Package: ngme2 (via r-universe)

October 14, 2026

Type Package

Title Linear Latent Non-Gaussian Models with Flexible Distributions

Version 1.0.1

Maintainer Xiaotian Jin <xiaotian.jin@kaust.edu.sa>

Description Fits and analyzes linear latent non-Gaussian models for temporal, spatial, and space-time data. The package provides model components for autoregressive and Ornstein-Uhlenbeck processes, random walks, Matern fields based on stochastic partial differential equations, separable and non-separable space-time models, graph-based Matern models, bivariate type-G fields, and user-defined sparse operators. Latent fields and observation models can use Gaussian and non-Gaussian noise distributions, including normal inverse Gaussian, generalized asymmetric Laplace, and skew-t distributions. Functions are included for simulation, likelihood-based estimation, prediction, cross-validation, convergence diagnostics, stochastic gradient optimization, batch-means confidence intervals, and posterior-like sampling. The modeling framework is described in Bolin, Jin, Simas and Wallin (2026) ``A Unified and Computationally Efficient Non-Gaussian Statistical Modeling Framework" <doi:10.48550/arXiv.2602.23987>.

License GPL (>= 3)

Encoding UTF-8

LazyData true

Imports Matrix, Rcpp, methods, rlang, ggplot2, graphics, fmesher, stats, gridExtra, withr

LinkingTo Matrix, Rcpp, RcppEigen

Suggests knitr, rmarkdown, INLA, RcppEigen, sf, inlabru, dplyr, plyr, MASS, grid, testthat (>= 3.0.0), xml2, splancs, lattice, viridis, rSPDE, tidyr, fields, lme4, plotly, usethis, devtools, MetricGraph (>= 1.1.2), zoo, mvtnorm, kableExtra, reshape2

Additional_repositories https://inla.r-inla-download.org/R/testing

SystemRequirements GNU make

VignetteBuilder knitr

Config/testthat/edition 3

URL https://davidbolin.github.io/ngme2/,
https://github.com/davidbolin/ngme2

Depends R (>= 3.5.0)

BugReports https://github.com/davidbolin/ngme2/issues

Copyright The R package and code, and the main programs, were written by and are Copyright by David Bolin, Xiaotian Jin, Alexandre Simas, and Jonas Wallin, and are redistributable under the GNU Public License, version 3 or later. The package also includes bundled SuiteSparse components. For details see inst/COPYRIGHTS.

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author David Bolin [aut, cph], Xiaotian Jin [aut, cre], Alexandre Simas [aut], Jonas Wallin [aut], Andrea V. Rocha [ctb] (contributed to many vignettes and examples), Timothy A. Davis [ctb, cph] (SuiteSparse components), Patrick R. Amestoy [ctb, cph] (SuiteSparse AMD and CAMD components), Iain S. Duff [ctb, cph] (SuiteSparse AMD and CAMD components), John K. Reid [ctb, cph] (SuiteSparse AMD-derived code), Yanqing Chen [ctb, cph] (SuiteSparse CAMD components), Sivasankaran Rajamanickam [ctb, cph] (SuiteSparse CCOLAMD components), Stefan Larimore [ctb, cph] (SuiteSparse CCOLAMD and COLAMD components), William W. Hager [ctb, cph] (SuiteSparse CHOLMOD Modify components), University of Florida [cph] (SuiteSparse CHOLMOD and CCOLAMD components), Regents of the University of Minnesota [cph] (METIS/GKlib components bundled with SuiteSparse), Free Software Foundation, Inc. [cph] (GKlib regex/getopt code bundled with SuiteSparse), Makoto Matsumoto [ctb, cph] (Mersenne Twister code bundled with SuiteSparse), Takuji Nishimura [ctb, cph] (Mersenne Twister code bundled with SuiteSparse), Alexander Chemeris [ctb, cph] (MS integer headers bundled with SuiteSparse)

Config/pak/sysreqs make

Repository https://cran.r-universe.dev

Date/Publication 2026-10-14 06:40:18 UTC

RemoteUrl https://github.com/cran/ngme2

RemoteRef HEAD

RemoteSha 48ee7236504c39fb8a3c0bd54c977a0f19e3b3bf

Contents

adagrad . . . 5
adam . . . 6
adamW . . . 7
adaptive_gd . . . 8
ar . . . 8
ar1 . . . 9
argo_float . . . 9
arma . . . 10
arma11 . . . 11
batch_decay . . . 12
batch_means_ci . . . 12
batch_means_estimator . . . 13
bfgs . . . 14
bv . . . 15
bv_matern . . . 16
calibrate_inv_exp_lambda_driven_nig . . . 17
cienaga . . . 18
cienaga.border . . . 19
compare_noise_kld . . . 19
compute_index_corr_from_map . . . 20
compute_log_like . . . 20
compute_ngme_ci . . . 21
compute_ngme_sgld_samples . . . 23
compute_score_given_pred . . . 24
control_ngme . . . 25
control_opt . . . 26
control_opt_batch_ci . . . 36
create_paired_cv_splits . . . 37
cross_validation . . . 38
f . . . 40
gal . . . 42
generic . . . 44
generic_ns . . . 46
get_data_from_formula . . . 48
get_parameter_distance . . . 49
get_trace_trajectories . . . 50
get_trajectories . . . 50
gig . . . 51
group_cv . . . 52
ig . . . 56
igam . . . 58
iid . . . 59
make_time_series_cv_index . . . 60
matern . . . 61
mean_list . . . 62
merge_noise . . . 63

merge_replicates . . . 63
momentum . . . 64
name2fun . . . 64
ngme . . . 65
ngme_as_sparse . . . 66
ngme_batch_ci . . . 67
ngme_cov_matrix . . . 68
ngme_make_mesh_repls . . . 68
ngme_model_types . . . 69
ngme_noise . . . 69
ngme_noise_types . . . 73
ngme_optimizers . . . 74
ngme_parse_formula . . . 74
ngme_post_samples . . . 75
ngme_prior_types . . . 76
ngme_result . . . 76
ngme_sgld_ci . . . 78
ngme_sgld_samples . . . 78
ngme_ts_make_A . . . 79
ngme_update . . . 80
nig . . . 80
noise_moments . . . 82
openmp_test . . . 83
ou . . . 83
plot.ngme_noise . . . 85
plot.parameter_distance . . . 86
pm25_quarterly_2022 . . . 87
poly_decay . . . 87
posterior_plot . . . 88
precision_matrix_multivariate . . . 89
precision_matrix_multivariate_spde . . . 90
precond_sgd . . . 92
predict.ngme . . . 93
print.ngme . . . 94
print.ngme_model . . . 95
print.ngme_noise . . . 95
print.ngme_operator . . . 96
print.ngme_replicate . . . 96
print.ngme_trajectories . . . 97
print.noise_kld_comparison . . . 97
print.parameter_distance . . . 98
prior_half_cauchy . . . 98
prior_inv_exponential . . . 99
prior_none . . . 99
prior_normal . . . 100
prior_pc_nu . . . 100
prior_pc_sd . . . 101
priors . . . 102

re . . . 102
rmsprop . . . 103
rw1 . . . 103
rw2 . . . 105
sgd . . . 106
sgld . . . 106
simulate.ngme . . . 107
simulate.ngme_model . . . 108
simulate.ngme_noise . . . 108
spacetime . . . 109
stepsize_control . . . 111
stepsize_decay . . . 111
stepsize_schedule . . . 112
summary.ngme . . . 113
summary.ngme_batch_ci . . . 114
tp . . . 114
traceplot . . . 115
var1 . . . 116

Index **118**

`adagrad` *AdaGrad SGD optimization*

Description

AdaGrad SGD optimization

Usage

```
adagrad(stepsize = 0.05, epsilon = 1e-08)
```

Arguments

`stepsize`	stepsize for SGD
`epsilon`	epsilon for numerical stability

Details

The update rule for AdaGrad is:

$$v_t = v_{t-1} + g_t^2$$

$$x_{t+1} = x_t - \text{stepsize} * \frac{g_t}{\sqrt{v_t} + \epsilon}$$

Value

a list of control variables for optimization (used in `control_opt` function)

adam *Adam SGD optimization*

Description

Adam SGD optimization

Usage

```
adam(stepsize = 0.05, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-08)
```

Arguments

stepsize	stepsize for SGD
beta1	beta1 for Adam
beta2	beta2 for Adam
epsilon	epsilon for numerical stability

Details

The update rule for Adam is:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = m_t / (1 - \beta_1^t)$$

$$\hat{v}_t = v_t / (1 - \beta_2^t)$$

$$x_{t+1} = x_t - \text{stepsize} * \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Value

a list of control variables for optimization (used in control_opt function)

adamW *AdamW SGD optimization*

Description

AdamW SGD optimization

Usage

```
adamW(
  stepsize = 0.05,
  beta1 = 0.9,
  beta2 = 0.999,
  lambda = 0.01,
  epsilon = 1e-08
)
```

Arguments

stepsize	stepsize for SGD
beta1	beta1 for AdamW
beta2	beta2 for AdamW
lambda	lambda (weight decay) for AdamW
epsilon	epsilon for numerical stability

Details

The update rule for AdamW is:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = m_t / (1 - \beta_1^t)$$

$$\hat{v}_t = v_t / (1 - \beta_2^t)$$

$$x_{t+1} = x_t - \text{stepsize} * \left(\lambda x_t + \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \right)$$

Value

a list of control variables for optimization (used in control_opt function)

adaptive_gd *Adaptive gradient descent*

Description

Adaptive gradient descent optimizer.

