr,family=Gamma(log),init=list(phi=1))

# To obtain cond. SE for phi after outer optimization, use the 'refit' control:
fitme(y~1,data=gr,family=Gamma(log),,init=list(phi=1),
      control=list(refit=list(phi=TRUE))) ## or ...refit=TRUE...

### Outer-optimization is not necessarily the best way to find a global maximum,
#  particularly when there is little statistical information in the data:

if (spaMM.getOption("example_maxtime")>1.6) {
  data("blackcap")
```

```
  fitme(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap) # poor
  #  Compare with the following two ways of avoiding outer-optimization of lambda:
  corrHLfit(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap,
            method="ML")
  fitme(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap,
        init=list(lambda=NaN))
}
```

`inverse.Gamma` *Distribution families for Gamma and inverse Gamma-distributed random effects*

Description

For dispersion parameter λ, a `Gamma` random-effect family means that random effects are distributed as u Gamma(`shape`=1/λ,**scale=**λ), so u has mean 1 and variance λ. Both the log ($v = \log(u)$) and identity ($v = u$) links are possible, though in the latter case the variance of u is constrained below 1 (otherwise Laplace approximations fail).

The two-parameter inverse Gamma distribution is the distribution of the reciprocal of a variable distributed according to the Gamma distribution Gamma with the same shape and scale parameters. `inverse.Gamma` implements the one-parameter inverse Gamma family with shape=1+1/λ and **rate**=1/λ) (rate=1/scale). It is used to model the distribution of random effects. Its mean=1; and its variance =$\lambda/(1-\lambda)$) if $\lambda < 1$, otherwise infinite. The default link is `"-1/mu"`, in which case `v=-1/u` is "-Gamma"-distributed with the same shape and rate, hence with mean $-(\lambda+1)$ and variance $\lambda(\lambda+1)$, which is a different one-parameter Gamma family than the above-described `Gamma`. The other possible link is `v=log(u)` in which case
$v \sim -\log(X \sim \text{Gamma}(1+1/\lambda, 1/\lambda))$, with mean $-(\log(1/\lambda)+\text{digamma}(1+1/\lambda))$ and variance trigamma$(1+1/\lambda)$.

Usage

```
inverse.Gamma(link = "-1/mu")
# Gamma(link = "inverse") using stats::Gamma
```

Arguments

`link`	For `Gamma`, allowed links are `log` and `identity` (the default link from `Gamma`, `"inverse"`, cannot be used for the random effect specification). For `inverse.Gamma`, allowed links are `"-1/mu"` (default) and `log`.

Examples

```
# see help("HLfit") for fits using the inverse.Gamma distribution.
```

`is_separated` *Checking for (quasi-)separation in binomial-response model.*

Description

Separation occurs in binomial response models when a combination of the predictor variables perfectly predict a level of the response. In such a case the estimates of the coefficients for these variables diverge to (+/-)infinity, and the numerical algorithms typically fail. To anticipate such a problem, the fitting functions in `spaMM` try to check for separation by default. The check may take much time, and is skipped if the "problem size" exceeds a threshold defined by `spaMM.options(separation_max=<.>)`, in which case a message will tell users by how much they should increase `separation_max` to force the check (its exact meaning and default value are subject to changes without notice but the default value aims to correspond to a separation check time of the order of 1s on the author's computer).

`is_separated` is a convenient interface to procedures from the `ROI` package, allowing them to be called explicitly by the user to check bootstrap samples (see Example in `anova`). `is_separated.formula` is a variant (not yet a formal S3 method) that performs the same check, but using arguments similar to those of `fitme(., family=binomial())`.

Usage

```
is_separated(x, y, verbose = TRUE, solver=spaMM.getOption("sep_solver"))
is_separated.formula(formula, ..., separation_max=spaMM.getOption("separation_max"),
                     solver=spaMM.getOption("sep_solver"))
```

Arguments

`x`	Design matrix for fixed effects.
`y`	Numeric response vector
`formula`	A model formula
`...`	`data` and possibly other arguments of a `fitme` call. `family` is ignored if present.
`separation_max`	numeric: non-default value allow for easier local control of this spaMM option.
`solver`	character: name of linear programming solver used to assess separation; passed to `ROI_solve`'s `solver` argument. One can select another solver if the corresponding ROI plugin is installed.
`verbose`	Whether to print some messages (e.g., pointing model terms that cause separation) or not.

Value

Returns a boolean; `TRUE` means there is (quasi-)separation. Screen output may give further information, such as pointing model terms that cause separation.

References

The method accessible by `solver="glpk"` implements algorithms described by

Konis, K. 2007. Linear Programming Algorithms for Detecting Separated Data in Binary Logistic Regression Models. DPhil Thesis, Univ. Oxford.

See Also

See also the 'safeBinaryRegression' and 'detectseparation' package.

Examples

```
set.seed(123)
d <- data.frame(success = rbinom(10, size = 1, prob = 0.9), x = 1:10)
is_separated.formula(formula= success~x, data=d) # FALSE
is_separated.formula(formula= success~I(success^2), data=d) # TRUE
```

Leuca *Leucadendron data*

Description

A data set from Tonnabel et al. (2021) to be fitted by models with sex-specific spatial random effects. Leucadrendron rubrum is a dioecious shrub from South Africa. Various phenotypes were recorded on individuals from a small patch of habitat.

Usage

```
data("Leuca")
```

Format

Leuca is

```
'data.frame': 156 obs. of  12 variables:
 $ name   : Factor w/ 156 levels "f_101","f_102",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ sex    : Factor w/ 2 levels "f","m": 1 1 1 1 1 1 1 1 1 1 ...
 $ area   : num  0.857 0.9 0.827 0.654 0.733 ...
 $ diam   : int  60 30 180 50 70 80 130 90 27 59 ...
 $ fec    : num  0.013 0.0137 5.1171 0.2905 1.042 ...
 $ fec_div: num  0.0128 0.0135 5.037 0.2859 1.0257 ...
 $ x      : num  42 41 62.5 58.5 42.5 33.5 24 26.5 25 41 ...
 $ y      : num  23 46 58 63 51 51 55.5 55.5 58.5 63 ...
 $ diamZ  : num  -0.713 -1.479 2.352 -0.968 -0.457 ...
 $ areaZ  : num  0.72 0.92 0.586 -0.2 0.158 ...
 $ male   : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
 $ female : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
```

Source

Tonnabel, J., Klein, E.K., Ronce, O., Oddou-Muratorio, S., Rousset, F., Olivieri, I., Courtiol, A. and Mignot, A. (2021), Sex-specific spatial variation in fitness in the highly dimorphic Leucadendron rubrum. Mol Ecol, 30: 1721-1735. doi:10.1111/mec.15833

See Also

`MaternCorr` and `composite-ranef` for examples using these data.

Examples

```
data(Leuca)
```

`lev2bool` *Conversion of factor to 0/1 variable*

Description

It may be straightforward to add columns of indicator variables for each level of a factor to the data, by
`<data> <- cbind(<data>, model.matrix( ~ <factor> - 1, data = <data>))`. Alternatively, indicator variables can be created on the fly for given levels, using the `lev2bool` function.

Usage

```
lev2bool(fac, lev)
```

Arguments

`fac`	An object coercible to `factor`.
`lev`	The level of `fac` to be converted to 1.

Value

A one-column matrix.

See Also

Example in `GxE` for alternative to using `lev2bool` in specification of random effects with heteroscedasticity, useful when the latter is controlled by a factor with many levels.

Examples

```
## Elementary bivariate-response model

# Data preparation
#
fam <- rep(c(1,2),rep(6,2)) # define two biological 'families'
ID <- gl(6,2) # define 6 'individuals'
resp <- as.factor(rep(c("x","y"),6)) # distinguishes two responses per individual
set.seed(123)
toymv <- data.frame(
  fam = factor(fam), ID = ID, resp = resp,
 y = 1 + (resp=="x") + rnorm(4)[2*(resp=="x")+fam] + rnorm(12)[6*(resp=="x")+as.integer(ID)]
)
toymv <- cbind(toymv, model.matrix( ~ resp - 1, data = toymv))

# fit response-specific variances of random effect and residuals:
#
(fitme(y ~ resp+ (0+respx|fam)+ (0+respy|fam),
        resid.model = ~ 0+resp ,data=toymv))

# Same result by different syntaxes:

#  * by the lev2bool() specifier:
 (fitme(y ~ resp+ (0+lev2bool(resp,"x")|fam)+ (0+lev2bool(resp,"y")|fam),
        resid.model = ~ 0+resp ,data=toymv))

#  * or by random-coefficient model using 'resp' factor:
(fitme(y ~ resp+ (0+resp|fam), resid.model = ~ 0+resp ,data=toymv,
       fixed=list(ranCoefs=list("1"=c(NA,0,NA)))))
#
# For factors with more levels, the following function may be useful to specify
# through partially fixed ranCoefs that covariances are fixed to zero:
ranCoefs_for_diag <- function(nlevels) {
  vec <- rep(0,nlevels*(nlevels+1L)/2L)
  vec[cumsum(c(1L,rev(seq(nlevels-1L)+1L)))] <- NA
  vec
}
# see application in help("GxE").

#  * or by the dummy() specifier from lme4:
# (fitme(y ~ resp+ (0+dummy(resp,"x")|fam)+ (0+dummy(resp,"y")|fam),
#        resid.model = ~ 0+resp ,data=toymv))
```

`llm.fit` *Link-linear regression models (LLMs)*

Description

Some "family" objects in **spaMM** describe models with non-GLM distribution families, such as the `negbin1` or `beta_resp` families already widely considered in previous works and other packages.

These models are characterized by a linear predictor, a link function, and a distribution for residual variation that does not belong to the exponential family from which GLMs are defined.

These family objects are conceived for use with **spaMM**'s fitting functions. They cannot generally be used as argument to the `glm` function, except when this function is highjacked by use of the `method="llm.fit"` argument, where llm stands for Link-Linear (as in "log-linear", say) regression Model.

Mixed-effect models fitted by such methods cannot use approximations based on the expected information matrix, in contrast to GLM distribution families.

Usage

```
# glm(..., method="llm.fit")
## See also 'beta_resp', 'negbin1', 'betabin', and possibly later additions.
```

Details

These family objects are lists, formally of class `c("LLF", "family")`. Compared to a `family` object, they have additional elements, not documented here.

As `stats::` GLM family objects do, they provide deviance residuals through the `dev.resids` member function. There are various definitions of deviance residuals for non-GLM families in the literature. Here they are defined from the unit deviances "2*(saturated_logLik - logLik)$_i$" for each observation i (provided by the `dev.resids` function defined by the family), where the likelihood for the saturated model is the likelihood maximized wrt to the mean parameter μ_i for each observation y_i independently. The maximizing μ_i is not equal to the observation y_i, in contrast to the standard result for GLMs.

`llm.fit` provides the deviance (the sum of the above unit deviances), but not the null deviance provided by `stats::glm.fit`. The latter is the deviance of an intercept-only (or offset only) model. There, the intercept estimate is the mean response value for GLMs, so that the null deviance is immediately obtained; but this is not generally so for other families.

Examples

```
data(scotlip)

### negbin1 response:

# Fixed-effect model
#
(var_shape <- fitme(cases~I(prop.ag/10)+offset(log(expec)),family=negbin1(),
                    data=scotlip))

# Highjacking glm(): the family parameter must be given
#
fitted_shape <- residVar(var_shape,which="fam_parm")
glm(cases~I(prop.ag/10)+offset(log(expec)),family=negbin1(shape=fitted_shape),
    method="llm.fit", data=scotlip)

### Similar exercice with Beta family:
```

```
set.seed(123)
beta_dat <- data.frame(y=runif(100),grp=sample(2,100,replace = TRUE))

# Fixed-effect model
(var_prec <- fitme(y ~1, family=beta_resp(), data= beta_dat))

# Highjacking glm():
fitted_prec <- residVar(var_prec,which="fam_parm")
glm(y ~1, family=beta_resp(prec=fitted_prec), data= beta_dat, method="llm.fit")
```

`Loaloa` *Loa loa prevalence in North Cameroon, 1991-2001*

Description

This data set describes prevalence of infection by the nematode *Loa loa* in North Cameroon, 1991-2001. This is a superset of the data discussed by Diggle and Ribeiro (2007) and Diggle et al. (2007). The study investigated the relationship between altitude, vegetation indices, and prevalence of the parasite.

Usage

```
data("Loaloa")
```

Format

The data frame includes 197 observations on the following variables:

latitude latitude, in degrees.

longitude longitude, in degrees.

ntot sample size per location

npos number of infected individuals per location

maxNDVI maximum normalised-difference vegetation index (NDVI) from repeated satellite scans

seNDVI standard error of NDVI

elev1 altitude, in m.

elev2,elev3,elev4 Additional altitude variables derived from the previous one, provided for convenience: respectively, positive values of altitude-650, positive values of altitude-1000, and positive values of altitude-1300

maxNDVI1 a copy of maxNDVI modified as `maxNDVI1[maxNDVI1>0.8] <- 0.8`

Source

The data were last retrieved on March 1, 2013 from P.J. Ribeiro's web resources at `www.leg.ufpr.br/doku.php/pessoais:paulojus:mbgbook:datasets`. A current (2022-06-18) source is https://www.lancaster.ac.uk/staff/diggle/moredata/Loaloa.txt).

References

Diggle, P., and Ribeiro, P. 2007. Model-based geostatistics, Springer series in statistics, Springer, New York.

Diggle, P. J., Thomson, M. C., Christensen, O. F., Rowlingson, B., Obsomer, V., Gardon, J., Wanji, S., Takougang, I., Enyong, P., Kamgno, J., Remme, J. H., Boussinesq, M., and Molyneux, D. H. 2007. Spatial modelling and the prediction of Loa loa risk: decision making under uncertainty, Ann. Trop. Med. Parasitol. 101, 499-509.

Examples

```
data("Loaloa")
if (spaMM.getOption("example_maxtime")>5) {
  fitme(cbind(npos,ntot-npos)~1 +Matern(1|longitude+latitude),
        data=Loaloa, family=binomial())
}

### Variations on the model fit by Diggle et al.
###    on a subset of the Loaloa data
### In each case this shows the slight differences in syntax,
###    and the difference in 'typical' computation times,
###    when fit using corrHLfit() or fitme().

if (spaMM.getOption("example_maxtime")>4) {
  corrHLfit(cbind(npos,ntot-npos)~elev1+elev2+elev3+elev4+maxNDVI1+seNDVI
                  +Matern(1|longitude+latitude),method="HL(0,1)",
                data=Loaloa,family=binomial(),ranFix=list(nu=0.5))
}
if (spaMM.getOption("example_maxtime")>1.6) {
  fitme(cbind(npos,ntot-npos)~elev1+elev2+elev3+elev4+maxNDVI1+seNDVI
                  +Matern(1|longitude+latitude),method="HL(0,1)",
                data=Loaloa,family=binomial(),fixed=list(nu=0.5))
}

if (spaMM.getOption("example_maxtime")>5.8) {
  corrHLfit(cbind(npos,ntot-npos)~elev1+elev2+elev3+elev4+maxNDVI1+seNDVI
            +Matern(1|longitude+latitude),
              data=Loaloa,family=binomial(),ranFix=list(nu=0.5))
}
if (spaMM.getOption("example_maxtime")>2.5) {
  fitme(cbind(npos,ntot-npos)~elev1+elev2+elev3+elev4+maxNDVI1+seNDVI
            +Matern(1|longitude+latitude),
              data=Loaloa,family=binomial(),fixed=list(nu=0.5),method="REML")
}

## Diggle and Ribeiro (2007) assumed (in this package notation) Nugget=2/7:
if (spaMM.getOption("example_maxtime")>7) {
  corrHLfit(cbind(npos,ntot-npos)~elev1+elev2+elev3+elev4+maxNDVI1+seNDVI
           +Matern(1|longitude+latitude),
             data=Loaloa,family=binomial(),ranFix=list(nu=0.5,Nugget=2/7))
}
if (spaMM.getOption("example_maxtime")>1.3) {
```

```
  fitme(cbind(npos,ntot-npos)~elev1+elev2+elev3+elev4+maxNDVI1+seNDVI
           +Matern(1|longitude+latitude),method="REML",
             data=Loaloa,family=binomial(),fixed=list(nu=0.5,Nugget=2/7))
}

## with nugget estimation:
if (spaMM.getOption("example_maxtime")>17) {
  corrHLfit(cbind(npos,ntot-npos)~elev1+elev2+elev3+elev4+maxNDVI1+seNDVI
           +Matern(1|longitude+latitude),
             data=Loaloa,family=binomial(),
             init.corrHLfit=list(Nugget=0.1),ranFix=list(nu=0.5))
}
if (spaMM.getOption("example_maxtime")>5.5) {
  fitme(cbind(npos,ntot-npos)~elev1+elev2+elev3+elev4+maxNDVI1+seNDVI
           +Matern(1|longitude+latitude),
             data=Loaloa,family=binomial(),method="REML",
             init=list(Nugget=0.1),fixed=list(nu=0.5))
}
```

LRT *Likelihood ratio tests of fixed and random effects.*

Description

LRT performs a likelihood ratio (LR) test between two model fits. It differs from another function with the same effect, fixedLRT, by its arguments (model fits for LRT, but all arguments required to fit the models for fixedLRT), and by the format of its return value. LRT determines which model is the more complete one by comparing model components including the fixed-effect, random-effect, residual-dispersion model specifications, and response families (offsets are ignored). Then, a standard test based on the asymptotic chi-square distribution is performed. In addition, parametric bootstrap p-values can be computed, either using the *raw bootstrap* distribution of the likelihood ratio, or a bootstrap estimate of the Bartlett correction of the LR statistic.

These different tests perform diffferently depending on the differences between the two models:

* If the models differ only by their fixed effects, the asymptotic LRT may be anticonservative, but the Bartlett-corrected one is generally well-calibrated.

* If the two models differ by their random effects, tests based on the chi-square distribution (including their Bartlett-corrected version) may be poorly behaved, as such tests assume unbounded parameters, as contrasted to, e.g., necessarily positive variances.
In such cases the raw boostrap test may be the only reliable test. The procedure aims to detect and report such issues, but may not report all problems: users remain responsible for applying the tests in valid conditions (see **Caveats** in Details section). In simple cases (such as comparing a fixed-effect to a mixed-effect model with the same fixed-effect term), the chi-square tests may not be reported. In other cases (see Examples) they may otherwise be reported, with a warning when the procedure detects some cases of estimates at the boundary for the full model, or detects cases where the LR statistic of bootstrap replicates is often zero (also suggesting that estimates are at the

boundary in such replicates).

* If the fits differ by the fixed effects terms of their residual-dispersion models (but not by any random effect specification), tests based on the chi-square distribution are reported. A bootstrap can be performed as in other cases.
* Tests for some cases of nested response families (e.g., the Poisson versus its extensions) are tentatively allowed.
* In some cases the full and the null models cannot be identified and the basic LRT based on the chi-square distribution will not be returned, but a bootstrap test may still be performed.
* The case where residual-dispersion models of either fit include random effects is problematic as, for such fits, the fitting procedure does not maximize the reported likelihood. The basic LRT is not returned when the two fits differ by their random effects, but is still performed otherwise (see Examples); and a bootstrap test may still be performed in both cases.

Usage

```
LRT(object, object2, boot.repl = 0L, resp_testfn = NULL,
    simuland = eval_replicate, include="call",
    #     many further arguments can be passed to spaMM_boot via the '...'
    #     These include arguments for parallel computations, such as
    # nb_cores, fit_env,
    #     as well as other named arguments and spaMM_boot's own '...'
    ...)
```

Arguments

`object`	Fit object returned by a **spaMM** fitting function.
`object2`	Optional second model fit to be compared to the first (their order does not matter, except in non-standard cases where the second model is taken as teh null one and a message is issued).
`boot.repl`	the number of bootstrap replicates.
`resp_testfn`	See argument `resp_testfn` of `spaMM_boot`.
`simuland`	a function, passed to `spaMM_boot`. See argument `eval_replicate` for default value and requirements.
`include`	character string, either `"fit"` or `"call"` (default; other values are ignored). This argument controls whether the fits compared, or only their calls, are included in the return value.
`...`	Further arguments, passed to `spaMM_boot` (e.g., for parallelization, or a random seed).

Details

* **Identifying a nested model**: **spaMM** tries to guess which is the nested model, but this is not always possible.

In particular, the fit with (substantially) lower logLik is not necessarily the one with fewer degrees of freedom, if the models are not nested, or even for some form of "nestedness" with constrained parameters. The latter case may occur if one first fits a simple model, then fits a more complex model

containing the first model **but** one constrains estimates of shared parameters to the values given by the first fit: the second fit should then have a higher likelihood but it may still have fewer estimated parameters than the first one. A likelihood ratio test is invalid anyway when such constraints are used.

Second, although nestedness is relatively easy to assess for fixed effects, equivalent random-effect models can be specified by diverse syntaxes, so a simple textual comparison of the random-effect terms may not be enough, and model specifications that hinder such a comparison should be avoided. When differences in random effects are tested, the null distribution of the LR may include a probability mass in 1: the discussion in Details of `get_RLRsim_args` applies.

* `data` **checks**: LRTs are invalid if the two models are fitted on different **informative** data. Informative data may differ even if the input data are identical, because the fitting procedures ignore lines of data with missing information. The retained data may then differ for the two models, if additional variables of the more complete model are missing from some lines of data retained in the fit of the less complete model.

`LRT` perform some checks based on comparison of the data used by the two models. However, a perfect check all cases where the informative data differ is not easily defined. For example, data may appear different but still contain the same information if lines of the data are reordered. Hence, users remain ultimately responsible for checking whether informative data are equivalent in the two models. They should keep in mind that data processing is applied on a per-submodel basis for multivariate-response fits, and `pois4mlogit` has further criteria for definition of informative data for each model.

* **Bootstrap LRTs**: A raw bootstrap p-value can be computed from the simulated distribution as `(1+sum(t >= t0))/(N+1)` where `t0` is the original likelihood ratio, `t` the vector of bootstrap replicates and `N` its length. See Davison & Hinkley (1997, p. 141) for discussion of the adjustments in this formula. However, a computationally more economical use of the bootstrap is to provide a Bartlett correction for the likelihood ratio test in small samples. According to this correction, the mean value m of the likelihood ratio statistic under the null hypothesis is computed (here estimated by a parametric bootstrap) and the original LR statistic is multiplied by n/m where n is the number of degrees of freedom of the test.

Bootstrap samples are not returned, and they are by default simulated internally (by a call to `spaMM_boot` itself calling `simulate4boot`). There are several ways to recover the samples if you need them: (1) simulate them beforehand and provide them as the `boot_samples` argument, passed to `spaMM_boot` through the ... argument; (2) provide a `seed` value (passed down to `simulate4boot` through the ... argument) and re-run `simulate4boot` with this `seed` and with other relevant arguments of the original `LRT` call; (3) same as (2), but using the `set.seed` function insted of the `seed` argument; (4) given only an `LRT` output where the previous solutions were ignored, you can still use the `RNGstates` value contained in the object returned by `LRT` to reset the generator state and re-run `simulate4boot`, as shown in the Examples for that function.

* **Caveats**: (1) An evaluated log-likelihood ratio can be slightly negative, e.g. when a fixed-effect model is compared to a mixed one, or a spatial random effect to a block effect, if parameters of the more complete model are estimated within bounds (e.g., variance>1e-06, or Matern smoothness>0.005) designed to avoid numerical singularities, while the less complete model corresponds to a boundary case (e.g., variance=0, or smoothness=0). The bootstrap procedure tries to identify these cases and then corrects slightly negative logL ratios to 0. (2) The Bartlett correction is applicable when the true distribution of the LRT departs smoothly from the chi-square distribution, but not in cases where it has a probability mass in zero (at typically occurs in the same boundary cases).

Value

LRT returns an object of class fixedLRT, unless some exception occurs. This is a list with typical elements (depending on the options)

fullfit, nullfit
: the fit objects for the full and null models, or only their calls, depending on the include argument;

basicLRT
: A data frame including values of the likelihood ratio chi2 statistic, its degrees of freedom, and the p-value;

and, if a bootstrap was performed:

rawBootLRT
: A data frame including values of the likelihood ratio chi2 statistic, its degrees of freedom, and the raw bootstrap p-value;

BartBootLRT
: A data frame including values of the Bartlett-corrected likelihood ratio chi2 statistic, its degrees of freedom, and its p-value;

bootInfo
: a list with the following elements:

bootreps A table of fitted likelihoods for bootstrap replicates;

meanbootLRT The mean likelihood ratio chi-square statistic for bootstrap replicates;

RNGstates the state of .Random.seed at the beginning of the simulation of the bootstrap samples (present only is such simulation was run).

References

Bartlett, M. S. (1937) Properties of sufficiency and statistical tests. Proceedings of the Royal Society (London) A 160: 268-282.

Davison A.C., Hinkley D.V. (1997) Bootstrap methods and their applications. Cambridge Univ. Press, Cambridge, UK.

See Also

See also fixedLRT and anova.HLfit

Examples

```
## Using resp_testfn argument for bootstrap LRT:
## Not run:
set.seed(1L)
d <- data.frame(success = rbinom(10, size = 1, prob = 0.9), x = 1:10)
xx <- cbind(1,d$x)
table(d$success)
m_x <- fitme(success ~ x, data = d, family = binomial())
m_0 <- fitme(success ~ 1, data = d, family = binomial())
#
# Bootstrap LRTs:
anova(m_x, m_0, boot.repl = 100,
      resp_testfn=function(y) {! is_separated(xx,as.numeric(y),verbose=FALSE)})
```

```
## End(Not run)

#### Various cases were asymptotic tests may be unreliable:

set.seed(123)
dat <- data.frame(g = rep(1:10, e = 10), x = (x<-rnorm(100)),
                  y = 0.1 * x + rnorm(100))
m0 <- fitme(y ~ 1, data=dat)

## (1) Models differing both by fixed and random effects:
#
# (note the warning for variance at boundary):
#
if (spaMM.getOption("example_maxtime")>11) {
  m <- fitme(y ~ x + (1|g), data=dat)
  LRT(m,m0, boot.repl = 199L)
}
## See help("get_RLRsim_args") for a fast and accurate test procedure

## (2) Models differing also by residual-dispersion models:
#
if (spaMM.getOption("example_maxtime")>25) {
  m <- fitme(y ~ x + (1|g), data=dat, resid.model= ~x)
  LRT(m,m0, boot.repl = 99L)
}

## (3) Models differing (also) by their random-effects in resid.model:
#
m <- fitme(y ~ x, data=dat, resid.model= ~1+(1|g))
LRT(m,m0) # no test performed
```

`make_scaled_dist` *Scaled distances between unique locations, and Earth distances*

Description

This function computes scaled distances from whichever relevant argument it can use (see Details). The result can directly by used as input for computation of the Matérn correlation matrix. It is usually called internally by HLCor, so that users may ignore it, except if they wish to control the distance used through `control.dist$method`, or the parametrization of the scaling through `control.dist$rho.mapping`. `control.dist$method` provide access to the distances implemented in the `proxy` package, as well as to `"EarthChord"` and `"Earth"` methods defined in `spaMM` (see Details).

Usage

```
make_scaled_dist(uniqueGeo, uniqueGeo2=NULL, distMatrix, rho,
```

```
rho.mapping=seq_len(length(rho)),
dist.method="Euclidean",
return_matrix=FALSE)
```

Arguments

uniqueGeo	A matrix of geographical coordinates (e.g. 2 columns for latitude and longitude), without replicates of the same location.
uniqueGeo2	NULL, or a second matrix of geographical coordinates, without replicates of the same location. If NULL, scaled distances among uniqueGeo locations are computed. Otherwise, scaled distances between locations in the two input matrices are computed.
distMatrix	A distance matrix.
rho	A scalar or vector of positive values. Scaled distance is computed as <distances in each coordinate> * rho, unless a non-trivial rho.mapping is used.
rho.mapping	A set of indices controlling which elements of the rho scale vector scales which dimension(s) of the space in which (spatial) correlation matrices of random effects are computed. Scaled distance is generally computed as <distances in each coordinate> * rho[rho.mapping]. As shown in the Example, if one wishes to combine isotropic geographical distance and some environmental distance, the coordinates being latitude, longitude and one environmental variable, the scaled distance may be computed as (say) (lat,long,env) *rho[c(1,1,2)] so that the same scaling rho[1] applies for both geographical coordinates. In this case, rho should have length 2 and rho.mapping should be c(1,1,2).
dist.method	method argument of proxy::dist function (by default, "Euclidean", but other distances are possible (see Details).
return_matrix	Whether to return a matrix rather than a proxy::dist or proxy::crossdist object.

Details

The function uses the distMatrix argument if provided, in which case rho must be a scalar. Vectorial rho (i.e., different scaling of different dimensions) is feasible only by providing uniqueGeo.

The dist.method argument gives access to distances implemented in the proxy package, or to user-defined ones that are made accessible to proxy through its database. Of special interest for spatial analyses are distances computed from longitude and latitude (proxy implements "Geodesic" and "Chord" distances but they do not use such coordinates: instead, they use Euclidean distance for 2D computations, i.e. Euclidean distance between points on a circle rather than on a sphere). spaMM implements two such distances: "Earth" and "EarthChord", using longitude and latitude inputs **in that order** (see Examples). The "EarthChord" distance is the 3D Euclidean distance "through Earth". The "Earth" distance is also known as the orthodromic or great-circle distance, on the Earth surface. Both distances functions **return values in km** and are based on approximating the Earth by a sphere of radius 6371.009 km. The **input values** of longitude and latitude (in the data) must be **in degrees**.

Value

A matrix or dist object. If there are two input matrices, rows of the return value correspond to rows of the first matrix.

Examples

```
data("blackcap")
## a biologically not very meaningful, but syntactically correct example of rho.mapping
fitme(migStatus ~ 1 + Matern(1|longitude+latitude+means),
      data=blackcap, fixed=list(nu=0.5,phi=1e-6),
      init=list(rho=c(1,1)), control.dist=list(rho.mapping=c(1,1,2)))

## Using orthodromic distances:
# order of variables in Matern(.|longitude+latitude) matters;
# Matern(1|latitude+longitude) should cause a warning
fitme(migStatus ~ 1 + Matern(1|longitude+latitude),data=blackcap,
      method="ML", fixed=list(nu=0.5,phi=1e-6),
      control.dist=list(dist.method="Earth"))
```

---

mapMM *Colorful plots of predictions in two-dimensional space.*

Description

These functions provide either a map of predicted response in analyzed locations, or a predicted surface. mapMM is a straightforward representation of the analysis of the data, while filled.mapMM uses interpolation to cope with the fact that all predictor variables may not be known in all locations on a fine spatial grid. map_ranef maps a single spatial random effect. These three functions takes an HLfit object as input. mapMM calls spaMMplot2D, which is similar but takes a more conventional (x,y,z) input.

Using filled.mapMM may involve questionable choices. Plotting a filled contour generally requires prediction in non-observed locations, where predictor variables used in the original data analysis may be missing. In that case, the original model formula cannot be used and an alternative model (controlled by the map.formula argument) must be used to interpolate (not smooth) the predicted values in observed locations (these predictions still resulting from the original analysis based on predictor variables). filled.mapMM always performs such interpolation (it does not allow one to provide values for the predictor variables). As a result (1) filled.mapMM will be slower than a mere plotting function, since it involves the analysis of spatial data; (2) the results may have little useful meaning if the effect of the original predictor variables is not correctly represented by this interpolation step. For example, prediction by interpolation may be biased in a way analogous to prediction of temperature in non-observed locations while ignoring effect of variation in altitude in such locations. Likewise, the variance argument of filled.mapMM allows one only to plot the prediction variance of its own interpolator, rather than that of the input object.

map_ranef is free of the limitations of filled.mapMM.

Usage

```
spaMMplot2D(x, y, z, xrange=range(x, finite = TRUE),
  yrange=range(y, finite = TRUE), margin=1/20, add.map= FALSE,
  nlevels = 20, color.palette = spaMM.colors, map.asp=NULL,
  col = (color.palette)(n = nlevels), plot.title=NULL, plot.axes=NULL,
  decorations=NULL, key.title=NULL, key.axes=NULL, xaxs = "i",
  yaxs = "i", las = 1, axes = TRUE, frame.plot = axes, ...)

mapMM(fitobject,Ztransf=NULL,coordinates,
  add.points,decorations=NULL,plot.title=NULL,plot.axes=NULL,envir=-3, ...)

filled.mapMM(
  fitobject, Ztransf = NULL, coordinates, xrange = NULL, yrange = NULL,
  margin = 1/20, map.formula, phi = 1e-05, gridSteps = 41,
  decorations = quote(points(pred[, coordinates], cex = 1, lwd = 2)),
  add.map = FALSE, axes = TRUE, plot.title = NULL, plot.axes = NULL,
  map.asp = NULL, variance = NULL, var.contour.args = list(),
  smoothObject = NULL, return.="smoothObject", ...)

map_ranef(fitobject, re.form, Ztransf=NULL, xrange = NULL, yrange = NULL,
  margin = 1/20, gridSteps = 41,
  decorations = quote(points(fitobject$data[, coordinates], cex = 1, lwd = 2)),
  add.map = FALSE, axes = TRUE, plot.title=NULL, plot.axes=NULL,
  map.asp = NULL, mv_it=NULL, ...)
```

Arguments

fitobject	A fit object of class HLfit, as returned by fitme or corrHLfit.
x, y, z	Three vectors of coordinates, with z being expectedly the response.
re.form	A model formula giving the single random effect term to plot, needed only if there are several spatial random effects in the fitted model. In that case, it must be formatted as . ~ <term>, as for the re.form argument of predict.HLfit.
Ztransf	A transformation of the predicted response, given as a function whose only required argument can be a one-column matrix. The name of this argument must be Z (not x), as is appropriate for use in do.call(Ztransf,list(Z=Zvalues)).
coordinates	Vector of character strings, giving the names of the geographical coordinates. By default they are deduced from the model formula. For example if this formula is resp ~ 1 + Matern(1| x + y) the default coordinates are c("x","y"). If this formula is resp ~ 1 + Matern(1| x + y + z), the user must choose two of the three coordinates.
xrange	The x range of the plot (a vector of length 2); by default defined to cover all analyzed points.
yrange	The y range of the plot (a vector of length 2); by default defined to cover all analyzed points.
margin	This controls how far (in relative terms) the plot extends beyond the x and y ranges of the analyzed points, and is overriden by explicit xrange and yrange arguments.

`map.formula` NULL, or a formula whose left-hand side is ignored. Provides the formula used for interpolation. If NULL, a default formula with the same spatial effect(s) as in the input `fitobject` is used.

`phi` This controls the phi value assumed in the interpolation step. Ideally `phi` would be zero, but problems with numerically singular matrices may arise when `phi` is too small.

`gridSteps` The number of levels of the grid of x and y values

`variance` Either NULL, or the name of a component of variance of prediction *by the interpolator* to be plotted. Must name one of the components that can be returned by `predict.HLfit`. `variance="predVar"` is suitable for uncertainty in point prediction.

`var.contour.args`
A list of control parameters for rendering of prediction variances. See `contour` for possible arguments (except `x`, `y`, `z` and `add`).

`add.map` Either a boolean or an explicit expression, enclosed in `quote` (see Examples). If `TRUE`, the `map` function from the `maps` package (which much therefore the loaded) is used to add a map from its default `world` database. `xrange` and `yrange` are used to select the area, so it is most convenient if the `coordinates` are longitude and latitude (in this order and in standard units). An explicit expression can also be used for further control.

`nlevels` if `levels` is not specified, the range of z, values is divided into *approximately* this many levels (a call to `pretty` determines the actual number of levels).

`color.palette` a color palette function to be used to assign colors in the plot.

`map.asp` the y/x aspect ratio of the 2D plot area (not of the full figure including the scale). By default, the scales for x and y are identical unless the x and y ranges are too different. Namely, the scales are identical if (plotted y range)/(plotted x range) is $1/4 < . < 4$, and map.asp is 1 otherwise.

`col` an explicit set of colors to be used in the plot. This argument overrides any palette function specification. There should be one less color than levels

`plot.title` statements which add titles to the main plot. See Details for differences between functions.

`plot.axes` statements which draw axes (and a box) on the main plot. See Details for differences between functions.

`decorations` Either NULL or Additional graphic statements (`points`, `polygon`, etc.), enclosed in `quote` (the default value illustrates the latter syntax). .

`add.points` Obsolete, use `decorations` instead.

`envir` Controls the environment in which `plot.title`, `plot.axes`, and `decorations` are evaluated. `mapMM` calls `spaMM2Dplot` from where these graphic arguments are evaluated, and the default value -3 means that they are evaluated within the environment from where `mapMM` was called.

`key.title` statements which add titles for the plot key.

`key.axes` statements which draw axes on the plot key.

`xaxs` the x axis style. The default is to use internal labeling.

yaxs	the y axis style. The default is to use internal labeling.
las	the style of labeling to be used. The default is to use horizontal labeling.
axes, frame.plot	logicals indicating if axes and a box should be drawn, as in plot.default.
smoothObject	Either NULL, or an object inheriting from class HLfit (hence, an object on which predict.HLfit can be called), predicting the response surface in any coordinates. See Details for typical usages.
return.	character string: see Value
mv_it	NULL or integer: for multivariate-response fits, specify a submodel.
...	further arguments passed to or from other methods. For mapMM, all such arguments are passed to predict.HLfit and spaMMplot2D; for spaMMplot2D, currently only additional graphical parameters passed to title() (see Details). For filled.mapMM and map_ranef, these parameters are those that can be passed to spaMM.filled.contour.

Details

The smoothObject argument may be used to redraw a figure faster by recycling the predictor of the response surface returned invisibly by a previous call to filled.mapMM.

For smoothObject=NULL (the default), filled.mapMM interpolates the predicted response, with sometimes unpleasant effects. For example, if one interpolates probabilities, the result may not be within [0,1], and then (say) a logarithmic Ztransf may generate NaN values that would otherwise not occur. The smoothObject argument may be used to overcome the default behaviour, by providing an alternative predictor.

If you have values for all predictor variables in all locations of a fine spatial grid, filled.mapMM may not be a good choice, since it will ignore that information (see map.formula argument). Rather, one should use predict(<fitobject>,newdata= <all predictor variables >) to generate all predictions, and then either spaMM.filled.contour or some other raster functions.

The different functions are (currently) inconsistent among themselves in the way they handle the plot.title and plot.axes argument:

spaMM.filled.contour behaves like graphics::filled.contour, which (1) handles arguments which are calls such as title(.) or {axis(1);axis(2)}; (2) ignores ... arguments if plot.title is missing; and (3) draws axes by default when plot.axes is missing, given axes = TRUE.

By contrast, **filled.mapMM** handles arguments which are language expressions such as produced by quote(.) or substitute(.) (see Examples).

mapMM can handles language expressions, but also accepts at least some calls.

Value

filled.mapMM by default returns invisibly the fit object predicting the interpolated response surface; however, for any non-default return. argument (return.="raster" would be recommended to ensure future back-compatibility), it will return a raster of values as a list with elements x, y and z. map_ranef returns invisibly a 3-column matrix containing the spatial coordinates, and the predicted effect z on the linear predictor scale (which is also the scale of the plot, unless a Ztransf is used). mapMM returns invisibly a list with elements x, y and z. Plots are produced as side-effects.

See Also

seaMask for masking areas in a filled map; https://gitlab.in2p3.fr/francois.rousset/spamm-ref/-/blob/master/vignettePlus/example_raster.html for more elaborate plot procedures.

Examples