Usage

```
adaptive_gd(stepsize = 0.01)
```

Arguments

stepsize	initial stepsize for SGD

Details

Based on the method described in https://arxiv.org/pdf/1910.09529. The update rule for adaptive gradient descent is:

$$\lambda_k = \min(\sqrt{1+\theta_{k-1}}\lambda_{k-1}, \frac{||x_k - x_{k-1}||}{2||\nabla f(x_k) - \nabla f(x_{k-1})||})$$

$$x_{k+1} = x_k - \lambda_k \nabla f(x_k)$$

$$\theta_k = \lambda_k / \lambda_{k-1}$$

Value

a list of control variables for optimization (used in control_opt function)

ar *ngme AR(p) model specification*

Description

ngme AR(p) model specification

Usage

```
ar(mesh = NULL, rho = NULL, order = NULL)
```

Arguments

mesh	integer vector or inla.mesh.1d object, index to build the mesh
rho	vector of AR coefficients of length p (should satisfy stationarity conditions)
order	integer, the AR order p. If provided and rho is not specified, rho will be initialized as rep(0, p)

Value

ngme_operator object

Examples

```
# AR(2) model with specified coefficients
ar(c(1:10), rho = c(0.5, -0.3))
# AR(3) model with specified coefficients
ar(c(1:10), rho = c(0.4, 0.2, -0.1))
# AR(2) model with default coefficients (zeros)
ar(c(1:10), order = 2)
# AR(3) model with specified order and coefficients
ar(c(1:10), order = 3, rho = c(0.4, 0.2, -0.1))
```

---

ar1 *ngme AR(1) model specification*

Description

ngme AR(1) model specification

Usage

```
ar1(mesh = NULL, rho = 0)
```

Arguments

mesh	integer vector or inla.mesh.1d object, index to build the mesh
rho	the correlation parameter (between -1 and 1)

Value

ngme_operator object

---

argo_float *Argo float dataset*

Description

Argo floats measurements.

Usage

```
data("argo_float")
```

Format

Data frame containing 274 observations on 4 variables.

lat Latitude.

lon Longitude.

sal Salinity.

temp Temperature.

Details

The floats have a pressure case made of aluminium that is about 1.3m long and about 20cm diameter. They weigh about 40kg. On the top is an antenna to communicate with the satellites that fix the float's position and receive the data. Also on the top are the temperature, salinity and pressure sensors.

Source

Data can be obtained from Argo float website.

`arma` *ngme ARMA(p, q) model specification*

Description

General ARMA(p, q) latent operator under AZW: K W = Z eps, where K encodes the AR part and Z encodes the MA part.

Usage

```
arma(
  mesh = NULL,
  ar_order = 1,
  ma_order = 1,
  ar = NULL,
  ma = NULL,
  fix_ar = FALSE,
  fix_ma = FALSE
)
```

Arguments

`mesh`	Integer vector or 'inla.mesh.1d' object.
`ar_order`	Integer p (0, 1, or 2).
`ma_order`	Integer q (0, 1, or 2).
`ar`	Numeric length-p vector of AR coefficients in (-1, 1). Defaults to zeros.
`ma`	Numeric length-q vector of MA coefficients in (-1, 1). Defaults to zeros.
`fix_ar`	Logical or length-p logical vector to fix AR coefficients.
`fix_ma`	Logical or length-q logical vector to fix MA coefficients.

Details

- Supports p, q <= 2. The user provides standard AR/MA coefficients (ar, ma). For estimation, these are transformed in R into an unbounded parameter vector ‘theta_K‘ via PACF-style mappings so optimizers (e.g., Adam/SGD) operate on unconstrained variables while K and Z remain valid: - AR(1): phi1 = tanh(raw1/2). AR(2): pi1=tanh(raw1/2), pi2=tanh(raw2/2), phi2 = pi2, phi1 = pi1 * (1 - pi2). - MA(1): theta1 = tanh(raw1/2). MA(2): psi1=tanh(raw1/2), psi2=tanh(raw2/2), theta2 = psi2, theta1 = psi1 * (1 - psi2). - The returned operator includes both K (AR part) and Z (MA part). During estimation the backend updates Z synchronously every time ‘theta_K‘ changes.

Value

An ‘ngme_operator‘ describing the ARMA(p, q) latent model.

Examples

```
mesh <- 1:10
op <- arma(mesh, ar_order = 2, ma_order = 2, ar = c(0.5, 0.3), ma = c(0.6, 0.4))
```

`arma11` *Convenience wrapper for ARMA(1,1)*

Description

Convenience wrapper for ARMA(1,1)

Usage

```
arma11(mesh = NULL, ar = 0, ma = 0, fix_ar = FALSE, fix_ma = FALSE)
```

Arguments

`mesh`	Integer vector or ‘inla.mesh.1d‘ object.
`ar`	Numeric length-p vector of AR coefficients in (-1, 1). Defaults to zeros.
`ma`	Numeric length-q vector of MA coefficients in (-1, 1). Defaults to zeros.
`fix_ar`	Logical or length-p logical vector to fix AR coefficients.
`fix_ma`	Logical or length-q logical vector to fix MA coefficients.

Value

An `ngme_operator` object describing an ARMA(1,1) latent operator. The object contains the AR precision component `K`, the MA filter component `Z`, integration weights `h`, and the unconstrained AR and MA parameters used by the optimizer. If `mesh` is `NULL`, returns an `ngme_operator_def` specification for delayed construction inside `f()`.

batch_decay *Batch/checkpoint decay helper*

Description

Batch/checkpoint decay helper

Usage

```
batch_decay(
  patience = 3,
  gamma = 0.5,
  min_delta = 0,
  warmup = 0,
  min_stepsize = 0
)
```

Arguments

patience	number of consecutive checkpoints without improvement before decay.
gamma	decay factor in $(0, 1)$.
min_delta	minimum required decrease to count as improvement.
warmup	number of initial checkpoints to skip.
min_stepsize	lower bound for absolute stepsize.

Details

Convenience helper that enables grad-norm plateau decay and keeps iteration schedule constant.

Value

a `stepsize_control()` object.

batch_means_ci *Pooled Batch-Means Confidence Intervals from Multiple Chains*

Description

Compute pooled point estimates, covariance, and Wald confidence intervals from multiple chain trajectories.

Usage

```
batch_means_ci(
  chain_trajectories,
  level = 0.95,
  alpha = 0.501,
  M = NULL,
  N = NULL,
  drop_burnin = TRUE,
  burnin_iter = 0
)
```

Arguments

chain_trajectories	list of numeric matrices, each with rows = iterations and columns = parameters.
level	confidence level.
alpha	stepsize decay exponent in $(1/2, 1)$.
M	number of retained batches per chain (excluding burn-in batch 0).
N	decorrelation constant used in the batch boundary formula.
drop_burnin	logical; if 'TRUE', discard batch 0.
burnin_iter	non-negative integer. Explicitly discard the first 'burnin_iter' iterations of each chain before Xi-style batching.

Value

A list with pooled estimates, covariance, standard errors, and confidence intervals.

---

batch_means_estimator *Batch-Means Covariance Estimator for SGD Trajectories*

Description

Estimate the asymptotic covariance from a single SGD trajectory using the increasing-batch construction from Xi et al. (2020).

Usage

```
batch_means_estimator(
  trajectory,
  alpha = 0.501,
  M = NULL,
  N = NULL,
  drop_burnin = TRUE,
  burnin_iter = 0
)
```

Arguments

`trajectory`	numeric matrix with rows = iterations and columns = parameters.
`alpha`	stepsize decay exponent in $\eta_i = \eta i^{-\alpha}$. Must satisfy $1/2 < \alpha < 1$.
`M`	number of retained batches (excluding burn-in batch 0). If 'NULL', use $\lfloor n^{(1-\alpha)/2} \rfloor$.
`N`	decorrelation constant in $e_k = \lfloor ((k+1)N)^{1/(1-\alpha)} \rfloor$. If 'NULL', use $N = n^{1-\alpha}/(M+1)$.
`drop_burnin`	logical; if 'TRUE', discard batch 0.
`burnin_iter`	non-negative integer. Explicitly discard the first 'burnin_iter' iterations before building Xi-style batches. After trimming, batch boundaries are rebuilt from iteration 1 of the retained trajectory.

Value

A list containing the covariance estimate, pooled mean, batch sizes, batch boundaries, and chain-level metadata.

`bfgs` *BFGS optimization*

Description

BFGS optimization

Usage

```
bfgs(line_search = "wolfe")
```

Arguments

`line_search`	line search method, can be c("backtracking", "wofle") "bfgs" means use BFGS preconditioner

Value

a list of control variables for optimization (used in `control_opt` function)

bv *Ngme bivariate model specification*

Description

Giving 2 sub_models, build a correlated bivariate operator based on K = D(theta, rho)

$$D(\theta, \rho) = \begin{pmatrix} \cos(\theta) + \rho \sin(\theta) & -\sin(\theta)\sqrt{1+\rho^2} \\ \sin(\theta) - \rho \cos(\theta) & \cos(\theta)\sqrt{1+\rho^2} \end{pmatrix}$$

Usage

```
bv(
  mesh,
  sub_models,
  theta = 0,
  rho = 0,
  c1 = 1,
  c2 = 1,
  share_param = FALSE,
  fix_theta = FALSE,
  use_c_param = FALSE
)
```

Arguments

mesh	the mesh where model is defined
sub_models	a list of sub_models (total 2 sub_models)
theta	the rotation parameter (starting point should be from -pi/4 to pi/4)
rho	the parameter related to correlation
c1	scaling factor for the first sub-model (default: 1)
c2	scaling factor for the second sub-model (default: 1)
share_param	TRUE if share the same parameter for 2 sub_models (of same type)
fix_theta	TRUE if fix the theta of bv model
use_c_param	TRUE to estimate c1 and c2 as parameters, FALSE to fix them at specified values (default: FALSE)

Details

The bivariate model combines two sub-models with a rotation matrix D and optional scaling factors c1 and c2. When use_c_param = TRUE, c1 and c2 are estimated as parameters (on log scale). When use_c_param = FALSE (default), c1 and c2 are fixed at their specified values (both default to 1).

Value

a ngme_operator of bivariate model

bv_matern *Ngme bivariate model with Matern sub_models*

Description

Giving 2 sub_models, build a correlated bivaraite operator based on K = D(theta, eta)

$$D(\theta, \rho) = \begin{pmatrix} \cos(\theta) + \rho \sin(\theta) & -\sin(\theta)\sqrt{1+\rho^2} \\ \sin(\theta) - \rho \cos(\theta) & \cos(\theta)\sqrt{1+\rho^2} \end{pmatrix}$$

Usage

```
bv_matern(
  mesh,
  sub_models,
  theta = 0,
  rho = 0,
  sd1 = 1,
  sd2 = 1,
  group = NULL,
  share_param = FALSE,
  fix_theta = FALSE,
  ...
)
```

Arguments

mesh	the mesh where model is defined
sub_models	a list of sub_models (should be two matern models)
theta	the parameter related to rotation
rho	the parameter related to correlation
sd1	scaling parameter for the first sub-model
sd2	scaling parameter for the second sub-model
group	group vector, can be inherited from ngme() function
share_param	TRUE if share the same parameter for 2 sub_models (of same type)
fix_theta	TRUE if the rotation parameter is fixed
...	ignore

Value

a list of specification of model

`calibrate_inv_exp_lambda_driven_nig` *Calibrate Inverse-Exponential Prior for NIG Driven Noise*

Description

Calibrate λ in the prior $\kappa = 1/\nu \sim \mathrm{Exp}(\lambda)$ using a tail-inflation target for driven NIG noise.

The calibration target is

$$\Pr(R_c(\nu) > r_{\text{target}}) = \alpha,$$

where

$$R_c(\nu) = \frac{\Pr(|U| > c \mid \nu)}{\Pr(|Z| > c)}, \quad Z \sim N(0,1)$$

and

$$U = \frac{\mu(V-h) + \sigma\sqrt{V}Z}{\sigma\sqrt{h}}, \quad V \sim \mathrm{GIG}(-1/2, \nu, \nu h^2).$$

The function solves $R_c(\nu_r) = r_{\text{target}}$ using log-scale bisection, then returns

$$\lambda = -\nu_r \log(\alpha).$$

Usage

```
calibrate_inv_exp_lambda_driven_nig(
  r_target = 2,
  alpha = 0.1,
  c = 3,
  mu = 0,
  sigma = 1,
  h = 1,
  n_samples = 1e+05,
  nu_lower = 0.1,
  nu_upper = 100,
  tol = 0.05,
  max_iter = 30,
  max_expand = 30,
  seed = NULL
)

calibrate_inv_exp_lambda(...)
```

Arguments

`r_target`	target tail inflation level $r_{\text{target}} > 1$
`alpha`	target prior probability in $(0, 1)$
`c`	tail threshold for $\lvert U\rvert > c$, typically ‘2.5‘ or ‘3‘

`mu`	NIG drift parameter in the driven noise term
`sigma`	NIG scale parameter (> 0)
`h`	positive increment scaling (> 0)
`n_samples`	Monte Carlo sample size used per evaluation of $R_c(\nu)$
`nu_lower`	initial lower bracket for ν
`nu_upper`	initial upper bracket for ν
`tol`	relative tolerance for solving $R_c(\nu_r) = r_{\text{target}}$
`max_iter`	maximum number of bisection iterations
`max_expand`	maximum bracket expansion steps on each side
`seed`	optional integer seed for reproducible calibration
`...`	arguments forwarded to `calibrate_inv_exp_lambda_driven_nig()`

Value

A list with calibrated ‘lambda‘, solved ‘nu_r‘, achieved ‘rc_nu_r‘, and diagnostics.

`cienaga` *The swamp of Cienaga Grande in Santa Marta, Colombia*

Description

There is a total of 114 locations where some properties of the swamp were measured. Those measurements were taken twice, however there is no information available about their chronological order so this data cannot be treated as spatiotemporal, despite that, the multiple measurements can be treated as replicates.

Usage

```
cienaga
```

Format

A data frame with 218 rows and 6 columns.

East, North location

depth depth of the swamp

temp temperature

oxyg oxygen

measurement 1 means the first measurement, 2 the second

Source

..

cienaga.border *The x y location of the border of the swamp of Cienaga Grande in Santa Marta, Colombia*

Description

The data is of dimension 472 * 2. It contains the x and y coordinates of the border of the swamp.

Usage

```
cienaga.border
```

Format

A data frame with 472 rows and 2 columns.

East, North location

Source

..

compare_noise_kld *Compare noise objects using Kullback-Leibler divergence*

Description

This function compares multiple noise objects by calculating the KLD of each noise against the first noise object (reference).

Usage

```
compare_noise_kld(x = NULL, ..., xlim = c(-10, 10), n_points = 1000)
```

Arguments

x	first noise object (reference)
...	additional noise objects to compare, and optional parameters
xlim	x-axis range for evaluation (default: c(-10, 10))
n_points	number of evaluation points (default: 1000)

Value

named vector of KLD values

Examples

```
n1 <- noise_nig(mu = 0, sigma = 1, nu = 1)
n2 <- noise_nig(mu = 0.5, sigma = 1.2, nu = 0.8)
compare_noise_kld(n1, method2 = n2)
```

`compute_index_corr_from_map` *Helper function to compute the index_corr vector*

Description

Helper function to compute the index_corr vector

Usage

```
compute_index_corr_from_map(map, eps = 0.1)
```

Arguments

`map`	used as location to compute distance, can be 1d (numerical) or 2d (data.frame)
`eps`	threshold to determine if two points are close (if close, we consider them as the same point)

Value

the index_corr vector for ngme correlated measurement noise

Examples

```
x_coord <- c(1.11, 1.12, 2, 1.3, 1.3)
y_coord <- c(2.11, 2.11, 2, 3.3, 3.3)
coord <- data.frame(x_coord, y_coord)
compute_index_corr_from_map(map = coord, 0.1)
```

`compute_log_like` *Compute Gaussian log-likelihood*

Description

Creates the underlying C++ block models and evaluates their Gaussian log-likelihood when applicable.

Usage

```
compute_log_like(ngme_model)
```

Arguments

ngme_model	An object created by ‘ngme()‘.

Value

A numeric scalar. Returns ‘0‘ if any replicate is non-Gaussian.

---

compute_ngme_ci *Refit an Existing ngme Object with SGD and Compute Batch-Means CI*

Description

Run one additional SGD stage (warm-started from an existing ‘ngme‘ fit), then compute Xi-style batch-means confidence intervals from the newly stored trajectories.