```
data("blackcap")
bfit <- fitme(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap,
              fixed=list(lambda=0.5537,phi=1.376e-05,rho=0.0544740,nu=0.6286311))
mapMM(bfit,color.palette = function(n){spaMM.colors(n,redshift=1/2)},add.map=TRUE)
map_ranef(bfit) # providing argument re.form= . ~ Matern(1|longitude+latitude)

if (spaMM.getOption("example_maxtime")>1) {
  ## filled.mapMM takes a bit longer
  # showing 'add.map', 'nlevels', and contour lines for 'variance'
  filled.mapMM(bfit, nlevels=30, add.map=TRUE, plot.axes=quote({axis(1);axis(2)}),
             variance="respVar",
             plot.title=title(main="Inferred migration propensity of blackcaps",
                              xlab="longitude",ylab="latitude"))

  ## Similar plots by ggplot2:
  ## Not run:
   library(rnaturalearth) # provides sea mask through 'ne_download' function
   library(ggplot2)
   library(sp)

   # sea mask
  sea <- ne_download(scale = 10, type = 'ocean', category = "physical", returnclass = "sf")

   # Generation of data.frame for ggplot:
   rastr <- filled.mapMM(bfit, return.="raster")
  spdf <- data.frame(Long=rep(rastr$x, nc), Lat=rastr$y[gl(nr,nc)], z = as.vector(rastr$z))

   ggplot(spdf) +
     geom_contour_filled(aes(Long,Lat,z=z), bins = 20) +
     guides(fill = "none") +
     geom_sf(data = sea, fill = "black") +
     coord_sf(ylim = range(rastr$y), xlim = range(rastr$x), expand = FALSE)

## End(Not run)

}

if (spaMM.getOption("example_maxtime")>3) {
 data("Loaloa")
 lfit <- fitme(cbind(npos,ntot-npos)~elev1+elev2+elev3+elev4+maxNDVI1+seNDVI
                 +Matern(1|longitude+latitude), method="PQL", data=Loaloa,
                 family=binomial(), fixed=list(nu=0.5,rho=2.255197,lambda=1.075))

 ## longer computation requiring interpolation of 197 points
 ## Also illustrating effect of 'return.' argument
 res <- filled.mapMM(lfit,add.map=TRUE,plot.axes=quote({axis(1);axis(2)}),
```

```
    decorations=quote(points(pred[,coordinates],pch=15,cex=0.3)),
    return.="raster", # so that 'res' is a list representing a raster.
    plot.title=title(main="Inferred prevalence, North Cameroon",
                     xlab="longitude",ylab="latitude"))
}
```

`MaternCorr` *Matern correlation function and Matern formula term.*

Description

The Matérn correlation function describes realizations of Gaussian spatial processes with different smoothnesses (i.e. either smooth or rugged surfaces, controlled by the ν parameter). It also includes a ρ scaling parameter and an optional 'nugget' parameter. A random effect specified in a model formula as `Matern(1|<...>)` has pairwise correlations given by the Matérn function at the scaled Euclidean distance between coordinates specified in `<...>`, using "+" as separator (e.g., `Matern(1|longitude+latitude)`). The Matern family can be used in Euclidean spaces of any dimension; and also for correlations on a sphere (with maximum smoothness `nu=0.5`).

A syntax of the form `Matern(1|longitude+latitude %in% grp)` can be used to specify a Matern random effect with independent realizations (but identical correlation parameters) for each level of the grouping variable `grp`. Alternatively, the `Matern(<T/F factor>|longitude+latitude)` may be used to specify Matern effects specific to individuals identified by the `<T/F factor>` (see Example with females and males). In that case distinct correlation parameters are fitted for each such Matern term.

When group-specific autocorrelated random effects are fitted, it may be wise to allow for different means for each group in the Intercept (a message will point this out if the fit results for Matern or Cauchy terms suggest so).

By default, `fitme` and `corrHLfit` performs optimization over the ρ and ν parameters. It is possible to estimate different scaling parameters for the different Euclidean dimensions: see examples in `make_scaled_dist`.

The `MaternCorr` function may be used to vizualise these correlations, using distances as input.

Usage

```
## Default S3 method:
MaternCorr(d, rho = 1, smoothness, nu = smoothness, Nugget = NULL)
# Matern(1|...)
```

Arguments

`d`	A distance or a distance matrix.
`rho`	A scaling factor for distance. The 'range' considered in some formulations is the reciprocal of this scaling factor

smoothness	The smoothness parameter, >0. $\nu = 0.5$ corresponds to the exponential correlation function, and the limit function when ν goes to ∞ is the squared exponential function (as in a Gaussian).
nu	Same as smoothness
Nugget	(Following the jargon of Kriging) a parameter describing a discontinuous decrease in correlation at zero distance. Correlation will always be 1 at $d = 0$, and from which it immediately drops to (1-Nugget)
...	Names of coordinates, using "+" as separator (e.g., `Matern(1|longitude+latitude)`. The coordinates are numeric values found in the `data` data frame provided to the fitting function. No additional declaration of groups, factors, or other specific formatting is required.

Details

The correlation at distance $d > 0$ is

$$(1 - \text{Nugget}) \frac{(\rho d)^\nu K_\nu(\rho d)}{2^{(\nu-1)}\Gamma(\nu)}$$

where K_ν is the `besselK` function of order ν.

By default the Nugget is set to 0. See one of the examples on data set `Loaloa` for a fit including the estimation of the Nugget.

Value

Scalar/vector/matrix depending on input.

References

Stein, M.L. (1999) Statistical Interpolation of Spatial Data: Some Theory for Kriging. Springer, New York.

See Also

See `corMatern` for an implementation of this correlation function as a `corSpatial` object for use with `lme` or `glmmPQL`.

Examples

```
## See examples in help("HLCor"), help("Loaloa"),  help("make_scaled_dist"), etc.
## Matern correlations in 4-dimensional space:
set.seed(123)
randpts <- matrix(rnorm(20),nrow=5)
distMatrix <- as.matrix(proxy::dist(randpts))
MaternCorr(distMatrix,nu=2)

## Group-specific random effects:
if (spaMM.getOption("example_maxtime")>1.6) {
  data(Leuca)
 subLeuca <- Leuca[c(1:10,79:88),] # subset of 10 females and 10 males, for faster examples.
```

```
  # Independent Matern random effect with different covariance parameters for each sex:
  fitme(fec_div ~ sex + Matern(female|x + y) + Matern(male|x + y),  data = subLeuca)

  # Independent Matern random effect with the same covariance parameters for each sex:
  fitme(fec_div ~ sex + Matern(1|x+y %in% sex),data=subLeuca)

  # Matern models with random-effects distinct but correlated across sexes
  # can also be fitted: see Matern examples in help("composite-ranef").
}
```

`MaternIMRFa` *corrFamily constructor for Interpolated Markov Random Field (IMRF) covariance structure approximating a 2D Matern correlation model.*

Description

Reimplements the `IMRF` correlation model approximating a Matern correlation function, through a `corrFamily` constructor. This allows the efficient joint estimation of the `alpha` parameter of the approximating Markov random field (in principle related to the smoothness parameter of the `Matern` correlation function) together with its `kappa` parameter. By contrast, random effects terms specified as `IMRF(1| . , model = <INLA::inla.spde2.matern result>)` assume a fixed `alpha`.

Using this feature requires that the not-on-CRAN package **INLA** (`https://www.r-inla.org`) is installed so that `INLA::inla.spde2.matern` can be called for each `alpha` value.

Usage

```
# corrFamily constructor:
MaternIMRFa(mesh, tpar = c(alpha = 1.25, kappa = 0.1), fixed = NULL, norm=FALSE)
```

Arguments

`mesh`	An `inla.mesh` object as produced by `INLA::inla.mesh.2d`.

and consistently with the general format of `corrFamily` constructors:

`tpar`	Named numeric vector: template values of the parameters of the model. Better not modified unless you know what you are doing.
`fixed`	NULL or numeric vector, to fix the parameters of this model.
`norm`	Boolean: whether to apply a normalization so that the random effect is homoscedastic (see `IMRF`) for details.

Value

A list suitable as input in the `covStruct` argument, with the following elements:

`f`	function returning a precision matrix for the random effect in mesh vertices;

tpar	template parameter vector (see general requirements of a corrFamily descriptor);
Af	function returning a matrix that implements the prediction of random effect values in data locations by interpolation of values in mesh locations (similarly to INLA::inla.spde.make.A);
type	specifies that the matrix returned by Cf is a precision matrix rather than a correlation matrix;

and possibly other elements which should not be considered as stable features of the return value.

References

Lindgren F., Rue H., Lindström J. (2011) An explicit link between Gaussian fields and Gaussian Markov random fields: the stochastic partial differential equation approach Journal of the Royal Statistical Society: Series B (Statistical Methodology), 73: 423-498. doi:10.1111/j.1467-9868.2011.00777.x

Examples

```
## Not run:

if(requireNamespace("INLA", quietly = TRUE)) {

data("Loaloa")

mesh <- fmesher::fm_mesh_2d_inla(loc = Loaloa[, c("longitude", "latitude")],
                     max.edge = c(3, 20))

### Fit with fixed alpha

(fit_MaternIMRF <- fitme(
  cbind(npos,ntot-npos) ~ elev1 + elev2 + elev3 + elev4 + maxNDVI1 +
        seNDVI +  MaternIMRFa(1|longitude+latitude, mesh=mesh, fixed=c(alpha=1.05)),
  family=binomial(),
  data=Loaloa, verbose=c(TRACE=interactive())) )

# For data sets with a small number of locations (as here), fitting
# the Matern model as follows is faster than fitting its MaternIMRFa approximation.
# Here this appears more than twofold faster

fit_Matern <- fitme(
  cbind(npos,ntot-npos) ~ elev1 + elev2 + elev3 + elev4 + maxNDVI1 +
        seNDVI + Matern(1|longitude+latitude),
  fixed=list(nu=0.05),                          # in principle similar to alpha=0.05
  data=Loaloa,family=binomial())



### Same with variable alpha

(fit_MaternIMRF <- fitme(
```

```
  cbind(npos,ntot-npos) ~ elev1 + elev2 + elev3 + elev4 + maxNDVI1 +
        seNDVI + MaternIMRFa(1|longitude+latitude, mesh),
  family=binomial(),
  data=Loaloa, verbose=c(TRACE=interactive())) )

# Comparable Matern fit:
fit_Matern <- fitme(
  cbind(npos,ntot-npos) ~ elev1 + elev2 + elev3 + elev4 + maxNDVI1 +
        seNDVI + Matern(1|longitude+latitude),
  init=list(nu=0.25), lower=list(nu=0), upper=list(nu=1),
  data=Loaloa,family=binomial())

# Note that the fitted nu and alpha parameters do not quite match each other,
# and that the IMRF likelihood does not really approximate the Matern likelihood.
# Mesh design would also be seen to matter.

} else print("INLA must be installed to run this example.")


## End(Not run)
```

`mat_sqrt` *Computation of "square root" of symmetric positive definite matrix*

Description

`mat_sqrt` is not usually directly called by users, but arguments may be passed to it through higher-level calls (see Examples). For given matrix **C**, it computes a factor **L** such that **C** = **L** * t(**L**), handling issues with nearly-singular matrices. The default behavior is to try Cholesky factorization, and use `eigen` if it fails. Matrix roots are not unique (for example, they are lower triangular for `t(chol(.))`, and symmetric for `svd(.)`. As matrix roots are used to simulate samples under the fitted model (in particular in the parametric bootstrap implemented in `fixedLRT`), this implies that for given seed of random numbers, these samples will differ with these different methods (although their distribution should be identical).

Usage

```
mat_sqrt(m = NULL, symSVD = NULL, try.chol = TRUE, condnum=1e12)
```

Arguments

`m`	The matrix whose 'root' is to be computed. This argument is ignored if `symSVD` is provided.
`symSVD`	A list representing the symmetric singular value decomposition of the matrix which 'root' is to be computed. Must have elements `$u`, a matrix of eigenvectors, and `$d`, a vector of eigenvalues.
`try.chol`	If `try.chol=TRUE`, the Cholesky factorization will be tried.

condnum (large) numeric value. In the case chol() was tried and failed, the matrix is regularized so that its (matrix 2-norm) condition number is reduced to condnum (in version 3.10.0 this correction has been implemented more exactly than in previous versions).

Value

For non-NULL m, its matrix root, with rows and columns labelled according to the columns of the original matrix. If eigen was used, the symmetric singular value decomposition (a list with members u (matrix of eigenvectors) and d (vector of eigenvalues)) is given as attribute.

Examples

```
## Not run:
## try.chol argument passed to mat_sqrt
## through the '...' argument of higher-level functions
## such as HLCor, corrHLfit, fixedLRT:
data("scotlip")
HLCor(cases~I(prop.ag/10) +adjacency(1|gridcode)+offset(log(expec)),
      ranPars=list(rho=0.174),adjMatrix=Nmatrix,family=poisson(),
      data=scotlip,try.chol=FALSE)

## End(Not run)
```

---

method *Fitting methods (objective functions maximized)*

Description

The method argument of the fitting functions, with possible values "ML", "REML","PQL", "PQL/L", and so on, controls whether restricted likelihood techniques are used to estimate residual variance and random-effect parameters, and the way likelihood functions are approximated.

By default, Laplace approximations are used, as selected by "ML" and "REML" methods. The Laplace approximation to (log-)marginal likelihood can be expressed in terms of the joint log-likelihood of the data and the random effects (or the *h*-likelihood in works of Lee and Nelder). The Laplace approximation is the joint likelihood minus half the log-determinant of the matrix of second derivatives (Hessian matrix) of the negative joint likelihood with respect to the random effects (observed information matrix). The Laplace approximation to restricted likelihood (for REML estimation) is similarly defined from the Hessian matrix with respect to random effects **and** fixed effects (for the adventurous, **spaMM** allows some non-standard specification of the fixed effects included in the definition of he Hessian).

Various additional approximations have been considered. Penalized quasi-likelihood (PQL), as originally defined for GLMMs by Breslow and Clayton (1993), uses a Laplace approximation of restricted likelihood to estimate dispersion parameters, and estimates fixed effects by maximizing the joint likelihood (h-likelihood). Although PQL has been criticized as an approximation of likelihood (and actual implementations may diverge from the original version), it has some interesting inferential properties. **spaMM** allows one to use an ML variant of PQL, named PQL/L.

Further approximations defined by Lee, Nelder and collaborators (e.g., Noh and Lee, 2007, for some overview) may mostly be seen as laying between PQL and the full Laplace method in terms of approximation of the likelihood, and as extending them to models with non-gaussian random effects ("HGLMs"). In practice the ML, REML, PQL and PQL/L methods should cover most (all?) needs for GLMMs, and EQL extends the PQL concept to HGLMs. `method="EQL+"` stands for the EQL method of Lee and Nelder (2001). The '+' signals that it includes the d v/ d tau correction described p. 997 of that paper, while `method="EQL-"` ignores it. `"PQL"` is equivalent to `EQL-` for GLMMs. `"PQL/L"` is PQL without the leverage corrections that characterize REML estimation of random-effect parameters.

spaMM uses the observed information matrix by default in its Laplace approximation (this was not so before version 4.0.0). By contrast, in Laplace approximations of likelihood described in the work of Lee & Nelder, i.e. for mixed-effect models with GLM response families, the information matrix is written in terms of the GLM weights (e.g., Lee & Nelder 2001, p.1004), and is thus effectively the expected information matrix, which differs from the observed information matrix in the case of GLM families with non-canonical link (McCullagh & Nelder 1989, p.42). Therefore, the likelihood approximation based on the expected information matrix differs from the one based on the observed information matrix in the same conditions.

For non-GLM distribution families (currently, the `negbin1`, `beta_resp` and `betabin`), only observed information is available (expected information would at best be quite difficult to evaluate, with no benefits). For GLM families, use of expected information matrix can be required at a global level by setting `spaMM.options(obsInfo=FALSE)` or in a specific fit by adding `"exp"` as a second specifier in the method (e.g., `method=c("ML","exp")`). This can be distinctly useful (in terms of speed) for fitting models with `Gamma(log)` family. Conversely, the `"obs"` specifier will enforce use of observed information matrix when the alternative is set at a global level.

Details

The `method` (or `HLmethod`) argument of fitting functions also accepts values of the form `"HL(<...>)"`, `"ML(<...>)"` and `"RE(<...>)"`, e.g. `method="RE(1,1)"`, which allow one to experiment with further combinations of approximations. HL and RE are equivalent (both imply an REML correction). The first '1' means that a Laplace approximation to the likelihood is used to estimate fixed effects (a '0' would instead mean that the h likelihood is used as the objective function). The second '1' means that a Laplace approximation to the likelihood or restricted likelihood is used to estimate dispersion parameters, this approximation including the dv/d tau term specifically discussed by Lee & Nelder 2001, p. 997 (a '0' would instead mean that these terms are ignored). It is possible to enforce the EQL approximation for estimation of dispersion parameter (i.e., Lee and Nelder's (2001) method) by adding a third index with value 0. `"EQL+"` is thus `"HL(0,1,0)"`, while `"EQL-"` is `"HL(0,0,0)"`. `"PQL"` is `EQL-` for GLMMs. `"REML"` is `"HL(1,1)"`. `"ML"` is `"ML(1,1)"`.

Some of these distinctions make sense for **GLMs**, and may help in understanding idiosyncrasies of `stats::glm` for Gamma GLMs. In particular (as stated in the `stats::logLik` documentation) the logLik of a Gamma GLM fit by `glm` differs from the exact likelihood. An `"ML(0,0,0)"` approximation of true ML provides the same log likelihood as `stats::logLik`. Further, the dispersion estimate returned by `summary.glm` differs from the one implied by `logLik`, because `summary.glm` uses Pearson residuals instead of deviance residuals. This may be confusing, and no `method` in **spaMM** tries to reproduce simultaneously these distinct features (however, `spaMM_glm` may do so). The dispersion estimate returned by an `"HL(.,.,0)"` fit matches what can be computed from residual deviance and residual degrees of freedom of a `glm` fit, but this is not the estimate displayed by `summary.glm`.

For GLMs, the fixed effect estimates are not affected by these tinkerings, but their reported standard errors may be affected by them and by a further distinction: using the observed versus the expected information matrix in some computations. This makes a difference for GLMs with non-canonical link (see Examples). Using the observed one has been the default for mixed-effect models since version 4.0.0, but has become the default for GLMs in version 4.7.0. This brings more consistency with computations of the Hessian matrix of log-likelihood by numerical derivation (as in numInfo), or with computations by automatic differentiation methods also using the observed information matrix.

References

Breslow, NE, Clayton, DG. (1993). Approximate Inference in Generalized Linear Mixed Models. Journal of the American Statistical Association 88, 9-25.

Lee, Y., Nelder, J. A. (2001) Hierarchical generalised linear models: A synthesis of generalised linear models, random-effect models and structured dispersions. Biometrika 88, 987-1006.

McCullagh, P. and Nelder, J.A. (1989) Generalized Linear Models, 2nd edition. London: Chapman & Hall.

Noh, M., and Lee, Y. (2007). REML estimation for binary data in GLMMs, J. Multivariate Anal. 98, 896-915.

Examples

```
data("wafers") # Differences more this with smaller data, so using [1:30,]:
sqrt(diag(vcov(glm(y ~X1+X2+X1*X3+X2*X3+I(X2^2),data=wafers[1:30,],
                   family=Gamma(log))))) # glm() != fitme():
sqrt(diag(vcov((spfit <- fitme(y ~X1+X2+X1*X3+X2*X3+I(X2^2),data=wafers[1:30,],
                               family=Gamma(log))))))
sqrt(diag(solve(numInfo(spfit)))[-1]) # numInfo() = fitme()
# Using method=c("ML", "exp") breaks the latter result:
sqrt(diag(vcov((spfitexp <- fitme(y ~X1+X2+X1*X3+X2*X3+I(X2^2),data=wafers[1:30,],
                               family=Gamma(log), method=c("ML","exp"))))))
sqrt(diag(solve(numInfo(spfitexp)))[-1]) # numInfo(..."exp") != fitme(... "exp")
```

MSFDR *Multiple-Stage False Discovery Rate procedure*

Description

This implements the procedure described by Benjamini and Gavrilov (2009) for model-selection **of fixed-effect terms** based on False Discovery Rate (FDR) concepts. It uses forward selection based on penalized likelihoods. The penalization for the number of parameters is distinct from that in Akaike's Information Criterion, and variable across iterations of the algorithm (but functions from the stats package for AIC-based model-selection are still called, so that some screen messages refer to AIC).

Usage

```
MSFDR(nullfit, fullfit, q = 0.05, verbose = TRUE)
```

Arguments

nullfit	An ML fit to the minimal model to start the forward selection from; an object of class HLfit.
fullfit	An ML fit to the maximal model; an object of class HLfit.
q	Nominal error rate of the underlying FDR procedure (expected proportion of incorrectly rejected null out of the rejected). Benjamini and Gavrilov (2009) recommend q=0.05 on the basis of minimizing mean-squared prediction error in various simulation conditions considering only linear models.
verbose	Whether to print information about the progress of the procedure.

Value

The fit of the final selected model; an object of class HLfit.

References

A simple forward selection procedure based on false discovery rate control. Ann. Appl. Stat, 3, 179-198 (2009).

Examples

```
if (spaMM.getOption("example_maxtime")>1.4) {
data("wafers")
nullfit <- fitme(y~1+(1|batch), data=wafers,family=Gamma(log))
fullfit <- fitme(y ~X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch), data=wafers, family=Gamma(log))
MSFDR(nullfit=nullfit,fullfit=fullfit)
}
```

---

multIMRF *Interpolated Markov Random Field models*

Description

spaMM can fit random-effect terms of the forms considered by Lindgren et al. (2011) or Nychka et al. (2015, 2018). The random effects considered here all involve a multivariate Gaussian random effect over a lattice, from which the random-effect value in any spatial position is determined by interpolation of values on the lattice. **IMRF** stands for **I**nterpolated **M**arkov **R**andom **F**ield because the specific process considered on the lattice is currently known as a Gaussian Markov Random Field (see the Details for further information). Lindgren et al. considered irregular lattices designed to approximate of the Matern correlation model with fixed smoothness <= 2, while Nychka et al. considered regular grids.

The correlation model of Lindgren et al. (2011) can be fitted by **spaMM** by declaring an IMRF random effect term in the model formula, with a model argument in the right-hand side whose value is the result of INLA::inla.spde2.matern (or INLA::inla.spde2.pcmatern) for given smoothness. The **spaMM** functions for such a fit do not call **INLA** functions. Alternatively, the same model with variable smoothness can be fitted by declaring a corrFamily term whose structure is

described through the MaternIMRFa function, whose respective documentations should be considered for more details. In the latter case INLA::inla.spde2.matern is called internally by spaMM. The correlation models thus defined are fitted by the same methods as other models in **spaMM**.

Regular lattices can also be declared as an IMRF term (with arguments distinct from model). The multIMRF syntax implements the multiresolution model of Nychka et al. Any multIMRF term in a formula is immediately converted to IMRF terms for regular grids wih different step sizes. This has distinct implications for controlling the parameters of these or other random effects in the model by init or fixed values: see Details if you need such control.

Usage

```
# IMRF( 1 | <coordinates>, model, nd, m, no, ce, ...)
# multIMRF( 1 | <coordinates>, levels, margin, coarse=10L,
#            norm=TRUE, centered=TRUE )
```

Arguments

model	An inla.spde2 object as produced by INLA::inla.spde2.matern or INLA::inla.spde2.pcmatern (see Examples below, and https://www.r-inla.org for further information).
levels	integer; Number of levels in the hierarchy, i.e. number of component IMRFs.
margin, m	integer; width of the margin, as a number of additional grid points on each side (applies to all levels of the hierarchy).
coarse	integer; number of grid points (excluding the margins) per dimension for the coarsest IMRF. The number of grids steps nearly doubles with each level of the hierarchy (see Details).
nd	integer; number of grid steps (excluding the margins) per dimension for the given IMRF.
norm, no	Boolean; whether to apply normalization (see Details), or not.
centered, ce	Boolean; whether to center the grid in all dimensions, or not.
...	Not documented, for programming purposes

Details

Formulation of the covariance models:

Gaussian Markov Random Field (MRF) and conditional autoregressive models are essentially the same thing, apart from details of specification. adjacency and AR1 random effects can be seen as specific MRFs. The common idea is the Markov-like property that the distribution of each element b_i of the random-effect **b**, given values of a few specific elements (the "neighbours" of i), is independent of other elements (i.e., of non-neighbours). The non-zero non-diagonal elements of a precision matrix characterize the neighbours.

Given the inferred vector **b** of values of the MRF on the lattice, the interpolation of the MRF in any focal point is of the form **Ab** where each row of **A** weights the values of **b** according to the position of the focal point relative to the vertices of the lattice. Following the original publications,
* for grids given by model=<inla.spde2 object>, the non-zero weights are the barycentric coordinates of the focal point in the enclosing triangle from the mesh triangulation (points from outside

the mesh would have zero weights, so the predicted effect **Ab=0**);
* for regular grids (NULL `model`), the weights are computed as <Wendland function>(<scaled Euclidean distances between focal point and vertices>).

The `IMRF` model defines both a lattice in space, the precision matrix for a Gaussian MRF over this lattice, and the **A** matrix of weights. The full specification of the MRF on **irregular lattices** is complex. The κ (`kappa`) parameter considered by `spaMM` is the κ scale parameter considered by Lindgren et al and comparable to the ρ scale factor of the Matérn model. The α argument of the `INLA::inla.spde2.matern` controls the smoothness of the approximated Matern model, as $\alpha = \nu + d/2$) where d is the dimension of the space. Correlation models created by `INLA::inla.spde2.pcmatern` are handled so as to give the same correlation values as when `INLA::inla.spde2.matern` is used with the same `mesh` and `alpha` argument (thus, the extra functionalities of "pc"`matern` are ignored).

Not all options of the INLA functions may be compatible or meaningful when used with spaMM (only the effects of `alpha` and `cutoff` have been checked).

Normalization:

For the MRFs on default **regular grids** (missing `model` argument), the precision matrix is defined (up to a variance parameter) as **M'M** where the diagonal elements m_{ii} of **M** are 4+κ^2 and the m_{ij} for the four nearest neighbours are -1 (note that **M'M** involves more than these four neighbours). The precision matrix defined in this way is the inverse of an heteroscedastic covariance matrix **C**, but (following Nychka et al.) by default a normalization is applied so that the random effect in each data position is homoscedastic (the precision matrix for the latent effect in grid positions is not modified, but it is the **A** matrix of weights which is is modified). As for other random effects, the variance is further controlled by a multiplicative factor λ.

Without normalization, the covariance matrix of the random effect in data locations is λ**ALL'A'** (**A** being the above-described weight matrix, and **L** is a "square root" of **C**), and **AL** is the original "design matrix" of the random effect. λ may then be quite different from the marginal variance of the random effect, and is difficult to describe in a simple way. For normalization, **A** is modified as **WA** where **W** is a diagonal matrix such that **WAL** is a correlation matrix (**WALL'A'W'** has unit diagonal); then, λ is the marginal variance of the random effect.

For irregular grids specified using the `model` argument, the precision matrix described by this object is also the inverse of an heteroscedastic covariance matrix, but here (again following original publicatiosn such as Lindgren at al. 2011) the normalization is not applied by default (and was not even an option before version 4.3.23). But for ease of presentation and interpretation, if for no other reason, the normalized model may be preferable.

Details for rectangular grids:

By default (meaning in particular that `model` is not used to specify a lattice defined by the INLA procedures), the IMRF lattice is rectangular (currently the only option) and is made of a core lattice, to which margins of `margin` steps are added on each side. The core lattice is defined as follows: in each of the two spatial dimensions, the range of axial coordinates is determined. The largest range is divided in `nd-1` steps, determining `nd` values and step length L. The other range is divided in steps of the same length L. If it extends over (say) $2.5L$, a grid of 2 steps and 3 values is defined, and by default centered on the range (the extreme points therefore typically extend slightly beyond the grid, within the first of the additional steps defined by the `margin`; if not centered, the grid start from the lower coordinate of the range).

`multIMRF` implements multilevel IMRFs. It defines a sequence of IMRFs, with progressively finer lattices, a common κ value `hy_kap` for these IMRFs, and a single variance parameter `hy_lam` that

determines λ values decreasing by a factor of 4 for successive IMRF terms. By default, each component IMRF is normalized independently as described above (as in Nychka et al. 2019), and hy_lam is the sum of the variances of these terms (e.g., if there are three levels and hy_lam=1, the successive variances are (1,1/4,1/16)/(21/16)). The nd of the first IMRF is set to the coarse value, and its lattice is defined accordingly. If coarse=4 and margin=5, a grid of 14 coordinates is therefore defined over the largest range. In the second IMRF, the grid spacing is halved, so that new steps are defined halfway between the previous ones (yielding a grid of 27 step in the widest range). The third IMRF proceeds from the second in the same way, and so on.

To control initial or fixed values of multIMRF κ and variance parameters, which are hyper-parameter controlling several IMRF terms, the hyper syntax shown in the Examples should be used. hyper is a nested list whose possible elements are named "1", "2", ... referring to successive multIMRF terms in the input formula, not to successive random effect in the expanded formula with distinct IMRF terms (see Examples). But the different IMRF terms should be counted as distinct random effects when controlling other parameters (e.g., for fixing the variances of other random effects).

References

D. Nychka, S. Bandyopadhyay, D. Hammerling, F. Lindgren, S. Sain (2015) A multiresolution gaussian process model for the analysis of large spatial datasets. Journal of Computational and Graphical Statistics 24 (2), 579-599. doi:10.1080/10618600.2014.914946

D. Nychka, D. Hammerling, Mitchel. Krock, A. Wiens (2018) Modeling and emulation of nonstationary Gaussian fields. Spat. Stat. 28: 21-38. doi:10.1016/j.spasta.2018.08.006

Lindgren F., Rue H., Lindström J. (2011) An explicit link between Gaussian fields and Gaussian Markov random fields: the stochastic partial differential equation approach Journal of the Royal Statistical Society: Series B (Statistical Methodology), 73: 423-498. doi:10.1111/j.1467-9868.2011.00777.x

Examples

```
if (spaMM.getOption("example_maxtime")>6) {

data("blackcap") ## toy examples; but IMRF may be useful only for much larger datasets
## and when using the 'cutoff' argument
## of fmesher::fm_mesh_2d_inla() / INLA::inla.mesh.2d()

########################### Irregular lattice specified by 'model':
#
data("small_spde") ## load object of class 'inla.spde2', created and saved by :
  # spd <- sp::SpatialPointsDataFrame(coords = blackcap[, c("longitude", "latitude")],
  #                                   data = blackcap)
  # small_mesh <- INLA::inla.mesh.2d(loc = INLA::inla.mesh.map(sp::coordinates(spd)),
  #                                  max.n=100, # only for demonstration purposes
  #                                  max.edge = c(3, 20))
  # small_spde <- INLA::inla.spde2.matern(small_mesh)
  # save(small_spde, file="small_spde.RData", version=2)
#
fit_SPDE <- fitme(migStatus ~ means + IMRF(1|longitude+latitude, model=small_spde),
                  data=blackcap)

########################### Regular lattices:
```

```
#
#Using 'hyper' to control fixed hyper-parameters
#
(mrf <- fitme(migStatus ~ 1 + (1|pos) +
                          multIMRF(1|longitude+latitude,margin=5,levels=2),
              data=blackcap, fixed=list(phi=1,lambda=c("1"=0.5),
              hyper=list("1"=list(hy_kap=0.1,hy_lam=1)))) )

# Using 'hyper' to control initial hyper-parameters
#
(mrf <- fitme(migStatus ~ 1 + multIMRF(1|longitude+latitude,margin=5,levels=2),
                data=blackcap, method="ML", fixed =list(phi=1),
                init=list(hyper=list("1"=list(hy_kap=0.1,hy_lam=1)))) )

# *Independent* IMRF terms with default rectangular lattice (often giving dubious results)
#
(mrf <- fitme(migStatus ~ 1 + IMRF(1|longitude+latitude,margin=5, nd=4L)
                            + IMRF(1|longitude+latitude,margin=5, nd=7L),
          data=blackcap,
          fixed=list(phi=1,lambda=c(1/4,1/16),
                     corrPars=list("1"=list(kappa=0.1),"2"=list(kappa=0.1)))))

}
```

`multinomial` *Analyzing multinomial data*

Description

The features described here were implemented to facilitate the analysis of multinomial data, before `fitmv` was developed, as a series of nested binomial data. `fitmv` should now be considered, as well as its wrapper `pois4mlogit` which allows one to fit models known as multinomial logit.

The main interface described here is the `multi` "family", to be used in the `family` argument of the fitting functions. Fits using it call `binomialize`, which can be called directly to check how the data are converted to nested binomial data, and to use these data directly. The `fitted.HLfitlist` method of the `fitted` generic function returns a matrix of fitted multinomial probabilities. The `logLik.HLfitlist` method of the `logLik` generic function returns a log-likelihood for the joint fits.

Usage

```
multi(binResponse=c("npos","nneg"),binfamily=binomial(),input="types",...)
binomialize(data,responses,sortedTypes=NULL,binResponse=c("npos","nneg"),
             depth=Inf,input="types")
## S3 method for class 'HLfitlist'
fitted(object, version=2L, ...)
## S3 method for class 'HLfitlist'
logLik(object,which,...)
```

Arguments

data	The data frame to be analyzed.
object	A list of binomial fits returned by a multinomial analysis
responses	column names of the data, such that <data>[,<responses>] contain the multinomial response data, as levels of factor variables.
sortedTypes	Names of multinomial types, i.e. levels of the multinomial response factors. Their order determines which types are taken first to define the nested binomial samples. By default, the most common types are considered first.
binResponse	The names to be given to the number of "success" and "failures" in the binomial response.
depth	The maximum number of nested binomial responses to be generated from the multinomial data.
binfamily	The family applied to each binomial response.
input	If input="types", then the responses columns must contain factor levels of the binomial response. If input="counts", then the responses columns must contain counts of different factor levels, and the column names are the types.
which	Which element of the APHLs list to return. The default depends on the fitting method.In particular, if it was REML or one of its variants, the function returns the log restricted likelihood (exact or approximated).
version	Integer, for fitted.HLfitlist (i.e. for multinomial fits using multi); 1 will provide the result of past versions up to 3.5.0 (See Value).
...	Other arguments passed from or to other functions.

Details

A multinomial response, say counts 17, 13, 25, 8, 3, 1 for types type1 to type6, can be represented as a series of nested binomials e.g. type1 against others (17 vs 50) then among these 50 others, type2 versus others (13 vs 37), etc. The binomialize function generates such a representation. By default the representation considers types in decreasing order of the number of positives, i.e. first type3 against others (25 vs 42), then type1 against others within these 42, etc. It stops if it has reached depth nested binomial responses. This can be modified by the sortedTypes argument, e.g. sortedTypes=c("type6","type4","type2"). binomialize returns a list of data frames which can be directly provided as a data argument for the fitting functions, with binomial response.

Alternatively, one can provide the multinomial response data frame, which will be internally converted to nested binomial data if the family argument is a call to multinomial (see Examples).

For mixed models, the multinomial data can be fitted to a model with the same correlation parameters, and either the same or different variances of random effects, for all binomial responses. Which analysis is performed depends on whether the variances are fitted by "outer optimization" or by HLfit's "inner iterative" algorithm, as controlled by the init or init.corrHLfit arguments (see Examples, and inner-vs-outer). These initial values therefore affect the definition of the model being fitted. fitmv and pois4mlogit provide better control of model definition than multi does.

corrHLfit will fit different variances by default. Adding an init.corrHLfit will force estimation of a single variance across models. fitme's default optimization strategy is more complex, and has changed and still change over versions. This creates a **back-compatibility issue** where the model to be fitted may change over versions of spaMM. To avoid that, it is strongly advised to use an explicit initial value when fitting a multi model by fitme.

Value

`binomialize` returns a list of data frames appropriate for analysis as binomial response. Each data frame contains the original one plus two columns named according to `binResponse`.

The main fitting functions, when called on a model with `family=multi(.)`, return an object of class `HLfitlist`, which is a list with attributes. The list elements are fits of the nested binomial models (objects of class `HLfit`). The attributes provide additional information about the overall multinomial model, such as global log-likelihood values and other information properly extracted by the `how()` function.

`multi` is a function that returns a list, but users may never need to manipulate this output.

`fitted.HLfitlist` returns a matrix. The current default `version=2L` provides meaningful fitted values (predicted multinomial frequencies for each response type) even for data rows where the nested binomial fit for a type had no response information remaining. By contrast, the first version provided a matrix with `0`s for these row*fit combinations, except for the last column; in many cases this may be confusing.

Examples