Usage

```
compute_ngme_ci(
  fit,
  iterations = 4000,
  alpha = 0.501,
  optimizer = sgd(stepsize = 0.03),
  burnin = 100,
  n_batch = 20,
  n_parallel_chain = 4,
  t0 = 1,
  start_sd = 0.2,
  seed = ngme_random_seed(),
  verbose = FALSE,
  level = 0.95,
  name = "all",
  M = NULL,
  N = NULL,
  apply_transform = TRUE,
  drop_burnin = TRUE,
  burnin_iter = 0,
  control_opt = NULL,
  ...
)

compute_ngme_CI(
  fit,
  iterations = 4000,
  alpha = 0.501,
  optimizer = sgd(stepsize = 0.03),
  burnin = 100,
```

```
  n_batch = 20,
  n_parallel_chain = 4,
  t0 = 1,
  start_sd = 0.2,
  seed = ngme_random_seed(),
  verbose = FALSE,
  level = 0.95,
  name = "all",
  M = NULL,
  N = NULL,
  apply_transform = TRUE,
  drop_burnin = TRUE,
  burnin_iter = 0,
  control_opt = NULL,
  ...
)
```

Arguments

fit	existing fitted 'ngme' object (can be obtained with any optimizer).
iterations	optimization iterations for the SGD CI stage.
alpha	Xi-style exponent used for both polynomial schedule and batch-means inference.
optimizer	optimizer for the CI stage; must be 'sgd(...)' for Xi theory.
burnin	burn-in iterations before SGD optimization.
n_batch	number of optimization checkpoints.
n_parallel_chain	number of parallel chains for SGD.
t0	non-negative schedule offset in $\eta_t = \eta_0(t + t0)^{-\alpha}$.
start_sd	standard deviation for randomized chain initialization.
seed	random seed for CI-stage optimization. Defaults to a seed drawn from the current R random number stream, so `set.seed()` makes the result reproducible.
verbose	logical; print optimization progress.
level	confidence level for CI.
name	parameter block for CI ('"all"', latent name, or '"general"').
M	number of retained Xi batches (excluding batch 0).
N	decorrelation constant in Xi batch boundaries.
apply_transform	logical; if 'TRUE', apply Post-delta transform after raw-space inference.
drop_burnin	logical; if 'TRUE', discard Xi batch 0.
burnin_iter	non-negative integer used both as optimizer schedule warmup ('stepsize_schedule_burnin_iter') and as explicit CI trajectory trimming before Xi-style batching.
control_opt	optional pre-built 'control_opt' object for the CI stage. If supplied, it is used directly (with 'store_traj' forced to 'TRUE').
...	additional arguments forwarded to [control_opt_batch_ci()] when 'control_opt' is 'NULL'.

Value

An object of class 'ngme_batch_ci' with extra fields: 'refit' (the SGD-refit model), 'refit_control_opt', and 'refit_source'.

`compute_ngme_sgld_samples` *Refit an Existing ngme Object with SGLD and Extract Samples*

Description

Run one additional SGLD stage (warm-started from an existing 'ngme' fit), then extract posterior-like samples from stored trajectories.

Usage

```
compute_ngme_sgld_samples(
  fit,
  iterations = 4000,
  optimizer = sgld(stepsize = 0.01, temperature = 1),
  burnin = 100,
  n_batch = 20,
  n_parallel_chain = 4,
  alpha = 0.6,
  t0 = 10,
  start_sd = 0.2,
  seed = ngme_random_seed(),
  verbose = FALSE,
  name = "all",
  burnin_iter = 0,
  thinning = 1,
  n_gibbs_samples = NULL,
  apply_transform = TRUE,
  combine_chains = TRUE,
  control_opt = NULL,
  ...
)
```

Arguments

`fit`	existing fitted 'ngme' object (can be obtained with any optimizer).
`iterations`	optimization iterations for the SGLD stage.
`optimizer`	optimizer for the sampling stage; must be 'sgld(...)'.
`burnin`	burn-in iterations before optimization.
`n_batch`	number of optimization checkpoints.

n_parallel_chain	number of parallel chains.
alpha	polynomial schedule exponent used by ‘poly_decay(alpha, t0)‘.
t0	non-negative schedule offset.
start_sd	standard deviation for randomized chain initialization.
seed	random seed for SGLD stage. Defaults to a seed drawn from the current R random number stream, so `set.seed()` makes the result reproducible.
verbose	logical; print optimization progress.
name	parameter block to extract: ‘"all"‘ (default), latent model name, or ‘"general"‘.
burnin_iter	non-negative integer used both as optimizer schedule warmup (‘stepsize_schedule_burnin_iter‘) and as explicit trajectory trimming before sampling.
thinning	positive integer thinning interval.
n_gibbs_samples	optional positive integer. If supplied, override ‘control_ngme$n_gibbs_samples‘ for the SGLD refit stage only; otherwise inherit the value from ‘fit‘.
apply_transform	logical; apply parameter transforms to user scale.
combine_chains	logical; if ‘TRUE‘, return one combined data.frame, otherwise return one data.frame per chain.
control_opt	optional pre-built ‘control_opt‘ object for the SGLD stage. If supplied, it is used directly (with ‘store_traj‘ forced to ‘TRUE‘).
...	additional arguments forwarded to [control_opt()] when ‘control_opt‘ is ‘NULL‘.

Value

A data.frame (or list of data.frames) from [ngme_sgld_samples()] with extra attributes ‘refit‘, ‘refit_control_opt‘, and ‘refit_source‘.

`compute_score_given_pred` *Compute the scores given the prediction*

Description

Compute the scores given the prediction

Usage

```
compute_score_given_pred(
  Y_N_1_thin,
  Y_N_2_thin,
  y_data,
  group_data,
```

```
  merge_groups = FALSE,
  merged_group_name = NULL,
  metric = NULL,
  transform = identity
)
```

Arguments

Y_N_1_thin	posterior predictive draws (rows = observations, columns = samples)
Y_N_2_thin	posterior predictive draws (rows = observations, columns = samples)
y_data	a vector of length n_obs
group_data	a vector of length n_obs
merge_groups	logical, if TRUE and there are exactly two groups, combine them into a vector-valued score
merged_group_name	optional label for the merged group
metric	optional custom metric function used to combine group-wise values before scoring
transform	element-wise map applied to the observations and to every posterior draw before scoring, so the scores are on the transformed scale. Applied before ‘metric‘: ‘transform‘ changes the scale of each value, ‘metric‘ combines values across groups, and the two compose in that order.

---

control_ngme *Generate control specifications for the ngme model*

Description

Generate control specifications for the ngme model

Usage

```
control_ngme(
  n_gibbs_samples = 5,
  post_burnin = 100,
  fix_beta = FALSE,
  n_post_samples = 100,
  beta_init = NULL,
  debug = FALSE,
  ...
)
```

Arguments

n_gibbs_samples	number of gibbs samples at each iteration. Raising it lowers the variance of each gradient but costs proportionally more per iteration, so the number of iterations needed to satisfy the convergence checks is largely unchanged and the wall-clock time to converge goes up. Raise it when the estimates are noisy at convergence.
post_burnin	number of Gibbs sweeps to discard before recording the posterior samples of W and V returned by ngme(). A burn-in is needed because the chain is restarted from the stored W and V, which are averaged over the parallel chains and so are not themselves a draw from the posterior. Set to 0 only if you know the stored state is a valid draw.
fix_beta	logical, fix fixed effect estimation
n_post_samples	number of posterior samples, see ?ngme_post_samples()
beta_init	fixed effect initial value on the original design scale. If control_opt(standardize_fixed = TRUE) (the default), the vector is internally rotated/scaled to match the SVD-standardized design matrix before being passed to the optimizer. Provide values on the raw design scale; no manual rescaling is required.
debug	debug mode
...	additional arguments. Legacy aliases feff and fix_feff are still recognized and mapped to beta_init and fix_beta. Anything else is ignored with a warning naming it.

Value

a list of control variables for ngme

---

control_opt *Generate control specifications for* ngme() *function.*

Description

These are configurations for ngme optimization process.

Usage

```
control_opt(
  seed = ngme_random_seed(),
  burnin = 100,
  iterations = 500,
  estimation = TRUE,
  standardize_fixed = TRUE,
  n_batch = NULL,
  iters_per_check = 25,
  optimizer = precond_sgd(),
```

```
  start = NULL,
  start_sd = 0.5,
  n_parallel_chain = 4,
  max_num_threads = n_parallel_chain,
  print_check_info = FALSE,
  max_relative_step = 0.5,
  max_absolute_step = 0.5,
  step_clip = c("adaptive", "norm", "value"),
  step_clip_factor = 5,
  rao_blackwellization = TRUE,
  n_trace_iter = 10,
  polish_iterations = 2000L,
  polish_stepsize_factor = 1,
  polish_decay_alpha = 0.6,
  polish_decay_t0 = 13.5,
  selinv_max_fill = 4,
  selinv_cost_ratio = 2,
  n_fisher_probes = NA_integer_,
  selinv_max_fill_k = Inf,
  trace_adapt = TRUE,
  trace_adapt_frac = 0.1,
  trace_adapt_every = 100L,
  trace_adapt_min = 5L,
  trace_adapt_max = 200L,
  trace_adapt_rule = c("cost", "share"),
  trace_probing = TRUE,
  trace_probing_max_dist = 4L,
  trace_probing_raise_budget = 1,
  n_trace_iter_k = NA_integer_,
  trace_adapt_k = FALSE,
  trace_block_probe = FALSE,
  sampling_strategy = "all",
 solver_backend = if (Sys.info()["sysname"] == "Darwin") "accelerate" else "cholmod",
  solver_type = "llt",
  nonsym_solver = "normal_equations",
  solver_order = c("auto", "default", "amd", "metis"),
  verbose = FALSE,
  store_traj = TRUE,
  robust = FALSE,
  nig_param_std = 3,
  precond_meas_sigma = c("auto", "fisher", "complete"),
  fisher_refresh_every = 10L,
  stepsize_control = NULL,
  n_min_batch = 1,
  n_slope_check = 3,
  trend_std_conv_check = TRUE,
  std_lim = 0.01,
  trend_lim = 2,
```

```
  trend_rel_lim = 0.01,
  use_std_check = FALSE,
  n_conv_batch = 2,
  warn_no_convergence = TRUE,
  schedule_auto_start = FALSE,
  schedule_min_scale = 0.1,
  schedule_arm_z = 2,
  stationarity_window = 0L,
  stationarity_eff = 1.5,
  polish_floor_frac = 0.25,
  stationarity_ratio_lim = 1,
  stationarity_dir_lim = 0.5 * sqrt(6),
  stationarity_min_checks = 6L,
  stationarity_dir_scaled = TRUE,
  conv_criterion = c("stationarity", "drift"),
  mc_se_conv_check = TRUE,
  mc_se_lim = 0.7,
  max_stepsize_decays = 1L,
  n_settle_checks = 3L,
  stepsize_decay_precision_gamma = 0.5,
  continue_chains = TRUE,
  R_hat_conv_check = TRUE,
  max_R_hat = 1.1
)
```

Arguments

seed	random seed. Defaults to a seed drawn from the current R random number stream, so `set.seed()` makes the result reproducible.
burnin	interations for burn-in periods (before optimization)
iterations	optimization iterations
estimation	run the estimation process (call C++ in backend)
standardize_fixed	whether or not standardize the fixed effect. When `TRUE` (default), the design matrix is SVD-standardized internally and any user-supplied `control_ngme(beta_init = ...)` is automatically mapped from the original design scale to that standardized basis. Set to `FALSE` to keep both the design matrix and any provided `beta_init` on their original scale.
n_batch	**Deprecated**; use `iters_per_check`. It set the checkpoint interval as a division of the iteration budget, so raising `iterations` silently checked less often.
iters_per_check	how many iterations between convergence checkpoints (default 25). A fixed count, independent of `iterations`, which need not be a multiple of it.
optimizer	choose different sgd optimization method, currently support "sgd", "sgld", "precond_sgd", "momentum", "adagrad", "rmsprop", "adam", "adamW" see ?sgd, ?precond_sgd, ?momentum, ?adagrad, ?rmsprop, ?adam, ?adamW

`start` deprecated guard argument. Do not pass model starts through `control_opt()`; use `ngme(..., start = previous_fit)` instead.

`start_sd` standard deviation of the initial parameter (1st chain fixed, other chains random), set 0 to be fixed for all chains

`n_parallel_chain`
number of parallel chains

`max_num_threads`
maximum number of threads used for parallel computing, by default will be set same as n_parallel_chain. If it is more than n_parallel_chain, the rest will be used to parallel different replicates of the model.

`print_check_info`
print the convergence information

`max_relative_step`
max relative step allowed in 1 iteration

`max_absolute_step`
max absolute step allowed in 1 iteration

`step_clip` how the per-iteration step is limited.

- `"value"` the historical behaviour: clip each component at `max_absolute_step`. Because components are truncated independently, it rotates the search direction.
- `"norm"` rescale the whole vector when its length exceeds `max_absolute_step * sqrt(p)`.
- `"adaptive"` the default: rescale only steps far out of line with the recent typical step length (`step_clip_factor` times a running average of it), with the `"norm"` cap as a backstop. Blow-ups are orders of magnitude out and are caught; a step that is merely larger than usual, because the method is accumulating one, is not.

`step_clip_factor`
for `step_clip = "adaptive"`, how many times the recent typical step length is still considered a normal step.

`rao_blackwellization`
replace the sampled latent field by its conditional expectation `E[W | Y, V]` in the gradient (default `TRUE`).

`n_trace_iter` use how many iterations to approximate the trace (Hutchinson's trick). The starting probe budget; with `trace_adapt = TRUE` it is retuned during the run.

`polish_iterations`
maximum iterations of the post-convergence polish, whose averaged iterates (Polyak-Ruppert) are the reported estimate. It is a cap and the polish stops earlier once the precision and drift tests pass. The polish runs only if the search converged. A search that exhausts `iterations` gets no polish at all, and its estimate is the last-batch mean rather than an average. The budget is capped at the iterations actually run only under `conv_criterion = "drift"`; under `"stationarity"` (the default) the full `polish_iterations` is available however short the search was.

`polish_stepsize_factor`
multiplies the step size at the moment the polish begins. Default 1.0.

`polish_decay_alpha`
: exponent of the step-size decay applied during the polish, as `(1 + i/polish_decay_t0)^-alpha` in the polish iterations `i`. Set 0 to hold the polish step size constant.

`polish_decay_t0`
: timescale of that decay, in polish iterations.

`selinv_max_fill`
: use the exact selected (Takahashi) inverse for the Rao-Blackwell traces of the block precision when the Cholesky factor of QQ has `nnz(L)/n` at or below this, and Hutchinson probes otherwise. Triangular operators (ar1, ou, arma) and 1-d meshes give a banded QQ with a ratio near 2-3, where the exact route is both cheaper and free of probe noise; a 2-d mesh is nearer 35, where it is far slower. Set to 0 to always probe. This is a fill threshold rather than a cost comparison on purpose: the two routes reach very different fractions of peak throughput per operation – a probe is a triangular solve over a block of right-hand sides and vectorises better the denser the factor gets, while the Takahashi recursion is a scalar scatter/gather – and the spread between them is wider than the margin that separates the models wanting one route from those wanting the other, so counting operations does not decide it.

`selinv_cost_ratio`
: how much dearer one exact selected inverse of the OPERATOR may be than the probe budget it replaces and still be preferred. This governs the operator traces only; the block precision is gated by `selinv_max_fill`. Both sides are counted in operations, read off the factor's sparsity pattern once at the start of a fit, and the choice is then fixed for the run. Counting rather than timing is what keeps a fit reproducible: a rule that consults the clock can decide differently on a busy machine. Above one because the exact route carries no estimation variance, which probing cannot buy at any finite budget. The comparison is only trusted here because on the operator side it is not close – the operator's factor is far sparser than the probes it replaces, and every model tested clears the threshold by more than an order of magnitude.

`n_fisher_probes`
: probe budget for the expected-information estimate the preconditioner uses for the measurement scale. `NA` (the default) follows `n_trace_iter`, which is how it has been sized until now – but that budget is chosen against gradient variance, which is a different target, so the two can be set apart.

`selinv_max_fill_k`
: the same bound for the operator traces, and `Inf` by default, so there the operation-count comparison always decides and no fill threshold pre-empts it. The operator can afford that where the block precision cannot: its traces are taken once per optimizer iteration, whereas the block-precision traces are retaken on every Gibbs pass, so the exact route is charged far less over a run, and for operators that factor into a tensor product it decomposes through the factors. Set a finite value to restore a hard gate.

`trace_adapt`
: size the Hutchinson probe count automatically (default `TRUE`). The estimator's own variance falls as 1/N and is measured from the spread of the probes, while the Gibbs sampling noise is measured across iterations; the budget is set so the former is `trace_adapt_frac` of the latter. Only active when Rao-Blackwellisation is on, since that is where the trace estimators feed the gradient.

`trace_adapt_frac`
: target share of the total gradient variance carried by the trace estimator (default 0.1, i.e. it adds about 5% to the gradient standard deviation). A deadband leaves the budget alone while the measured share is within a factor of two of this, so the achieved share settles between roughly `0.5 * trace_adapt_frac` and `2 * trace_adapt_frac`. Updates are damped and capped at 25% per step, so the budget approaches its target over several updates rather than jumping.

`trace_adapt_every`
: minimum iterations between probe-budget updates, rounded up to the next convergence checkpoint. The budget is changed only at those checkpoints, because that is where the parallel chains meet: every chain is given the same budget, since `R_hat` reads the spread between chains as evidence about the spread within them and chains probing at different budgets carry different estimator variances.

  Because each update is capped at 25%, this also sets how far the budget can travel in a run, and the default is deliberately slow: it leaves the budget near where it started. Lowering it lets the budget reach what the measurement asks for, which is worth doing when the trace estimator is the only noise in the gradient – an all-Gaussian Rao-Blackwell fit – and is wasted work when it is not. With a non-Gaussian noise the Gibbs sampling carries most of the gradient variance, extra probes do not move the estimate, and the budget is sized off the worst-served parameter, which raises it anyway. Measure before lowering it.

`trace_adapt_min`, `trace_adapt_max`
: bounds on the adapted probe count.

`trace_adapt_rule`
: how `trace_adapt` sizes the probe budget.

  `"cost"` (default) hold the budget through the search and size it in the polish to minimise the variance of the reported estimate per unit of work: $N^* = \sqrt{(P/V_{gibbs})(a/b)}$, capped where the probes cost as much as the rest of a pass. Under this rule an all-Gaussian Rao-Blackwell fit – where the probes are the only gradient noise – also gets the cheapest usable colouring from the first iteration.

  `"share"` size the budget so the trace estimator carries `trace_adapt_frac` of the gradient variance, retuned throughout the search. The behaviour before the cost rule existed.