```
## Adding colour to the famous 'iris' dataset:
iriscol <- iris
set.seed(123) # Simulate colours, then fit colour frequencies:
iriscol$col <- sample(c("yellow", "purple", "blue"),replace = TRUE,
                      size = nrow(iriscol), prob=c(0.5,0.3,0.2))
colfit <- fitme(cbind(npos,nneg) ~ 1+(1|Species), family=multi(responses="col"),
                data=iriscol, init=list(lambda=NA)) # note warning if no 'init'...
head(fitted(colfit))

# To only generate the binomial datasets:
binomialize(iriscol,responses="col")

## An example considering pseudo-data at one diploid locus for 50 individuals
set.seed(123)
genecopy1 <- sample(4,size=50,prob=c(1/2,1/4,1/8,1/8),replace=TRUE)
genecopy2 <- sample(4,size=50,prob=c(1/2,1/4,1/8,1/8),replace=TRUE)
alleles <- c("122","124","126","128")
genotypes <- data.frame(type1=alleles[genecopy1],type2=alleles[genecopy2])
## Columns "type1","type2" each contains an allele type => input is "types" (the default)
datalist <- binomialize(genotypes,responses=c("type1","type2"))

## two equivalent fits:
f1 <- HLfit(cbind(npos,nneg)~1,data=datalist, family=binomial())
f2 <- HLfit(cbind(npos,nneg)~1,data=genotypes, family=multi(responses=c("type1","type2")))
fitted(f2)

if (spaMM.getOption("example_maxtime")>1.7) {

##### Control of lambda estimation over different binomial submodels

genoInSpace <- data.frame(type1=alleles[genecopy1],type2=alleles[genecopy2],
                          x=runif(50),y=runif(50))
method <- "PQL" # for faster exampple
```

```
## Fitting distinct variances for all binomial responses:

multifit <- corrHLfit(cbind(npos,nneg)~1+Matern(1|x+y),data=genoInSpace,
                      family=multi(responses=c("type1","type2")),
                      ranFix=list(rho=1,nu=0.5), method=method)
length(unique(unlist(lapply(multifit, get_ranPars, which="lambda")))) # 3

multifit <- fitme(cbind(npos,nneg)~1+Matern(1|x+y),data=genoInSpace,
                  family=multi(responses=c("type1","type2")),
                  init=list(lambda=NaN), # forcing 'inner' estimation for fitme
                  fixed=list(rho=1,nu=0.5), method=method)
length(unique(unlist(lapply(multifit, get_ranPars, which="lambda")))) # 3

## Fitting the same variance for all binomial responses:

multifit <- fitme(cbind(npos,nneg)~1+Matern(1|x+y),data=genoInSpace,
                  family=multi(responses=c("type1","type2")),
                  init=list(lambda=NA), # forcing 'outer' estimation for fitme
                  fixed=list(rho=1,nu=0.5), method=method)
length(unique(unlist(lapply(multifit, get_ranPars, which="lambda")))) # 1

multifit <-
  corrHLfit(cbind(npos,nneg)~1+Matern(1|x+y),data=genoInSpace,
            family=multi(responses=c("type1","type2")),
            init.corrHLfit=list(lambda=1), # forcing 'outer' estimation for corrHLfit
            ranFix=list(rho=1,nu=0.5), method=method)
length(unique(unlist(lapply(multifit, get_ranPars, which="lambda")))) # 1
}
```

---

mv *Virtual factor for multivariate responses*

Description

Motivation: In a multivariate-response model fitted by `fitmv`, one may wish to fit a random-coefficient term appearing in *s* submodels, that is a random effect with realized values for each of these submodels and each group, with values possibly correlated among submodels within groups. Hence one might wish to specify it as a term of the form `(<submodel>|group)`, where `<submodel>` would represent a factor for the *s* submodels. But the data are not expected to contain a factor for these submodels, so such a syntax would not work without substantial data reshaping. Instead, this effect can be stated as `mv(...)` where the `...` are the indices of the submodels where the random effect appears. For example if submodels 2 and 3 include this random-coefficient term, the term can be specified as `(mv(2,3)|group)`. *No* "argument name" should be attached to the indices, as any named element will be treated distinctly (cf. `lhs` argument), or ignored.

The `mv` syntax can be used further to declare *composite random effects*. As defined in `composite-ranef` and illustrated by a bivariate-response quantitative-genetic model (Examples in `Gryphon`), these effects combine correlations among response variables of different submodels as specified by `mv()`,

and correlations within each response variable(as specified by a relatedness matrix in the same example).

The `mv(...)` expression is treated as a factor for all purposes, meaning or example that `(0+mv(2,3)|group)` can also be used, leading (as for any factor) to an alternative parametrization of the same random-coefficient model (see Examples). **The parametrization through `(0+mv...)` is generally recommended as its results are easier to interpret**. The random-effect term is treated as a random-coefficient term for all purposes, meaning for example that fixed values can be specified for its parameters using the `ranCoefs` syntax (see Examples).

Usage

```
# mv(..., xpr)
```

Arguments

`...`	Indices of all the submodels (possibly more than two) where the random effect involving this virtual factor appears. *No* "argument name" should be attached to the indices.
`xpr`	(experimental) a character string which, when evaluated as an expression in the environment specified by the internal copy of the data, evaluates to a vector (one element per row of the data). `as.numeric()` is then applied to the vector, so that the resulting weights are numeric values by which the rows of the design matrix for the random effects will be pre-multiplied.

Value

Not a function, hence no return value. In the summary of the fit, levels for the different submodels *s* within each group are labelled `.mv`*s*.

See Also

`Gryphon` for example of composite random effects.

The `X2X` argument of `fitmv` for fixed effects shared across sub-models.

Examples

```
if (spaMM.getOption("example_maxtime")>1.1) {
## data preparation
data("wafers")
me <- fitme(y ~ 1+(1|batch), family=Gamma(log), data=wafers, fixed=list(lambda=0.2))

set.seed(123)
wafers$y1 <- simulate(me, type="marginal")
wafers$y2 <- simulate(me, type="marginal")

## fits
(fitmv1 <- fitmv(
  submodels=list(mod1=list(formula=y1~X1+(mv(1,2)|batch), family=Gamma(log)),
                 mod2=list(formula=y2~X1+(mv(1,2)|batch), family=Gamma(log))),
  data=wafers))
```

```
# alternative '0+' parametrization of the same model:
(fitmv2 <- fitmv(
  submodels=list(mod1=list(formula=y1~X1+(0+mv(1,2)|batch), family=Gamma(log)),
                 mod2=list(formula=y2~X1+(0+mv(1,2)|batch), family=Gamma(log))),
  data=wafers))
# relationship between the *correlated* effects of the two fits
ranef(fitmv2)[[1]][,2]-rowSums(ranef(fitmv1)[[1]]) # ~ 0

# fit with given correlation parameter:
update(fitmv2, fixed=list(ranCoefs=list("1"=c(NA,-0.5,NA))))
}
```

`negbin` *Family function for negative binomial "2" response (including truncated variant).*

Description

Returns a GLM `family` object for the negative-binomial family with variance quadratically related to the mean μ: variance=$\mu + \mu^2$/shape, which can be represented as a Poisson-Gamma mixture, where the conditional Poisson mean is μ times a Gamma random variable with mean 1 and variance `1/shape` (as produced by `rgamma(., shape=shape,scale=1/shape)`).

The zero-truncated variant can be specified as `negbin2(., trunc = 0L)`. See `negbin1` for the alternative negative-binomial model with variance "linearly" related to the mean.

A **fixed-effect** residual-dispersion model can be fitted, using the `resid.model` argument, which is used to specify the form of the logarithm of the shape parameter. Thus the variance of the response become $\mu + \mu^2$/`exp(<specified linear expression>)`.

`negbin(.)` is an alias for `negbin2(.)` (truncated or not), and `Tnegbin(.)` is an alias for `negbin2(., trunc = 0L)`.

Usage

```
negbin2(shape, link = "log", trunc = -1L, LLgeneric = TRUE)
Tnegbin(shape, link = "log")
```

Arguments

`shape`	Shape parameter. The "outer-optimizing" functions, such as `fitme` (see `inner-vs-outer`) will estimate this parameter if called with `family=negbin2()` (no given `shape` value). In other usages of this family, `shape` must be specified.
`link`	log, sqrt or identity link, specified by any of the available ways for GLM links (name, character string, one-element character vector, or object of class `link-glm` as returned by `make.link`).
`trunc`	Either `0L` for zero-truncated distribution, or `-1L` for default untruncated distribution.
`LLgeneric`	For development purposes, not documented.

Details

shape is the k parameter of McCullagh and Nelder (1989, p.373) and the theta parameter of Venables and Ripley (2002, section 7.4). The latent Gamma variable has mean 1 and variance 1/shape.

The name NB_shape should be used to set values of shape in optimization control arguments of the fitting functions (e.g., fitme(.,init=list(NB_shape=1))); but fixed values are set by the shape argument.

The returned family object is formally suitable for usage with glm if the shape argument is specified, but such usage is *not* recommended as it will lead to incorrect results for the zero-truncated case.

Value

A family object with structure similar to stats::family objects but with additional member functions for usage with **spaMM** fitting functions.

References

McCullagh, P. and Nelder, J.A. (1989) Generalized Linear Models, 2nd edition. London: Chapman & Hall.

Venables, W. N. and Ripley, B. D. (2002) Modern Applied Statistics with S-PLUS. Fourth Edition. Springer.

Examples

```
## Fitting negative binomial model with estimated scale parameter:
data("scotlip")
fitme(cases~I(prop.ag/10)+offset(log(expec)),family=negbin(), data=scotlip)
negfit <- fitme(I(1+cases)~I(prop.ag/10)+offset(log(expec)),family=Tnegbin(), data=scotlip)
simulate(negfit,nsim=3)
```

---

negbin1 *Alternative negative-binomial family*

Description

Returns a family object suitable as a fitme argument for fitting negative-binomial models with variance linearly (affinely) related to the mean μ: variance=$\mu + \mu$/shape. This distribution can be represented as a Poisson-Gamma mixture, where the conditional Poisson mean is μ times a Gamma random variable with mean 1 and variance 1/(shape*μ) as produced by rgamma(., shape=sh,scale=1/sh) where sh=shape*μ, meaning that the family shape parameter controls, but differs from, the gamma shape parameter.

The family described by this object is not a GLM family, so it should not be used as a glm argument unless one additionally uses method="llm.fit" (see Examples). But it is still characterized by the same concepts as GLMs: a linear predictor, a link function, and the given distribution of residual variation.

The zero-truncated variant of this family is also handled, and is never suitable as argument for glm.

A **fixed-effect** residual-dispersion model can be fitted, using the `resid.model` argument, which is used to specify the form of the logarithm of the shape parameter. Thus the variance of the response become $\mu + \mu$/exp(<specified linear expression>).

Usage

```
negbin1(shape, link = "log", trunc = -1L)
```

Arguments

shape	Shape parameter. The "outer-optimizing" functions, such as `fitme` (see `inner-vs-outer`) will estimate this parameter if called with `family=negbin1()` (no given shape value). In other usages of this family, shape must be specified.
link	log, sqrt or identity link, specified by any of the available ways for GLM links (name, character string, one-element character vector, or object of class `link-glm` as returned by `make.link`).
trunc	Either `0L` for zero-truncated distribution, or `-1L` for default untruncated distribution.

Details

The name `NB_shape` should be used to set values of shape in optimization control arguments of the fitting functions (e.g., `fitme(.,init=list(NB_shape=1))`); but fixed values are set by the shape argument.

Value

A list, formally of class `c("LLF", "family")`. See `LL-family` for details about the structure and usage of such objects.

See Also

Examples in `LL-family`. `resid.model` for an example with a residual-dispersion model.

Examples

```
data(scotlip)
## Shape estimated:
fitme(I(1+cases)~I(prop.ag/10)+offset(log(expec)),family=negbin1(), data=scotlip)

## Shape fixed:
fitme(I(1+cases)~I(prop.ag/10)+offset(log(expec)),family=negbin1(1), data=scotlip)

# => consistent with:
glm(I(1+cases)~I(prop.ag/10)+offset(log(expec)),family=negbin1(1),
    data=scotlip, method="llm.fit")

# !! but not with !!
# glm(I(1+cases)~I(prop.ag/10)+offset(log(expec)),family=negbin1(1),
#     data=scotlip) # WRONG !!
```

numInfo *Information matrix*

Description

Computes by numerical derivation the observed information matrix for (ideally) all parameters for mean response model, that is, the matrix of second derivatives of negative log likelihood. The default value of the `which` argument shows all classes of parameters that should be handled (except for PQL/L fits: see Details), including random-effect parameters (`lambda`, `ranCoefs`, `corrPars`, and `hyper`), residual dispersion parameters (`phi`, `NB_shape` for `negbin1` and `negbin2`, and `beta_prec` for `beta_resp` and `betabin`), and fixed-effect coefficients (`beta`).

Approximate covariance matrices of parameters estimates can be deduced from the inverse of the information matrix. Inverses of blocks of this matrix provide asymptotic covariance matrices given the values of (nuisance) parameters excluded from a given block: these covariances calculations ignore the additional variance stemming from the uncertainty in the nuisance parameter values. For example, the inverse of the block for fixed-effect coefficients should contain variances corresponding to the standard errors traditionally reported in tables of fixed effects (see Examples).

The function calls algorithms from **numDeriv** and share their limitations. Notably, they may request a re-fit of the model with some parameters fixed at values that are out of the allowed ranges. This can be controlled by the `method.args` argument, passed through the `...` to these algorithms (see Examples).

PQL/L fits raise specific problems, as do other fits by methods maximizing two different likelihood approximations for different subsets of parameters (see Details).

Model fits including a `phi-resid.model` are not fully handled, in two ways: the information matrix does not include their parameters; and if the residual dispersion model include random effects, there is good reason for the `numInfo` calculation to detect that the fit has not maximized marginal likelihood with respect to most parameters.

Usage

```
numInfo(fitobject, transf = FALSE, which = NULL, check_deriv = TRUE,
        sing=1e-05, verbose=FALSE, refit_hacks=list(), attrs=NULL,
        return.="", ...)
```

Arguments

`fitobject`	Fit object returned by a **spaMM** fitting function.
`transf`	Whether to perform internal computations on a transformed scale (but computation on transformed scale may be implemented for fewer classes of models than default computation).
`which`	NULL, or character vector giving the sets of parameters with respect to which derivatives are to be computed. The NULL default is equivalent to `c("lambda", "ranCoefs", "corrPars", "hyper", "phi", "NB_shape", "beta_prec", "beta")` for ML fits, and to the same except `"beta"` (fixed effects) for REML fits.
`check_deriv`	Boolean; whether to perform some checks for possible problems (see Details).

sing	numeric value, or FALSE; if it is a nonzero numeric value, eigenvalues of the matrix are checked and values smaller than sing are highlighted in output (see Value). This will highlight nearly-singular information matrices, but also those with large negative eigenvalues.
verbose	Boolean: whether to print (as a list) the estimates of the parameters for which the information matrix will be computed, additional information about possibly ignored parameters, possible misuse of REML fits, and a (sort of) progress bar if the procedure is expected to last more than a few seconds.
refit_hacks	list of arguments; its name anticipates that it might allow hazardous manipulations in a later version of **spaMM**. But currently only the innocuous element verbose of the list will be taken into account. Notably, refit_hacks=list(verbose=c(TRACE=TRUE)) can be used to give information on parameter values used in the computation of numerical derivatives.
attrs	NULL or vector of character strings, for names of optional attributes to be added to the return value; in particular, "df" for a data frame of the parameter points and their likelihood values used to compute the information matrix.
return.	character string. If set to "grad", the function returns the numerical gradient of the objective function instead of the numerical information matrix.
...	Arguments passed to hessian and grad, or functions with a similar interface. In particular, method.args can be used as shown in the Examples.

Details

The computation of a second derivatives is not necessarily meaningful if a first derivative does not vanish at the parameter estimates. This may occur in particular when the objective function (say, marginal likelihood) is maximized at a boundary of the parameter space (say, at zero for lambda estimates). Further, for such values at a boundary, only one-sided derivatives can be computed, and this is not handled by numDeriv::hessian. So, some checks may be requested to detect non-zero gradients and parameters estimated at their boundaries. The boundary checks are currently performed for lambda and ranCoefs estimates, if check_deriv is set to TRUE or to NULL. Other parameters are not (yet) checked, so numInfo may sometimes fails when such other parameter estimates are at a boundary. If check_deriv is set to TRUE, an additional systematic check of the gradient with respect to all target parameters is performed.

For PQL/L fits, the gradient of the log-likelihood approximation used to infer random-effect parameters (θ, say) is not zero at the fixed-effect estimates, since the fixed-effect coefficients β are then estimated by maximizing the distinct h-likelihood. Further, trying to compute the full information matrix (including fixed-effect coefficients) in this case means that fixed effects are fixed to they estimates $\hat{\beta}$ when θ is varied, while $\hat{\theta}$ maximize the likelihood approximation only when β is refitted for any given θ. Thus, the gradient for θ with β fixed is also not zero. For these reasons, the only information matrix that can be safely computed excludes the fixed effects, as in the latter case the β are refitted for any given θ.

Value

NULL or a matrix.

NULL is returned if no parameter is found with respect to which a numerical information "should" be computed (where what should be done depends on the which and check_derivs arguments).

Otherwise a matrix is returned, with an `eigvals` attribute if `sing` was non-zero. This attribute is a numeric vector of eigenvalues of the matrix. If some eigenvalue(s) were lower than `sing`, the vector has additional class `"singeigs"` so that its printing is controlled by an ad-hoc `print.singeigs` method highlighting the small eigenvalues.

Examples

```
data("wafers")
lmmfit <- fitme(y ~X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch),data=wafers)
numinfo <- numInfo(lmmfit)
covmat <- solve(numinfo[-(1:2),-(1:2)])
(cond.SEs <- sqrt(diag(covmat))) # = SEs for fixed effects in the summary()

(SEs <- sqrt(diag(solve(numinfo))))
# => SEs[-(1:2)] here, *but not generally*, equivalent to cond.SEs.
# SEs[1:2] are those of lambda and phi; they can be compared
# to the more approximate ones, for log(phi) and log(lambda), given by
# > update(lmmfit, control=list(refit=TRUE))
# These SEs translate to 1118*0.5289 ~ 591.3... for lambda and
# 10840*0.1024 ~ 1110 for phi, while numInfo() gives 735.9 and 1120, respectively.

data("blackcap")
maternfit <- fitme(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap)
numInfo(maternfit)

## Not run:
# Using 'method.args':
set.seed(123)
fx <- as.vector(na.omit(stats::filter(rnorm(400L), rep(0.025,40))))
tsl <- length(fx)
ts1 <- data.frame(y=fx+rnorm(tsl, sd=0.1),time=seq(tsl))
ts2 <- data.frame(y=fx+rnorm(tsl, sd=0.1),time=seq(tsl))
toydata <- rbind(ts1,ts2)
(fts <- fitme(y ~ 1 +AR1(1|time), data=toydata))
# numInfo(fts) # Fails as a value of AR1 correlation parameter > 1 is tried
numInfo(fts, method.args=list(d=2e-5)) # OK

## End(Not run)
```

`optimBounds` *Optimization bounds*

Description

Higher-level fitting functions (`fitme`, its derived function `fitmv` and `pois4mlogit`, and the older function `corrHLfit`), perform box-constrained numerical optimization of the likelihood objective function. The constraints can be specified in calls of these functions by the `lower` and `upper` arguments.

Specifying upper bounds for variances of random effects, and lower and upper bounds for the correlation parameters of random-coefficient covariance matrices, may be useful to prevent divergence of their estimates. Such divergence may otherwise occur notably in REML or ML fits of binary-response models, and more broadly of count-response data with low count values.

The experimental `optimBounds` function may be used to extract information from a fit object about the lower and upper bounds used in optimization.

Usage

```
optimBounds(x, transf, ...)
```

Arguments

`x`	Fit object produced by `fitme` or related higher-level fitting functions.
`transf`	boolean; whether to report transformed values of (co-)variance parameters or not. Random-coefficients parameter bounds are always returned in transformed space, for reasons given in Details.
`...`	For development purposes.

Details

Optimization of random-coefficient parameters is performed in a transformed parameter space. Box constraints on the elements of the `ranCoefs` vectors (i.e., variances and correlations), do not translate into box constraints in the transformed space, and vice versa. For this reason, simple transformation of lower or upper bounds from `ranCoefs` space to transformed space is generally inadequate, and `optimBounds` will not provide random-coefficient bounds untransformed as `ranCoefs`, even when `transf=FALSE`. It will instead keep any transformed values `trRanCoefs` in the result, with a message. Such `trRanCoefs` do not take into account optional box constraints declared by the user as `ranCoefs` elements of the `lower` or `upper` arguments of the fitting function that returned the fit object. Yet, since version 4.6.36, the fit object does satisfy such constraints correctly.

Value

A structured list, with elements `lower` and `upper` and themselves being lists.

Examples

```
# Example from VarCorr() documentation in 'nlme' package:
data("Orthodont",package = "nlme")
sp1 <- fitme(distance ~ age+(age|Subject), data = Orthodont, method="REML")
optimBounds(sp1, transf=TRUE) # showing transformed values 'trRanCoefs'

# Constrained fit (here with clear effect on first variance):
sp2 <- fitme(distance ~ age+(age|Subject), data = Orthodont, method="REML",
             upper=list(ranCoefs=list("1"=c(1,0.999,1))))
optimBounds(sp2, transf=TRUE) # showing transformed values 'trRanCoefs'

# Specifying bounds on transformed parameters is possible:
sp3 <- fitme(distance ~ age+(age|Subject), data = Orthodont, method="REML",
             upper=list(trRanCoefs=list("1"=c(1,0.999,0.0447))))
```

```
optimBounds(sp3, transf=TRUE)
```

`options` *spaMM options settings*

Description

Allow the user to set and examine a variety of *options* which affect operations of the spaMM package.

Usage

```
spaMM.options(..., warn = TRUE)

spaMM.getOption(x)
```

Arguments

`x`	a character string holding an option name.
`warn`	Boolean: whether to warn if a previously undefined options is being defined (a protection against typos).
`...`	A named value or a list of named values. The following values, with their defaults, are used in `spaMM`:

`LevenbergM=NULL`**:** NULL or boolean. Whether to use a Levenberg-Marquardt-like algorithm (see Details) by default in most computations. But it is advised to use instead `control.HLfit=list(LevenbergM=...)` to control this on a case-by-case basis. The joint default behaviour is that Levenberg-Marquardt is used by default for binomial response data that takes only extreme values (in particular, for binary 0/1 response), and that for other models the fitting algorithm switches to it if divergence is suspected. `FALSE` inhibits its use; `TRUE` forces its use for all iterative least-square fits, except when 'confint()' is called.

`example_maxtime=0.7`**:** Used in the documentation and tests to control whether the longer examples should be run. The approximate running time of given examples on one author's laptop is compared to this value.

`optimizer1D="optimize"`**:** Optimizer for one-dimensional optimization. If you want to control the initial value, you should select another optimizer.

`optimizer=".safe_opt"`**:** Optimizer for optimization in several dimensions. The default `".safe_opt"` uses `nloptr` (with default method `"NLOPT_LN_BOBYQA"`) except in some cases where it expects or detects problems with it, and it uses `bobyqa` instead (the source code should be consulted for details). Use `optimizer="nloptr"` or `optimizer="bobyqa"` to force the use of either optimizer. Use `optimizer="L-BFGS-B"` to call `optim` with method `"L-BFGS-B"`. The optimizer can also be specified on a fit-by-fit basis as the

value of control$optimizer in a fitme call, or as the value of control.corrHLfit$optimizer in a corrHLfit call.

nloptr: Default control values of nloptr calls.

bobyqa: Default control values of bobyqa calls.

allow_outer_phiGLM=TRUE: Control of fitting method for some residual-dispersion models. See phi-resid.model

maxLambda=1e10: The maximum value of lambda: higher fitted lambda values in HLfit are reduced to this. Since version 3.1.0, a much smaller lambda bound is deduced from maxLambda for COMPoisson and log-link response families.

regul_lev_lambda Numeric (default: 1e-8); lambda leverages numerically 1 are replaced by 1- regul_lev_lambda

COMP_maxn: Number of terms for truncation of infinite sums that are evaluated in the fitting of COMPoisson models.

CMP_asympto_cond: Function returning the condition for applying an approximation or the COMPoisson family, as detailed in COMPoisson.

Gamma_min_y=1e-10: A minimum response value in Gamma-response models; used to check data, and in simulate() to correct the simulation results.

QRmethod: A character string, to control whether dense matrix or sparse matrix methods are used in intensive matrix computations, overcoming the default choices made by spaMM in this respect. Possible values are "dense" and "sparse".

matrix_method: A character string, to control the factorization of dense model matrices. Default value is "def_sXaug_EigenDense_QRP_scaled". The source code should be consulted for further information.

Matrix_method: A character string, to control the factorization of sparse model matrices. Default value is "def_sXaug_Matrix_QRP_CHM_scaled". The source code should be consulted for further information.

stylefns: Default colors of some screen output (notably that of some fitting functions when called with argument verbose=c(TRACE=TRUE))

barstyle: Integer, or Boolean interpreted as Integer, or quoted expression evaluating to such types; controlling the display of some progress bars. If zero, no progress bar should be displayed; otherwise, a bar should be displayed. Further, when txtProgressBar is called, barstyle is passed as its style argument. Default is quote(if(interactive()) {3L} else {0L}) (in a parallel setting, child processes may display the bar if the parent process is interactive).

and many other undocumented values for programming or development purposes. Additional options without default values can also be used (e.g., see algebra).

Details

spaMM.options() provides an interface for changing maximal values of parameters of the Matérn correlation function. However, it is not recommended to change these values unless a spaMM message specifically suggests so.

By default spaMM use Iteratively Reweighted Least Squares (IRLS) methods to estimate fixed-effect parameters (jointly with predictions of random effects). However, a Levenberg-Marquardt algorithm, as described by Nocedal & Wright (1999, p. 266), is also implemented. The Levenberg-Marquardt algorithm is designed to optimize a single objective function with respect to all its parameters. It is thus well suited to compute a PQL fit, which is based on maximization of a single function, the h-likelihood. By contrast, in a fit of a mixed model by (RE)ML, one computes jointly fixed-effect estimates that maximizes marginal likelihood, and random-effect values that maximize h-likelihood given the fixed-effect estimates. The gradient of marginal likelihood with respect to fixed-effect coefficients does not generally vanishes at the solution (although it remains close to zero except in "difficult" cases with typically little information in the data). The Levenberg-Marquardt algorithm is not directly applicable in this case, as it may produce random-effect values that increases marginal likelihood rather than h-likelihood. The (RE)ML variant of the algorithm implemented in spaMM may therefore use additional nested h-likelihood-maximizing steps for correcting random-effect values. In version 3.1.0 this variant was revised for improved performance in difficult cases.

Value

For `spaMM.getOption`, the current value set for option `x`, or `NULL` if the option is unset.

For `spaMM.options()`, a list of all set options. For `spaMM.options(<name>)`, a list of length one containing the set value, or `NULL` if it is unset. For uses setting one or more options, a list with the previous values of the options changed (returned invisibly).

References

Jorge Nocedal and Stephen J. Wright (1999) Numerical Optimization. Springer-Verlag, New York.

See Also

Details of `algebra` for global controls of matrix methods.

Examples

```
  spaMM.options()
  spaMM.getOption("example_maxtime")
  ## Not run:
  spaMM.options(maxLambda=1e06)

## End(Not run)
```

`pedigree` *Fit mixed-effects models incorporating pedigrees*

Description

This example illustrates how to use spaMM for quantitative genetic analyses. spaMM appears competitive in terms of speed for GLMMs with large data sets, particularly when using the PQL method, which may be a quite good approximation in such cases. For large pedigrees it may be useful to compute the inverse of the relationship matrix using some efficient ad hoc algorithm, then to provide it as argument of the fit using the `covStruct(list(precision=...))` syntax. If the precision matrix is not specified, spaMM will generally evaluate it to assess whether it should use sparse-precision methods. see `sparse_precision` for further control of this computation, on another example from quantitative genetics.

See Also

`sparse_precision`

Examples

```
## Not run:
# if(requireNamespace("pedigreemm", quietly=TRUE)) {
    ## derived from help("pedigreemm")
    # p1 <- new("pedigree",
              sire = as.integer(c(NA,NA,1, 1,4,5)),
              dam  = as.integer(c(NA,NA,2,NA,3,2)),
              label = as.character(1:6))
    # A <- pedigreemm::getA(p1) ## relationship matrix
# }
## => Manually-built matrix:
A <- matrix(NA, ncol=6,nrow=6)
A[lower.tri(A,diag=TRUE)] <- c(8,0,4,4,4,2, 8,4,0,2,5, 8,2,5,4.5, 8,5,2.5, 9,5.5, 9)/8
A <- Matrix::forceSymmetric(A,uplo = "L")
colnames(A) <- rownames(A) <- 1:6

## data simulation
cholA <- chol(A)
varU <- 0.4; varE <- 0.6; rep <- 20
n <- rep*6
set.seed(108)
bStar <- rnorm(6, sd=sqrt(varU))
b <- crossprod(as.matrix(cholA),bStar)
ID <- rep(1:6, each=rep)
e0 <- rnorm(n, sd=sqrt(varE))
y <- b[ID]+e0
obs <- data.frame(y=y,IDgen=ID,IDenv=ID) ## two copies of ID for readability of GLMM results

## fits
fitme(y ~ 1+ corrMatrix(1|IDgen) , corrMatrix=A,data=obs,method="REML")
obs$y01 <- ifelse(y<1.3,0,1)
fitme(y01 ~ 1+ corrMatrix(1|IDgen)+(1|IDenv), corrMatrix=A,data=obs,
      family=binomial(), method="REML")

prec_mat <- solve(A)
colnames(prec_mat) <- rownames(prec_mat) <- rownames(A) # important
```

```
fitme(y01 ~ 1+ corrMatrix(1|IDgen)+(1|IDenv) , covStruct=list(precision=prec_mat),
      data=obs, family=binomial(), method="REML")

## End(Not run)
```

`phi-resid.model` *Residual dispersion model for gaussian and Gamma response*

Description

A model can be specified for the residual-dispersion parameter ϕ of gaussian and Gamma response families. This model may or may not include random effects. The `resid.model` argument of all fitting functions is used to specify this model and to control its fit. `resid.model` is **either** a formula (without left-hand side) for the dispersion parameter `phi` of the residual error (a log link is assumed); **or** a list of arguments similar to those of a standard `fitme` call.

The residual-dispersion model may be fitted by a specific method (see Details) involving estimation of its parameters by the fit of a Gamma-response model to response values computed by the parent fitting function (i.e., the fitting function called to fit the joint models for main response and for residual dispersion). For mixed-effect residual-dispersion models, the `fitme` function is used internally to fit this Gamma-response model (irrespective of the parent fitting function used, which may not be `fitme`).

For fixed-effect residual-dispersion models, this specific method may be slower than a more generic optimization method. By default, **spaMM** now uses the latter when it guesses it is faster and safe. However, this feature is recent and all extractor methods may not have yet been updated to handle its results. So its use can be prevented by setting `spaMM.options(allow_outer_phiGLM=FALSE)`.

Usage

```
# 'resid.model' argument of fitting functions (fitme(), HLfit(), etc)
```

Arguments

If `resid.model` is a list, it must include a `formula` element (model formula without left-hand side, as when `resid.model` is only a `formula`). The following additional arguments may be useful:

`family`	The family is always Gamma. The default link is log. The identity link can be tried but may fail because only the log link ensures that the fitted ϕ is positive.
`fixed`	fixed values of parameters of the residual dispersion model itself. Same usage as documented in `fitme`, except that it is better not to try to fix its `phi` (see Details).
`etaFix`	To fix some of the fixed-effect coefficients, as in the mean response, and with the same format. Note that the same effect can usually be acheived by an offset in the `formula`.
`control.dist`	A list of arguments that control the computation of the distance argument of the correlation functions. Same usage as documented in `HLCor`
`rand.family`	A `family` object or a `list` of family objects describing the distribution of the random effect(s). Same usage as documented for `HLfit`

`init`, `lower`, `upper`, `control`
: with same usage as documented in `fitme`. These arguments may (partly) be ignored in some cases.

Other arguments should be ignored (see Details).

Details

The following elements should not be specified in `resid.model`, for the specified reasons:

method which is constrained to be identical to the method from the parent call;

control.HLfit, control.glm constrained to be identical to the same-named controls from the parent call;

resid.model constrained: no `resid.model` for a `resid.model`;

the `phi` **of the Gamma family of the residual dispersion model** This parameter is by default set to 1, in agreement with the theory underlying the estimation procedure for the residual model; it can be set to another value, and a `resid.model`'s `fixed=list(phi=NA)` will even force its estimation, but this is not warranted;

REMLformula constrained to NULL;

data The data of the parent call are used, so they must include all the variables required for the `resid.model`;

prior.weights constrained: no prior weights;

verbose constrained: use the `verbose` argument of the fitting function instead to control a progress line summarizing the results of the `resid.model` fit at each iteration of main loop of the parent call (see the next section of the Details).

init.HLfit if used, this argument may affect the fits. However, it is best ignored in practice: users would have hard time guessing good initial values, and bad ones might have unwarranted effects.

Progress reports of the fitting procedure: Fits with a mixed-effect residual-dispersion model involve repeated "nested" fits of the latter model (each of them itself tipycally involving "double-nested" fits of a mixed-effect model with fixed random-effect parameters). This can be slow particularly when the residual-dispersion model involve spatial effects. A specific element `phifit` of the `verbose` vector controls screen information about progress of such fits during the full-model fit: when set to 0 (or FALSE) there is no report. For higher values a one-line message is output at the end of each "nested" call, but it may be overwritten by the next one-line message. So the ultimately visible output depends on control of overwriting. When `verbose["phifit"]` is set to 1 (or TRUE) each output overwrites the previous one so the ultimately visible output is from the last "nested" call; when it is set to 2, the final line of output of each "nested" call remains visible; when set to 3, a line of output remains visible from each "double-nested" call.

Methods: The present implementation of the Gamma-response procedure used to fit (fixed-effect) residual dispersion models is based on its exact components as detailed by Smyth et al. 2001. **spaMM** also implements an ML version of this REML procedure (where the leverage corrections used in the REML procedure are set to zero). Smyth et al. discuss more approximate versions of the components, considered in the early h-likelihood literature and elsewhere.

Lee and Nelder (2006) extend the REML procedure to mixed-effects residual dispersion models. Again, **spaMM** also implements ML and REML versions of these procedures (using distinct "standardizing" leverages as detailed in `hatvalues.HLfit`), and additionally use by default the full

Laplace approximation (with observed information matrix) for fitting the Gamma-response mixed-effect model, instead of the approximation using expected information matrix considered in the h-likelihood literature.

When the residual-dispersion model includes random effects, no single likelihood objective function appears to be maximized by the joint fit of mean-response and residual dispersion models. A procedure such as `numInfo` may then detect that the likelihood gradient does not vanish for all parameters. Indeed, this limitation is "relatively obvious" in original formulations relying on both marginal likelihood and restricted likelihood concepts to fit different parameters of the joint model. But this limitation is also true in the case where marginal likelihood (actually, its Laplace approximation, although the issue could persist even if exact Gamma-GLMM likelihood were used) is used in the residual-dispersion fit.

Value

When such dispersion models are fitted, the resulting fits are embedded in the main fit object. The `get_fittedPars` extractor will by default)as controlled by its argument `phiPars`) include in its return value the `rdisPars` element, which is the list of parameters of the residual-dispersion fit, in the same format as a `get_fittedPars` value for the mean-response model (`rdisPars` may also include fits of other residual-dispersion models described in `resid.model`). The `phi` element of the `get_fittedPars` value will further contain the residual-dispersion fit itself, as a `"glm"` or, when it includes random effects, as a `"HLfit"` object.

References

Lee, Y. and Nelder, J.A. (2006), Double hierarchical generalized linear models (with discussion). Journal of the Royal Statistical Society: Series C (Applied Statistics), 55: 139-185. doi:10.1111/j.14679876.2006.00538.x

Lee, Y., Nelder, J. A. and Pawitan, Y. (2006) Generalized linear models with random effects: unified analysis via h-likelihood. Chapman & Hall: London.

Smyth, G. K., Huele, F., and Verbyla, A. P. 2001. Exact and approximate REML for heteroscedastic regression, Stat. Modelling 1, 161–175.

Examples

```
data("crack") # crack data, Lee et al. 2006 chapter 11 etc
hlfit <- HLfit(y~crack0+(1|specimen), family=Gamma(log),
               data=crack, rand.family=inverse.Gamma(log),
               resid.model=list(formula=~cycle+(1|specimen)) )
```

`plot.HLfit` *Model checking plots for mixed models*

Description

This function provides diagnostic plots for residual errors from the mean model and for random effects. Plots for the mean models are similar to those for GLMs. However, "std_dev_res" ended as the type of the residuals plotted by the old version of plot.HLfit instead of the intended "std_dev_rt" (the standardized deviance residuals as described by Lee et al. 2006, p.52), by historical accident and lack of interest for the dubious results even with "std_dev_rt" (see Details). The "RQR" type may be more interesting type, for count data in particular.

Plots for random effects likewise consider standardized values.

Usage

```
## S3 method for class 'HLfit'
plot(x,
    which=c("mean","ranef"),
    res_type="std_dev_rt",
    form = residuals(., type=res_type) ~ fitted(.),
    titles = list(
    meanmodel=list(outer="Mean model",devres="residuals", absdevres="|residuals|",
                    resq="Residual quantiles", devreshist="residuals"),
    ranef=list(outer="Random effects and leverages",qq="Random effects Q-Q plot",
                levphi=expression(paste("Leverages for ",phi)),
                levlambda=expression(paste("Leverages for ",lambda)))
    ),
    control = list() , ask=TRUE, ...)
```

Arguments

x	An object of class HLfit, as returned by the fitting functions in spaMM.
which	A vector of keywords for different types of plots. By default, two types of plots are presented on different devices: diagnostic plots for mean values, and diagnostic plots for random effects. Either one can be selected using this argument. Use keyword "predict" for a plot of predicted response against actual response.
titles	A list of the main (inner and outer) titles of the plots. See the default value for the format.
control	A list of default options for the plots. Defaults are pch="+" and pcol="blue" for points, and lcol="red" for curves.
ask	Logical; passed to devAskNewPage which is run when a new device is opened by code.HLfit.
res_type	Character string: the type of residual in the plots, passed to residuals.HLfit(., type).
form	Either a formula specifying the desired type of plot for residuals, wherein the fit object x can be referenced using the symbol "."; or (more for programming purposes) directly a vector of values for the residuals.
...	Options passed from plot.HLfit to par.

Details

In principle the standardized deviance residuals for the mean model should have a nearly Gaussian distribution hence form a nearly straight line on a Q-Q plot. However this is (trivially) not so for well-specified (nearly-)binary response data nor even for well-specified Poisson response data with moderate expectations. Hence this plot is not so useful. The DHARMa package proposes better-behaved diagnostic plots, but the p-value that appears on one of these plots may not stand for a valid goodness-of-fit test; see instead the gof procedure using the randomized quantile residuals ("RQR"). The current version of DHARMa should handle spaMM fit objects; otherwise, see https://github.com/florianhartig/DHARMa/issues/95 for how to run DHARMa procedures on spaMM output.

Value

Returns the input object invisibly.

References

Lee, Y., Nelder, J. A. and Pawitan, Y. (2006). Generalized linear models with random effects: unified analysis via h-likelihood. Chapman & Hall: London.

Examples

```
data("blackcap")
fit <- fitme(migStatus ~ 1+ Matern(1|longitude+latitude),data=blackcap,
             fixed=list(lambda=1,nu=1,rho=1))
plot(fit)

# compare Q-Q plots for count data
set.seed(123)
dat <- data.frame(
  x=(x <- seq(1,2,1/100)),
  y=rpois(101, lambda=x)
)
fit <- fitme(y ~ x, family=poisson(), data=dat)
plot(fit, res_type="RQR", which="mean")
plot(fit, which="mean")
```

---

plot_effects *Partial-dependence effects and plots*

Description

The function pdep_effects evaluates, and the function plot_effects plots, *partial-dependence* effects.

pdep_effects evaluates the effect of a given fixed-effect variable, as (by default, the average of) predicted values on the response scale, over the empirical distribution of all other fixed-effect variables in the data, and of inferred random effects. This can be seen as the result of an experiment where specific treatments (given values of the focal variable) are applied over all conditions defined

by the other fixed effects and by the inferred random effects. Thus, apparent dependencies induced by associations between focal and non-variables are avoided (see Friedman, 2001, from which the name "partial dependence plot" is taken; or Hastie et al., 2009, Section 10.13.2). This also avoids biases of possible alternative ways of plotting effects. In particular, such biases occur if the response link is not identity, and if averaging is performed on the linear-predictor scale or when other variables are set to some conventional value other than its average.

Ignoring dependencies between the focal and non-focal variables, as described above, is not always appropriate. In particular, when such dependencies are inherent consequences of the data-generating process, it may make sense to perform predictions that take such dependencies into account.

`pdep_effects` also compute intervals of the type defined by its `intervals` argument (by default, "prediction intervals", but see `predVar` for the diverse interpretations of this concept) and of nominal coverage defined by the `level` argument (it may make particular sense to choose a `level`<0.95 to better visualize effects as a prediction interval can be much larger than a confidence interval for, say, the fixed-effect terms). By default, it returns a data frame of average values of point predictions and interval bounds for each value of the focal variable (so the intervals may briefly be described as mean prediction intervals, for want of better), but it can also return lists of all predictions.

A plot function is available for numeric or factor predictors: `plot_effects` calls `pdep_effects` and produces a simple plot (using only base graphic functions) of its results, including prediction bands representing the two average one-sided widths of intervals. The last section of the Examples shows how to obtain more elaborate plots including the same information using **ggplot2**.

If added to the plot, the raw data may appear to depart from the partial-dependence predictions, since the data are a priori affected by the associations between variables which the predictions free themselves from. An adapted plot of fit residuals may be then be more useful, and the Examples also show how it can be performed.

Usage

```
pdep_effects(object, focal_var, newdata = object$data, length.out = 20,
             focal_values=NULL, level=0.95, levels = NULL, submodel=NULL,
             intervals = "predVar", indiv = FALSE, verbose = NULL, ...)
plot_effects(object, focal_var, newdata = object$data, focal_values=NULL,
             effects = NULL, submodel=NULL, xlab = focal_var, ylab = NULL,
             rgb.args = col2rgb("blue"), add = FALSE, ylim = NULL, ...)
```

Arguments

`object`	An object of class `HLfit`, as returned by the fitting functions in `spaMM`.
`focal_var`	Character string: the name of the predictor variable whose effect is to be represented. The variable must be numeric for `plot_effects` but not necessarily so for `pdep_effects`.
`newdata`	If non-NULL, a data frame passed to `predict.HLfit`, whose documentation should be consulted for further details.
`effects`	If non-NULL, a data frame to substitute to the one produced by default by `pdep_effects`.
`submodel`	Integer, required for multivariate-response fits (it should remain NULL otherwise): the submodel for which predictions are computed.

xlab	If non-NULL, a character string: X-axis label for the plot.
ylab	If non-NULL, a character string: Y-axis label for the plot.
ylim	The plot's ylim argument. Default is based on the (0.025,0.975) quantiles of the response.
rgb.args	Color control arguments, in the format produced by col2rgb.
add	Boolean: whether to add graphic elements of a previous plot produced by plot_effects
length.out	Integer: for a numeric predictor variable, this controls the number of values at which predictions are evaluated. By default, predictions are made at regular intervals over the range of the predictor variable. If length.out=0, predictions are made for the actual values of the focal predictor in the data. The default behaviour is also overriden by using focal_values, in which case predictions are evaluated at the given focal_values (as if length.out=0), unless a non-zero length.out is also specified. In the latter case, predictions are evaluated at regular intervals over the range of focal_values.
intervals, level	Passed to predict.HLfit, whose documentation should be consulted for further details.
focal_values, levels	focal_values may be used to specify the values of the focal variable at which predictions are evaluated. For factor variables, levels is an older implementation of this control, and is now redundant.
indiv	Boolean: whether to return all predictions given the values of other predictors in the newdata, or only their means.
verbose	Passed to predict.HLfit, whose documentation should be consulted for further details. Element na_once is always set to TRUE within pdep_effects.
...	Further arguments passed by plot_effects to pdep_effects, or by pdep_effects to predict.HLfit.

Value

For pdep_effects, a nested list, or a data frame storing values of the focal_var, average point predictions pointp and bounds low and up of intervals, depending on the indiv argument. When indiv is TRUE, each sublist contains vectors for pointp, low and up.

For plot_effects, the same value, returned invisibly.

References

J.H. Friedman (2001). Greedy Function Approximation: A Gradient Boosting Machine. Annals of Statistics 29(5):1189-1232.

J. Friedman, T. Hastie and R. Tibshirani (2009) The Elements of Statistical Learning, 2nd ed. Springer.

Examples

```
data("scotlip")
hlcor <- HLCor(cases~I(prop.ag/10) +adjacency(1|gridcode)+offset(log(expec)),
           adjMatrix=Nmatrix,family=poisson(),data=scotlip)
plot_effects(hlcor,focal_var="prop.ag",ylim=c(0,max(scotlip$cases)))
points(cases~prop.ag, data=scotlip, col="blue",pch=20)

# Impose specific values of a numeric predictor using 'focal_values':
plot_effects(hlcor, focal_var="prop.ag", focal_values=1:5)

### Adding 'partial residuals' [residuals relative to predict(<fit object>),
###  but plotted relative to pdep_effects() predictions]:

# One first needs predictions for actual values of the predictor variable,
# provided by pdep_effects(.,length.out=0L):
#
pdep_points <- pdep_effects(hlcor,focal_var="prop.ag",length.out=0L)

# Rename for easy prediction for each observation, and add the residuals
# of the actual fit, using the default residuals() i.e. deviance ones:
#
rownames(pdep_points) <- pdep_points$focal_var
pdep_res <- pdep_points[paste(hlcor$data$prop.ag),"pointp"] +
              residuals(hlcor)

points(x = hlcor$data$prop.ag, y = pdep_res, col = "red", pch = 20)

## Not run:

## Plotting pdep-effects for different categories, using ggplot.
library(ggplot2)

data("Gryphon")
tmp <- na.omit(Gryphon_df)
spfit <- spaMM::fitme(TARSUS ~ BWT*sex, data = tmp)

tmp$sex <- "1"
pdep_1 <- pdep_effects(spfit,"BWT", newdata=tmp, level=qnorm(0.75))
#                   qnorm(0.75)  to get the so-called 'probable error'.
tmp$sex <- "2"
pdep_2 <- pdep_effects(spfit,"BWT", newdata=tmp, level=qnorm(0.75))
pdep_1$sex <- "1" ; pdep_2$sex <- "2"
pdep <- rbind(pdep_1,pdep_2)

ggplot(pdep,aes(y = pointp , x = focal_var ,col = sex, fill=sex)) + geom_point() +
  geom_ribbon(aes(ymin = low, ymax = up), alpha = 0.3) + xlab("BWT") +
  ylab("TARSUS")



## End(Not run)
```

`pois4mlogit` *Fit multinomial logit models*

Description

`pois4mlogit` is a procedure that fits a multinomial logit model by calling `fitmv` to fit multivariate `poisson(log)` surrogate models, according to the following logic. In the (mixed or not) multinomial logit model, the probabilities p_{ic} of the different categories (or `types`) $c = 1, ..., C$ for the ith multinomial draw $(n_{i1}, ..., n_{iC})$ are of the form

$$p_{ic} = \frac{e^{\eta_{ic}}}{\sum_{c=1}^{C} e^{\eta_{ic}}}$$

where each η_{ic} is a linear predictor (a sum of fixed and random effects as considered by other fitting functions, including all forms allowed by `fitmv` in particular). By contrast, the denominator makes $\log(p_{ic})$ non-linear. This type of model can can be fitted as a Poisson non-linear mixed-effect model (e.g., Chen & Kuo, 2001). **spaMM** does not have general procedures for fitting non-linear mixed-effect models, but its current procedures for GLMMs have been hacked to fit this specific model, and this includes the use of an iteratively updated offset term (see Details for more information). The correctness of the procedure can be verified for binomial data by comparison to a standard binomial GLMM fit (see Examples).

In a `pois4mlogit` call, the `formula` for each type of the multinomial response must contain a term `offset(.dynoffset)` that declares the above-defined offset term, in addition to the terms specifying each type-specific $\eta_{.c}$. The values of this offset for all i are iteratively updated between successive calls to surrogate Poisson multivariate-response GLMMs, each fitted using `fitmv`.

This procedure is **experimental**, in the sense that not all post-fit procedures may run or return meaningful results from objects returned by `pois4mlogit`, and some caveats should be kept in mind when handling the outputs (see Examples). But fits and basic post-fit inference procedures are operational and appear reliable. There is a specific `predict` method for such fits: this method returns by default the predicted frequencies, summing to 1 for each multinomial draw (see Examples). `plot_effects` and `LRT` can also handle `pois4mlogit` fits. `simulate` and bootstrap procedures based on it may be expected to work, except possibly for some esoteric options. `confint` has a method for these fits, that can be used to obtain intervals for fixed-effect coefficients, but so far it appears frustratingly slow. `numInfo` can be used to compute a numerical information matrix (at least when its argument `which` is NULL, retaining all random-effect parameters in the matrix dimensions). This computation may be distinctly useful to provide standard errors of parameters (see Details and Examples).

Since `fitmv` is called, its various specific features can be used, such as random effects correlated across the different response types (`mv` syntax), or the additional relationships between random effects that can be specified by the `aliases` argument, or shared fixed-effect coefficients across response types (`X2X` argument). Further, additional submodels not belonging to the multinomial model can also be included, but most post-fit procedures have not been checked in this case, and are likely to fail. However, `pois4mlogit` cannot fit a model including several multinomial models.

Users should quickly learn to avoid **non-identifiable models**, in particular the following two simple cases: (i) models where all submodels have their own intercept; and (ii) models where a fixed-effect term (which may be simply an intercept) has the same value in the different submodels, as occurs if its fitted coefficient is shared among submodels, and the values of its regressor are also identical among submodels.

Usage

```
pois4mlogit(submodels, data, to.long=FALSE, init=list(),
            control=list(wdfac=2, p4m="", grad=FALSE), ...,
            next_inits=c("ranPars","v_h","fixef"),
            types, n_iter = 1000L, tol=c(1e-3,1e-5),
            initfn=get_inits_from_fit, progress = FALSE)
## S3 method for class 'pois4mlogit'
predict(object, newdata=NULL, verbose=NULL,
                              na.action=na.omit, ...)

reshape2long(data, types)
```

Arguments

submodels	Passed to fitmv: see its submodels argument. This must contain as many poisson(log) submodels as there are response types in the multinomial model.
data	Data frame; each line contains a multinomial draw and, as usual, the required predictor variables. The counts for the different response types must form different columns of the data (this is convenient as, e.g., exactly the same data format can be used in binomial fits). The data are passed unchanged to fitmv, unless to.long is set to TRUE. An initial .dynoffset can be provided in the data. It will be internally overwritten after the first iteration.
init	list; passed to fitmv. Beyond initiating the first fitmv call, it controls which parameters have explicit initial values in further iterations (though the initial values themselves are then distinct from those of the first iteration).
next_inits	Character vector. Controls initiation of the next fitmv call from the result of the previous one. If "ranPars" is included, the function specified by the initfn argument is applied to this previous fit to provide initial values for ("outer-optimized") random-effect parameters in the next fitmv call. See Details for possible alternative to the *next-init default*. If "v_h" is included, the previous fitmv result is used to provide initial values for the random effects. If "fixef" is included, the previous fitmv result is used to provide initial values for the fixed effects.
to.long	Boolean; for optional reformatting of the data. Maybe no longer useful (see Details).
control	list; passed to fitmv, but may also contain extra elements intrinsic to pois4mlogit, as shown in its default value. These elements are currently for development purposes.
object, newdata, verbose, na.action	these arguments are respectively a pois4mlogit fit result, a data frame, and objects of any of the formats accepted for the verbose and na.action arguments of predict.HLfit. The first three are indeed passed to predict.HLfit after a local change of the .dynoffset in the data.
...	Further arguments passed to fitmv or to predict.HLfit.
types	Character vector: type labels, i.e., column names of the counts for the different mutlinomial response categories in data.

`n_iter`	Integer: maximum number of iteration of iterative algorithm.
`tol`	Numeric: tolerance thresholds for determining convergence. For development purposes, not documented.
`initfn`	function: a function that returns initial values. Currently for development purposes, and may become deprecated in the future.
`progress`	Boolean or numeric: whether to print information about number of iterations, and (if `progress>1L`) about each iteration. Negative values suppress convergence warnings.

Details

The fitting procedure combines (1) the iterative adjustment of GLMMs where the $\log(\sum_{c=1}^{C} e^{\eta_{ic}})$ values are represented by an offset term, modified over iterations; and (2) ad-hoc modifications of the gradient vector and information matrix of the GLMMs (the latter modifications being ignored in a first step of the fitting procedure). In mixed-effect models, the procedure iteratively updating the ad-hoc offset has to be called for each combination of random-effect parameters considered by the overall fitting procedure. At convergence, the dynamic offset should not change over iterations, and the Poisson model predictions (offset included) for each multinomial draw should sum to the sample size of the draw. Two convergence criteria, the `Ocrit` and the `Scrit` respectively, assess these properties. Non-convergence may signal a problem with the fitting procedure, *but* may also signal that the model is not identifiable, in which case it should be modified.

To fit fixed-effect models, by default only the first step of the above procedure is applied, involving iterative calls to `fitmv`, with iterative adjustments of the offset term, as this is sufficient to get correct final resutls.

An alternative fitting procedure that performs iterative adjustment of the offset *within* a surrogate Poisson fit has been implemented and may be faster in some cases (particularly with not too strict `tol`). However, preliminary tests show that convergence of iterations is not well controlled, in particular in mixed-effect models with given large variance of random effects (which is not really avoidable in practice). Fixing this convergence issue is not a top priority.

The **standard errors for fixed-effect coefficients** shown in the summary of the fit are not those for the multinomial model (see Examples), and may be quite misleading. This stems from the fact that these SEs do not take into account the effect of the fixed-effect coefficients on the likelihood through the offset. By contrast, SEs which take this effect into account can be deduced from the `numInfo` computation (see Examples, and what the `numInfo` documentation says about SEs). Further, likelihood ratio tests will provide more reliable tests of fixed effects.

Missing data are handled. This includes the detection and correct handling of cases where (i) a multinomial draw is uninformative because only one type has full information (i.e., response value and all required predictor variables); and (ii) a multinomial draw is informative about some but not all types. In that case, the basic form of the model for a multinomial draw still holds for the relative counts of these types.

When `to.long=TRUE`, the surrogate Poisson model is fitted to data specified in a long form where each multinomial draw of size s_i is described as s_i multinomial draws of size 1 (so that each Poisson submodel is fitted to a response vector of 0s and 1s). In this long format, counts for the different response types still form different columns (here containing only 0 or 1) of the `data`. The utility function `reshape2long` can also be used to convert the original data frame to the long format. This format was used by Chen & Kuo (2001), but currently it is not needed, and it is not memory-efficient.

Value

A `list` also inheriting from classes `pois4mlogit` and `HLfit`.

References

Chen, Z. and Kuo, L. (2001) A note on the estimation of the multinomial logit model with random effects. The American Statistician 55, 89-95. https://www.jstor.org/stable/2685993

See Also

`vignette("multinomial-logit", package = "spaMM")` for more examples. On an example used notably by Chen & Kuo, this vignette illustrates in particular how to use the following `fitmv` arguments, also handled by `pois4mlogit`, to describe relationships between model terms from different sub-models: `mv` and `aliases` syntaxes for random effects, and `X2X` for fixed effects.

Examples

```
#### Fitting a binomial(logit) model by a bivariate poisson(log) surrogate:
## Toy data: Let us say we observe a color polymorphism of irises in 10 populations...
set.seed(123)
ssize <- 10L
shape <- 0.35
toydata <- data.frame(
  yellow=rbinom(ssize, 16, prob=rbeta(ssize,shape,shape)),
  purple=rbinom(ssize, 16, prob=rbeta(ssize,shape,shape)), # (purple ignored below)
  blue=rbinom(ssize, 16, prob=rbeta(ssize,shape,shape)),
  phenotype=rnorm(ssize)
)
## Standard binomial fit (purple flowers are ignored here)
(byB <- fitme(cbind(yellow,blue) ~ phenotype, family = binomial(),
      data=toydata))

## Surrogate fit
(byP2 <- pois4mlogit(submodels = list(
  list(yellow ~ offset(.dynoffset) + phenotype, family = poisson()),
  list(blue ~ offset(.dynoffset) + 0, family = poisson())),
  data = toydata, types=c("yellow","blue")))

# => Note the different standard errors: the surrogate Poisson fit does not
# provide the "correct" standard errors of the binomial fit.
# However, numInfo(), from which standard errors can be deduced,
# returns correct results:
if (spaMM.getOption("example_maxtime")>0.65) {
  (infoP <- numInfo(byP2)) # practically identical to numInfo(byB)
  sqrt(diag(solve(infoP))) # SEs as in byB
}

# The 'over-parametrized model' below gives standard errors
# closer to the ones from the standard binomial fit.
```

```
# Add a trivial gaussian submodel just to show that this can be done:
(just4fun <- pois4mlogit(submodels = list(
  list(yellow ~ offset(.dynoffset) + phenotype, family = poisson()),
  list(blue ~ offset(.dynoffset) + 0, family = poisson()),
  list(phenotype ~1)),
  data = toydata, types=c("yellow","blue")))

#### Over-parametrized model:

(byP2over <- pois4mlogit(submodels = list(
  list(yellow ~ offset(.dynoffset) + phenotype, family = poisson()),
  list(blue ~ offset(.dynoffset) + phenotype, family = poisson())),
  data = toydata, types=c("yellow","blue")))

## Coefficients of binomial model recovered as:
  fixef(byP2over)[1:2]-fixef(byP2over)[3:4]

## SEs of coefficients of binomial model recovered as:
  {
    P2B <- rbind(c(1,0,-1,0),c(0,1,0,-1))
    (vcovP2B <- P2B %*% vcov(byP2over) %*% t(P2B))
  }
  # practically equivalent to
  vcov(byB)
  # and the original SEs:
  sqrt(diag(vcovP2B))

## Probabilities of each type:
  matrix(predict(byP2),ncol=2,
         dimnames=list(NULL, c("yellow","blue")))

## logLiks
# The likelihood values contained in 'pois4mlogit' objects are now those for
# the multinomial model rather than for an underlying surrogate Poisson model,
# so examples previous given here interpreting differences between these
# likelihoods are no longer relevant.

# Have a look at the 'See also' section for useful features
# not illustrated in the present examples.
```

`Poisson` *Family function for GLMs and mixed models with Poisson and zero-truncated Poisson response.*

Description

`Poisson` (with a capital P) is a `family` that specifies the information required to fit a Poisson generalized linear model. It differs from the base version `stats::poisson` only in that it handles the zero-truncated variant, which can be specified either as `Tpoisson(<link>)` or as `Poisson(<link>, trunc = 0L)`. The truncated poisson with mean μ_T is defined from the un-truncated poisson with

mean μ_U, by restricting its response to strictly positive values. $\mu_T = \mu_U/(1-p0)$, where $p0 := \exp(-\mu_U)$ is the probability that the response is 0.

Usage

```
Poisson(link = "log", trunc = -1L, LLgeneric=TRUE)
Tpoisson(link="log")
# <Poisson object>$linkfun(mu, mu_truncated = FALSE)
# <Poisson object>$linkinv(eta, mu_truncated = FALSE)
```

Arguments

`link`	log, sqrt or identity link, specified by any of the available ways for GLM links (name, character string, one-element character vector, or object of class `link-glm` as returned by `make.link`).
`trunc`	Either `0L` for zero-truncated distribution, or `-1L` for default untruncated distribution.
`eta`, `mu`	Numeric (scalar or array). The linear predictor; and the expectation of response, truncated or not depending on `mu_truncated` argument.
`mu_truncated`	Boolean. For `linkinv`, whether to return the expectation of truncated (μ_T) or un-truncated (μ_U) response. For `linkfun`, whether the `mu` argument is μ_T, or is μ_U but has μ_T as attribute (μ_U without the attribute is not sufficient).
`LLgeneric`	For development purposes, not documented.

Details

Molas & Lesaffre (2010) developed expressions for deviance residuals for the truncated Poisson distribution, which were the ones implemented in **spaMM** until version 3.12.0. Later versions implement the (non-equivalent) definition as "2*(saturated_logLik - logLik)".

`predict`, when applied on an object with a truncated-response distribution family, by default returns μ_T. The simplest way to predict μ_U is to get the linear predictor value by `predict(.,type="link")`, and deduce μ_U using `linkinv(.)` (with default argument `mu_truncated=FALSE`), since getting μ_U from μ_T is comparatively less straightforward. The `mu.eta` member function is that of the base `poisson` family, hence its `mu` argument represents μ_U.

`simulate`, when applied on an object with a truncated-response distributionfamily, simulates the truncated family. There is currently no clean way to override this (trying to pass`type="link"` to `predict` will not have the intended effect).

Value

A `family` object suitable for use with `glm`, as `stats::family` objects.

References

McCullagh, P. and Nelder, J.A. (1989) Generalized Linear Models, 2nd edition. London: Chapman & Hall.

Molas M. and Lesaffre E. (2010). Hurdle models for multilevel zero-inflated data via h-likelihood. Statistics in Medicine 29: 3294-3310.

Examples

```
data("scotlip")
logLik(glm(I(1+cases)~1,family=Tpoisson(),data=scotlip))
logLik(fitme(I(1+cases)~1+(1|id),family=Tpoisson(),fixed=list(lambda=1e-8),data=scotlip))
```

---

post-fit *Applying post-fit procedures from other packages on spaMM results*

Description

Packages implementing post-fit procedures define helper functions which may not handle **spaMM**'s fit objects, or which have not always handled them, or which can handle them correctly only with some non-default arguments. This documentation topic gives further directions to apply some such post-fit procedures (from packages **DHARMa**, **RLRsim**, **multcomp** and **lmerTest**) to these fit objects.

`emmeans` can be tentatively applied to **spaMM**'s fit objects but the underlying code is experimental. It may be safer to use `multcomp::glht`.

For other procedures not considered here, diagnosing a failure in a debugging session may suggest a simple solution (as it did for `multcomp::glht`).

Details

For multiple comparison procedures by `multcomp::glht`, one has to explicitly give the argument `coef.=fixef.HLfit` (see Examples; `fixef.HLfit` is the **spaMM** method for the generic function `fixef`);

For **DHARMa** plots, see Details of `plot.HLfit`;

For using **RLRsim::RLRTSim()**, see `get_RLRTSim_args`.

For using **lmerTest::contest()** or **lmerTest::anova()**, see `as_LMLT`.

Examples

```
if (requireNamespace("multcomp", quietly = TRUE)) {
  library(multcomp)
  set.seed(123)
  irisr <- cbind(iris,id=sample(4,replace=TRUE,size=nrow(iris)))
  irisfit <- fitme(Petal.Length~ Species +(1|id), data=irisr, family=Gamma(log))
  summary(glht(irisfit,mcp("Species" = "Tukey"), coef.=fixef.HLfit))
}
```

predict *Prediction from a model fit*

Description

The following functions can be used to compute point predictions and/or various measures of uncertainty associated to such predictions:
* `predict` can be used for prediction of the response variable by its expected value obtained as (the inverse link transformation of) the linear predictor (η) and more generally for terms of the form $\mathbf{X}_n\beta+\mathbf{Z}_n\mathbf{L}\mathbf{v}$, for new design matrices $\mathbf{X}_n$ and $\mathbf{Z}_n$.
* Various components of prediction variances and predictions intervals can also be computed using `predict`. The `get_...` functions are convenient extractors for such components;
* `get_predCov_var_fix` extracts a block of a prediction covariance matrix. It was conceived for the specific purpose of computing the spatial prediction covariances between two "new" sets of geographic locations, without computing the full covariance matrix for both the new locations and the original (fitted) locations. When one of the two sets of new locations is fixed while the other varies, some expensive computations can be performed once for all sets of new locations, and be provided as the `fix_X_ZAC.object` argument. The `preprocess_fix_corr` extractor is designed to compute this argument.

Usage

```
## S3 method for class 'HLfit'
predict(object, newdata = newX, newX = NULL, re.form = NULL,
        variances=list(), binding = FALSE, intervals = NULL,
        level = 0.95, blockSize = 2000L, type = "response", verbose=NULL,
        control=list(), na.action=na.omit, cluster_args=list(), ...)
get_predCov_var_fix(object, newdata = NULL, fix_X_ZAC.object, fixdata,
        re.form = NULL, variances=list(disp=TRUE, residVar=FALSE, cov=FALSE),
        control=list(),  ...)
preprocess_fix_corr(object, fixdata, re.form = NULL,
        variances=list(residVar=FALSE, cov=FALSE), control=list())
get_fixefVar(...)
get_predVar(..., variances=list(), which="predVar")
get_residVar(...)
get_respVar(...)
get_intervals(..., intervals="predVar")
```

Arguments

`object`	The return object of fitting functions `HLfit`,`corrHLfit`,`HLCor`... returning an object inheriting from `HLfit` class.
`newdata`	**Either** NULL, a matrix or data frame, or a numeric vector. If `NULL`, the original data are reused. Otherwise, all variables required to evaluate model formulas must be included. Which variables are required may depend on other arguments: see "prediction with given phi's" example, also illustrating the syntax when formulas include an offset.

If newdata is a numeric vector, its names (if any) are ignored. This makes it easier to use predict as an objective function for an optimization procedure such as optim, which calls the objective function on unnamed vectors. However, one must make sure that the order of elements in the vector is the order of first occurrence of the variables in the model formula. This order can be checked in the error message returned when calling predict on a newX vector of clearly wrong size, e.g. predict(<object>,newdata=numeric(0)).

newX — equivalent to newdata, available for back-compatibility

re.form — formula for random effects to include. By default, it is NULL, in which case all random effects are included. If it is NA, no random effect is included. If it is a formula, only the random effects it contains are retained. The other variance components are removed from both point prediction and variances calculations. If you want to retain only the spatial effects in the point prediction, but all variances, either use re.form and add missing variances (on linear predictor scale) manually, or ignore this argument and see Details and Examples for different ways of controlling variances.

variances — A list whose elements control the computation of different estimated variances. predict can return four components of prediction variance: fixefVar, predVar, residVar and respVar, whose definitions is detailed in predVar. They are all returned as attributes of the point predictions.

In particular, variances=list(predVar=TRUE) is suitable for uncertainty in point prediction, distinguished from the response variance given by list(respVar=TRUE). See the predVar help page for further explanations and other options.

intervals — NULL or character string or vector of strings. Provides prediction intervals with nominal level level, deduced from the given prediction variance term, e.g. intervals="predVar". Currently only intervals from fixefVar and predVar (and for LMMs respVar including the residual variance) may have a probabilistic meaning. Intervals returned in other cases are (currently) meaningless.

which — any of "predVar","respVar","residVar", "fixefVar", "intervals", or "naive".

level — Coverage of the intervals.

binding — If binding is a character string, the predicted values are bound with the newdata and the result is returned as a data frame (except for multivariate-response fits: see Details). The predicted values column name is the given binding, or a name based on it if the newdata already include a variable with this name. If binding is FALSE, The predicted values are returned as a one-column matrix and the data frame used for prediction is returned as an attribute (unless it was NULL). If binding is NA, a vector is returned, without the previous attributes.

fixdata — A data frame describing reference data whose covariances with variable newdata may be requested.

fix_X_ZAC.object — The return value of calling preprocess_fix_corr (see trivial Example). This is a more efficient way of providing information about the fixdata for repeated calls to get_predCov_var_fix with variable newdata.

blockSize — For data with many rows, it may be more efficient to perform some operations on slices of the data, and this gives the maximum number or rows of each slice.

Further, parallelisation of computations over the slices is possible, as controlled by the cluster_args argument. Slicing and parallelisation may operate only if covariance matrices are not requested.

type — character string; The returned point predictions are on the response scale if type="response" (the default; for binomial response, a frequency 0<.<1). It is on the linear predictor scale if type="link".
* The "prediction variance" (as opposed to the response variance, see predVar) that may be returned as a "predVar" attribute of the point predictions is always on the linear predictor scale, even when type="response". If you want to extract this predVar transformed to the response scale, use predict(.,variances=list(respVar=TRUE)) and take the difference between the respVar and residVar attributes of the result.
* Prediction intervals (as opposed to the response intervals) will be on the linear predictor or response scale depending on type (new to versions more recent than 3.12.0).

control — A list; a warning will direct you to relevant usage when needed.

cluster_args — Passed to makeCluster. Parallel computations are possible if the slicing mechanism (as controlled by argument blockSize) is effective.

verbose — a list or a vector; The NULL default is interpreted as c(showpbar=eval(spaMM.getOption("barstyle")), na=TRUE, na_once=FALSE). showpbar controls whether to show a progress bar (and its style) in certain prediction variance computations; na controls whether to output a message when values required for some predictions are missing from the data; and setting na_once=TRUE is useful to prevent repetitive messages when predict is called repeatedly by an higher-level function (e.g., pdep_effects sets na_once=TRUE internally).

na.action — Possible values of this argument are some of the functions dealing with NAs in data frames (see na.omit). if it is set to na.exclude, NAs will be included in the returned point predictions, for rows of the newdata which do not provide information for all required predictor variables. The effect of the default na.omit is to not include such NAs (this differs from the default of, e.g., predict.lm). Implementation is limited; in particular, na.exclude currently does not have the effect of including NAs in the optional attributes providing (co-)variance information, except the "mv" attribute for predictions of multivariate-response fits.

... — further arguments passed to or from other methods. For the get_... functions, they are passed to predict.

Details

See the predVar help page for information about the different concepts of prediction variances handled by spaMM (uncertainty of point prediction vs. of response) and about options controlling their computation.

If newdata is NULL, predict returns the fitted responses, including random effects, from the object. Otherwise it computes new predictions including random effects as far as possible. For spatial random effects it constructs a correlation matrix $\mathbf{C}$ between new locations and locations in the original fit. Then it infers the random effects in the new locations as $\mathbf{C}\ (\mathbf{L}')^{-1}\ \mathbf{v}$ (see spaMM for notation). For non-spatial random effects, it checks whether any group (i.e., level of a random

effect) in the new data was represented in the original data, and it adds the inferred random effect for this group to the prediction for individuals in this group.

In the **point prediction** of the linear predictor, the unconditional expected value of u is assigned to the realizations of u for unobserved levels of non-spatial random effects (it is zero in GLMMs but not for non-gaussian random effects), and the inferred value of u is assigned in all other cases. Corresponding values of v are then deduced. This computation yields the classical "BLUP" or empirical Bayes predictor in LMMs, but otherwise it may yield less well characterized predictors, where "unconditional" v may not be its expected value when the `rand.family` link is not identity.

There are cases where prediction without a `newdata` argument may give results of different length than prediction with `newdata=`<original data>, as for `predict`. Notably, for multivariate-response fits, different subsets of lines of the data may be used for each submodel depending on the availability of all variables (including the response variable) for each submodel, and the resulting fitted values from each submodel will be used from prediction; while prediction with `newdata` does not check the availability of a response variable.

Intervals computations use the relevant variance estimates plugged in a Gaussian approximation, except for the simple linear model where it uses Student's *t* distribution.

Binding the predicted values to a **single** data frame of predictors is not implemented for multivariate-response fits. It would not generally be feasible in this case without various side effects, because there is a data frame of predictors for each submodels, each with possibly different predictors. Instead, binding of predictions for each submodel to their respective frames is attempted. This is an **experimental** feature which may not work for all options. If this fails, the returned object when `binding=FALSE` can nevertheless be used to achieve suitable forms of binding: in particular, the frames are returned as the `"frame"` attribute (a list of data frames), and a list, over submodels, of the predictions is returned as the `"mv"` attribute.

Value

See Details in `Tpoisson` for questions specific to truncated distributions.

`predict` returns a matrix or data frame (according to the `binding` argument), with optional attributes including `frame`, `intervals`, `predVar`, `fixefVar`, `residVar`, and/or `respVar`, the last four holding one or more variance vector or covariance matrices. The further attribute `fittedName` contains the binding name, if any.

The `frame` is a data frame of predictors, and includes information about any `na.action` effect on the new data, except for predictions from multivariate response fits, where the `frame` is a list of data frames, each equivalent to the `frame` attribute for the univariate case. In the multivariate case, the additional `mv` attribute gives the predictions as a list with elements being the predictions for each submodel.

For multivariate response fits, the additional `mv` attribute gives the predictions as a list with elements for each submodel, and the `frame` is a list of data frames, each equivalent to the `frame` attribute for the univariate case.

The `get_...` extractor functions call `predict` and extract from its result the attribute implied by the name of the extractor. By default, `get_intervals` will return prediction intervals using `predVar`. `get_predVar` with non-default `which` argument has the same effect as the `get_...` function whose name is implied by `which`.

See Also

The residVar function is an alternative extractor for residual variances, with additional functionalities compared to get_residVar. For example, it can return the phi dispersion parameter, distinct from the variance in particular for fits with Gamma family.

The predVar documentation provides additional information specific to prediction variances sensu lato, including the definitions of the four components of prediction variance, fixefVar, predVar, residVar and respVar, that can be requested through the variances argument.

The get_cPredVar function returns a bootstrap-corrected version of variances returned by get_predVar.

Examples

```
data("blackcap")
fitobject <- fitme(migStatus ~ 1 + Matern(1|longitude+latitude),data=blackcap,
                       fixed=list(nu=4,rho=0.4,phi=0.05))
predict(fitobject)

#### multiple controls of prediction variances
## (1) fit with an additional random effect
grouped <- cbind(blackcap,grp=c(rep(1,7),rep(2,7)))
fitobject2 <- fitme(migStatus ~ 1 +  (1|grp) +Matern(1|longitude+latitude),
                       data=grouped,  fixed=list(nu=4,rho=0.4,phi=0.05))

## (2) re.form usage to remove a random effect from point prediction and variances:
predict(fitobject2,re.form= ~ 1 +  Matern(1|longitude+latitude))

## (3) comparison of covariance matrices for two types of new data
moregroups <- grouped[1:5,]
rownames(moregroups) <- paste0("newloc",1:5)
moregroups$grp <- rep(3,5) ## all new data belong to an unobserved third group
cov1 <- get_predVar(fitobject2,newdata=moregroups,
                    variances=list(linPred=TRUE,cov=TRUE))
moregroups$grp <- 3:7 ## all new data belong to distinct unobserved groups
cov2 <- get_predVar(fitobject2,newdata=moregroups,
                    variances=list(linPred=TRUE,cov=TRUE))
cov1-cov2 ## the expected off-diagonal covariance due to the common group in the first fit.