  `"share"` targets a share of a quantity that contains the thing being controlled, so on a converging fit it reduces to holding probe noise to a fraction of the drift, and the budget climbs to `trace_adapt_max` whatever it measures. `"cost"` has no such hole and responds to a better estimator by asking for fewer probes.

  Two limits. A fit that never converges never polishes, so its budget stays where the search put it – fine unless the probes are what is holding it back; `"share"` is the fallback if a fit needs its budget to grow. And the cost model treats the per-probe cost as constant when wider probe blocks are cheaper per column, so it under-spends rather than over-spends.

`trace_probing`
: structure the Hutchinson probes against the sparsity of the model rather than drawing them densely (default `TRUE`).

  The error of a Hutchinson estimate is carried by the off-diagonal terms, and no choice of probe distribution removes them. What does is the support: colour

the graph of the precision so that two indices share a colour only when they are several edges apart, and give each probe signs on one colour class only. The surviving terms are pairs within a class, whose entry of the inverse has already decayed. Still exactly unbiased; the variance falls geometrically in the colouring distance where more dense probes buy only $1/\sqrt{N}$.

It does not raise the probe count: the distance is the largest whose colour count fits the budget in force, leftover budget goes into repeat sign draws, and a budget too small for a distance-1 colouring falls back to dense probes. The colour count is set by the mesh's local connectivity, not its size, so the same budget buys the same structure at any n.

The gain is in the block-precision traces and $tr(K^{-1}dK)$. The Hessian pair $tr(K^{-1}K_kK^{-1}K_j)$ gains little – two inverses roughly double the decay length – but rides the same probe block.

`trace_probing_max_dist`

largest colouring distance `trace_probing` may consider (default 4). The budget bounds the probe count either way, so this only widens the search for a usable colouring; each colouring is computed once per sparsity pattern and cached.

`trace_probing_raise_budget`

the most `n_trace_iter` may be multiplied by to reach the smallest budget at which `trace_probing` engages (default `1`, i.e. never). A colouring costs one probe per colour and that count is a property of the mesh, so a small `n_trace_iter` can sit below the point where any colouring fits. Raising this lets the budget climb to that point and no further, bounded also by `trace_adapt_max`; a bound below the floor raises nothing rather than raising partway. It is off by default because the extra probes were measured to buy accuracy but not convergence. Ignored under `trace_adapt_rule = "cost"`, which makes this decision itself.

`n_trace_iter_k` probe budget for the operator traces, which feed both the gradient of the operator parameters and the operator block of the preconditioner. `NA` (the default) uses `n_trace_iter`. A probe against the operator is cheaper than one against the full precision, so the two need not be carried at the same value.

`trace_adapt_k` whether `trace_adapt` also retunes the probe budget used for the operator traces, which enter the gradient of the operator parameters. Without it only the budget for the block precision is retuned and the operator keeps the budget it was built with. Off by default.

`trace_block_probe`

estimate the operator-side traces of a non-separable spacetime model with structured probes rather than a dense inverse of each spatial block. Default `FALSE`.

`sampling_strategy`

subsampling method of replicates of model, c("all", "ws") "all" means using all replicates in each iteration, "ws" means weighted sampling (each iteration use 1 replicate to compute the gradient, the sample probability is proption to its number of observations)

`solver_backend` backend in ("eigen", "cholmod", "accelerate", "pardiso"). Defaults to "accelerate" on macOS and "cholmod" elsewhere. Note that the speed of "cholmod" is set by the BLAS R itself is linked against, and with R's bundled reference BLAS those kernels run one to two orders of magnitude below a tuned one. Thus, link R against OpenBLAS or MKL before fitting large models.

`solver_type` factorization type: "llt" or "ldlt"

`nonsym_solver` how the operator matrix `K` of a non-symmetric model is factorized when estimating `tr(K^-1 dK)`. "normal_equations" (default) takes a Cholesky of `t(K) K` and applies `K^-1` as `(t(K) K)^-1 t(K)`; "lu" takes a sparse LU of `K`. The Cholesky is the cheaper of the two for every non-symmetric operator here. Its one cost is that it squares the condition number of `K`, so switch to "lu" for an operator so ill-conditioned that the Cholesky of `t(K) K` fails.

`solver_order` fill-reducing ordering for the `"accelerate"` backend. The ordering fixes the fill of the Cholesky factor and so the cost of every factorization, which is the largest single item in a fit with a high-dimensional latent field. `"auto"` (default) analyzes the pattern under each candidate ordering and keeps the one giving the smallest factor. Its cost is a fixed amount of extra symbolic work at setup, so pass `"default"` (leave Apple's choice, which is AMD) for a short fit. `"amd"` and `"metis"` (nested dissection) force one ordering.

`verbose` print estimation

`store_traj` store the optimizer trajectory for diagnostics (set FALSE to reduce memory)

`robust` use robust mode in the backend optimizer/model updates

`nig_param_std` coordinates the optimiser uses for stationary NIG measurement noise, following Cabral, Bolin and Rue (2023). They apply when the measurement noise is NIG with scalar, free and stationary `mu`, `sigma` and `nu`, unshifted `nu` and no correlation; otherwise the native coordinates are used. In the native parameters the variance `h(sigma^2 + mu^2/nu)` is shared by all three, giving a long flat ridge; these coordinates turn that ridge into an axis. Only the optimiser's coordinates change and the objective, the priors and the reported estimates stay native, so fits remain comparable across settings.

- **0** native `(theta_mu, log sigma, log nu)`
- **1** standardised `(log sigma_marg, zeta, log eta)` with `eta = 1/nu`, `zeta = mu/sigma` and `sigma_marg = sqrt(sigma^2 + mu^2/nu)` the marginal SD
- **2** additionally orthogonalised, `zeta* = zeta sqrt(eta)`, `eta* = eta / xi^2` with `xi = 1 + zeta*^2 - |zeta*| sqrt(1 + zeta*^2)`, which makes the kurtosis invariant to skewness
- **3** as 2 with `zeta*` carried as `asinh(zeta*)`, which keeps the skewness coordinate unbounded and better scaled (default)

`precond_meas_sigma`

curvature `precond_sgd()` uses for the measurement noise `sigma`.

- **"auto"** (default) the marginal Fisher information for non-Gaussian measurement noise, the complete-data Hessian for Gaussian noise.
- **"fisher"** always the marginal Fisher information, with the latent field integrated out. It keeps `sigma` from stalling near zero when the starting values are poor, e.g. fixed effects started far from their OLS estimates, at some extra cost per iteration.
- **"complete"** always the complete-data Hessian. Only supported for Gaussian measurement noise; other noise uses `"fisher"`, with a warning.

Only used for uncorrelated measurement noise.

`fisher_refresh_every`
: how often, in iterations, the Fisher information for the measurement `sigma` is recomputed (default 10). It is also recomputed as soon as `sigma` has moved by more than 5%, and is never computed more than once per iteration.

`stepsize_control`
: unified stepsize configuration created by `stepsize_control()`, `poly_decay()`, or `batch_decay()`. For polynomial schedule, `poly_decay(..., burnin_iter = B)` keeps schedule scale at 1 for the first `B` iterations, then starts decay with reset local time index.

`n_min_batch`
: minimum number of checkpoints before any convergence diagnostic is attempted (default 1).

`n_slope_check`
: number of checkpoints used as the regression window for the trend test

`trend_std_conv_check`
: enable the trend/std diagnostic (uses `std_lim`, `trend_lim`, `n_slope_check`)

`std_lim`
: maximum allowed standard deviation

`trend_lim`
: reported in `attr(fit, "conv_diag")` as a reference for the fitted slope's t-statistic; it no longer gates convergence.

`trend_rel_lim`
: how fast a parameter must actually be MOVING before a detectable drift counts as non-convergence, as a fraction of the parameter's own scale per 100 iterations The default 0.01 reads as "1% of itself per 100 iterations". A parameter fails the trend test only when its drift is both statistically detectable and faster than this.

`use_std_check`
: include the relative-standard-deviation part of the trend/std diagnostic. The raw coefficient of variation conflates chain disagreement with parameter imprecision, so weakly identified parameters can never pass it.

`n_conv_batch`
: number of consecutive checkpoints that must satisfy the criteria before convergence is declared (default 2). Guards against a single lucky checkpoint.

`warn_no_convergence`
: emit a warning when the iteration budget is exhausted without the convergence criteria being met. Set `FALSE` for short runs where convergence is not expected (e.g. fast unit tests).

`schedule_auto_start`
: arm the polynomial step-size schedule automatically at the first checkpoint where the drift criterion passes. Defaults to FALSE.

`schedule_min_scale`
: minimum multiplier allowed for the scheduled step size (default 0.1).

`schedule_arm_z`
: threshold for arming the optional step-size schedule (`stepsize_control`). It does not gate the stationarity criterion, which uses `stationarity_ratio_lim`.

`stationarity_window`
: length of the comparison window, in checkpoints. `0` (default) derives it from the measured correlation length.

`stationarity_eff`
: effective samples per half-window when the window is derived (default 1.5).

`polish_floor_frac`
: lower bound on the polish step, as a fraction of the step the polish starts from (default 0.25).

`stationarity_ratio_lim`
: how far a parameter may move between the two halves of the window, as a multiple of `se_stat` (default 1.0).

`stationarity_dir_lim`
: a parameter failing the magnitude test is called a flat direction and is excluded from the stopping rule and reported, only if its successive half-differences cancel, `|sum| / sum|.|` below this (default 0.5). One still travelling toward stationarity moves the same way each time and keeps the ratio high.

`stationarity_min_checks`
: checkpoints of history required before a parameter may be called a flat direction (default 6).

`stationarity_dir_scaled`
: compare `dir * sqrt(n)` against `stationarity_dir_lim` rather than `dir` alone, and hold the direction history to `stationarity_min_checks` rather than twice it. `dir = |sum d| / sum|d|` has expectation `1/sqrt(n)` under a stationary null, so a fixed threshold gets steadily more permissive as the history fills, and the window length ends up setting the test's strictness. Scaling by `sqrt(n)` holds it constant, which matters because the direction this errs in is premature release of a drifting parameter.

`conv_criterion`
: which stopping rule the search phase uses: `"stationarity"` (default) or `"drift"`.

`mc_se_conv_check`
: use the Monte Carlo standard-error diagnostic as an additional convergence condition during the search phase (default TRUE).

`mc_se_lim`
: target for the polish, as a multiple of the statistical error: it runs until the Monte Carlo error of the reported average falls under `mc_se_lim * se_stat`. Total error is then about `sqrt(1 + mc_se_lim^2) * se_stat` and the cost goes as `1 / mc_se_lim^2`, so tightening it is quadratically expensive for a shrinking return. It touches only the polish phace, the search is unaffected.

`max_stepsize_decays`
: maximum number of precision-driven step-size reductions (default 1).

`n_settle_checks`
: number of checkpoints to wait after a step-size reduction before assessing convergence again (default 3).

`stepsize_decay_precision_gamma`
: factor by which to multiply the step size at each precision-driven reduction (default 0.5).

`continue_chains`
: make `ngme(start = previous_fit)` a true continuation (default `TRUE`). Three things follow from it: `start_sd` is not applied, each chain resumes from its OWN final parameters and its own latent state (`W` and the mixing variables `V`); and the stored optimisation trajectory is concatenated across the restarts instead of replaced. A run split into chunks is then the same optimisation as one long run.

  Set `FALSE` to recover the old behaviour, which is what you want if you are deliberately re-dispersing chains from a fitted point to test whether they return to it. The per-chain part is skipped, with a warning, when the previous fit has a

different parameterization; the no-jitter part still applies, since it needs nothing from the previous fit but the fact that it is one.

`R_hat_conv_check`
: use the R-hat diagnostic for convergence checking

`max_R_hat`
: maximum allowed R_hat

Details

Convergence diagnostics (multi-chain): * R-hat: per-parameter Gelman–Rubin statistic; passes if `R_hat <= max_R_hat`. * Trend: fits a weighted linear trend over a window of `n_slope_check` points and passes when the drift is slower than `trend_rel_lim`, as relative movement per 100 iterations. This is scale-free and independent of `n_batch`. `use_std_check = TRUE` additionally requires `sqrt(var)/|mean| <= std_lim`. The window points are sub-batch means spread evenly over the SECOND HALF of the run so far, so the test is available from the first checkpoint and never regresses through the optimiser's initial transient. Checks are evaluated every `iters_per_check` iterations. A parameter is marked converged only if every enabled parameter-level diagnostic (R-hat and Trend/Std) passes, so a single diagnostic cannot declare convergence on its own; the run stops when all parameters converge. Disable a diagnostic (`R_hat_conv_check = FALSE` or `trend_std_conv_check = FALSE`) to drop it from the requirement. The criteria must hold on `n_conv_batch` consecutive checkpoints, since a single passing checkpoint is weak evidence.

Value

list of control variables

`control_opt_batch_ci` *Generate CI-focused control settings for batch-means inference*

Description

Convenience wrapper for `control_opt()` with defaults tailored for Xi-style batch-means confidence intervals.

Usage

```
control_opt_batch_ci(
  optimizer = sgd(stepsize = 0.03),
  burnin = 100,
  iterations = 2000,
  iters_per_check = 100,
  n_parallel_chain = 4,
  alpha = 0.501,
  t0 = 1,
  schedule_burnin_iter = 0,
  ...
)
```

Arguments

optimizer	optimizer object, default sgd(stepsize = 0.03).
burnin	burn-in iterations before optimization.
iterations	optimization iterations.
iters_per_check	iterations between checkpoints (default 100).
n_parallel_chain	number of parallel chains.
alpha	polynomial stepsize exponent for poly_decay(alpha, t0).
t0	non-negative schedule offset.
schedule_burnin_iter	non-negative integer. Initial optimization iterations where polynomial schedule scaling is disabled.
...	additional arguments forwarded to control_opt() to override defaults.

Details

This helper enforces trajectory-friendly defaults:

- store_traj = TRUE
- trend_std_conv_check = FALSE
- R_hat_conv_check = FALSE
- stepsize_control = poly_decay(alpha, t0, schedule_burnin_iter)

Any of these can still be overridden through ....

Value

object of class control_opt.

---

create_paired_cv_splits

Create paired indices for bivariate cross-validation Ensures that paired observations (e.g., u_wind and v_wind at same location) are kept together in the same fold

---

Description

Create paired indices for bivariate cross-validation Ensures that paired observations (e.g., u_wind and v_wind at same location) are kept together in the same fold

Usage

```
create_paired_cv_splits(data, loc_col, group, k, seed = NULL)
```

Arguments

data	data frame containing the observations
loc_col	character vector of column names defining locations (e.g., c("lon", "lat") or c("x", "y"))
group	character, name of the group column (e.g., "direction" for wind data)
k	number of folds
seed	random seed

Value

list with test_idx and train_idx, each containing k lists of indices

Examples

```
# Example with wind data
data_long <- data.frame(
  lon = rep(c(1, 2, 3, 4), 2),
  lat = rep(c(1, 1, 2, 2), 2),
  direction = rep(c("u_wind", "v_wind"), each = 4),
  wind = rnorm(8)
)
splits <- create_paired_cv_splits(data_long, c("lon", "lat"), "direction", k = 2)
```

---

cross_validation *Compute the cross-validation for the ngme model Perform cross-validation for ngme model first into sub_groups (a list of target, and train data)*

Description

Compute the cross-validation for the ngme model Perform cross-validation for ngme model first into sub_groups (a list of target, and train data)

Usage

```
cross_validation(
  ngme,
  type = "k-fold",
  seed = NULL,
  print = TRUE,
  N_sim = 5,
  n_gibbs_samples = 500,
  n_burnin = 100,
  k = 5,
  percent = 0.2,
  times = 10,
```

```
  metric = NULL,
  transform = identity,
  test_idx = NULL,
  train_idx = NULL,
  keep_pred = FALSE,
  parallel = FALSE,
  thining_gap = 1,
  max_num_threads = NULL,
  cores_layer1 = if (parallel) min(parallel::detectCores(), 2) else 1,
  cores_layer2 = if (parallel) min(parallel::detectCores(), 2) else 1,
  merge_groups = FALSE,
  merged_group_name = NULL,
  data = NULL,
  chain_combine = c("param_mean", "predictive_average")
)
```

Arguments

ngme	a ngme object, or a list of ngme object (if comparing multiple models)
type	character, in c("k-fold", "loo", "lpo", "custom") k-fold is k-fold cross-validation, provide k loo is leave-one-out, lpo is leave-percent-out, provide percent from 1 to 100 custom is user-defined group, provide target and data
seed	random seed. If NULL (the default), a seed is drawn from the current R random number stream, so set.seed() makes the result reproducible.
print	print information during computation
N_sim	integer, number of simulations (e.g., estimate MAE, MSE, .. N times)
n_gibbs_samples	number of gibbs samples of latent process, used for computing CRPS, sCRPS
n_burnin	number of burnin
k	integer (only for k-fold type)
percent	how many percent for testing? from 0 to 1 (for lpo type)
times	how many test cases (only for lpo type)
metric	Optional function or list of functions (one per model) that maps the group-wise observations/predictions for a single location to the quantity that should be scored. The function receives a list containing at least y (a named numeric vector of group values) and may optionally use samples1 and samples2 (matrices with rows named by group and columns indexing posterior draws). The function must return either a numeric scalar (optionally named), or a list with component y and optional components samples1, samples2, and label. When NULL, the original per-group scores are computed. Example: metric = function(data) 2 * data$y["A"] + data$y["B"]. To sum all groups, return sum(data$y).
transform	maps each value to a comparison scale, element-wise, applied to the observations and to every posterior draw.
test_idx	a list of indices of the data (which data points to be predicted) (only for custom type)

train_idx	a list of indices of the data (which data points to be used for re-sampling (not re-estimation)) (only for custom type)
keep_pred	logical, keep test information (pred_1, pred_2) in the return (as attributes), pred_1 and pred_2 are the prediction of the two chains
parallel	logical, run in parallel mode
thining_gap	integer, the gap between samples for thinning, if 0, then no thinning, if 1, then keep 50% of the samples for CRPS, sCRPS, etc.
max_num_threads	cap on OpenMP threads for the scoring passes. Fitting sets this from control_opt, but scoring does not, so a session that only cross-validates inherits one thread per core; several concurrent runs then oversubscribe. NULL (default) leaves the current setting untouched.
cores_layer1	integer, number of cores for the first layer (over testing samples)
cores_layer2	integer, number of cores for the second layer (over computing scores for N_sim simulations)
merge_groups	logical, if TRUE, merge groups as vector components (e.g., for vector-valued wind data with north_wind, east_wind). MAE becomes Euclidean distance, MSE becomes squared Euclidean distance, etc.
merged_group_name	character, name for the merged group when merge_groups=TRUE. If NULL, uses "group1_group2" format (default: NULL)
data	optional data.frame used to replace the original fitting data before running CV. If ‘NULL‘, the data stored in ‘ngme‘ is used. If provided, the model is rebuilt on ‘data‘ while reusing fitted parameters from ‘ngme‘.
chain_combine	how to combine multiple optimization chains: ‘"param_mean"‘ uses the fitted object directly (default), while ‘"predictive_average"‘ computes predictions from each optimization chain and averages at the predictive level.