## Not run:
#### Other extractors:
#
fix_X_ZAC.object <- preprocess_fix_corr(fitobject,fixdata=blackcap)
#
# ... for use in multiple calls to get_predCov_var_fix():
#
get_predCov_var_fix(fitobject,newdata=blackcap[14,],fix_X_ZAC.object=fix_X_ZAC.object)

#### Prediction with distinct given phi's in different locations,
#   as specified by a resid.model:
#
varphi <- cbind(blackcap,logphi=runif(14))
vphifit <- fitme(migStatus ~ 1 + Matern(1|longitude+latitude),
                    resid.model = list(formula=~0+offset(logphi)),
```

```
                    data=varphi,  fixed=list(nu=4,rho=0.4))
#
# For respVar computation (i.e., response variance, often called prediction variance),
#   one then also needs to provide the variables used in 'resid.model', here 'logphi':
#
get_respVar(vphifit,newdata=data.frame(latitude=1,longitude=1,logphi=1))
#
# For default 'predVar' computation (i.e., uncertainty in point prediction),
#   this is not needed:
#
get_predVar(vphifit,newdata=data.frame(latitude=1,longitude=1))

#### point predictions and variances with new X and Z
#
if(requireNamespace("rsae", quietly = TRUE)) {
  data("landsat", package = "rsae")
  fitobject <- fitme(HACorn ~ PixelsCorn + PixelsSoybeans + (1|CountyName),
                     data=landsat[-33,])
  newXandZ <- unique(data.frame(PixelsCorn=landsat$MeanPixelsCorn,
                                PixelsSoybeans=landsat$MeanPixelsSoybeans,
                                CountyName=landsat$CountyName))
  predict(fitobject,newdata=newXandZ,variances = list(predVar=TRUE))
  get_predVar(fitobject,newdata=newXandZ,variances = list(predVar=TRUE))
}


## End(Not run)
```

---

predVar *Prediction and response variances*

Description

spaMM allows computation of four variance components of prediction, returned by `predict` as "...Var" attributes: `predVar`, `fixefVar`, `residVar`, or `respVar`. The phrase "prediction variance" is used inconsistently in the literature. Often it is used to denote the uncertainty in the response (therefore, including the residual variance), but **spaMM** follows some literature for mixed models in departing from this usage. Here, this uncertainly is called the response variance (`respVar`), while prediction variance (`predVar`) is used to denote the uncertainty in the linear predictor (as in Booth & Hobert, 1998; see also Jeske & Harville, 1988). The `respVar` is the `predVar` plus the residual variance `residVar`.

Which components are returned is controlled in particular by the `type` and `variances` arguments of the relevant functions. `variances` is a list of booleans whose possible elements either match the possible returned components: `predVar`, `fixefVar`, `residVar`, or `respVar`; or may additionally include `linPred`, `disp`, `cov`, `as_tcrossfac_list` and possibly other cryptic ones.

The `predict` default value for all elements is `NULL`, which jointly translate to no component being computed, equivalently to setting all elements to `FALSE`. However, setting one component to `TRUE` may reverse the default effect for other components. In particular, by default, component `predVar`

implies `linPred=TRUE`, `disp=TRUE` and component `respVar` additionally implies `residVar=TRUE`; in both cases, the `linPred=TRUE` default by default implies `fixefVar=TRUE`. Calling for one variance may imply that some of its components are not only computed but also returned as a distinct attribute.

By default the returned components are vectors of variances (with exceptions for some `type` value). To obtain covariance matrices (when applicable), set `cov=TRUE`. `as_tcrossfac_list=TRUE` can be used to return a list of matrices X_i such that the `predVar` covariance matrix equals $\sum_i X_i X_i'$. It thus provides a representation of the `predVar` that may be useful in particular when the `predVar` has large dimension, as the component X_is may require less memory (being possibly non-square or sparse).

`residVar=TRUE` evaluates `residVar` the residual variance. For families without a dispersion parameter (e.g., `binomial` or `poisson`), this is as given by the `variance` function of the `family` object (in the binomial case, it is thus the variance of a single binary draw). For families with a dispersion parameter (such as ϕ for gaussian or Gamma families, negative-binomial, beta), it is the residual variance as function of the dispersion parameter, whether this parameter is a single scalar or follows a more complex residual-dispersion model. Prior weights are however ignored (see the `residVar` extractor for the opposite feature). For the beta-binomial family, it is also the variance of a single binary draw; although this family has a residual-dispersion parameter the latter variance is not affected by it.

`fixefVar=TRUE` evaluates `fixefVar`, the variance due to uncertainty in fixed effects ($\mathbf{X}\beta$).

Computations implying `linPred=TRUE` will take into account the variances of the linear predictor η, i.e. the uncertainty in fixed effects ($\mathbf{X}\beta$) and random effects (**ZLv**), **for given dispersion parameters** (see Details). For fixed-effect models, the `fixefVar` calculations reduces to the `linPred` one.

Computations implying `disp=TRUE` additionally include the effect of uncertainty in estimates of dispersion parameters (λ and ϕ), with some limitations (see Details). `variances=list(predVar=TRUE)`, which evaluates the uncertainty of linear predictor, implies `disp=TRUE` by default, meaning that it includes such effects of uncertainty in dispersion parameters on the linear predictor. `variances=list(respVar=TRUE)` performs similarly but additionally includes the residual variance in the returned variance.

Details

`fixefVar` is the (co)variance of $\mathbf{X}\beta$, deduced from the asymptotic covariance matrix of β estimates.

`linPred` is the prediction (co)variance of η=$\mathbf{X}\beta$+**Zv** (see `HLfit` Details for notation, and keep in mind that new matrices may replace the ones from the fit object when `newdata` are used), by default computed for given dispersion parameters. It takes into account the joint uncertainty in estimation of β and prediction of **v**. In particular, for new levels of the random effects, `predVar` computation takes into account uncertainty in prediction of **v** for these new levels. For **prediction covariance** with a new **Z**, it matters whether a single or multiple new levels are used: see Examples.

For computations implying `disp=TRUE`, prediction variance may also include a term accounting for uncertainty in ϕ and λ, computed following Booth and Hobert (1998, eq. 19). This computation acheives its originally described purpose for a scalar residual variance (ϕ) and for several random effects with scalar variances (λ). This computation ignores uncertainties in spatial correlation parameters.

The (1998) formulas for the effect of uncertainty in dispersion parameters are here also applied to the variance parameters in random-coefficient terms, but with a one-time warning. Not only this is

not expected to account for the uncertainty of the correlation parameter(s) of such terms, but the result is then only heuristic as it depends on the internal representation (the "square root") of the covariance matrix, which may differ among the different fitting algorithms that may be used by **spaMM**.

`respVar` is the sum of `predVar` (pre- and post-multiplied by $\partial\mu/\partial\eta$ for models with non-identity link) and of `residVar`.

These variance calculations are approximate except for LMMs, and cannot be guaranteed to give accurate results.

References

Booth, J.G., Hobert, J.P. (1998) Standard errors of prediction in generalized linear mixed models. J. Am. Stat. Assoc. 93: 262-272.

Jeske, Daniel R. & Harville, David A. (1988) Prediction-interval procedures and (fixed-effects) confidence-interval procedures for mixed linear models. Communications in Statistics - Theory and Methods, 17: 1053-1087. doi:10.1080/03610928808829672

Examples

```
## Not run:
# (but run in help("get_predVar"))
data("blackcap")
fitobject <- fitme(migStatus ~ 1 + Matern(1|longitude+latitude),data=blackcap,
                       fixed=list(nu=4,rho=0.4,phi=0.05))

#### multiple controls of prediction variances
# (1) fit with an additional random effect
grouped <- cbind(blackcap,grp=c(rep(1,7),rep(2,7)))
fitobject <- fitme(migStatus ~ 1 +  (1|grp) +Matern(1|longitude+latitude),
                       data=grouped,  fixed=list(nu=4,rho=0.4,phi=0.05))

# (2) re.form usage to remove a random effect from point prediction and variances:
predict(fitobject,re.form= ~ 1 +  Matern(1|longitude+latitude))

# (3) comparison of covariance matrices for two types of new data
moregroups <- grouped[1:5,]
rownames(moregroups) <- paste0("newloc",1:5)
moregroups$grp <- rep(3,5) ## all new data belong to an unobserved third group
cov1 <- get_predVar(fitobject,newdata=moregroups,
                    variances=list(linPred=TRUE,cov=TRUE))
moregroups$grp <- 3:7 ## all new data belong to distinct unobserved groups
cov2 <- get_predVar(fitobject,newdata=moregroups,
                    variances=list(linPred=TRUE,cov=TRUE))
cov1-cov2 ## the expected off-diagonal covariance due to the common group in the first fit.


## End(Not run)
## see help("get_predVar") for further examples
```

pseudoR2 *Pseudo R-squared*

Description

Generalization of R-squared based on likelihood ratios, called pseudo-R2 below, and variously attributed to Cragg & Uhler (1970), Cox & Snell (1989), Magee (1990) and some other authors (see comments in the References section). The null model used in the definition of R2 can be modified by the user.

If you are looking for a goodness-of-fit test, the `gof` function may be more interesting that R2 computations.

Usage

```
pseudoR2(fitobject, nullform = . ~ 1, R2fun = LR2R2, rescale=FALSE, verbose=TRUE)
```

Arguments

`fitobject`	The fitted model object, obtained as the return value of a **spaMM** fitting function.
`nullform`	Mean-response formula for the null model. The default value (including only an intercept) represents the traditional choice in R2 computation for linear models. Alternative formulas (including, e.g., random effects) can be specified using either the `update.formula` syntax (e.g., with a `'.'` on the right hand side; note that **spaMM**'s updating conventions differ from those implemented by `stats::update.formula`, see `update.HLfit`), or a full formula (which may be a safer syntax).
`R2fun`	The backend function computing R2 given the fitted and null model. The default implements the pseudo-R2. For linear models, it reduces to the canonical R2 and the value adjusted as in `summary.lm` is also returned.
`rescale`	Boolean or formula, controlling whether and how to rescale R2 so that its maximum possible value is 1 (often considered for discrete-response models). If a formula, it should specify the model with maximal R2. If TRUE, rescaling is performed in a way meaningful only for binary logistic regression (see Examples for how this is implemented).
`verbose`	Boolean; whether to display various informations about the procedure (most notably, to warn about some potential problem in applying the default procedure to `fitobject`).

Details

None of the R2-like computations I am aware of helps in addressing, for the general class of models handled by **spaMM**, a well-defined inference task (comparable to, say, formally testing goodness of fit, or measuring accuracy of prediction of new data as considered for AIC). This problem has been well-known (e.g., Magee, 1990), and the canonical R2 itself for linear models is not devoid of weaknesses from this perspective (e.g., https://stats.stackexchange.com/

questions/13314/is-r2-useful-or-dangerous). As a consequence, strong statements about the properties that R2 should have are difficult to follow (and this includes the claim that it should always have maximum value 1).

Given the above problems, (1) ultimately the main reason for computing R2 may be to deal with requests by misguided reviewers; (2) no attempt has been made here to implement the wide diversity of R2-like descriptors discussed in the literature. The `LR2R2` backend function implements the pseudo-R2, chosen on the basis that this is the simplest general method that makes at least as much sense as any other computation I have seen; and implementation of rescaling by maximal R2 is minimal (the examples explain some of its details). `LR2R2` allows adaptation of the R2 definition for mixed-effect models, by including some random effect(s) in the null model, using the `nullform` argument.

Value

As returned by the function specified by argument `R2fun`. The default function returns a numeric vector of length 2 for linear models and a single value otherwise.

References

Cox, D.R., Snell, E.J. (1989). The analysis of binary data (2nd ed.). Chapman and Hall.
Often cited in this context, but they barely mention the topic, in an exercise p. 208-209.

Pseudo-R2 is known to go back at least to
Cragg, J. G., & Uhler, R. S. (1970). The demand for automobiles. The Canadian Journal of Economics, 3(3), 386. doi:10.2307/133656
where they already discussed its rescaling by a maximum value, in the context of binary regression.

Magee, L. (1990) R2 Measures based on Wald and likelihood ratio joint significance tests. The American Statistician, 44, 250-253. doi:10.1080/00031305.1990.10475731
also often cited for the pseudo-R2, this paper reformulates some related descriptors and concisely reviews earlier literature.

Nagelkerke, N.J.D. (1991) A note on a general definition of the coefficient of determination. Biometrika, Vol. 78, No. 3. (Sep., 1991), pp. 691-692. doi:10.1093/biomet/78.3.691
details the properties of pseudo-R2 (including the way it "partitions" variation). Argues emphatically for its rescaling, for which it is often cited.

See Also

gof

Examples

```
#### Pseudo-R2 *is* R2 for linear models:
#
# lmfit <- lm(sr ~ pop15+pop75+dpi+ddpi , data = LifeCycleSavings)
# summary(lmfit) # Multiple R-squared = 0.3385, adjusted = 0.2797
#
spfit <- fitme(sr ~ pop15+pop75+dpi+ddpi , data = LifeCycleSavings)
pseudoR2(spfit)  # consistent with summary(lmfit)

#### Toy example of pseudo-R2 for binary data
```

```
#
set.seed(123)
toydf <- data.frame(x=seq(50), y=sample(0:1,50,TRUE))
#
##   Binary logistic regression:
#
binlog <- fitme(y~x, data=toydf, family=binomial())
(blR2 <- pseudoR2(binlog)) # quite low, despite the model being correct
#
##   Rescaling by 'maximum possible' R2 for binary logistic regression:
#
pseudoR2(binlog, rescale=TRUE)
#
#    which is acheived by silently computing the maximum possible R2 value
#    by the following brutal but effective way:
#
perfbinlog <- fitme(y~I(y), data=toydf, family=binomial())
(maxblR2 <- pseudoR2(perfbinlog)) # = 0.7397...
#
# (this 'maximum possible' value would be modified if the null model were modified).
#
blR2/maxblR2       # again, rescaled value
#
##   Same by more general syntax:
#
pseudoR2(binlog, rescale=y~I(y))
```

`random-effects` *Structure of random effects*

Description

The structure of random-effect models adjustable by **spaMM** can generally be described by the following steps.

First, independent and identically distributed (iid) random effects **u** are drawn from one of the following distributions: **Gaussian** with zero mean, unit variance, and identity link; **Beta**-distributed, where $u \sim B(1/(2\lambda), 1/(2\lambda))$ with mean=1/2, and var$= \lambda/[4(1+\lambda)]$; and with logit link `v=logit(u)`; **Gamma**-distributed random effects, where $u \sim$ `Gamma(shape=1+1/`λ`,`**scale=**$1/\lambda$`)`: see `Gamma` for allowed links and further details; and **Inverse-Gamma**-distributed random effects, where $u \sim$ inverse-Gamma(`shape=1+1/`λ`,`**rate=**$1/\lambda$`)`: see `inverse.Gamma` for allowed links and further details.

Second, a transformation **v**$= f($**u**$)$ is applied (this defines **v** whose elements are still iid). By default, **v=u** for gaussian random effects but not necessarily for other distributions of random effects (see `Gamma`).

For discussion of these alternative distributions, see Lee and Nelder 2001 or Lee et al. 2006, p. 178-.

Third, correlated random effects are obtained as **Mv**, where the matrix **M** can describe spatial correlation between observed locations, block effects (or repeated observations in given locations), and possibly also correlations involving unobserved locations (as is often the case for autoregressive

models). In most cases **M** is determined from the model formula, but it can also be controlled by covStruct argument. **M** takes the form **ZL** or **ZAL**, where **Z** is determined from the model formula, the optional **A** factor is given by the optional `"AMatrices"` attribute of argument `covStruct` (see Examples), and **L** can be determined from the model formula or from `covStruct`. In particular:

* **Z** is typically, though not always, an incidence matrix: its elements z_{ij} are 1 if the ith observation is affected by the jth element of `ALb`, and zero otherwise. The specification of **Z** is controlled by the left- and right-hand side of the "bar" formula terms (`(<LHS>|<RHS>)`), as further described in `spaMM-package`.

* For spatial random effects, **L** is typically the Cholesky "square root" of a correlation matrix determined by the random effect specification (e.g., `Matern(...)`), or given by the `covStruct` argument. This may be meaningful only for Gaussian random effects. Coefficients for each level of a random-coefficient model can also be represented as **Lv** where **L** is the "square root" of a correlation matrix.

* If there is one response value par location, **L** for a spatial random effect is thus a square matrix whose dimension is the number of observations. Alternatively, several observations may be taken in the same location, and a matrix **Z** (automatically constructed) tells which element of **Lv** affects each observation. The linear predictor then contains a term of the form **ZLv**, where `dim(Z)` is (number of observations,number of locations).

* in `IMRF` random effects (IMRF for Interpolated Markov Random Fields), the realized random effects in response locations are defined as linear combinations **ALv** of random effects **Lv** in distinct locations. In that case the dimension of **L** is the number of such distinct locations, an automatically constructed **A** matrix maps them to the observed locations, and **Z** again maps them to possibly repeated observations in observed locations.

References

Lee, Y., Nelder, J. A. (2001) Hierarchical generalised linear models: A synthesis of generalised linear models, random-effect models and structured dispersions. Biometrika 88, 987-1006.

Lee, Y., Nelder, J. A. and Pawitan, Y. (2006). Generalized linear models with random effects: unified analysis via h-likelihood. Chapman & Hall: London.

Examples

```
set.seed(123)
fac <- factor(sample(letters[1:3],100,replace=TRUE))
toydata <- data.frame(fac=fac,
                      y=c(a=-1,b=2,c=0)[fac]+rnorm(100))
ranef(fitme(y~0+(1|fac), data=toydata)) # 3 values

## Merge levels "b" and "c" by (the odd way of) using an 'A' matrix:

# There is no non-trivial covariance structure, but the A matrix
# can still be provided through the 'covStruct' argument
amatrix <- matrix(c(1,0,
                0,1,
                0,1), byrow=TRUE,ncol=2)
amatrix <-as(amatrix,"CsparseMatrix")
covstruct <- structure(list(),
```

```
                        AMatrices=list("1"=amatrix))

toyfit <- fitme(y~0+(1|fac), data=toydata,
                covStruct=covstruct, method="REML")
ranef(toyfit) # 2 values
get_ZALMatrix(toyfit) # two-column incidence matrix for random effect

# See also examples with Beta(logit) random-effects in help("spaMM-package").
```

---

rankinfo *Checking the rank of the fixed-effects design matrix*

Description

By default, fitting functions in spaMM check the rank of the design matrix for fixed effects, as stats::lm or stats::glm do (but not, say, nlme::lme). This computation can be quite long. To save time when fitting different models with the same fixed-effect terms to the same data, the result of the check can be extracted from a return object by get_rankinfo(), and can be provided as argument control.HLfit$rankinfo to another fit. Alternatively, the check will not be performed if control.HLfit$rankinfo is set to NA.

Usage

```
get_rankinfo(object)
```

Arguments

object	An object of class HLfit, as returned by the fitting functions in spaMM.

Details

The check is performed by a call to qr() methods for either dense or sparse matrices. If the design matrix is singular, a set of columns from the design matrix that define a non-singular matrix is identified. Note that different sets may be identified by sparse- and dense-matrix qr methods.

Value

A list with elements rank, whichcols (a set of columns that define a non-singular matrix), and method (identifying the algorithm used).

Examples

```
## Data preparation
# Singular matrix from ?Matrix::qr :
singX <- cbind(int = 1,
           b1=rep(1:0, each=3), b2=rep(0:1, each=3),
           c1=rep(c(1,0,0), 2), c2=rep(c(0,1,0), 2), c3=rep(c(0,0,1),2))
rownames(singX) <- paste0("r", seq_len(nrow(singX)))
donn <- as.data.frame(singX)
```

```
set.seed(123)
donn$y <- runif(6)

fitlm <- fitme(y~int+ b1+b2+c1+c2+c3,data=donn)
get_rankinfo(fitlm)
```

`register_cF` *Declare corrFamily constructor for use in formula*

Description

`register_cF` registers the name of a new corrFamily constructor so that it can be used as the keyword of a random effect in a formula (as in `y ~ 1 + ARp()`). `unregister_cF` cancels this.

Usage

```
register_cF(corrFamilies = NULL, reset = FALSE)
unregister_cF(corrFamilies)
```

Arguments

`corrFamilies`	NULL, or character vector of names of corrFamily constructors.
`reset`	Boolean. Set it to `TRUE` in order to reset the list of registered constructors to the **spaMM** built-in default, before registering the ones specified by `corrFamilies`.

Value

No value; operates through side-effects on internal variables.

Examples

```
ts <- data.frame(lh=lh,time=seq(48)) ## using 'lh' data from 'stats' package

myARp <- ARp                  #  defines 'new' corrFamily from built-in one

# Now, this would not yet work:

# fitme(lh ~ 1 + myARp(1|time), data=ts, method="REML")

# but this works if we first register "myARp"

register_cF("myARp")          #  registers it

fitme(lh ~ 1 + myARp(1|time), data=ts, method="REML")
#
# same as
#
fitme(lh ~ 1 + corrFamily(1|time), data=ts, method="REML",
```

```
                  covStruct=list(corrFamily=myARp()))
#
# showing it's possible not to register myARp,
# although this has limitations (see Details in help("corrFamily")).

## Specifying arguments of the corrFamily constructor:

fitme(lh ~ 1 + myARp(1|time, p=3), data=ts, method="REML")
#
# same as
#
fitme(lh ~ 1 + corrFamily(1|time), data=ts, method="REML",
                  covStruct=list(corrFamily=ARp(p=3)))

unregister_cF("myARp") # Tidy things before leaving.
```

`remove_fixef` *Remove specified fixed-effect terms from a model formula*

Description

As the Title says.

Usage

```
remove_fixef(formula, remove, nofixef = NULL, keep_intercept = TRUE, keep_offset = TRUE)
```

Arguments

`formula`	The model formula from which terms should be removed.
`remove`	The fixed-effect terms to be removed, specified either as a formula from which fixed-effect terms will be deduced (any random effet in such a formula will be ignored), or as a vector of character strings. The variable names should match those in the `formula` argument (not to be confused with the labels of fitted coefficients of fixed effect, when these differ, as for factor terms). Permutation of variables in interaction terms are handled (see Examples).
`nofixef`	Optional `expression` or formula allowing control of terms included in the returned formula, beyond fixed effects not first removed according to the `remove` argument (see Details). By default, these additional terms are all the offsets and random effects from the original formula. See Details for non-default usage.
`keep_intercept`	Boolean: whether to keep an original intercept in the returned formula (does not add one if not originally present).
`keep_offset`	Boolean: whether to keep the offsets in the returned formula.

Details

The following toy example explains the results with a non-default `nofixef`:

```
form <- ~ offset(o) + a*b + (1|grp)
remove_fixef(form, "a:b", nofixef=form)
```

which gives `~1 + a + b + offset(o) + a * b + (1 | grp)`.

First, fixed effect terms are extracted from the `formula`. Next, the terms implied by `remove` are removed, yielding an expression containing only other fixed-effect terms (here, `1 + a + b`). Then, this expression is joined to the terms included in `nofixef`. The `nofixef` argument by default contains only the offset and random effect terms from the original formula (hence its name), but this example shows that anything in `nofixef`, including fixed effects, is included in the result.

Value

A model formula. The response variable is retained in it if it was present in the input formula.

Examples

```
remove_fixef( y ~ a*b, ~ a*b) #  y ~ 1, different from:
# remove_fixef(y ~ a*b, "a*b") # Warning and nothing removed
# (because a*b not in terms(formula), which instead contains a, b and a:b)

remove_fixef( y ~ a*b, ~ a*b) # => y ~ 1, same result as:
remove_fixef( y ~ a*b, ~ b*a)

remove_fixef( y ~ a*b, "a:b") # Same result as:
remove_fixef( y ~ a*b, "b:a") # a:b removed

# Remove all fixed effects term by setting 'remove' equal to 'formula':
form <- ~ offset() + a*b + (1|grp)
remove_fixef(form, form)
remove_fixef(form, form, keep_intercept = FALSE, keep_offset = FALSE)
```

`resid.model` *Structured dispersion models*

Description

The `resid.model` argument of fitting functions can be used to specify a model for a residual-dispersion parameter of various response families, that is, either
(1) the ϕ parameter of the gaussian and Gamma GLM families;
(2a) the dispersion parameter of some other GLM families, such as the shape parameter of the negbin1 and negbin2 families; or
(2b) the dispersion parameter of some other (non-GLM) distribution families, such as the precision parameter of the beta family.

This documentation is more specifically for case (2). Case (1) is more specifically documented as phi-resid.model.

In case (2) the model for the dispersion parameter is constrained as a fixed-effect model, of the form dispersion parameter = exp(**X**β+offset),
and specified using the standard formula syntax. Random effects cannot be included, in contrast to dispersion models for case (1).

Usage

```
# 'resid.model' argument of fitme() and fitmv()
```

Arguments

The resid.model for case (2) is simply a formula (without left-hand side) for the logarithm of the dispersion parameter. Fixed β values can be specified through the rdisPars element of the fixed argument in the fitme call (or through the fixed argument of each submodel of a fitmv call). Likewise, initial values can be specified through the init argument.

Details

In case (2) a fixed "heteroscedastic" model can also be specified directly through the family specification, e.g., family=negbin1(shape=<vector>) where the vector has the length of the response vector, but this may not be suitable if the model is to be used for prediction purposes (where the residual-dispersion model should be specified in such a way that one can "predict" new dispersion values from it).

The design matrix for the specified model is internally rescaled to avoid numerical problems. That means that there is no need to rescale the predictor variable, even if it tends to take large (cf 'population' variable in the Examples) of small values (this is also true for fixed-effect predictors of the mean-response model).

Value

The fit results for the residual model are accessible through the summary and various extractors. In particular, the get_fittedPars extractor will by default include in its return value the rdisPars element, which is here the vector of fitted β coefficients. residVar(., which="fam_parm") will return the vector of fitted values of the dispersion parameter.

Examples

```
data("scotlip")
if (spaMM.getOption("example_maxtime")>3) {
(toyfit <- fitme(cases~1+(1|id),family=negbin1(), data=scotlip,
                 resid.model = ~ population))
}

# => This toy example is a bit challenging to fit because the data set is small and
# individual-level variation is here described both by a random effect
# and by a two-parameter negbin1 residual variation. Such fits might often stop
# at a local maximum of the logLik (although there is no evidence
# that this is presently the case).
```

`residuals.HLfit` *Extract model residuals*

Description

Extracts several types of residuals from an object of class `HLfit`. Note that the default type (`"deviance"`) of returned residuals differs from the default of equivalent functions in base R.

Usage

```
## S3 method for class 'HLfit'
residuals(object,
  type = c("deviance", "pearson", "response", "working",
           "RQR", "std_RQR", "std_dev_res", "std_dev_rt"),
  force=FALSE, ...)
```

Arguments

`object`	An object of class `HLfit`, as returned by the fitting functions in `spaMM`.
`type`	The type of residuals which should be returned. See Details for additional information.
`force`	Boolean: to force recomputation of the `"std_dev_res"` residuals even if they are available in the object, for checking purposes.
`...`	For consistency with the generic.

Details

The first four types `"deviance"` (default), `"pearson"`, `"response"` are `"working"` are, for GLM families, the same that are returned by `residuals.glm`. `"working"` residuals may be returned only for fixed-effect models.

The default type is `"deviance"` as for `residuals.glm`, but since this may be confusing to some users, a gentle one-time warning will be issued when the `object` is a GLMM.

GLMs may be formulated as exponential dispersion models, i.e., models where the probability or probability density of each observation y can be written in the form $f(y) = b(y,\phi)\exp[-d(y,\mu)/(2\phi)]$ where $d(y,\mu)$ is the *unit deviance* for the observation, independent of the dispersion parameter ϕ (e.g., Dunn & Smyth, 2018). The sum over observations, $\sum_i d(y_i,\mu_i)$ is called the *deviance*, and $\sum_i d(y_i,\mu_i)/\phi$ is called the *scaled deviance*, by McCullagh & Nelder (1989, pp. 33-34). In the **spaMM** documentation the former deviance is called the *unscaled* deviance to emphasize this distinction.

Prior weights w may be included in "unscaled" quantities even though ϕ is not. This is consistent with what base R does: unscaled quantities returned by `stats::residuals(<glm>)` and `stats::deviance(<glm>)` will be different for equivalent fits with different prior weights (as produced by `glm(., family=gaussian, weights=rep(2,nrow<data>))` versus the `glm` call without weights).

`residuals(., type="deviance")` returns the vector of deviance residuals for the different observations (as `residuals.glm`) does. The deviance residual for an observation is the signed square root of the (prior-weighted) unit deviance for this observation, $\text{sign}(y-\mu)\sqrt{wd(y,\mu)}$.

The extractor `dev_resids` is equivalent to the `family` object's `dev.resids` function. Hence, it does not really returns residuals but rather their squares, i.e. the (prior-weighted) unit deviances, because this is already so for families defined in base R.

The `deviance` extractor returns the sum of these prior-weighted unit deviances, i.e., the unscaled, prior-weighted deviance. This is consistent with what `stats::deviance(<glm>)` does.

`residuals(., type="std_dev_rt")` returns the (prior-weighted) "standardized deviance residuals" as defined by Lee et al. (2006, p.52), i.e. the deviance residuals divided by $\sqrt{\phi(1-q)}$, where ϕ is the dispersion parameter of the distribution family (a vector of values, for heteroscedastic cases), and q is a vector of leverages given by `hatvalues(., type="std")` (see `hatvalues` for details about these specific standardizing leverages). They are thus scaled by ϕ, and signed.

`residuals(.type="std_dev_res")` returns the signed **squares** of such standardized residuals.

Some definitions must be extended for non-GLM response families. as in the latter case, there may not be a concept of unit deviance, and there is no conceptual distinction between scaled and unscaled residuals since the residual dispersion parameter (if any) is not generally a scale factor ϕ. This amounts to the convention that $\phi = 1$. The deviance residuals for non-GLMs are as defined in Details of `llm.fit`. They are analogous to the scaled ones for GLMs as they depend on any dispersion parameter. `"std_dev_res"` and `"std_dev_rt"` residuals are defined from them as detailed above for GLM response families, and from the additional convention that $\phi = 1$. Pearson residuals and response residuals are defined as in `stats:::residuals.glm`. The `"working"` residuals are defined for each observation as $-[d\log(\text{clik})/d\eta]/[d^2\log(\text{clik})/d\eta^2]$ where 'clik' is the conditional likelihood.

`residuals(.type="RQR")` returns the randomized quantile residuals (Dunn & Smyth, 1996), used by `gof`. `type="std_RQR"` returns the RQR residuals divided by $\sqrt{1-q}$, following the same logic as for standardized deviance residuals.

Value

A vector of residuals

References

Dunn, K. P., and Smyth, G. K. (1996). Randomized quantile residuals. Journal of Computational and Graphical Statistics 5, 1-10.

Dunn, P. K., & Smyth, G. K. (2018). Generalized Linear Models With Examples in R. Springer.

Lee, Y., Nelder, J. A. and Pawitan, Y. (2006). Generalized linear models with random effects: unified analysis via h-likelihood. Chapman & Hall: London.

McCullagh, P. and Nelder J. A. (1989) Generalized linear models. Second ed. Chapman & Hall: London.

Examples

```
data("wafers")
fit <- fitme(y ~X1+(1|batch) ,data=wafers, init=list(phi=NaN))  # : this 'init'
```

```
#                 implies that standardized deviance residuals are saved in the
#                 fit result, allowing the following comparison:

r1 <- residuals(fit, type="std_dev_res") # gets stored value
r2 <- residuals(fit, type="std_dev_res", force=TRUE) # forced recomputation
if (diff(range(r1-r2))>1e-14) stop()

#####
## Not run:
glmfit <- glm(I(y/1000)~X1, family=gaussian(), data=wafers)
deviance(glmfit) #           3...  (a)
sum(residuals(glmfit)^2) #   3...  (b)

# Same model, with different parametrization of residual variance
glmfit2 <- glm(I(y/1000)~X1, family=gaussian(), data=wafers, weights=rep(2,198))
deviance(glmfit2) #          6...  (c)
sum(residuals(glmfit2)^2) #  6...  (d)

# Same comparison but for HLfit objects:
spfit <- fitme(I(y/1000)~X1, family=gaussian(), data=wafers)
deviance(spfit) #            3...  (e)
sum(residuals(spfit)^2) #    3...  (f) ~ sum(abs(residuals(.,"std_dev_res")))*phi
sum(dev_resids(spfit)) #     3...
# Gaussian case: "RQR" residuals ~ ("deviance" residuals)/sqrt(phi)

spfit2 <- fitme(I(y/1000)~X1, family=gaussian(), data=wafers, prior.weights=rep(2,198))
deviance(spfit2) #           6...  (g) ~ (c,d) # post v4.2.0
sum(residuals(spfit2)^2) #   6...  (h) ~ (c,d)
                                       ~ sum(abs(residuals(.,"std_dev_res")))*phi/pw
sum(dev_resids(spfit2)) #    3...
# "RQR" residuals still ~ ("deviance" residuals)/sqrt(phi)

# Unscaled residuals should not depend on arbitrarily fixed residual variance:
spfit3 <- fitme(I(y/1000)~X1, family=gaussian(), data=wafers, fixed=list(phi=2),
                prior.weights=rep(2,198))
deviance(spfit3) #           6...  (i) ~ (g)
sum(residuals(spfit3)^2) #   6...  (k) ~ (h)
sum(dev_resids(spfit3)) #    3...


## End(Not run)
```

`residVar` *Residual variance extractor*

Description

`residVar` extracts from a fit object the residual variance or, depending on the `which` argument, a family dispersion parameter phi (which is generally not the residual variance itself except for gaussian-response models without prior weights), or a vector of values of the dispersion parameter, or further information about the residual variance model.

For gaussian and Gamma response families, the return values for `which = "var"` and `"phi"` include prior weights, if any.

Given that base R defines the generic function `sigma` to extract the estimated residual standard deviation, a `sigma.HLfit` method for fit objects produced by spaMM fitting functions is defined, effectively as
`sqrt(residVar(object, which="fit", ...))` when this is numeric.

Usage

```
residVar(object, which = "var", submodel = NULL, newdata = NULL)
## S3 method for class 'HLfit'
sigma(object, ...)
```

Arguments

`object`	An object of class `HLfit`, as returned by the fitting functions in `spaMM`.
`which`	Character: `"var"` for the fitted residual variances, `"phi"` for the fitted phi values, `"fam_parm"` for the dispersion parameter of `COMPoisson`, `negbin1`, `negbin2`, `beta_resp` or `betabin` families, `"fit"` for the fitted residual model (a GLM or a mixed model for residual variances, if not a simpler object), and `"family"` or `"formula"` for such properties of the residual model.
`submodel`	integer: the index of a submodel, if `object` is a multivariate-response model fitted by `fitmv`. This argument is mandatory for all `which` values except `"var"` and `"phi"`.
`newdata`	**Either** NULL, a matrix or data frame, or a numeric vector. See `predict.HLfit` for details.
`...`	Further arguments passed to `residVar`: for example, `submodel` may be needed for mutlivariate-response fits.

Value

`which="var"` (default) and `"phi"` always return a vector of residual variances (or, alternatively, phi values) of length determined by the `newdata` and `submodel` arguments.
`which="fit"` returns an object of class `HLfit`, `glm`, or a single scalar depending on the residual dispersion model (`which="fit"` is the option to be used to extract the scalar phi value).
`which="fam_parm"` returns either NULL (for families without such a parameter), a vector (if a `resid.model` was specified for relevant families), a single scalar (relevant families, without `resid.model`), or a list of such objects (for multivariate-response models).
Other `which` values return an object of class `family` or `formula` as expected.

See Also

`get_residVar` is a alternative extractor of residual variances with different features inherited from `get_predVar`. In particular, it is more suited for computing the residual variances of new realizations of a fitted model, not accounting for prior weights used in fitting the model (basic examples of using the **IsoriX** package provide a context where this is the appropriate design decision). By contrast, `residVar` aims to account for prior weights.

Examples

```
## residVar optional arguments:

# data preparation: simulated trivial life-history data
set.seed(123)
nind <- 20L
u <- rnorm(nind)
lfh <- data.frame(
  id=seq_len(nind), id2=seq_len(nind),
  feco= rpois(nind, lambda = exp(1+u)),
  growth=rgamma(nind,shape=1/0.2, scale=0.2*exp(1+u)) # mean=exp(1+u), var= 0.2*mean^2
)
# multivariate-response fit
fitlfh <- fitmv(submodels=list(list(feco ~ 1+(1|id), family=poisson()),
                               list(growth ~ 1+(1|id), family=Gamma(log))),
                data=lfh)
#
residVar(fitlfh)
residVar(fitlfh, which="phi") # shows fixed phi=1 for Poisson responses
residVar(fitlfh, submodel=2)
residVar(fitlfh, which="family", submodel=2)
residVar(fitlfh, which="formula", submodel=2)
residVar(fitlfh, which="fit", submodel=2) # Fit here characterized by a single scalar

## Prior weights in residVar() vs. get_residVar():
  data(wafers)
  spfit <- fitme(I(y/1000)~X1, family=gaussian(), data=wafers)
  residVar(spfit)[1] #      0.015...
  get_residVar(spfit)[1] #  0.015...

  spfit2 <- fitme(I(y/1000)~X1, family=gaussian(), data=wafers, prior.weights=rep(2,198))
  head(residVar(spfit2)) #     0.015... = phi/prior.weights
  head(get_residVar(spfit2)) # 0.030... = phi

  spfit3 <- fitme(I(y/1000)~X1, family=gaussian(), data=wafers, fixed=list(phi=2),
                  prior.weights=rep(2,198))
  residVar(spfit3)[1] #     1 = phi/prior.weights
  get_residVar(spfit3)[1] # 2 = phi
```

`salamander` *Salamander mating data*

Description

Data from a salamander mating experiment discussed by McCullagh and Nelder (1989, Ch. 14). Twenty males and twenty females from two populations (Rough Butt and Whiteside) were each paired with 6 individuals from their own or from the other population. The experiments were later published by Arnold et al. (1996).

Usage

```
data("salamander")
```

Format

The data frame includes 360 observations on the following variables:

Female Index of the female;

Male Index of the male;

Mate Whether the pair successfully mated or not;

TypeF Population of origin of female;

TypeM Population of origin of male;

Cross Interaction term between `TypeF` and `TypeM`;

Season A factor with levels `Summer` and `Fall`;

Experiment Index of experiment

Source

The data frame was borrowed from the `HGLMMM` package (Molas and Lesaffre, 2011), version 0.1.2.

References

Arnold, S.J., Verrell, P.A., and Tilley S.G. (1996) The evolution of asymmetry in sexual isolation: a model and a test case. Evolution 50, 1024-1033.

McCullagh, P. and Nelder, J.A. (1989). Generalized Linear Models, 2nd edition. London: Chapman & Hall.

Molas, M., Lesaffre, E. (2011) Hierarchical Generalized Linear Models: The R Package HGLMMM. Journal of Statistical Software 39, 1-20.

Examples

```
data("salamander")

## Not run:
HLfit(cbind(Mate,1-Mate)~TypeF+TypeM+TypeF*TypeM+(1|Female)+(1|Male),
      family=binomial(),data=salamander,method="ML")
# equivalent fo using fitme(), but here a bit faster

## End(Not run)
```

scotlip *Lip cancer in Scotland 1975 - 1980*

Description

This data set provides counts of lip cancer diagnoses made in Scottish districts from 1975 to 1980, and additional information relative to these data from Clayton and Kaldor (1987) and Breslow and Clayton (1993). The data set contains (for each district) counts of disease events and estimates of the fraction of the population involved in outdoor industry (agriculture, fishing, and forestry) which exposes it to sunlight.

`data("scotlip")` actually loads a data frame, `scotlip`, and an adjacency matrix, `Nmatrix`, between 56 Scottish districts, as given by Clayton and Kaldor (1987, Table 1).

Usage

```
data("scotlip")
```

Format

The data frame includes 56 observations on the following 7 variables:

gridcode alternative district identifier.

id numeric district identifier (1 to 56).

district district name.

cases number of lip cancer cases diagnosed 1975 - 1980.

population total person years at risk 1975 - 1980.

prop.ag percent of the population engaged in outdoor industry.

expec offsets considered by Breslow and Clayton (1993, Table 6, 'Exp' variable)

The rows are ordered according to `gridcode`, so that they match the rows of `Nmatrix`.

References

Clayton D, Kaldor J (1987). Empirical Bayes estimates of age-standardized relative risks for use in disease mapping. Biometrics, 43: 671 - 681.

Breslow, NE, Clayton, DG. (1993). Approximate Inference in Generalized Linear Mixed Models. Journal of the American Statistical Association: 88 9-25.

Examples

```
data("scotlip")
fitme(cases~I(log(expec)), data=scotlip, family=poisson)

## see 'help(autoregressive)' for additional examples involving 'scotlip'.
```

seaMask *Masks of seas or lands*

Description

These convenient masks can be added to maps of (parts of) the world to mask map information for these areas.

However, many other tools may be available since this documentation was conceived. See e.g. the **rnaturalearth** package, used to provide a sea mask in an example for `filled.mapMM`

Usage

```
data("seaMask")
data("landMask")
# data("worldcountries") # deprecated and removed
# data("oceanmask") # deprecated and removed
```

Format

`seaMask` and `landMask` are data frames with two variables, `x` and `y` for longitude and latitude. Its contents are suitable for use with `polypath`: they define different polygones, each separated by a row of `NA`s.

`worldcountries` and `oceanmask` were `sp::SpatialPolygonsDataFrame` objects previously included in spaMM (see Details for replacement). Such objects were useful for creating land masks for different geographical projections.

Details

The removed objects `worldcountries` and `oceanmask` were suitable for plots involving geographical projections not available through `map`, and more generally for raster plots. A land mask could be produced out of `worldcountries` by filling the countries, as by `fill="black"` in the code for `country.layer` in the Examples in https://gitlab.in2p3.fr/francois.rousset/spamm-ref/-/blob/master/vignettePlus/example_raster.html. These objects may now be available through the same web page, but a better place to look for the same functionality is the `IsoriX` package (objects `CountryBorders` and `OceanMask`).

`seaMask` and `landMask` were created from the world map in the maps package. `polypath` requires polygons, while `map(interior=FALSE,plot=FALSE)` returns small segments. `landMask` is the result of reconnecting the segments into full coastlines of all land blocks.

See Also

https://gitlab.in2p3.fr/francois.rousset/spamm-ref/-/blob/master/vignettePlus/example_raster.html for access to, and use of `worldcountries` and `oceanmask`; https://cran.r-project.org/package=IsoriX for replacement `CountryBorders` and `OceanMask` for these objects.

Examples

```
## Predicting behaviour for a land bird: simplified fit for illustration
data("blackcap")
bfit <- fitme(migStatus ~ means+ Matern(1|longitude+latitude),data=blackcap,
              fixed=list(lambda=0.5537,phi=1.376e-05,rho=0.0544740,nu=0.6286311))

## the plot itself, with a sea mask,
## and an ad hoc 'pointmask' to see better the predictions on small islands
#
def_pointmask <- function(xy,r=1,npts=12) {
  theta <- 2*pi/npts *seq(npts)
  hexas <- lapply(seq(nrow(xy)), function(li){
    p <- as.numeric(xy[li,])
    hexa <- cbind(x=p[1]+r*cos(theta),y=p[2]+r*sin(theta))
    rbind(rep(NA,2),hexa) ## initial NA before each polygon
  })
  do.call(rbind,hexas)
}
ll <- blackcap[,c("longitude","latitude")]
pointmask <- def_pointmask(ll[c(2,4,5,6,7),],r=0.8) ## small islands only
#
if (spaMM.getOption("example_maxtime")>1) {
  data("seaMask")

  filled.mapMM(bfit,add.map=TRUE,
             plot.title=title(main="Inferred migration propensity of blackcaps",
                              xlab="longitude",ylab="latitude"),
             decorations=quote(points(pred[,coordinates],cex=1,pch="+")),
             plot.axes=quote({axis(1);axis(2);
                       polypath(rbind(seaMask,pointmask),border=FALSE,
                                col="grey", rule="evenodd")
             }))
}
```

---

seeds *Seed germination data*

Description

A classic toy data set, "from research conducted by microbiologist Dr P. Whitney of Surrey University. A batch of tiny seeds is brushed onto a plate covered with a certain extract at a given dilution. The numbers of germinated and ungerminated seeds are subsequently counted" (Crowder, 1978). Two seed types and two extracts are here considered in a 2x2 factorial design.

Usage

```
data("seeds")
```

Format

The data frame includes 21 observations on the following variables:

plate Factor for replication;

seed Seed type, a factor with two levels O73 and O75;

extract Root extract, a factor with two levels Bean and Cucumber;

r Number of seeds that germinated;

n Total number of seeds tested

Source

Crowder (1978), Table 3.

References

Crowder, M.J., 1978. Beta-binomial anova for proportions. Appl. Statist., 27, 34-37.

Y. Lee and J. A. Nelder. 1996. Hierarchical generalized linear models (with discussion). J. R. Statist. Soc. B, 58: 619-678.

Examples

```
# An extended quasi-likelihood (EQL) fit as considered by Lee & Nelder (1996):
data("seeds")
fitme(cbind(r,n-r)~seed*extract+(1|plate),family=binomial(),
      rand.family=Beta(),
      method="EQL-", # see help("method") for difference with "EQL+" method
      data=seeds)
```

---

setNbThreads *Parallel computations in fits*

Description

A few steps of fitting can be parallelized. Currently it is possible to control the use of multiple threads by OpenMP by the Eigen library. By default only one thread will be used, but this may be modified by using `control.HLfit$NbThreads` in a fitting function's arguments, as in

```
avail_thr <- parallel::detectCores(logical=FALSE) - 1L
fitme(., control.HLfit=list(NbThreads=max(avail_thr, 1L)))
```

This control is distinct from that of post-fit steps such as bootstraps where some parallel computations are controlled e.g. the `nb_cores` argument of `spaMM_boot`. In cases where post-fits computation imply refits of models (as is typical of parametric bootstraps), the two parallelizations should not be combined, and the **spaMM** code for post-fit operations will in principle automatically take care of this.

According to https://cran.r-project.org/doc/manuals/r-devel/R-exts.html#OpenMP-support, using openMP may decrease the precision of some computations, and may be less efficient under Windows; and according to https://libeigen.gitlab.io/eigen/docs-nightly/TopicMultiThreading.html only a few Eigen computations will benefit from such parallelisation, mainly the dense matrix products. **spaMM** will *suggest* using parallelisation when random effects have many levels and dense-correlation methods are selected (see algebra), that is mainly for geostatiscal models with many locations. Speed gains appear moderate, as the slowest steps are not parallelized.

`simulate.HLfit` *Simulate realizations of a fitted model.*

Description

From an HLfit object, `simulate.HLfit` function generates new samples given the estimated fixed effects and dispersion parameters. Simulation may be unconditional (the default, useful in many applications of parametric bootstrap), or conditional on the predicted values of random effects, or may draw from the conditional distribution of random effects given the observed response. Simulations may be run for the original sampling design (i.e., original values of fixed-effect predictor variables and of random-effect levels, including spatial locations if relevant), or for a new design as specified by the `newdata` argument.

The `simulate(`*multivariate-response fit*`)` return value will contain `NA` elements when information is missing to predict and simulate these elements, while this is so neither in the univariate case nor for `predict` results. This goes together with differences in the format of the `data` element in the fitted object (see Value in fitmv). In both cases, the `simulate` result is directly usable in update_resp and suitable for standard bootstrap computations, with no need for additional user input. `simulate(`*multivariate-response fit*`,` `newdata=`*newdata*`)` may also contain `NA`s (see Details).

`simulate4boot` is a wrapper around `simulate.HLfit` that can be used to precompute the bootstrap samples to be used by spaMM_boot or spaMM2boot through their `boot_samples` argument (and is called internally by these functions when `boot_samples` is NULL).

`simulate_ranef` will only simulate and return a vector of random effects, more specifically some elements of the **b** vector appearing in the standard form `offset`+ $\mathbf{X}\beta + \mathbf{Z}\,\mathbf{b}$ for the linear predictor.

Usage

```
## S3 method for class 'HLfit'
simulate(object, nsim = 1, seed = NULL, newdata = NULL,
                         type = "marginal", re.form, conditional = NULL,
                         verbose = c(type=TRUE,
                                     showpbar=eval(spaMM.getOption("barstyle"))),
                    sizes = if (is.null(newdata)) get_drawSizes(object, p4m="M"),
                         resp_testfn = NULL, phi_type = "predict",
                     prior.weights = if (is.null(newdata)) object$prior.weights,
                         variances=list(), ...)

## S3 method for class 'HLfitlist'
simulate(object, nsim = 1, seed = NULL,
```

```
                              newdata = object[[1]]$data, sizes = NULL, ...)

simulate4boot(object, nsim, seed=NULL, resp_testfn=NULL, type="marginal",
              showpbar=eval(spaMM.getOption("barstyle")), ...)
simulate_ranef(object, which=NULL, newdata = NULL, nsim = 1L)
```

Arguments

`object`	The return object of HLfit or similar function.
`nsim`	number of response vectors to simulate. Defaults to '1'.
`seed`	A seed for `set.seed`. If such a value is provided, the initial state of the random number generator at a global level is restored on exit from simulate.
`newdata`	A data frame closely matching the original data, except that response values are not needed. May provide new values of fixed predictor variables, new spatial locations, new individuals within a block, or new values of the LHS in random-effect terms of the form `(<LHS>|<RHS>)`.
`prior.weights`	Prior weights that may be substituted to those of the original fit, with the same effect on the residual variance. See Details for the definition of the default when `newdata` are provided. For multivariate-response fits, this is a list with one element per submodel, each element being a vector whose size is the number of response levels to be simulated for each submodel (the `object$prior.weights` provides an example).
`sizes`	A vector of sample sizes to simulate in the case of a binomial, `betabin` of multinomial fits. See Details for the definition of the default when `newdata` are provided. For multivariate-response fits, the `sizes` argument should contain elements, for response levels that can be predicted and simulated (see Detaisl for handling of missing information), for all submodels whatever their response families (e.g. for submodels with families and response levels `poisson: 3` and `binomial: 2`, respectively, the `sizes` vector should contain 5 elements, e.g. 1 1 1 5 10, only the last two of which having nontrivial meaning).
`re.form`	formula for random effects to condition on. Default behaviour depends on the `type` argument. The joint default is the latter's default, i.e., unconditional simulation. `re.form` is currently ignored when `type="predVar"` (with a warning). Otherwise, `re.form=NULL` conditions on all random effects (as `type="residual"` does), and `re.form=NA` conditions on none of the random effects (as `type="marginal"` or `re.form=~0` do).
`type`	character string specifying which uncertainties are taken into account in the linear predictor and notably in the random effect terms. Whatever the `type`, the residual variance is always accounted in the simulation output. `"marginal"` accounts for the marginal variance of the random effect; `"predVar"` accounts for the conditional distribution of the random effects given the data (and, by default, also for the uncertainty in fixed effects: see Details); and `"residual"` should perhaps be `"none"` as no uncertainty is accounted in the linear predictor: the simulation variance is only the residual variance of the fitted model.
`conditional`	Obsolete and will be deprecated. Boolean; TRUE and FALSE are equivalent to `type="residual"` and `type="marginal"`, respectively.

verbose	Either a single boolean (which determines verbose[["type"]], or a vector of booleans with possible elements "type" (to display basic information about the type of simulation) and "showpbar" (see predict(.,verbose)).
resp_testfn	NULL, or a function that tests a condition which simulated samples should satisfy. This function takes a response vector as argument and return a boolean (TRUE indicating that the sample satisfies the condition).
phi_type	Character string, either "predict" or one of the values possible for type. This controls the residual variance parameter ϕ. The default is to use predicted ϕ values from the fit, which are the fitted ϕ values except when a structured-dispersion model is involved together with non-NULL newdata. Simulating residual variation for ϕ is a dubious concept and is no longer a documented option. However, when a mixed-effect dispersion model is involved, it is still possible to simulate new values of the predictor for ϕ values, and for such mixed models, the same types of simulation controlled by type for the mean response can be performed as controlled by phi_type.
variances	Used when type="predVar": see Details.
...	For simulate4boot, further arguments passed to simulate.HLfit (e.g., newdata). For simulate.HLfit, further arguments only passed to predict in a speculative bit of code (see Details).
which	Integer, or integer vector: the random effect(s) (indexed as ordered as in the model formula) to be simulated. If NULL, all of them are simulated.
showpbar	Controls display of progress bar. See barstyle option for details.

Details

If the original data *myData*, to which a model was fitted, contained missing information (missing predictor or response values in some lines), the format of the simulate results will depend whether a multivariate response model was fitted or not:

univariate case: simulate(*fitme result*) will contain only numbers for informative lines and no NAs; simulate(*fitme result*, newdata=*result's*$data) will have the same structure; and simulate(*fitme result*, newdata=*myData*) should have additional values for lines of data with missing (only) response values, and still no NAs.

multivariate case: simulate(*fitmv result*) will contain numbers for informative lines and NAs for other lines; while both simulate(*fitmv result*, newdata=*result's*$data) and simulate(*fitmv result*, newdata=*my data*) will contain NA for values that cannot be predicted due to missing predictors (but will contain simulated values otherwise, irrespective of the absence of response values). The sizes argument, when used, should match the non-NA values (hence differentvalues might need to be specified whether newdata is NULL or not).

type="predVar" accounts for the uncertainty of the linear predictor, by drawing new values of the predictor in a multivariate gaussian distribution with mean and covariance matrix of prediction. In this case, the user has to provide a variances argument, passed to predict, which controls what goes into this covariance matrix. For example, with variances=list(linPred=TRUE,disp=TRUE)), the covariance matrix takes into account the joint uncertainty in the fixed-effect coefficients and of any random effects given the response and the point estimates of dispersion and correlation parameters ("linPred" variance component), and in addition accounts for uncertainty in the dispersion parameters (effect of "disp" variance component as further described in predict.HLfit). The

total simulation variance is then the response variance. Uncertainty in correlation parameters (such a parameters of the Matern family) is not taken into account. The "linPred" uncertainty is known exactly in LMMs, and otherwise approximated as a Gaussian distribution with mean vector and covariance matrix given as per the Laplace approximation.

type="(ranef|response)" can be viewed as a special version of type="predVar" where variances=list(linPred=TRUE,disp=FALSE)) and only the uncertainty in the random effects is taken into account.

A full discussion of the merits of the different types is beyond the scope of this documentation, but these different types may not all be useful. type="marginal" is typically used for computation of confidence intervals by parametric bootstrap methods. type="residual" is used by get_cPredVar for its evaluation of a bias term. The other types may be used to simulate the uncertainty in the random effects, conditionally on the data, and may therefore be more akin to the computation of prediction intervals conditionally on an (unknown but inferred) realization of the random effects. But these should presumably not be used in a bootstrap computation of such intervals, as this would represent a double accounting of the uncertainty that the bootstrap aims to quantify.

There are cases where simulation without a newdata argument may give results of different length than simulation with newdata=<original data>, as for predict.

When newdata are provided but new values of prior.weights or sizes are missing, new values of these missing arguments are guessed, and warnings may be issued depending on the kind of guess made for response families dependent on such arguments. The prior.weights values used in the fit are re-used without warning when such values were identical (generally, unit) for all response values, and labelled as such in the object$prior.weights. Unit weights will be used otherwise, with a warning. Unit binomial sizes will be used, with a warning, whenever there are newdata.

Value

simulate.HLfit returns a vector (if nsim=1) or a matrix with nsim columns, each containing simulated vectors of responses (or simulated random effects, for simulated_ranef()). For multivariate-response simulations, each vector may contain NA elements when some information is missing (see Details), and an nobs attribute gives the number of non-NA responses for each submodel if no resp_testfn was applied.

simulate4boot returns a list with elements

bootreps the result of simulate.HLfit as a matrix with nsim columns, one for each simulated response vector;

RNGstate the state of .Random.seed at the beginning of the sample simulation.

The simulate.HLfitlist method returns a list of simulated responses.

Examples

```
data("Loaloa")
HLC <- HLCor(cbind(npos,ntot-npos)~Matern(1|longitude+latitude),
           data=Loaloa,family=binomial(),
           ranPars=list(lambda=1,nu=0.5,rho=1/0.7))
simulate(HLC,nsim=2)

## Structured dispersion model
```

```
data("wafers")
hl <- HLfit(y ~X1+X2+X1*X3+X2*X3+I(X2^2)+(1|batch),family=Gamma(log),
            resid.model = ~ X3+I(X3^2) ,data=wafers)
# 'phi_type' matters only for mixed models:
simulate(hl,type="marginal", phi_type="anything", nsim=2)
simulate_ranef(hl,nsim=2)

## Using the returned 'RNGstates' to reproduce sample simulation:
s1 <- simulate4boot(hl, nsim=3)
assign(".Random.seed", pos = .GlobalEnv, value = s1$RNGstates)
s2 <- simulate4boot(hl, nsim=3)
identical(s1$bootreps, s2$bootreps) # TRUE
```

---

spaMM *Inference in mixed models, in particular spatial GLMMs*

Description

Fits a range of models including (or not) random effects, spatially autocorrelated or not. This covers a range of models from the simple linear regression model to generalized linear mixed models (GLMMs), and their extensions with non-GLM response families, non-Gaussian random effects ("HGLMs"), and multivariate response. Both maximum likelihood (ML) and restricted likelihood (REML) can be used for linear mixed models, and extensions of these methods using Laplace approximations are used for other models.

The main fitting functions are `fitme`, or `fitmv` for multivariate-response models. A variety of post-fit procedures are available for prediction, simulation and testing (see, e.g., `fixedLRT`, `simulate` and `predict`). So-called multinomial logit (mixed) models can be fitted by `pois4mlogit`, with some limitations on post-fit procedures. Alternatively, the fit of multinomial models as nested binomial models is facilitated by `fitmv`.

Also available are previously conceived fitting functions `HLfit` (sometimes faster, for non-spatial models), `HLCor` (sometimes faster, for conditional-autoregressive models and fixed-correlation models), and `corrHLfit` (which may be viewed as a primitve version of `fitme`, for spatial models only).

A variety of special syntaxes defined for fixed effects in other packages, such as `poly`, `splines::ns` or `bs`, are handled natively. Additional syntaxes are defined, such as `X.GCA` for diallel/multi-membership fixed-effect terms (`lmDiallel::GCA` is handled but is not suitable for `predict`)). Note that packages implementing these terms must be attached to the search list as `::` will not be properly understood in a `formula`.

Several variants of ML and REML methods discussed in the literature are included (see Details in `HLfit`), the most notable of which may be "PQL/L" for binary-response GLMMs (see Example for `arabidopsis` data). PQL methods implemented in spaMM are closer to (RE)ML methods than those implemented in `MASS::glmmPQL`.

Details

The standard response families `gaussian`, `binomial`, `poisson`, and `Gamma` are handled, as well as negative binomial (see `negbin1` and `negbin2`), beta (`beta_resp`), beta-binomial (`betabin`),

COMPoisson, tweedie, and zero-truncated variants of the two negative binomial families and of the poisson (see the distinct Poisson for the latter). A multi family look-alike is also available for multinomial response, but appears superseded by the alternative approaches to fit multinomial models.

The variance parameter of residual error is denoted ϕ (phi): this is the residual variance for gaussian response, but for Gamma-distributed response, the residual variance is $\phi\mu^2$ where μ is expected response. A (possibly mixed-effects) linear predictor for ϕ, modeling heteroscedasticity, can be considered (see Examples).

The package fits models including several random effects, including autocorrelated ones. An interface is being developed allowing users to implement their own parametric correlation models (see corrFamily), beyond the following ones which are built in **spaMM**:
* geostatistical (Matern, Cauchy),
* interpolated Markov Random Fields (IMRF, MaternIMRFa),
* autoregressive time-series (AR1, ARp, ARMA),
* conditional autoregressive as specified by an adjacency matrix,
* pairwise interactions with individual-level random effects, such as diallel experiments (diallel),
* or any fixed correlation matrix (corrMatrix).

GLMMs and HGLMs are fit via Laplace approximations for (1) the marginal likelihood with respect to random effects and (2) the restricted likelihood (as in REML), i.e. the likelihood of random effect parameters given the fixed effect estimates. All handled models can be formulated in terms of a linear predictor of the traditional form offset+ $\mathbf{X}\beta$ + $\mathbf{Z}$ **b**, where $\mathbf{X}$ is the design matrix of fixed effects, β (beta) is a vector of fixed-effect coefficients, $\mathbf{Z}$ is a "design matrix" for the random effects (which is instead denoted **M=ZAL** elsewhere in the package documentation), and **b** a vector of random effect values. The general structure of **Mb** is described in random-effects.

Gaussian and non-gaussian random effects can be fitted. Different **gaussian** random-effect terms are handled, with the following effects:

```
* (1|<RHS>), for non-autocorrelated random effects as in lme4;
* (<LHS>|<RHS>), for random-coefficient terms as in lme4, *and
   additional terms depending on the <LHS> type* (further detailed below);
* (<LHS> || <RHS>) is interpreted as in lme4: any such term is immediately
   converted to ( (1|<RHS>) + (0+<LHS>|<RHS>) ). It should be counted as two
   random effects for all purposes (e.g., for fixing the variances of the
   random effects). However, this syntax is useless when the LHS includes a
   factor (see help('lme4::expandDoubleVerts')).
* <prefix>(1|<RHS>), to specify autocorrelated random effects,
   e.g. Matern(1|long+lat).
* <prefix>(<LHS>|<RHS>), where the <LHS> can be used to alter the
   autocorrelated random effect as detailed below.
```

Different LHS types of **gaussian** (<LHS>|<RHS>) random-effect terms are handled, with the following effects:

```
* <logical> (TRUE/FALSE): affects only responses for which <LHS> is TRUE.
* <factor built from a logical>: same a <logical> case;
* <factor not built from a logical>: random-coefficient term as in lme4;
* 0 + <factor not built from a logical>: same but contrasts are not used;
```

```
* factors specified by the mv(...) expression, generate random-coefficient
  terms specific to multivariate-response models fitted by fitmv() (see
  help("mv")). 0 + mv(...) has the expected effect of not using contrasts;
* <numeric> (but not '0+<numeric>'): random-coefficient term as in lme4,
  with 2*2 covariance matrix of effects on Intercept and slope;
* 0 + <numeric>: no Intercept so no covariance matrix (random-slope-only
   term);
```

The '0 + <numeric>' effect is achieved by direct control of the elements of the incidence matrix **Z** through the <LHS> term: for numeric z, such elements are multiplied by z values, and thus provide a variance of order O(z **squared**).

If one wishes to fit uncorrelated group-specific random-effects with distinct variances for different groups or for different response variables, three syntaxes are thus possible. The most general, suitable for fitting several variances (see GxE for an example), is to fit a (0 + <factor>| <RHS>) random-coefficient term with correlation(s) fixed to 0. Alternatively, one can define **numeric** (0|1) variables for each group (as as.numeric(<boolean for given group membership>)), and use each of them in a 0 + <numeric> LHS (so that the variance of each such random effect is zero for response not belonging to the given group). See lev2bool for various ways of specifying such indicator variables for several levels.

Gaussian <prefix>(<LHS not 1>|<RHS>) random-effect terms may be handled, with two main cases depending on the LHS type, motivated by the following example: independent Matérn effects can be fitted for males and females by using the syntax Matern(male|.) + Matern(female|.), where male and female are TRUE/FALSE (or a factor with TRUE/FALSE levels). In contrast to a (male|.) term, no random-coefficient correlation matrix is fitted. However, for some other types of RHS, one can fit *composite random effects* combining a random-coefficient correlation matrix and the correlation model defined by the "prefix". This combination is defined in composite-ranef. This leads to the following distinction:
* The terms are *not* composite random effects when the non-'1' LHS type is boolean or factor-from-boolean, a just illustrated, but also 0+<numeric>: for example, Matern(0+<numeric>|.) represents an autocorrelated random-slope (only) term or, equivalently, a direct specification of heteroscedasticity of the Matérn random effect.
* By contrast, Matern(<numeric>|.) implies estimating a random-coefficient covariance matrix and thus defines a composite random effects, as does an LHS that is a factor constructed from numeric or character levels.
Composite random effects can be fitted in principle for all "prefixes", including for <corrFamily> terms. In practice, this functionality has been checked for Matern, corrMatrix, AR1 and the ARp-corrFamily term. In these terms, the <.> %in% <.> form of nested random effect is allowed.

The syntax (z-1|.), for **numeric** z only, can also be used to fit **some heteroscedastic non-Gaussian** random effects. For example, a Gamma random-effect term (wei-1|block) specifies an heteroscedastic Gamma random effect u with constant mean 1 and variance wei^2 λ, where λ is still the estimated variance parameter. See Details of negbin for a possible application. Here, this effect is not implemented through direct control of **Z** (multiplying the elements of an incidence matrix **Z** by wei), as this would have a different effect on the distribution of the random effect term. (z|.) is not defined for *non-Gaussian* random effects. It could mean that a correlation structure between random intercepts and random slopes for (say) Gamma-distributed random effects is considered, but such correlation structures are not well-specified by their correlation matrix.

Author(s)

spaMM was initially published by François Rousset and Jean-Baptiste Ferdy, and is continually developed by F. Rousset and tested by Alexandre Courtiol.

References

Lee, Y., Nelder, J. A. and Pawitan, Y. (2006). Generalized linear models with random effects: unified analysis via h-likelihood. Chapman & Hall: London.

Rousset F., Ferdy, J.-B. (2014) Testing environmental and genetic effects in the presence of spatial autocorrelation. Ecography, 37: 781-790. doi:10.1111/ecog.00566

See Also

See the test directory of the package for many additional examples of **spaMM** usage beyond those from the formal documentation.

See fitmv for multivariate-response models.

Specific information for installation of **spaMM** dependencies from source may be found at https://gitlab.in2p3.fr/francois.rousset/spamm-ref#installation.

Examples

```
data("wafers")
data("scotlip") ## loads 'scotlip' data frame, but also 'Nmatrix'

##     Linear model
fitme(y ~ X1, data=wafers)

##     GLM
fitme(y ~ X1, family=Gamma(log), data=wafers)
fitme(cases ~ I(log(population)), data=scotlip, family=poisson)

##     Non-spatial GLMMs
fitme(y ~ 1+(1|batch), family=Gamma(log), data=wafers)
fitme(cases ~ 1+(1|gridcode), data=scotlip, family=poisson)
#
# Random-slope model (mind the output!)
fitme(y~X1+(X2|batch),data=wafers, method="REML")

## Spatial, conditional-autoregressive GLMM
if (spaMM.getOption("example_maxtime")>2) {
  fitme(cases ~ I(log(population))+adjacency(1|gridcode), data=scotlip, family=poisson,
        adjMatrix=Nmatrix) # with adjacency matrix provided by data("scotlip")
}
# see ?adjacency for more details on these models

## Spatial, geostatistical GLMM:
# see e.g. examples in ?fitme, ?corrHLfit, ?Loaloa, or ?arabidopsis;
# see examples in ?Matern for group-specific spatial effects.

##     Hierachical GLMs with non-gaussian random effects
```

```
 data("salamander")
if (spaMM.getOption("example_maxtime")>1) {
 # both gaussian and non-gaussian random effects
 fitme(cbind(Mate,1-Mate)~1+(1|Female)+(1|Male),family=binomial(),
        rand.family=list(gaussian(),Beta(logit)),data=salamander)

 # Random effect of Male nested in that of Female:
 fitme(cbind(Mate,1-Mate)~1+(1|Female/Male),
       family=binomial(),rand.family=Beta(logit),data=salamander)
 # [ also allowed is cbind(Mate,1-Mate)~1+(1|Female)+(1|Male %in% Female) ]
}

##    Modelling residual variance ( = structured-dispersion models)
# GLM response, fixed effects for residual variance
fitme( y ~ 1,family=Gamma(log),
      resid.model = ~ X3+I(X3^2) ,data=wafers)
#
# GLMM response, and mixed effects for residual variance
if (spaMM.getOption("example_maxtime")>1.5) {
  fitme(y ~ 1+(1|batch),family=Gamma(log),
        resid.model = ~ 1+(1|batch) ,data=wafers)
}
```

---

spaMM-conventions *spaMM conventions and differences from related fitting procedures*

Description

input arguments are generally similar to those of `glm` and `(g)lmer`, in particular for the `spaMM::fitme` function, with the exception of the `prior.weights` argument, which is simply `weights` in the other packages. The name `prior.weights` seems more consistent, since e.g. `glm` returns its input `weights` as output `prior.weights`, while its output `weights` are instead the weights in the final iteration of an iteratively weighted least-square fit.

The **default likelihood target** for dispersion parameters is restricted likelihood (REML estimation) for `corrHLfit` and (marginal) likelihood (ML estimation) for `fitme`. Model fits may provide restricted likelihood values(`ReL`) even if restricted likelihood is is not used as an objective function at any step in the analysis.

See good-practice for advice about the proper syntax of `formula`.

Computation times depend on control parameters given by `spaMM.getOption("spaMM_tol")` parameters (for iterative algorithms), and `spaMM.getOption("nloptr")` parameters for the default optimizer. Do not use `spaMM.options()` to control them globally, unless you know what you are doing. Rather control them locally by the `control.HLfit` argument to control `spaMM_tol`, and by the control arguments of `corrHLfit` and `fitme` to control `nloptr`. If `nloptr$Xtol_rel` is set above 5e-06, `fitme` will by default refit the fixed effects and dispersion parameters (but not other correlation parameters estimated by `nloptr`) by the iterative algorithm after `nloptr` convergence. Increasing `nloptr$Xtol_rel` value may therefore switches the bulk of computation time from the optimizer to the iterative algorithm, and may increase or decrease computation time depending on

which algorithm is faster for a given input. Use `control$refit` if you wish to inhibit this, but note that by default it provides a rescue to a poor `nloptr` result due to a too large `Xtol_rel`.

References

Chambers J.M. (2008) Software for data analysis: Programming with R. Springer-Verlag New York

---

`spaMM.colors` *A flashy color palette.*

Description

`spaMM.colors` is the default color palette for some color plots in `spaMM`.

Usage

```
spaMM.colors(n = 64, redshift = 1, adjustcolor_args=NULL)
```

Arguments

`n`	Number of color levels returned by the function. A calling graphic function with argument `nlevels` will typically take the first (i.e., bluest) `nlevels` color levels. If `n<nlevels`, the color levels are recycled
`redshift`	The higher it is, the more the palette blushes....
`adjustcolor_args`	Either NULL or a list of arguments for `adjustcolor`, in which case `adjustcolor` is called to modify `spaMM.colors`'s default vector of colors. See the documentation of the latter function for further information. All arguments except `col` are possible.

Details

If you don't like this color palette, have a look at the various ones provided by the `fields` package.

Value

A vector giving the colors in a hexadecimal format.

Examples

```
## see mapMM examples
```

spaMM.filled.contour *Level (Contour) Plots with better aspect ratio control (for geographical maps, at least)*

Description

This function is derived from filled.contour in the graphics package, and this documentation is likewise heavily based on that of filled.contour.

This function likewise produces a contour plot with the areas between the contours filled in solid color, and a key showing how the colors map to z values is likewise shown to the right of the plot. The only difference is the way the aspect ratio is determined and can be controlled (using the map.asp parameter instead of asp), They thus easily provide nice-looking maps with meaningful latitude/longitude ratio (see Examples). However, this does not work well with rstudio.

Usage

```
spaMM.filled.contour(x = seq(0, 1, length.out = nrow(z)),
               y = seq(0, 1, length.out = ncol(z)),
               z,
               xrange = range(x, finite = TRUE),
               yrange = range(y, finite = TRUE),
               zrange = range(z, finite = TRUE, na.rm=TRUE),
               margin=1/20,
               levels = pretty(zrange, nlevels), nlevels = 20,
               color.palette = spaMM.colors,
               col = color.palette(length(levels) - 1),
               plot.title, plot.axes, key.title=NULL, key.axes=NULL,
               map.asp = NULL, xaxs = "i", yaxs = "i", las = 1,
               axes = TRUE, frame.plot = axes, ...)
```

Arguments

x, y	locations of grid lines at which the values in z are measured. These must be in ascending order. (The rest of this description does not apply to .filled.contour.) By default, equally spaced values from 0 to 1 are used. If x is a list, its components x$x and x$y are used for x and y, respectively. If the list has component z this is used for z.
z	a numeric matrix containing the values to be plotted.. Note that x can be used instead of z for convenience.
xrange	x range of the plot.
yrange	y range of the plot.
zrange	z range of the plot.
margin	This controls how far (in relative terms) the plot extends beyond the x and y ranges of the analyzed points, and is overriden by explicit xrange and yrange arguments.

levels	a set of levels which are used to partition the range of z. Must be **strictly** increasing (and finite). Areas with z values between consecutive levels are painted with the same color.
nlevels	if levels is not specified, the range of z, values is divided into approximately this many levels.
color.palette	a color palette function to be used to assign colors in the plot.
col	an explicit set of colors to be used in the plot. This argument overrides any palette function specification. There should be one less color than levels
plot.title	statements which add titles to the main plot.
plot.axes	statements which draw axes (and a box) on the main plot. This overrides the default axes.
key.title	statements which add titles for the plot key.
key.axes	statements which draw axes on the plot key. This overrides the default axis.
map.asp	the y/x aspect ratio of the 2D plot area (not of the full figure including the scale). Default is (plotted y range)/(plotted x range) (i.e., scales for x and y are identical) as long as this does not conflict too much with the available plot area deduced from the device dimensions.
xaxs	the x axis style. The default is to use internal labeling.
yaxs	the y axis style. The default is to use internal labeling.
las	the style of labeling to be used. The default is to use horizontal labeling.
axes, frame.plot	logicals indicating if axes and a box should be drawn, as in plot.default.
...	additional graphical parameters, currently only passed to title().

Details

The values to be plotted can contain NAs. Rectangles with two or more corner values are NA are omitted entirely: where there is a single NA value the triangle opposite the NA is omitted.

Values to be plotted can be infinite: the effect is similar to that described for NA values.

Value

This returns invisibly a list with elements of the plot, the x, y, z coordinates and the contour levels.

Note

Builds heavily on filled.contour by Ross Ihaka and R-core. spaMM.filled.contour uses the layout function and so is restricted to a full page display.

The output produced by spaMM.filled.contour is actually a combination of two plots; one is the filled contour and one is the legend. Two separate coordinate systems are set up for these two plots, but they are only used internally – once the function has returned these coordinate systems are lost. If you want to annotate the main contour plot, for example to add points, you can specify graphics commands in the plot.axes argument. See the Examples.

References

Cleveland, W. S. (1993) *Visualizing Data*. Summit, New Jersey: Hobart.

See Also

contour, image, palette; contourplot and levelplot from package lattice.

Examples

```
spaMM.filled.contour(volcano, color.palette = spaMM.colors) # simple

## Comparing the layout with that of filled.contour:
#  (except that it does not always achieve the intended effect
#  in RStudio Plots pane).

x <- 10*1:nrow(volcano)
y <- 10*1:ncol(volcano)
spaMM.filled.contour(x, y, volcano, color.palette = terrain.colors,
    plot.title = title(main = "The Topography of Maunga Whau",
    xlab = "Meters North", ylab = "Meters West"),
    plot.axes = { axis(1, seq(100, 800, by = 100))
                  axis(2, seq(100, 600, by = 100)) },
    key.title = title(main = "Height\n(meters)"),
    key.axes = axis(4, seq(90, 190, by = 10)))  # maybe also asp = 1
mtext(paste("spaMM.filled.contour(.) from", R.version.string),
      side = 1, line = 4, adj = 1, cex = .66)

## compare with

filled.contour(x, y, volcano, color.palette = terrain.colors,
    plot.title = title(main = "The Topography of Maunga Whau",
    xlab = "Meters North", ylab = "Meters West"),
    plot.axes = { axis(1, seq(100, 800, by = 100))
                  axis(2, seq(100, 600, by = 100)) },
    key.title = title(main = "Height\n(meters)"),
    key.axes = axis(4, seq(90, 190, by = 10)))  # maybe also asp = 1
mtext(paste("filled.contour(.) from", R.version.string),
      side = 1, line = 4, adj = 1, cex = .66)
```

---

spaMM_boot *Parametric bootstrap*

Description

spaMM_boot simulates samples from a fit object inheriting from class "HLfit", as produced by spaMM's fitting functions, and applies a given function to each simulated sample. Parallelization is supported (see Details).

spaMM2boot is similar except that it assumes that the original model is refitted on the simulated data, and the given function is applied to the refitted model, and the value is in a format directly usable as input for boot::boot.ci.

Both of these functions can be used to apply standard parametric bootstrap procedures. spaMM_boot is suitable for more diverse applications, e.g. to fit by one model some samples simulated under another model (see Example).

Usage

```
spaMM_boot(object, simuland, nsim, nb_cores=NULL, seed=NULL,
           resp_testfn=NULL, control.foreach=list(),
           debug. = FALSE, type, fit_env=NULL, cluster_args=NULL,
           showpbar= eval(spaMM.getOption("barstyle")),
           boot_samples=NULL,
           ...)
spaMM2boot(object, statFUN, nsim, nb_cores=NULL, seed=NULL,
           resp_testfn=NULL, control.foreach=list(),
           debug. = FALSE, type="marginal", fit_env=NULL,
           cluster_args=NULL, showpbar= eval(spaMM.getOption("barstyle")),
           boot_samples=NULL,
           ...)
```

Arguments

object	The fit object to simulate from.
simuland	The function to apply to each simulated sample. See Details for requirements of this function.
statFUN	The function to apply to each fit object for each simulated sample. See Details for requirements of this function.
nsim	Number of samples to simulate and analyze.
nb_cores	Number of cores to use for parallel computation. The default is spaMM.getOption("nb_cores"), and 1 if the latter is NULL. nb_cores=1 prevents the use of parallelisation procedures.
seed	Passed to simulate.HLfit
resp_testfn	Passed to simulate.HLfit; NULL, or a function that tests a condition which simulated samples should satisfy. This function takes a response vector as argument and return a boolean (TRUE indicating that the sample satisfies the condition).
control.foreach	list of control arguments for foreach. These include in particular .combine (with default value "rbind"), and .errorhandling (with default value "remove", but "pass" is quite useful for debugging).
debug.	Boolean (or integer, interpreted as boolean). For debugging purposes, given that spaMM_boot does not stop when the fit of a bootstrap replicate fails. Subject to changes with no or little notice. In serial computation, debug.=2 will stop on an error. In parallel computation, this would be ignored. The effect of

debug.=TRUE depends on what simuland does of it. The default simuland for likelihood ratio testing functions, eval_replicate, shows how debug. can be used to control a call to dump.frames (however, debugging user-defined functions by such a call does not require control by debug.).

type — Character: passed to simulate.HLfit. Defaults, with a warning, to type="marginal" in order to replicate the behaviour of previous versions of spaMM_boot. This is an appropriate default for various parametric bootstrpa analyses, but not necessarily the appropriate type for all possible uses. See Details of simulate.HLfit for other implemented options.

fit_env — An environment or list containing variables necessary to evaluate simuland on each sample, and not included in the fit object. E.g., use fit_env=list(phi_fix=phi_fix) if the fit assumed fixed=list(phi=phi_fix): the name in list(phi_fix=<.>) must be the name of the object that will be sought by the called process when interpreting fixed=list(phi=phi_fix) (if still unsure about the proper syntax, see the clusterExport documentation, as fit_env is used in the following context: parallel::clusterExport(cl=<cluster>, varlist=ls(fit_env), envir=fit_env)).

cluster_args — NULL or a list of arguments, passed to makeCluster.

showpbar — Controls display of progress bar. See barstyle option for details.

boot_samples — NULL, or precomputed bootstrap samples from the fitted model, provided as a matrix with one column per bootstrap replicate (the format of the result of simulate.HLfit), or as a list including a bootreps element with the same matrix format.

... — Further arguments passed to the simuland function.

Details

The simuland function must take as first argument a vector of response values, and may have other arguments including '...'. When required, these additional arguments must be passed through the '...' arguments of spaMM_boot. Variables needed to evaluate them must be available from within the simuland function or otherwise provided as elements of fit_env.

The statFUN function must take as first argument (named refit) a fit object, and may have other arguments including '...' handled as for simuland.

spaMM_boot handles parallel backends with different features. pbapply::pbapply has a very simple interface (essentially equivalent to apply) and provides progress bars, but (in version 1.4.0, at least) does not have efficient load-balancing. doSNOW also provides a progress bar and allows more efficient load-balancing, but its requires foreach. foreach handles errors differently from pbapply (which will simply stop if fitting a model to a bootstrap replicate fails): see the foreach documentation.

spaMM_boot calls simulate.HLfit on the fit object and applies simuland on each column of the matrix returned by this call. simulate.HLfit uses the type argument, which must be explicitly provided.

Value

spaMM_boot returns a list, with the following element(s) (unless debug. is TRUE):

bootreps nsim return values in the format returned either by apply or parallel::parApply or by foreach::`%dopar%` as controlled by control.foreach$.combine (which is here "rbind" by default).

RNGstate (absent in the case the boot_samples argument was used to provide the new response values but not the RNGstate) the state of .Random.seed at the beginning of the sample simulation.

spaMM2boot returns a list suitable for use by boot.ci, with elements:

t nsim return values of the simulated statistic (in matrix format).

t0 nsim return the value of statFUN from the original fit.

sim The simulation type ("parametric").

R nsim

.Random.seed the state of .Random.seed at the beginning of the sample simulation.

(other elements of an object of class boot are currently not included.)

Examples

```
if (spaMM.getOption("example_maxtime")>7) {
 data("blackcap")

 # Generate fits of null and full models:
 lrt <- fixedLRT(null.formula=migStatus ~ 1 + Matern(1|longitude+latitude),
                 formula=migStatus ~ means + Matern(1|longitude+latitude),
                 method='ML',data=blackcap)

 # The 'simuland' argument:
 myfun <- function(y, what=NULL, lrt, ...) {
    data <- lrt$fullfit$data
   data$migStatus <- y ## replaces original response (! more complicated for binomial fits)
    full_call <- getCall(lrt$fullfit) ## call for full fit
    full_call$data <- data
    res <- eval(full_call) ## fits the full model on the simulated response
    if (!is.null(what)) res <- eval(what)(res=res) ## post-process the fit
    return(res) ## the fit, or anything produced by evaluating 'what'
  }
  # where the 'what' argument (not required) of myfun() allows one to control
  # what the function returns without redefining the function.

  # Call myfun() with no 'what' argument: returns a list of fits
  spaMM_boot(lrt$nullfit, simuland = myfun, nsim=1, lrt=lrt,
             type ="marginal")[["bootreps"]]

  # Return only a model coefficient for each fit:
  spaMM_boot(lrt$nullfit, simuland = myfun, nsim=7,
             what=quote(function(res) fixef(res)[2L]),
             lrt=lrt, type ="marginal")[["bootreps"]]

  ## Not run:
    # Parametric bootstrap by spaMM2boot() and spaMM_boot():
```

```
boot.ci_info <- spaMM2boot(lrt$nullfit, statFUN = function(refit) fixef(refit)[1],
                           nsim=99, type ="marginal")
boot::boot.ci(boot.ci_info, , type=c("basic","perc","norm"))

nullfit <- lrt$nullfit
boot_t <- spaMM_boot(lrt$nullfit, simuland = function(y, nullfit) {
  refit <- update_resp(nullfit, y)
  fixef(refit)[1]
}, nsim=99, type ="marginal", nullfit=nullfit)$bootreps
boot::boot.ci(list(R = length(boot_t), sim="parametric"), t0=fixef(nullfit)[1],
              t= t(boot_t), type=c("basic","perc","norm"))


## End(Not run)
}
```

---

spaMM_glm.fit *Fitting generalized linear models without initial-value or divergence headaches*

Description

spaMM_glm.fit is a stand-in replacement for glm.fit, which can be called through glm by using glm(<>, method="spaMM_glm.fit"). Input and output structure are exactly as for glm.fit. It uses a Levenberg-Marquardt algorithm to prevent divergence of estimates. For models families such as Gamma() (with default inverse link) where the linear predictor is constrained to be positive, if the **rcdd** package is installed, the function can automatically find valid starting values or else indicate that no parameter value is feasible. It also automatically provides good starting values in some cases where the base functions request them from the user (notably, for gaussian(log) with some negative response). spaMM_glm is a convenient wrapper, calling glm with default method glm.fit, then calling method spaMM_glm.fit, with possibly different initial values, if glm.fit failed.

Usage

```
spaMM_glm.fit(x, y, weights = rep(1, nobs), start = NULL, etastart = NULL,
              mustart = NULL, offset = rep(0, nobs), family = gaussian(),
              control = list(maxit=200), intercept = TRUE, singular.ok = TRUE)
spaMM_glm(formula, family = gaussian, data, weights, subset,
          na.action, start = NULL, etastart, mustart, offset,
        control = list(...), model = TRUE, method = c("glm.fit","spaMM_glm.fit"),
       x = FALSE, y = TRUE, singular.ok = TRUE, contrasts = NULL, strict=FALSE, ...)
```

Arguments

All arguments except strict are common to these functions and their stats package equivalents, glm and glm.fit. Most arguments operate as for the latter functions, whose documentation is repeated below. The control argument may operate differently.

formula — an object of class "formula" (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given in the 'Details' section of glm.

family — a description of the error distribution and link function to be used in the model. For spaMM_glm this can be a character string naming a family function, a family function or the result of a call to a family function. For spaMM_glm.fit only the third option is supported. (See family for details of family functions.)

data — an optional data frame, list or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which glm is called.

weights — an optional vector of 'prior weights' to be used in the fitting process. Should be NULL or a numeric vector.

subset — an optional vector specifying a subset of observations to be used in the fitting process.

na.action — a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The 'factory-fresh' default is na.omit. Another possible value is NULL, no action. Value na.exclude can be useful.

start — starting values for the parameters in the linear predictor.

etastart — starting values for the linear predictor.

mustart — starting values for the vector of means.

offset — this can be used to specify an *a priori* known component to be included in the linear predictor during fitting. This should be NULL or a numeric vector of length equal to the number of cases. One or more offset terms can be included in the formula instead or as well, and if more than one is specified their sum is used. See model.offset.

control — a list of parameters for controlling the fitting process. This is passed to glm.control, as for glm.fit. Because one can assume that spaMM_glm.fit will converge in many cases where glm.fit does not, spaMM_glm.fit allows more iterations (200) by default. However, if spaMM_glm.fit is called through glm(. . ., method="spaMM_glm.fit"), then the number of iterations is controlled by the glm.control call within glm, so that it is 25 by default, overriding the spaMM_glm.fit default.

model — a logical value indicating whether *model frame* should be included as a component of the returned value.

method — A 2-elements vector specifying first the method to be used by spaMM_glm in the first attempt to fit the model, second the method to be used in a second attempt if the first failed. Possible methods include those shown in the default, "model.frame", which returns the model frame and does no fitting, or user-supplied fitting functions. These functions can be supplied either as a function or a character string naming a function, with a function which takes the same arguments as glm.fit.

x, y — For spaMM_glm: x is a design matrix of dimension n * p, and y is a vector of observations of length n.

	For spaMM_glm.fit: x is a design matrix of dimension n * p, and y is a vector of observations of length n.
singular.ok	logical; if FALSE a singular fit is an error.
contrasts	an optional list. See the contrasts.arg of model.matrix.default.
intercept	logical. Should an intercept be included in the *null* model?
strict	logical. Whether to perform a fit by spaMM_glm.fit if glm.fit returned the warning "glm.fit: algorithm did not converge".
...	arguments to be used to form the default control argument if it is not supplied directly.

Value

An object inheriting from class glm. See glm for details.

Note

The source and documentation is derived in large part from those of glm.fit.

Examples

```
x <- c(8.752,20.27,24.71,32.88,27.27,19.09)
y <- c(5254,35.92,84.14,641.8,1.21,47.2)

# glm(.) fails:
(check_error <- try(glm(y~ x,data=data.frame(x,y),family=Gamma(log)), silent=TRUE))
if ( ! inherits(check_error,"try-error")) stop("glm(.) call unexpectedly succeeded")

spaMM_glm(y~ x,data=data.frame(x,y),family=Gamma(log))

## Gamma(inverse) examples
x <- c(43.6,46.5,21.7,18.6,17.3,16.7)
y <- c(2420,708,39.6,16.7,46.7,10.8)

# glm(.) fails (can't find starting value)
(check_error <- suppressWarnings(try(glm(y~ x,data=data.frame(x,y),family=Gamma()) , silent=TRUE)))
if ( ! inherits(check_error,"try-error")) stop("glm(.) call unexpectedly succeeded.")

if (requireNamespace("rcdd",quietly=TRUE)) {
  spaMM_glm(y~ x,data=data.frame(x,y),family=Gamma())
}

## A simple exponential regression with some negative response values

set.seed(123)
x <- seq(50)
y <- exp( -0.1 * x) + rnorm(50, sd = 0.1)
glm(y~ x,data=data.frame(x,y),family=gaussian(log), method="spaMM_glm.fit")

# => without the 'method' argument, stats::gaussian(log)$initialize() is called
# and stops on negative response values.
```

stripHLfit *Reduce the size of fitted objects*

Description

Large matrices and other memory-expensive objects may be stored in a fit object. This function removes them in order to reduce the size of the object, particularly when stored on disk. In principle, the removed objects can be regenerated automatically when needed (e.g., for a predict()).

Usage

```
stripHLfit(object, ...)
```

Arguments

object	The result of a fit (an object of class HLfit).
...	Further arguments, not currently used.

Value

The input fit objects with some elements removed.

Note

The effect may change without notice between versions as the efficiency of the operation is highly sensitive to implementation details.

Examples

```
## Not run:
## rather unconvincing example : quantitative effect is small.

# measure size of saved object:
saveSize <- function (object,...) {
    tf <- tempfile(fileext = ".RData")
    on.exit(unlink(tf))
    save(object, file = tf,...)
    file.size(tf)
  }
data("Loaloa")
lfit <- fitme(cbind(npos,ntot-npos)~elev1+elev2+elev3+elev4+maxNDVI1+seNDVI
                  +Matern(1|longitude+latitude), method="HL(0,1)",
              data=Loaloa, family=binomial(), fixed=list(nu=0.5,rho=1,lambda=0.5))
saveSize(lfit)
pfit <- predict(lfit,newdata=Loaloa,variances=list(cov=TRUE)) # increases size!
```

```
saveSize(lfit)
lfit <- stripHLfit(lfit)
saveSize(lfit)

## End(Not run)
```

---

summary.HLfit *Summary and print methods for fit and test results.*

Description

Summary and print methods for results from HLfit or related functions. summary may also be used as an extractor (see e.g. beta_table).

Usage

```
## S3 method for class 'HLfit'
summary(object, details=FALSE, max.print=100L, verbose=TRUE, ...)
## S3 method for class 'HLfitlist'
summary(object, ...)
## S3 method for class 'fixedLRT'
summary(object, verbose=TRUE, ...)
## S3 method for class 'HLfit'
print(x,...)
## S3 method for class 'HLfitlist'
print(x,...)
## S3 method for class 'fixedLRT'
print(x,...)
```

Arguments

object	An object of class HLfit, as returned by the fitting functions in spaMM.
x	The return object of HLfit or related functions.
verbose	For summary.HLfit, whether to print the screen output that is the primary purpose of summary. verbose=FALSE may be convenient when summary is used as an extractor. For summary.fixedLRT, whether to print the model fits or not.
max.print	Controls options("max.print") locally.
details	A list or vector, with elements controlling whether to print some obscure details. Handled non-default elements are: • ranCoefs=TRUE will print details about random-coefficients terms (see Details); • p_value="Wald" will print a p-value for the t-value of each fixed-effect coefficient, assuming a gaussian distribution of the test statistic (but, beyond the generally questionable nature of p-value tables, see e.g. LRT and fixedLRT for alternative testing approaches);

- p_value=TRUE is similar to p_value="Wald" but instead uses Student's t test;
- full.form=*numeric value* controls abbreviation of the display of model formulas if the number of output lines of a full display would exceed the given value. This may be particularly useful for multivariate-response fits (See Details for the way formulas are abbreviated). full.form=TRUE is interpreted as full.form=Inf, and FALSE is interpreted as 0 (so that multiline submodel formulas are always abbreviated). Both the default (no specified value) and an explicit NULL value are interpreted as 8 for multivariate-response fits and as Inf for other fits..

... further arguments passed to or from other methods.

Details

The random effect terms of the linear predictor are of the form **ZLv**. In particular, for **random-coefficients models** (i.e., including random-effect terms such as (z|group) specifying a random-slope component), correlated random effects are represented as **b = Lv** for some matrix **L**, and where the elements of **v** are uncorrelated. In the output of the fit, the Var. column gives the variances of the correlated effects, **b=Lv**. The Corr. column(s) give their correlation(s). If details is TRUE, estimates and SEs of the (log) variances of the elements of **v** are reported as for other random effects in the Estimate and cond.SE. columns of the table of lambda coefficients. However, this non-default output is potentially misleading as the elements of **v** cannot generally be assigned to specific terms (such as intercept and slope) of the random-effect formula, and the representation of **b** as **Lv** is not unique.

The condition for abbreviation of formulas, as defined by details$"full.form" is on the number of lines of the unabbreviated output. But the effect, if any, is to truncate individual formulas as function of getOption("width"). For example, when displaying the 10 formulas for a 10-responses multivariate-response fit, the default value of full.form is 8, but 10 lines will still be printed, so this only avoids display of 10 multiline formulas.

Value

The return value is a list whose elements may be subject to changes, but two of them can be considered stable, and are thus part of the API: the beta_table and lambda_table which are the displayed tables for the coefficients of fixed effects and random-effect variances.

Examples

```
## see examples of fitme() or corrHLfit() usage
```

---

transffit *Transformation of the data.*

Description

transffit is an experimental helper function to fit a transformation of the data, such as the Box-Cox transformation which is the implemented default. It is not particularly optimized, but is intended to help users to do it right. When a non-default transformation is used, it is important to provide the corresponding logDetJac argument.

Usage

```
transffit(object, lower= -2, upper=2, verbose = FALSE,
  extracts = quote({
    y <- object$y
  }),
  updates = quote({
    if (lambda == 0) {
        newy <- log(y)
    } else newy <- (y^lambda - 1)/lambda
    refit <- update_resp(object, newresp = newy)
  }),
  logDetJac = quote({
    (lambda - 1) * sum(log(y))
  }))
```

Arguments

object	A fit object to the untransformed data, as produced by fitme or fitmv.
lower, upper	Numeric vectors: optimization bounds (the function tentatively allows multiparameter transformations).
verbose	Boolean: whether to print inforamtion about the progress of the procedure.
extracts	A quoted R expression providing variables that should be available in the environment of the objective function, which refits the model on transformed variables. The default expression extracts the response variable so that it can be transformed by the objective function.
updates	A quoted R expression providing the transformation (whose mandatory parameter names is lambda) applied to the response variable (and optionally to other variables), *and* computes the refitted model (with mandatory named refit).
logDetJac	The logarithm of the determinant of the Jacobian of the transformation of the response variable, which is required to recover the likelihood, as the probability density of the original data, given the likelihod of the fit to the transformed data (Box and Cox 1964, eq.5).

Value

A list with elements the result of the optimization call, and the fit of the model to the best transformed data.

References

Box, G. E. P. and Cox, D. R. (1964) An analysis of transformations (with discussion). Journal of the Royal Statistical Society B, 26, 211–252.

Examples

```
set.seed(123)
data(wafers)
wafers$x <- (rnorm(198, mean=(wafers$y)^(1/2),sd=0.1))^2
toyfit <- fitme(y ~x+(1|batch),data=wafers)

# Default transformation (Cox-Box)
transffit(toyfit, lower=-2,upper=2)

## Not run:
# power transformation of response
transffit(toyfit,
       updates = quote({
         newy <- y^lambda
         refit <- update_resp(object, newresp = newy)
       }),
       logDetJac = quote({sum((lambda-1)*log(y)+log(lambda))}),
       lower=0.1, upper=2)

# Less standard power transformation of response *and* of predictor
# The update(object, data=.) syntax will be used
# so the 'data' must be extracted and updated:
transffit(toyfit,
       extracts = quote({
         locdata <- object$data
         y <- locdata$y
         x <- locdata$x
       }),
       updates = quote({
         locdata$x <- x^lambda
         locdata$y <- y^lambda
         refit <- update(object, data=locdata)
       }),
       logDetJac = quote({sum((lambda-1)*log(y)+log(lambda))}),
       lower=0.1, upper=2)

## End(Not run)
```

---

tweedie *Tweedie (compound Poisson Gamma) family*

Description

Returns a GLM family object for the Tweedie family of distributions (e.g., Dunn & Smyth, 2018). The variance of such distributions is related to the mean μ: variance=$\phi\mu^p$, where ϕ is a dispersion

parameter (as in Gaussian and Gamma families), and p is the `index` parameter of the distribution, which can be specified as the `index` argument of the function, and is referred to in other contexts (such as initial values for optimization) as `Tw_index`. The index need or need not be specified when calling `tweedie`, depending on usage. The current implementation handles values $1 < p < 2$, for which the distributions are a Poisson sum of Gamma distributions.

The family link is controlled by another argument, `link`, that may specify a log link, or the exponent q of the link relationship $\eta = \mu^q$.

`glm` can be used with this family object and given `index`.

Fitting Tweedie models by **spaMM**'s fitting functions requires some functions from the **tweedieDistr** package, or else from the **tweedie** package, so either one of these suggested packages must be installed. Various problems are theoretically expected for p close to zero (see Details), so the default parameter range when p is estimated is 1.1 – 1.999.

A structured-dispersion model can be fitted for ϕ but this is sensitive to numerical issues (see Details about "inner estimation" of ϕ).

Usage

```
tweedie(index, link = "log", numderiv=TRUE, ...)
```

Arguments

`index`	Missing or numeric value for the index parameter p, subject to the constraint $1 < p < 2$. The "outer-optimizing" functions, such as `fitme` (see `inner-vs-outer`) will estimate this parameter if called with `family=tweedie()` (no given `index` value). In other usages of this family, `index` must be specified.
`link`	Link function, specified either as the default `"log"` for a log link, or as the link exponent q. $q = 0$ is interpreted as the log link. There is no definite constraint on q, but using values that imply that η should be positive (such as 1, or non-integer values) may run into troubles. Values closes to 0 and 1 also lead to numerical errors. See Details if you want to estimate this parameter.
`numderiv`	Boolean; whether to use numerical derivation in some computation. `numderiv=FALSE` (not recommended) will be ignored if the **tweedie** package is not installed.
`...`	Additional arguments passed to some member functions of the returned family object. Currently used only for the `method` and `method.args` of `grad` if numerical derivatives are used (see `numderiv` argument).

Details

One can force **spaMM**'s "outer-optimization" fitting functions to estimate the `link` exponent by using `tweedie(link="fit")`, and specyfing init, lower and upper values for the link exponent (using the name `Tw_link` for their values: see Examples). However this will fail if negative values of η are generated during the fitting process.

Estimates of the residual dispersion parameter ϕ are by default constrained to be > 1e-05 because evaluations of the Tweedie density function are increasingly slow as ϕ approaches 0. Other minimal ϕ value can be set through the `lower` argument, as usual (or through `control.HLfit$min_phi` for inner estimation).

When p is close to 1, the Tweedie density is multimodal; further, when responses are recorded to a limited number of decimal places, the profile likelihood for p may be bimodal with a spurious maximum at $p = 1$ (Dunn & Smyth 2005). From limited numerical experiments, for low p such as 1.05, computed likelihood profiles with respect to ϕ also often exhibited several maxima. Overall, estimation for low p will be sensitive to many things.

If the fitting function uses inner-estimation methods to fit ϕ (see `inner-vs-outer`), it then requires evaluation of a derivative of the Tweedie density, which may be slow or not very accurate, depending on the method used (which can be controlled by the `numderiv` and optional ... arguments). So the fitting function avoids inner estimation for ϕ, when possible. When it is used,
• examples were found where the inner-estimation iterative algorithm did not converge due to cyclic values of ϕ estimates, for $p < 1.5$. An experimental correction has been implemented to fix this convergence issue;
• the results are sensitive to the algorithm used to compute the derivative of the conditional likelihood (with the default algorithm appearing better), and maybe more so when both p and ϕ are small.

Value

A family object with structure similar to `stats::family` objects, but with additional member functions for usage with **spaMM** fitting functions.

References

Dunn, Peter K and Smyth, Gordon K (2005) Series evaluation of Tweedie exponential dispersion model densities. Stat Comput 15, 267–280. doi:10.1007/s112220054070y

Dunn, Peter K and Smyth, Gordon K (2018) Generalized Linear Models With Examples in R. Springer, NY. doi:10.1007/9781441901187

Dunn, P. K. (2026). Tweedie: Evaluation of Tweedie exponential family models. R package. doi:10.32614/CRAN.package.tweedie

Examples

```
if (spaMM.getOption("example_maxtime")>1.4) {
### toy data
  set.seed(123)
  y <- rgamma(20,shape=rpois(20,lambda=rgamma(20,5)))
  x <- 1:20
  toyTw <- data.frame(x=x,y=y, grp=x)

### Fits
fitme(y~x+(1|grp),family=spaMM::tweedie(), data=toyTw)

# tentative estimation of the link exponent:
fitme(y~x+(1|grp),family=spaMM::tweedie(link="fit"), data=toyTw,
      init=list(lambda=NA, phi=NA, Tw_link=0.5),
      lower=list(Tw_link=0.4), upper=list(Tw_link=0.6))
# (this shows that this _can_ work).

### Interactions with other packages
```

```
  tweedie::logLiktweedie(toyglm <-
    glm(y~x,family=spaMM::tweedie(index=1.5,link= -0.5), data=toyTw))
  toyglm$fitted.values
  predict(toyglm, type="response")
  tweedie::logLiktweedie(spaMM_glm(y~x,
    family=spaMM::tweedie(index=1.5,link= -0.5), data=toyTw))
}
```

`update.HLfit` *Updates a fit*

Description

`update` and `update_resp` will update and (by default) re-fit a model. They do this mostly by extracting the call stored in the object, updating the call and evaluating that call. Using `update(<fit>)` is a risky programming style (see Details). `update_formulas(<mv fit>, ...)` can update formulas from a `fitmv` fit as well as the single formula of a fit by the other fitting functions.

`update_resp` handles a new response vector as produced by `simulate`.

Usage

```
## S3 method for class 'HLfit'
update(object, formula., ..., evaluate = TRUE)
update_resp(object, newresp, ..., evaluate = TRUE)

update_formulas(object, formula., ...)

# <fit object>$respName[s] : see Details.
```

Arguments

`object`	A return object from an HLfit call.
`formula.`	A standard `formula`; or a `formula` with a peculiar syntax only describing changes to the original model formula (see `update.formula` for details); or (for multivariate-response models) a list of `formula` of such types.
`newresp`	New response vector.
`...`	Additional arguments to the call, or arguments with changed values. Use *name* = `NULL` to remove the argument with given *name*.
`evaluate`	If TRUE, evaluate the new call else return the call.

Details

Controlling response updating: Updating the data may be tricky when the response specified in a formula is not simply the name of a variable in the data. For example, if the response was specified as `I(foo^2)` the variable `foo` is not what `simulate.HLfit` will simulate, so `foo` should not be updated with such simulation results, yet this is what should be updated in the `data`. For

some time **spaMM** has handled such cases by using an alternative way to provide updated response information, but this has some limitations. So **spaMM** now update the data after checking that this is correct, which the consequence that when response updating is needed (notably, for bootstrap procedures), the response should preferably be specified as the name of a variable in the data, rather than a more complicated expression.

However, in some cases, dynamic evaluation of the response variable may be helpful. For example, for bootstrapping hurdle models, the zero-truncated response may be specified as `I(count[presence>0] <- NA; count)` (where both the zero-truncated `count` and binary `presence` variables are both updated by the bootstrap simulation). In that case the names of the two variables to be updated is provided by setting (say)
`<fit object>$respNames <- c("presence", "count")`
for an hurdle model fit as a bivariate-response model, with first submodel for presence/absence, and second submodel for zero-truncated response. A full example is developed in the "Gentle introduction" to **spaMM** (https://gitlab.in2p3.fr/francois.rousset/spamm-ref/-/blob/master/vignettePlus/spaMMintro.pdf). Alternatively for univariate-response fits, use
`<fit object>$respName <- "count"`

Controlling formula updating: Early versions of **spaMM**'s `update` method relied on `stats::update.formula` whose results endorse `stats`'s (sometimes annoying) convention that a formula without an explicit intercept term actually includes an intercept. `spaMM::update.HLfit` was then defined to avoid this problem. **Formula updates should still be carefully checked**, as getting them perfect has not been on the priority list.

Various post-fit functions from base R may use `update.formula` directly, rather than using automatic method selection for `update`. `update.formula` is not itself a generic, which leads to the following problem. To make `update.formula()` work on multivariate-response fits, one would like to be able to redefine it as a generic, with an `HLfit` method that would perform what `update_formulas` does, but such a redefinition appears to be forbidden in a package distributed on CRAN. Instead it is suggested to define a new generic `spaMM::update`, which could have a `spaMM::update.formula` as a method (possibly itself a generic). This would be of limited interest as the new `spaMM::update.formula` would be visible to `spaMM::update` but not to `stats::update`, and thus the post-fit functions from base R would still not use this method.

Safe updating: `update(<fit>, ...)`, as a general rule, is tricky. `update` methods are easily affected in a non-transparent way by changes in variables used in the original call. For example

```
foo <- rep(1,10)
m <- lm(rnorm(10)~1, weights=foo)
rm(foo)
update(m, .~.) # Error
```

To avoid such problems, **spaMM** tries to avoid references to variables in the global environment, by enforcing that the data are explicitly provided to the fitting functions by the `data` argument, and that any variable used in the `prior.weights` argument is in the data.

Bugs can also result when calling `update` on a fit produced within some function, say function `somefn` calling `fitme(data=mydata,...)`, as e.g. `update(<fit>)` will then seek a global variable `mydata` that may differ from the fitted `mydata` which was local to `somefn`.

Value

`update.formula(object)` returns an object of the same nature as `formula(object)`. The other

functions and methods return an HLfit fit of the same type as the input object, or a call object, depending on the `evaluate` value. **Warning:** The object returned by `update_resp` cannot be used safely for further programming, for the reason explained in the Details section.

See Also

See also `HLCor`, `HLfit`.

Examples

```
data("wafers")
## First the fit to be updated:
wFit <- HLfit(y ~X1*X3+X2*X3+I(X2^2)+(1|batch),family=Gamma(log),
          resid.model = ~ X3+I(X3^2) ,data=wafers)

newresp <- simulate(wFit)
update_resp(wFit,newresp=newresp)

# For estimates given by Lee et al., Appl. Stochastic Models Bus. Ind. (2011) 27:  315-328:
# Refit with given beta or/and phi values:

betavals <- c(5.55,0.08,-0.14,-0.21,-0.08,-0.09,-0.09)
# reconstruct fitted phi value from predictor for log(phi)
Xphi <- with(wafers,cbind(1,X3,X3^2)) ## design matrix
phifit <- exp(Xphi %*% c(-2.90,0.1,0.95))
upd_wafers <- wafers
designX <- get_matrix(wFit)
upd_wafers$off_b <- designX %*% betavals
update(wFit,formula.= . ~ offset(off_b)+(1|batch), data=upd_wafers,
       ranFix=list(lambda=exp(-3.67),phi=phifit))

## There are subtlety in performing REML fits of constrained models,
##   illustrated by the fact that the following fit does not recover
##   the original likelihood values, because dispersion parameters are
##   estimated but the REML correction changes with the formula:
upd_wafers$off_f <- designX %*% fixef(wFit) ## = predict(wFit,re.form=NA,type="link")
update(wFit,formula.= . ~ offset(off_f)+(1|batch), data=upd_wafers)
#
## To maintain the original REML correction, Consider instead
update(wFit,formula.= . ~ offset(off_f)+(1|batch), data=upd_wafers,
       REMLformula=formula(wFit))  ## recover original p_v and p_bv
## Alternatively, show original wFit as differences from betavals:
update(wFit,formula.= . ~ . +offset(off_f), data=upd_wafers)
```

---

vcov *Extract covariance or correlation components from a fitted model object*

Description

summary(<fit object>)$beta_table returns the table of fixed-effect coefficients as it is printed by summary, including standard errors and t-values.

vcov returns the variance-covariance matrix of the fixed-effects coefficients (cf See Also for related computations involving random effects).

Corr by default returns correlation matrices of random effects (though see Details for user-defined correlation models).

VarCorr returns (co)variance parameters of random effects, and optionally the residual variance(s), from a fit object, in different possible formats (see Details). Other extractors to consider are get_ranPars and get_inits_from_fit, the latter providing parameters in a form suitable for initializing a fit.

The covariance matrix of residuals of a fit can be obtained as a block of the hat matrix (get_matrix(., which="hat_matrix")). This is (as other covariances matrices above) a matrix of expected values, generally assuming that the fitted model is correct and that its parameters are "well" estimated, and should not to be confused with the computation of diagnostic correlations among inferred residuals of a fit.

Usage

```
## S3 method for class 'HLfit'
vcov(object, ...)
## S3 method for class 'HLfit'
VarCorr(x, sigma = 1, add_residVars=TRUE, verbose=TRUE, format="data.frame",...)
Corr(object, A=TRUE, cov2cor.=TRUE, ...)
```

Arguments

object, x	A fitted model object, inheriting from class "HLfit", as returned by the fitting functions in spaMM.
add_residVars	Boolean; whether to include residual variance information in the returned table.
sigma	ignored argument, included for consistency with the generic function.
format	Selects the return format. See Details.
A	Boolean: Whether to return the correlation matrix described by the **AL** matrix product, when there is an **A** matrix (as for IMRF terms; see random-effects).
cov2cor.	Boolean: Whether to convert covariance matrices to correlation matrices (see Details).
verbose	Boolean: Whether to print some notes.
...	Other arguments that may be needed by some method.

Details

Any matrix returned by the Corr extractor is by default the unconditional correlation matrix of a vector "**ALv**" of random effects (as defined in random-effects).

But it may also be an heteroscedastic matrix for some random effects if cov2cor. is set to FALSE. In particular, the IMRF and MaternIMRFa models are by default defined in terms of the inverse of an

heteroscedastic covariance matrix (with tcrossprod factor **L**), and of a **A** matrix of weights. The product **AL** will be the tcrossprod factor of a covariance matrix rather than a correlation matrix, unless a non-default normalization was requested when declaring the random-effect terms. User-defined random-effects models may also be heteroscedastic. In all these cases Corr will by default return the correlation matrix, by applying cov2cor to the tcrossproduct.

For format="data.frame" (the default), VarCorr.HLfit returns a data.frame, whose format was inspired by that of objects produced by nlme::VarCorr.lme, in particular including columns with consistent names for easier extraction; but with several differences (e.g. numbers are converted to character strings in the nlme::VarCorr.lme, but not here).

For the alternative format="merMod", VarCorr.HLfit returns an object whose format is consistent with that of objects produced by lme4:::VarCorr.merMod, and inheriting from the class "VarCorr.merMod", so that the as.data.frame and print methods defined in **lme4** for this class can be selected for this returned object.

Value

vcov returns a matrix.

Corr returns a list, for the different random effect terms. For each random-effect term with nontrivial correlation structure, the returned element is a matrix, returned in base matrix format or in some class from **Matrix**. Otherwise the it is an information message.

By default, VarCorr returns either NULL (if no variance to report, as for a poisson GLM) or a data frame with columns for the grouping factor, term, variance of random effect, standard deviation (the root of the variance), and optionally for correlation of random effect in random-coefficient terms. Information about the residual variance is optionally included as the last row(s) of the data frame, when relevant (gaussian- or Gamma-response models with single scalar parameter; beware the meaning of the residual variance parameter for Gamma-response models). One may have to consult the summary of the fit object to check the meaning of the contents of this data frame (e.g., of 'variance' coefficients for non-gaussian random effects).

VarCorr(., format="merMod") returns a list of the same format as returned by lme4:::VarCorr.merMod.

Some variance parameters may be removed from the VarCorr output, with a message, such as the slope of the linear predictor describing the correlation model of an adjacency term (see autoregressive). The rare user of such parametrization should not consider this as a stable feature.

See Also

get_inits_from_fit and get_ranPars.

get_matrix(., which="beta_v_cov") for the joint covariance matrix of estimates/predictions of fixed-effect coefficients and random effects; and
get_matrix(., which="v_condcov") for the covariance matrix of predictions of random effects given fixed effects (the variances corresponding to the condsd reported in some **lme4** output). Both of these computations refer to the random effects **v** as defined in random-effects.

Examples

```
data("wafers")
m1 <- HLfit(y ~ X1+X2+(1|batch), resid.model = ~ 1 ,data=wafers, method="ML")
```

```
vcov(m1)

# Example from VarCorr() documentation in 'nlme' package
data("Orthodont",package = "nlme")
sp1 <- fitme(distance ~ age+(age|Subject), data = Orthodont, method="REML")
VarCorr(sp1)
VarCorr(sp1, format="merMod")
```

---

verbose *Details of the* verbose *argument of various functions*

Description

This partially documents the usage of the verbose argument in tow contexts: (1) as an argument of the fitting functions; (2) as an argument of confint.HLfit.

The verbosity of the fitting functions is controlled by the verbose["TRACE"] value. The default is TRACE=FALSE (or 0) which is self-explanatory. TRACE=TRUE (or 1) shows values of outer-estimated parameters (and possibly fixed values of parameters that would be outer-estimated), some cryptic progress bar, and the attained value of the likelihood objective function (see inner-vs-outer for these two types of estimates). See Details for other values.

Tracing the progress of residual-dispersion models is distinctly controlled by verbose["phifit"]: see phi-resid.model.

If the fitted model includes a residual-dispersion model, some tracing output for the latter may be confusingly intermingled with tracing output of the mean-response model. The Details below are valid only for the mean-response model.

confint(.,verbose=.) will not affect the verbosity of the fits performed by confint (but will rather affect messages specific to the progress of the confint procedure), **unless** verbose is a named vector. In particular, verbose["TRACE"] may affect the fit (but this has been implemented only for fit objects inheriting from class pois4mlogit). verbose["confint"] may then be used to control confint. Thus, for pois4mlogit fits,
verbose=FALSE and verbose=c(confint=FALSE) will affect only confint per se, in an equivalent way
verbose=c(TRACE=FALSE) will affect only the fits performed to find the intervals, and
verbose=c(TRACE=FALSE, confint=FALSE) will affect both.
Keep in mind that the verbosity of pois4mlogit fits is also controlled by a progress argument.

Details

0<TRACE<1 only shows the cryptic progress bar.
TRACE=1 was explained above.
TRACE=2 will further show information about the progress of Levenberg-Marquardt-like steps for linear-predictor coefficients.
TRACE=3 will further show information about the progress of distinct Levenberg-Marquardt-like steps random effects given fixed-effect coefficients.
TRACE=4 will further show cryptic information about which matrix computations are requested.
TRACE=5 will further report (through a call to the base trace function) the str(.) of the results of

such matrix computations.
TRACE=6 will further pause between iterations of the reweighted least-squares algorithm, allowing a browser session to be called.

(assuming no changes since this was first defined)

`wafers` *Data from a resistivity experiment for semiconductor materials.*

Description

This data set was reported and analyzed by Robinson et al. (2006) and reanalyzed by Lee et al. (2011). The data "deal with wafers in a single etching process in semiconductor manufacturing. Wafers vary through time since there are some variables that are not perfectly controllable in the etching process. For this reason, wafers produced on any given day (batch) may be different from those produced on another day (batch). To measure variation over batch, wafers are tested by choosing several days at random. In this data, resistivity is the response of interest. There are three variables, gas flow rate (x1), temperature (x2), and pressure (x3) and one random effect (batch or day)." (Lee et al 2011).

Usage

```
data("wafers")
```

Format

The data frame includes 198 observations on the following variables:

y resistivity.

batch batch, indeed.

X1 gas flow rate.

X2 temperature.

X3 pressure.

Source

This data set was manually pasted from Table 3 of Lee et al. (2011). Transcription errors may have occurred.

References

Robinson TJ, Wulff SS, Montgomery DC, Khuri AI. 2006. Robust parameter design using generalized linear mixed models. Journal of Quality Technology 38: 38–65.

Lee, Y., Nelder, J.A., and Park, H. 2011. HGLMs for quality improvement. Applied Stochastic Models in Business and Industry 27, 315-328.

Examples

```
## see examples in the main Documentation page for the package.
```

welding *Welding data set*

Description

The data give the results of an unreplicated experiment for factors affecting welding quality conducted by the National Railway Corporation of Japan (Taguchi and Wu, 1980, cited in Smyth et al., 2001). It is a toy example for heteroscedastic models and is also suitable for illustrating fit of overparameterized models.

Usage

```
data("welding")
```

Format

The data frame includes 16 observations on 10 variables:

Strengh response variable;

... nine two-level factors.

Source

The data were downloaded from http://www.statsci.org/data/general/welding.txt on 2014/08/19 and are consistent with those shown in table 5 of Bergman and Hynén (1997).

References

Bergman B, Hynén A (1997) Dispersion effects from unreplicated designs in the 2^{k-p} series. Technometrics, 39, 191–98.

Smyth GK, Huele AF, Verbyla AP (2001). Exact and approximate REML for heteroscedastic regression. Statistical Modelling 1, 161-175.

Taguchi G, Wu Y (1980) Introduction to off-line quality control. Nagoya, Japan: Central Japan Quality Control Association.

Examples

```
data("welding")
## toy example from Smyth et al.
fitme(Strength ~ Drying + Material,resid.model = ~ Material+Preheating ,data=welding, method="REML")
## toy example of overparameterized model
fitme(Strength ~ Rods+Thickness*Angle+(1|Rods),resid.model = ~ Rods+Thickness*Angle ,data=welding)
```

WinterWheat *Example of yield stability analysis*

Description

Translation of an example that may be found at
https://www.r-bloggers.com/2019/06/genotype-experiments-fitting-a-stability-variance-model-with-r/
based on yield of eight durum wheat genotypes over seven years, following a randomised block design with three replicates. A genotype-in-year random effect is used to quantify genotype-by-environment interactions. In the first fit (`constvar`), the variance of this random effect is constant over genotypes. In the second fit (`varvar`), different variances are fitted for the distinct genotypes, to assess the relative stability of yield of the different genotypes over environments. This second model can be fitted as a constrained random-coefficient model, where the constraint describes a diagonal covariance matrix for the random coefficients.

This example uses the fact that the argument `fixed=list(ranCoefs=<...>)` can be used to fit a covariance matrix with an arbitrary set of constrained elements. Only elements left as 'NA' (here the diagonal elements of the matrix) are fitted.

Examples

```
if (spaMM.getOption("example_maxtime")>1.5 &&
      requireNamespace("agridat", quietly = TRUE)) {

data("onofri.winterwheat", package="agridat")

(constvar <- fitme(
  yield ~ gen + (1|year) + (1|block %in% year)+(1|gen %in% year),
  data=onofri.winterwheat, method="REML"))

# Diagonal matrix of NA's, represented as vector for its lower triangle:
ranCoefs_for_diag <- function(nlevels) {
 ## Conceptual version
 # diagmat <- matrix(NA, ncol=nlevels,nrow=nlevels)
 # diagmat[lower.tri(diagmat,diag=FALSE)] <- 0
 # return(diagmat[lower.tri(diagmat,diag=TRUE)])
 ## which amounts to:
 vec <- rep(0,nlevels*(nlevels+1L)/2L)
 vec[cumsum(c(1L,rev(seq(nlevels-1L)+1L)))] <- NA
 vec
}

(varvar <- fitme(
  yield ~ gen + (1|year) + (1|block %in% year)+(0+gen|gen %in% year), method="REML",
  data=onofri.winterwheat, fixed=list(ranCoefs=list("3"=ranCoefs_for_diag(8L)))))

}
```

wrap_parallel *Selecting interfaces for parallelisation*

Description

spaMM implements three interfaces for parallelisation. Depending on their arguments, either serial computation (default), a socket cluster (parallelisation default), or a fork cluster (available in linux and alike operating systems) can be used by all interfaces.

dopar is called by default by its bootstrap procedures, and dofuture has been developed as an alternative, whose use is controlled by spaMM.options(wrap_parallel="dofuture") (versus the default, spaMM.options(wrap_parallel="dopar"). combinepar is the third and more recent interface; it is not a formally supported wrap_parallel option because its additional functionalities are of no use in **spaMM**'s bootstrap procedures.

dopar is based on a patchwork of backends: for socket clusters, depending whether the **doSNOW** package is attached, foreach or pbapply is called (**doSNOW** allows more efficient load balancing than pbapply); for fork clusters, parallel::mclapply is used. This makes it impossible to ensure consistency of options accross computation environments, notably of enforcing the .combine control of foreach; and this makes it error-prone to ensure identical control of random number generators in all cases (although dopar and combinepar still aim to ensure the latter control).

By contrast, dofuture is based only on the **future** and **future.apply** packages, in principle allowing a single syntax to control of random number generator across the different cases, hence repeatable results across them. This does **not** make a difference for bootstrap computations in **spaMM** as the bootstrap samples are never simulated in parallel: only refitting the models is performed in parallel, and fit results do not depend on random numbers. Further, the **future**-based code for socket clusters appears significantly slower than the one used by dopar. For these reasons, the latter function is used by default by **spaMM**.

combinepar is a third and more recent approach designed to address the other issue: it always uses **foreach** so that the .combine control is consistently enforced. It uses **future** only when no alternative is available to produce a progress bar (namely, for socket clusters when **doSNOW** is not available).

X.GCA *Fixed-effect terms for dyadic interactions*

Description

X.GCA and X.antisym are functions which, when called in a model formula, stand for terms designed to represent the effet of symmetric interactions between pairs of individuals (order of individuals in the pair does not matter). antisym likewise represents anti-symmetric interactions (the effect of reciprocal ordered pairs on the outcome are opposite, as in the so-called Bradley-Terry models). These constructs all account for multiple membership, i.e., the fact that the same individual may act as the first or the second individual among different pairs, or even within one pair if this makes sense).

The outcome of an interaction between a pair i, j of agents is subject to a symmetric overall effect a_{ij} when the effect "on" individual i (or viewed from the perspective of individual i) equals the effect on j: $a_{ij} = a_{ji}$. This may result from the additive effect of individual effects a_i and a_j: $a_{ij} = a_i + a_j$. A `X.GCA` call represents such symmetric additive effects. Conversely, antisymmetry is characterized by $a_{ij} = a_i - a_j = -a_{ji}$ and is represented by a `X.antisym` call. See the `diallel` documentation for similar constructs for random effects, for additional comments on semantics (e.g. about "GCA"), and for further references.

If individual-level factors ID1 + ID2 were included in a formula for dyadic interactions, this would result in different coefficients being fitted for the same level in each factor. By contrast, the present constucts ensure that a single coefficient is fitted for the same-named levels of factors ID1 and ID2.

Usage

```
X.GCA(term, contr="contr.treatment", ...)
X.antisym(term, contr="contr.treatment", ...)
```

Arguments

`term`	an expression of the form <.>:<.> where each <.> represents a factor (or a variable that will automaticall be converted to a factor) present in the `data` of the fitting function call.
`contr`	The `contrasts` used. Only the default and `"contr.sum"` are implemented.
`...`	For programming purposes, not documented.

Details

The fixed-effect terms (`GCA(Par1,Par2)`, etc) from the **lmDiallel** package (Onofri & Terzaroli, 2021), defined by functions returning design matrices for usage with `stats:lm`, work in a formula for a **spaMM** fit, and users can define use such functions as templates for additional functions that will work similarly. However, not all post-fit functions will handle terms defined in this way well: checking robustness of `predict` on small and permuted `newdata`, as shown in the Examples, is a good test. Such problems happen because the formula-handling machinery of R handles terms represented by either a matrix or a factor, while both the model matrix and the factor information used to construct dyadic-interaction terms are needed to correctly predict, with new data, from models including such terms.

The presently designed functions are defined to solve this issue. By using such functions as template, users can define additional functions with the same return format (as further explained in the documented source code of `X.antisym`), which will allow them to perform correct predictions from fitted models.

Value

The functions return design matrices with additional class `"factorS"` and attributes `"call"` and `"spec_levs"`.

References

Onofri A., Terzaroli N. (2021) lmDiallel: Linear fixed effects models for diallel crosses. Version 0.9.4. https://cran.r-project.org/package=lmDiallel.

See Also

ranGCA and diallel for random effects terms representing symmetric or antisymmetric dyadic interactions.

Examples

```
#### Simulate dyadic data

set.seed(123)
nind <- 10       # Beware data grow as O(nind^2)
x <- runif(nind^2)
id12 <- expand.grid(id1=seq(nind),id2=seq(nind))
id1 <- id12$id1
id2 <- id12$id2
u <-  rnorm(nind,mean = 0, sd=0.5)

## additive individual effects:
y <-  0.1 + 1*x + u[id1] +  u[id2] + rnorm(nind^2,sd=0.2)

## anti-smmetric individual effects:
t <-  0.1 + 1*x + u[id1] - u[id2] + rnorm(nind^2,sd=0.2)

dyaddf <- data.frame(x=x, y=y, t=t, id1=id1,id2=id2)
# : note that this contains two rows per dyad, which avoids identifiability issues.

# Enforce that interactions are between distinct individuals (not essential for the fit):
dyaddf <- dyaddf[- seq.int(1L,nind^2,nind+1L),]


# Fits:

(addfit <- fitme(y ~x +X.GCA(id1:id2), data=dyaddf))

(antifit <- fitme(t ~x +X.antisym(id1:id2), data=dyaddf))

if (FALSE) { #### check of correct handling by predict():

  # First scramble the data so that input factors are in no particular order
  set.seed(123)
  dyaddf <- dyaddf[sample(nrow(dyaddf)),]

  addfiti <- fitme(y ~x +X.GCA(id1:id2), data=dyaddf)
  foo <- rev(2:4)
  p1 <- predict(addfiti)[foo]
  p2 <- predict(addfiti, newdata=dyaddf[foo,])
  diff(range(p1-p2))<1e-10    # must be TRUE
}
```

X2X *Fitting fixed-effects coefficients shared between submodels*

Description

X2X is an argument of the fitmv function, whose use is illustrated in the Examples below. By providing an X2X matrix **M**, the default fixed-effect design matrix **X** of the model is replaced by **XM**. The fixed-effect term $\mathbf{X}\beta$ of the linear predictor is then modified to $\mathbf{XM}\beta^*$ for a new vector β^* of fixed-effect coefficients, distinct from the vector β of coefficients of the default fit.

In the examples, the vector β of fixed-effect coefficients of the default fit, without X2X argument, has four distinct elements, including one Intercept for each of the two sub-models. The **M** matrix is used to declare that the Intercept coefficient is identical in the two sub-models, so **M** has three columns and four rows.

M must have column names, labeling the β^* coefficients.

Because specifying the **M** matrix can be laborious and error-prone, the helper function genX2X is provided. The typical way of using it, illustrated below, is X2X=genX2X(matches=*list*), with the second argument of the function missing. This argument will be provided internally, so the user does not have to find its correct form. Without this argument, the function returns its own call, which is not of any immediate interest for the user.

The matches list is a named list: the names will become some of the column names of the matrix product (the new regressor variables) and each element should be a vector of column names of the original **X** matrix.

A common source of error in specifying X2X is not correctly predicting the latter column names (for example, forgetting the appended submodel indices). Under the (not otherwise necessary, but practically useful) premise that vector elements not matching original column names are wrongly predicted column names, genX2X checks that all vector elements match original column names.

The names of regressors for interaction terms, in particular, may be wrongly predicted because even if the user-level model formula contains $x:y$, a base R function may relabel it as $y:x$ (this happens if y appears before x in the formula), and thus the matches vector elements should be of the form $y:x_s$ for submodels s that are thus affected. Running attr(terms(*submodel formula*),"term.labels") will show whether such reversals of variables occur for some interaction terms.

Usage

```
genX2X(matches, names_ori)
```

Arguments

matches	A named list, whose elements are vectors of names (character strings). The list names are those of some of the coefficients to be fitted (notably, the names to be given to the new, shared coefficients); and the vector elements are some names of coefficients of the original model (the one without an X2X argument) that should be constrained to be identical to the corresponding shared coefficient.
names_ori	Either missing, or the names of all coefficients of the default model (the one without X2X argument).

Value

genX2X returns a matrix if names_ori is provided, and returns the call to the genX2X function is this argument is missing.

See Also

vignette("Multinomial-logit", package = "spaMM") for more examples.

Examples

```
### Data preparation
data(clinics)
climv <- clinics
(fitClinics <- HLfit(cbind(npos,nneg)~treatment+(1|clinic),
                     family=binomial(),data=clinics))
set.seed(123)
climv$np2 <- simulate(fitClinics, type="residual")

### fits

## Default fit without 'X2X' argument
(mvfit <- fitmv(
  submodels=list(mod1=list(formula=cbind(npos,nneg)~treatment+(1|clinic),family=binomial()),
                 mod2=list(formula=np2~treatment+(1|clinic),
                           family=poisson(), fixed=list(lambda=c("1"=1)))),
   data=climv))

## Fits with 'X2X' argument

# Suppose we want to fit the same intercept for the two submodels
# (there may be cases where this is meaningful, even if not here).
# The original fit has four coefficients corresponding to four columns
# of fixed-effect design matrix:

head(design_X <- model.matrix(mvfit))
#      (Intercept)_1 treatment_1 (Intercept)_2 treatment_2
# [1,]             1           1             0           0
#      ...
# where '_1' or '_2' identifies the submodel to which each coefficient belongs.

# The three coefficients of the intended model are (say)
# "(Intercept)" "treatment_1" "treatment_2"
# We build a matrix that relates the original 4 coefficients to these 3 ones:

X_4to3 <-
  matrix(c(1,0,0,
           0,1,0,
           1,0,0,
           0,0,1), nrow=4, ncol=3, byrow=TRUE,
         dimnames=list(NULL, c("(Intercept)","treatment_1","treatment_2")))

# defined such that design_X %*% X_4to3 will be the design matrix
# for the intended model, and the single "(Intercept)" coefficient
# of the three-parameter model will operate as a shared estimate
# of the "(Intercept)_1" and "(Intercept)_2" coefficients
# of the original 4-coefficients model, as intended.
```

```
# To define such matrices, it is *strongly advised* to either fit
# the unconstrained model first, and to examine the structure
# of its model matrix (as shown above), or to use genX2X().

# The 'X2X' argument provides the matrix:

(mvfit3m <- fitmv(
   submodels=list(mod1=list(formula=cbind(npos,nneg)~treatment+(1|clinic),family=binomial()),
                  mod2=list(formula=np2~treatment+(1|clinic),
                            family=poisson(), fixed=list(lambda=c("1"=1)))),
    X2X = X_4to3,
    data=climv))

# => the column names of 'X_4to3' are the fixed-effect names in all output.

# Alternatively, the 'X2X' argument provides a genX2X() call:

(mvfit3g <- fitmv(
   submodels=list(mod1=list(formula=cbind(npos,nneg)~treatment+(1|clinic),family=binomial()),
                  mod2=list(formula=np2~treatment+(1|clinic),
                            family=poisson(), fixed=list(lambda=c("1"=1)))),
    X2X = genX2X(list("(Intercept)"=c("(Intercept)_1","(Intercept)_2"))),
    data=climv))

# => the last two fits are equivalent (although the order of the coefficients may differ).
# The internally produced 'X2X' matrix is that provided by
genX2X(list("(Intercept)"=c("(Intercept)_1","(Intercept)_2")),
       names_ori=colnames(design_X) )
```

`ZAXlist` *S4 classes for structured matrices*

Description

A `ZAXlist` object is a representation of the "`ZAL`" matrix as an S4 class holding a list of descriptors of each `ZAL` block for each random effect. Its `envir` slot may store a cached copy of the `ZAL` matrix. The internal `.get_bind` extractor gets the `ZAL` matrix from a `ZAXlist` object, using the cached copy if already available, and computing and caching it otherwise.

A `Kronfacto` object is a representation of a Kronecker product as an S4 class holding its factors. Methods defined for this class may avoid the computation of the Kronecker product as an actual matrix of large dimensions.

This documentation is for development purposes and may be incomplete. The objects and methods are not part of the programming interface and are subject to modification without notice.

Usage

```
# new("ZAXlist", LIST=., as_matrix=., envir=.)
# new("Kronfacto", BLOB=.)
```

Slots

`LIST`: A list whose each element is a ZAL block represented as either a `(M|m)atrix`, or a list with two elements (and additional class `ZA_QCHM`): `ZA`, and the `Cholesky` factor `Q_CHMfactor` of the precision matrix (`L=solve(Q_CHMfactor,system="Lt")`).

`as_matrix`: boolean: whether ZAL should be stored as a base `matrix`. Otherwise, a sparse **Matrix** format is used.

`envir`: An environment. May store a cached copy `mMat` of the `ZAL` matrix.

`BLOB`: An environment holding `lhs` and `rhs`, the factors of the Kronecker product, and other objects initialized as promises. See the source code of the non-exported `.def_Kranfacto` constructor for further information.

Index

∗ **datagen**
simulate.HLfit, 206
∗ **datasets**
adjlg, 5
arabidopsis, 16
blackcap, 25
clinics, 27
freight, 89
Gryphon, 102
Leuca, 119
Loaloa, 123
salamander, 200
scotlip, 202
seaMask, 203
seeds, 204
wafers, 238
welding, 239
∗ **family**
multinomial, 148
∗ **hplot**
mapMM, 131
plot.HLfit, 166
∗ **htest**
anova, 13
fixedLRT, 85
get_RLRsim_args, 97
gof, 99
LRT, 125
spaMM_boot, 218
∗ **log-linear**
spaMM_glm.fit, 222
∗ **logistic**
spaMM_glm.fit, 222
∗ **loglinear**
spaMM_glm.fit, 222
∗ **manip**
multinomial, 148
∗ **models**
AIC, 6
autoregressive, 20
beta_resp, 23
betabin, 22
CauchyCorr, 26
COMPoisson, 28
corr_family, 55
MaternCorr, 136
MSFDR, 143
negbin, 153
negbin1, 154
Poisson, 176
spaMM_glm.fit, 222
tweedie, 229
∗ **model**
corrHLfit, 50
fitme, 76
fitmv, 79
HLCor, 106
HLfit, 109
make_scaled_dist, 129
multIMRF, 144
multinomial, 148
∗ **package**
spaMM, 210
∗ **print**
summary.HLfit, 226
∗ **regression**
beta_resp, 23
betabin, 22
COMPoisson, 28
get_RLRsim_args, 97
is_separated, 118
negbin, 153
negbin1, 154
Poisson, 176
spaMM_glm.fit, 222
tweedie, 229
∗ **spatial**
autoregressive, 20

CauchyCorr, 26
corr_family, 55
MaternCorr, 136
multIMRF, 144
spaMM, 210
∗ **ts**
ARp, 17
autoregressive, 20
.eval_replicate2 *(*eval_replicate*)*, 71
%*%,Kronfacto,numeric-method *(*ZAXlist*)*, 246
%*%,ZAXlist,Matrix-method *(*ZAXlist*)*, 246
%*%,ZAXlist,matrix-method *(*ZAXlist*)*, 246
%*%,ZAXlist,numeric-method *(*ZAXlist*)*, 246
%*%,numeric,Kronfacto-method *(*ZAXlist*)*, 246
%*%,numeric,ZAXlist-method *(*ZAXlist*)*, 246
%*%-methods *(*ZAXlist*)*, 246
'DHARMa-on-HLfit' *(*plot.HLfit*)*, 166

adjacency, *56*, *84*, *107*, *108*, *115*, *145*, *211*
adjacency *(*autoregressive*)*, 20
adjlg, 5
adjlgMat *(*adjlg*)*, 5
adjustcolor, *215*
AIC, 6
AIC.HLfit, *67*
algebra, 10, *37*, *38*, *47*, *114*, *161*, *162*, *206*
aliases, 12, *80*, *172*, *175*
AMatrices *(*random-effects*)*, 189
anova, 13, *118*
anova.glm, *14*
anova.HLfit, *19*, *128*
anova.lm, *14*
antisym *(*diallel*)*, 57
AR1, *84*, *108*, *145*, *211*
AR1 *(*autoregressive*)*, 20
arabidopsis, 16, *210*
ARMA, *211*
ARMA *(*ARp*)*, 17
ARp, 17, *20*, *40*, *42*, *44*, *47*, *96*, *211*
as.data.frame, *223*
as_LMLT, *14*, *15*, 19, *62*, *68*, *87*, *178*
as_precision *(*covStruct*)*, 56
autoregressive, 20, *108*

barstyle, *65*, *208*, *220*
barstyle *(*options*)*, 160
besselK, *137*
Beta *(*HLfit*)*, 109
Beta-distribution-random-effects *(*random-effects*)*, 189
beta_resp, *22*, 23, *121*, *142*, *156*, *210*
beta_table, *226*
beta_table *(*vcov*)*, 234
betabin, 22, *142*, *156*, *210*
binomialize *(*multinomial*)*, 148
blackcap, 25
bobyqa, *160*
boot, *221*
boot.ci, *35*
box, *217*
boxcox *(*transffit*)*, 227
bs, *210*

CAR *(*autoregressive*)*, 20
Cauchy, *84*, *211*
Cauchy *(*CauchyCorr*)*, 26
CauchyCorr, 26, *108*
Cholesky, *247*
class:Kronfacto *(*ZAXlist*)*, 246
class:LMLTslots *(*as_LMLT*)*, 19
class:missingOrNULL *(*ZAXlist*)*, 246
class:ZAXlist *(*ZAXlist*)*, 246
clinics, 27
clusterExport, *220*
clusterSetRNGStream, *63*, *65*
coef.corMatern *(*corMatern*)*, 38
coef.HLfit *(*extractors*)*, 72
coef<-.corMatern *(*corMatern*)*, 38
col2rgb, *170*
combinepar, *241*
combinepar *(*dopar*)*, 64
COMPoisson, 28, *161*, *211*
composite-ranef, 30
confint, *35*, *172*
confint *(*confint.HLfit*)*, 34
confint.HLfit, 34, *237*
contour, *133*, *218*
contourplot, *218*
contrasts, *242*
control.HLfit, 36, *110*, *230*
convergence, 37
corFactor.corMatern *(*corMatern*)*, 38
corMatern, 38, *137*
corMatrix.corMatern *(*corMatern*)*, 38

Corr, *95*, *112*
Corr *(vcov)*, 234
corr_family, 55
corrFamily, *18*, 40, *45*, *46*, *53*, *59*, *138*, *139*, *211*, *212*
corrFamily-definition, 45
corrFamily-design, 46
corrHLfit, 49, *113*, *210*
corrMatrix, *11*, 52, *107*, *108*, *211*
corrPars, *156*
corrPars *(fixed)*, 83
covStruct, *11*, *44*, *52*, 56, *107*, *163*, *190*
crossprod,Kronfacto,Matrix-method *(ZAXlist)*, 246
crossprod,Kronfacto,matrix-method *(ZAXlist)*, 246
crossprod,Kronfacto,numeric-method *(ZAXlist)*, 246
crossprod,ZAXlist,Matrix-method *(ZAXlist)*, 246
crossprod,ZAXlist,matrix-method *(ZAXlist)*, 246
crossprod,ZAXlist,numeric-method *(ZAXlist)*, 246
crossprod-methods *(ZAXlist)*, 246

dev_resids, *197*
dev_resids *(extractors)*, 72
deviance, *197*
deviance *(extractors)*, 72
df.residual *(extractors)*, 72
df.residual.HLfit, *62*
DHARMa *(post-fit)*, 178
diagnose_conv *(div_info)*, 60
diallel, *42*, *44*, 57, *70*, *211*, *242*, *243*
dim.Kronfacto *(ZAXlist)*, 246
dist, *131*
distMatrix *(corrMatrix)*, 52
div_info, 60
DoF, *9*, 61, *113*
dofuture, 62, *66*, *241*
dopar, *7*, *63*, 64, *241*
drop1 *(drop1.HLfit)*, 67
drop1.HLfit, *19*, 67
dyad, 69

Earth *(make_scaled_dist)*, 129
EarthChord *(make_scaled_dist)*, 129
eigen, *140*
emmeans *(post-fit)*, 178
etaFix, *111*
etaFix *(fixed)*, 83
eval_replicate, 71, *126*, *220*
expression, *193*
extractAIC, *7*
extractAIC *(AIC)*, 6
extractors, 72, *111*
extreme_eig, 75

factor, *120*
family, *28*, *51*, *77*, *122*, *153*, *154*, *176*, *177*, *223*, *229*, *231*
family *(extractors)*, 72
filled.mapMM, *203*
filled.mapMM *(mapMM)*, 131
fitme, *50*, 76, *79*, *109*, *113*, *116*, *164*, *165*, *210*
fitmv, *76*, 79, *148*, *149*, *152*, *173*, *206*, *210*, *213*, *244*
fitted *(extractors)*, 72
fitted.HLfitlist *(multinomial)*, 148
fix_predVar, 88
fixed, *77*, 83, *96*, *106*
fixedLRT, *51*, *52*, *72*, 85, *110*, *125*, *128*, *210*
fixef *(extractors)*, 72
formula, *50*, *76*, *109*, *223*
formula *(extractors)*, 72
formula_env *(good-practice)*, 100
freight, 89

Gamma, *117*, *189*
Gamma *(inverse.Gamma)*, 117
genX2X, *80*
genX2X *(X2X)*, 243
geometric *(COMPoisson)*, 28
get_any_IC, *62*, *112*
get_any_IC *(AIC)*, 6
get_cPredVar, 90, *183*, *209*
get_drawSizes *(extractors)*, 72
get_fittedPars, *95*, *97*, *111*, *166*
get_fittedPars *(get_ranPars)*, 95
get_fixefVar *(predict)*, 179
get_inits_from_fit, 92, *95*, *97*, *235*, *236*
get_intervals *(predict)*, 179
get_matrix, *75*, 93, *104*, *105*, *111*, *235*, *236*
get_predCov_var_fix *(predict)*, 179
get_predVar, *90*, *95*
get_predVar *(predict)*, 179
get_rankinfo *(rankinfo)*, 191

get_ranPars, *93*, 95, *112*, *235*, *236*
get_residVar, *95*, *97*, *199*
get_residVar *(*predict*)*, 179
get_respVar *(*predict*)*, 179
get_RLRsim_args, *75*, 97, *127*
get_RLRTSim_args, *178*
get_RLRTSim_args *(*get_RLRsim_args*)*, 97
get_ZALMatrix, *112*
get_ZALMatrix *(*get_matrix*)*, 93
getCovariate.corMatern *(*corMatern*)*, 38
getDistMat *(*extractors*)*, 72
glht *(*post-fit*)*, 178
glm, *22*, *23*, *28*, *111*, *142*, *154*, *223*, *224*, *230*
glm.control, *110*, *223*
glmmPQL, *40*
gof, 99, *168*, *187*, *188*, *197*
good-practice, 100
grad, *157*, *230*
graphical parameters, *217*
grep, *88*
Gryphon, *11*, *57*, 102, *151*, *152*
Gryphon_A *(*Gryphon*)*, 102
Gryphon_df *(*Gryphon*)*, 102
Gryphon_pedigree *(*Gryphon*)*, 102
GxE, *84*, *120*, *212*
GxE *(*WinterWheat*)*, 240

hatvalues, *75*, *104*, *112*, *197*
hatvalues *(*hatvalues.HLfit*)*, 104
hatvalues.HLfit, 104, *165*
hessian, *157*
HLCor, *51*, *77–80*, 106, *113*, *164*, *210*, *234*
HLfit, *36*, *50*, *51*, *77–80*, *106*, 109, *115*, *164*, *185*, *210*, *234*
how, *11*, *113*, 113
hyper, *156*
hyper *(*multIMRF*)*, 144

image, *218*
IMRF, *11*, *20*, *84*, *138*, *190*, *211*, *235*
IMRF *(*multIMRF*)*, 144
Initialize.corMatern *(*corMatern*)*, 38
inits, *38*, *77*, 114
inla.spde2.matern *(*multIMRF*)*, 144
inla.spde2.pcmatern *(*multIMRF*)*, 144
inner-vs-outer, 115
intervals *(*predict*)*, 179
inverse.Gamma, 117, *189*
is_separated, 118

keepInREML, *80*
keepInREML *(*fixed*)*, 83
kronecker, *31*
Kronfacto *(*ZAXlist*)*, 246
Kronfacto-class *(*ZAXlist*)*, 246

landMask *(*seaMask*)*, 203
layout, *217*
Leuca, 119
lev2bool, *84*, 120, *212*
levelplot, *218*
LevenbergM *(*options*)*, 160
LL-family *(*llm.fit*)*, 121
llm.fit, *110*, 121, *197*
lme, *40*
lmerTest *(*post-fit*)*, 178
LMLTslots *(*as_LMLT*)*, 19
LMLTslots-class *(*as_LMLT*)*, 19
Loaloa, *52*, 123, *137*
logDet.corMatern *(*corMatern*)*, 38
logLik, *142*
logLik *(*extractors*)*, 72
logLik.HLfitlist *(*multinomial*)*, 148
lower *(*optimBounds*)*, 158
lower.tri, *84*
LR2R2 *(*pseudoR2*)*, 187
LRT, *13*, *15*, *67*, *72*, *86*, *87*, *98*, 125, *172*

make.link, *22*, *24*, *153*, *155*, *177*
make_scaled_dist, *50*, *77*, *107*, 129, *136*
makeCluster, *181*, *220*
map_ranef *(*mapMM*)*, 131
mapMM, 131
mat_sqrt, *51*, *78*, *80*, *107*, 140
Matern, *38*, *50*, *77*, *84*, *138*, *211*
Matern *(*MaternCorr*)*, 136
MaternCorr, *39*, *108*, *120*, 136
MaternIMRFa, *20*, *42*, *44*, 138, *145*, *211*, *235*
mclapply, *65*
method, *14*, *51*, *67*, *77*, *78*, *86*, *108*, *110*, *111*, 141
missingOrNULL *(*ZAXlist*)*, 246
missingOrNULL-class *(*ZAXlist*)*, 246
mm *(*dyad*)*, 69
mmfn, *59*
mmfn *(*dyad*)*, 69
model.frame.HLfit *(*extractors*)*, 72
model.matrix.HLfit *(*extractors*)*, 72
model.matrix.LMLTslots *(*as_LMLT*)*, 19

model.offset, *223*
MSFDR, 143
multcomp *(post-fit)*, 178
multi, *51*, *77*, *80*, *149*
multi *(multinomial)*, 148
multIMRF, *80*, *96*, 144
multinomial, 148, *211*
mv, *31*, *79*, *81*, 151, *172*, *175*

na.exclude, *223*
na.fail, *223*
na.omit, *181*, *223*
negbin, 153, *212*
negbin1, *23*, *121*, *142*, *153*, 154, *156*, *210*
negbin2, *156*, *210*
negbin2 *(negbin)*, 153
nloptr, *78*
Nmatrix *(scotlip)*, 202
nobs *(extractors)*, 72
ns, *210*
numInfo, *19*, *34, 35*, *143*, 156, *172*, *174*

obsInfo *(method)*, 141
oceanmask *(seaMask)*, 203
offset, *223*
optim, *160*
optimBounds, *75*, *77*, 158
options, 160, *223*

PAIRfn *(dyad)*, 69
palette, *218*
pdep_effects, *181*
pdep_effects *(plot_effects)*, 168
pedigree, *11*, *57*, 162
phi-resid.model, 164
plot.default, *217*
plot.HLfit, 166, *178*
plot_effects, 168, *172*
pois4mlogit, *35*, *127*, *148, 149*, 172, *210*, *237*
Poisson, 176, *211*
poly, *210*
polypath, *203*
post-fit, 178
predict, 179, *182*, *209, 210*
predict.HLfit, *90*, *170*, *173*, *199*, *208*
predict.pois4mlogit *(pois4mlogit)*, 172
Predictor, *50*, *76*, *106*, *109*
Predictor *(covStruct)*, 56
predVar, *31*, *169*, *180, 181*, *183*, 184
preprocess_fix_corr *(predict)*, 179
pretty, *133*
print *(summary.HLfit)*, 226
print.corr_family *(corr_family)*, 55
print.ranef *(extractors)*, 72
print.singeigs *(numInfo)*, 156
prior.weights *(HLfit)*, 109
pseudoR2, 187

qqnorm, *99*

ranCoefs, *156*
ranCoefs *(fixed)*, 83
ranCoefs_for_diag, *84*
ranCoefs_for_diag *(WinterWheat)*, 240
rand.family *(HLfit)*, 109
random-effects, 189
ranef *(extractors)*, 72
ranFix, *50*, *111*
ranFix *(fixed)*, 83
ranGCA, *42*, *243*
ranGCA *(diallel)*, 57
rankinfo, 191
ranPars *(fixed)*, 83
recalc.corMatern *(corMatern)*, 38
refit *(update.HLfit)*, 232
register_cF, 192
regularize, *46*
regularize *(extreme_eig)*, 75
REMLformula, *80, 81*
REMLformula *(HLfit)*, 109
remove_fixef, 193
remove_from_parlist *(get_ranPars)*, 95
reshape2long *(pois4mlogit)*, 172
resid.model, *22, 23*, *96*, *110*, *153*, *155*, *166*, 194
residuals *(residuals.HLfit)*, 196
residuals.glm, *196, 197*
residuals.HLfit, *72*, *75*, *167*, 196
residVar, *75*, *81*, *95*, *97*, *112*, *183*, *185*, 198
respName *(update.HLfit)*, 232
response *(extractors)*, 72
rho.mapping *(make_scaled_dist)*, 129
RLRsim *(post-fit)*, 178
ROI_solve, *118*

salamander, 200
SAR_WWt *(corr_family)*, 55
scotlip, 202

seaMask, *135*, 203
seeds, 204
separation *(*is_separated*)*, 118
set.seed, *207*
setNbThreads, 205
shapiro.test, *100*
sigma, *199*
sigma.HLfit *(*residVar*)*, 198
simulate, *210*
simulate *(*simulate.HLfit*)*, 206
simulate.HLfit, *86*, *90*, 206, *219*, *220*
simulate4boot, *127*
simulate4boot *(*simulate.HLfit*)*, 206
simulate_ranef *(*simulate.HLfit*)*, 206
small_spde *(*multIMRF*)*, 144
spaMM, *50*, *76*, *77*, *109*, *181*, 210
spaMM-conventions, 214
spaMM-package *(*spaMM*)*, 210
spaMM.colors, 215
spaMM.filled.contour, *134*, 216
spaMM.getOption *(*options*)*, 160
spaMM.options, *51*, *77*
spaMM.options *(*options*)*, 160
spaMM2boot, *66*, *206*
spaMM2boot *(*spaMM_boot*)*, 218
spaMM_boot, *14*, *35*, *66*, *86*, *126*, *127*, *206*, 218
spaMM_glm, *142*
spaMM_glm *(*spaMM_glm.fit*)*, 222
spaMM_glm.fit, 222
spaMMplot2D *(*mapMM*)*, 131
sparse_precision, *163*
sparse_precision *(*algebra*)*, 10
sparseMatrix, *94*
str.inla.spde2 *(*multIMRF*)*, 144
stripHLfit, 225
summary *(*summary.HLfit*)*, 226
summary.HLfit, *15*, *31*, *75*, 226

t.ZAXlist *(*ZAXlist*)*, 246
tcrossprod,ZAXlist,missingOrNULL-method *(*ZAXlist*)*, 246
tcrossprod-methods *(*ZAXlist*)*, 246
terms *(*extractors*)*, 72
terms.object, *74*
title, *217*
Tnegbin *(*negbin*)*, 153
Tpoisson, *182*
Tpoisson *(*Poisson*)*, 176
transffit, 227
tweedie, *37*, *211*, 229
txtProgressBar, *161*

unregister_cF *(*register_cF*)*, 192
update.formula, *187*, *232*
update.formula *(*update.HLfit*)*, 232
update.HLfit, *100*, *110*, *187*, 232
update.pois4mlogit *(*pois4mlogit*)*, 172
update_formulas, *81*
update_formulas *(*update.HLfit*)*, 232
update_resp, *206*
update_resp *(*update.HLfit*)*, 232
upper *(*optimBounds*)*, 158

VarCorr, *93*, *95*, *97*, *112*
VarCorr *(*vcov*)*, 234
Variogram.corMatern *(*corMatern*)*, 38
vcov, *95*, *111*, *112*, 234
vcov.HLfit, *75*
verbose, *35*, *51*, *110*, 237

wafers, 238
weights *(*extractors*)*, 72
weights.glm, *73*
welding, 239
WinterWheat, 240
worldcountries *(*seaMask*)*, 203
wrap_parallel, *63*, *66*, 241

X.antisym *(*X.GCA*)*, 241
X.GCA, *59*, *210*, 241
X2X, *80*, *81*, *94*, *172*, *175*, 243

ZAXlist, *94*, 246
ZAXlist-class *(*ZAXlist*)*, 246