Value

A list with components:

mean.scores Mean of the N_sim estimates for MSE, MAE, CRPS, and sCRPS.

sd.scores Standard deviation of the N_sim estimates for MSE, MAE, CRPS, and sCRPS.

---

f *Specifying a latent process model (wrapper function for each model)*

Description

Function used for defining of smooth and spatial terms within ngme model formulae. The function is a wrapper function for specific submodels. (see ngme_models_types() for available models).

Usage

```
f(
  map,
  model,
  noise = noise_normal(),
  name = "field",
  data = NULL,
  group = NULL,
  which_group = NULL,
  replicate = NULL,
  A = NULL,
  W = NULL,
  fix_W = FALSE,
  fix_K = FALSE,
  prior = NULL,
  subset = rep(TRUE, length_map(map)),
  debug = FALSE
)
```

Arguments

map	symbol or numerical value: index or covariates to build index
model	an ngme_operator or ngme_operator_def created by model constructors such as ar1(), matern(), bv(), etc.
noise	ngme_noise object, noise_nig() or noise_gal() For a stationary nig or normal_nig noise whose nu prior was not set by the user, f() installs the penalised-complexity prior of prior_pc_nu on nu, that is $1/\nu \sim \mathrm{Exp}(\lambda)$ with λ calibrated from $\Pr(1/\nu > U) = \alpha$. The rate does not depend on the mesh or on the size of the domain; change it with options(ngme2.pc_nu_U =, ngme2.pc_nu_alpha =) or by passing priors(nu = prior_pc_nu(...)) to the noise.
name	name of the field, for later use, if not provided, will be "field1" etc.
data	specifed or inherit from ngme() function
group	group factor indicate resposne variable, can be inherited from ngme() function, (used for bivariate model)
which_group	belong to which group
replicate	factor indicating replicate structure for INLA-style replicates. When provided, operators and A matrices will be block-diagonalized across replicates.
A	observation matrix, automatically computed given map and model
W	starting value of the process
fix_W	stop sampling for W
fix_K	fix the estimation of parameter of K
prior	prior specification created by prior_*() or priors(...) for operator parameters (theta_K).
subset	subset of the model
debug	debug mode

Details

When using different meshes for different replicates, provide the mesh parameter as a list of mesh objects. The number of meshes should match the number of replicates. For example: `mesh = list(mesh1, mesh2, mesh3)` for 3 replicates.

When using the 'replicate' argument, the operator matrices will be block-diagonalized. For a generic operator K = param1 * Matrix1 + param2 * Matrix2, with 2 replicates, the resulting operator becomes: K = param1 * bdiag(Matrix1_rep1, Matrix1_rep2) + param2 * bdiag(Matrix2_rep1, Matrix2_rep2) Similarly, the A matrix will be block-diagonalized as bdiag(A_rep1, A_rep2, ...).

Value

a list for constructing latent model, e.g. A, h, C, G, which also has 1. Information about K matrix 2. Information about noise 3. Control variables

---

gal *The Generalized Asymmetric Laplace (GAL) Distribution*

Description

Density, distribution function, quantile function and random generation for the generalized asymmetric Laplace distribution with parameters `mu`, `sigma` and `nu`, `delta`.

Usage

```
dgal(x, delta, mu, nu, sigma, log = FALSE)

rgal(n, delta, mu, nu, sigma, seed = 0)

pgal(q, delta, mu, nu, sigma, lower.tail = TRUE, log.p = FALSE)

qgal(p, delta, mu, nu, sigma, lower.tail = TRUE, log.p = FALSE)
```

Arguments

`x`, `q`	vector of quantiles.
`delta`	A numeric value for the location parameter.
`mu`	A numeric value for the shift parameter.
`nu`	A numeric value for the shape parameter.
`sigma`	A numeric value for the scaling parameter.
`log`, `log.p`	logical; if `TRUE`, probabilities/densities p are returned as $log(p)$.
`n`	number of observations.
`seed`	Seed for the random generation.
`lower.tail`	logical; if `TRUE`, probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
`p`	vector of probabilities.

Details

The generalized asymmetric Laplace distribution has density given by

$$f(x;p,a,b) = \frac{e^{\nu+\mu(x-\delta)/\sigma^2}\sqrt{\nu\mu^2/\sigma^2+\nu^2}}{\pi\sqrt{\nu\sigma^2+(x-\delta)^2}} K_1(\sqrt{(\nu\sigma^2+(x-\delta)^2)(\mu^2/\sigma^4+\nu/\sigma^2)}),$$

where K_p is modified Bessel function of the second kind of order p, $x > 0$, $\nu > 0$ and $\mu, \delta, \sigma \in \mathbb{R}$. See Barndorff-Nielsen (1977, 1978 and 1997) for further details.

If the mixing variable V follows a Gamma distribution (same parameterization in R):

$$V \sim \Gamma(h\nu, \nu),$$

then the poserior follows the GAL distribution (a special case of GIG distribution):

$$-\mu + \mu V + \sigma\sqrt{V}Z \sim GIG(h\nu - 0.5, 2\nu + (\frac{\mu}{\sigma})^2, 0)$$

Value

dgal gives the density, pgal gives the distribution function, qgal gives the quantile function, and rgal generates random deviates.

Invalid arguments will result in return value NaN, with a warning.

The length of the result is determined by `n` for rgal.

References

Barndorff-Nielsen, O. (1977) Exponentially decreasing distributions for the logarithm of particle size. Proceedings of the Royal Society of London.

Series A, Mathematical and Physical Sciences. The Royal Society. 353, 401–409. doi:10.1098/rspa.1977.0041

Barndorff-Nielsen, O. (1978) Hyperbolic Distributions and Distributions on Hyperbolae, Scandinavian Journal of Statistics. 5, 151–157.

See Also

dgig, dig, digam

Examples

```
rgal(100, delta = 0, mu = 5, sigma = 1, nu = 1)
pgal(0.4, delta = 0, mu = 5, sigma = 1, nu = 1)
qgal(0.8, delta = 0, mu = 5, sigma = 1, nu = 1)
plot(function(x){dgal(x, delta = 0, mu = 5, sigma = 1, nu = 1)}, main =
"generalized asymmetric Laplace density", ylab = "Probability density",
xlim = c(0,10))
```

`generic` *Generic precision matrix operator*

Description

Creates a flexible precision matrix K by allowing linear combinations of base matrices with parameter-dependent coefficients. The model follows the form: K = sum_i f_i(theta) * matrices_i, where f_i depends on the parameter transformation.

Usage

```
generic(
  theta_K,
  matrices,
  h,
  trans = NULL,
  mesh = NULL,
  zero_trace = FALSE,
  model = "generic",
  param_name = NULL,
  param_trans = NULL,
  ...
)
```

Arguments

`theta_K`	The parameter vector (in real scale). If missing, initialized as double(0)
`matrices`	A list of matrices to be combined with parameter-dependent coefficients
`h`	The integration weights vector
`trans`	Transformation for each parameter. Must be a list where each element is a character vector defining transformations for each matrix. For example, 'trans=list(kappa=c("exp2", "identity", "null"))' means parameter 'kappa' affects matrix 1 with exp2 transformation, matrix 2 with identity, and doesn't affect matrix 3. Available transformations: "identity", "exp", "exp2", "exp4", "sqrt", "square", "log", "tanh", "sech"
`mesh`	The mesh object
`zero_trace`	Whether the trace of K should be zero
`model`	The model type name
`param_name`	Optional parameter names stored on the returned operator for diagnostics
`param_trans`	Optional parameter transformations stored on the returned operator for diagnostics
`...`	Additional arguments (ignored)

Details

This function can be used to represent various models like AR1, Matern, and RW1 in a unified framework.

The 'generic' function provides a flexible way to construct precision matrices by combining existing matrices with transformed parameters. Here's how it works:

1. Each parameter in 'theta_K' can affect one or more matrices in the 'matrices' list 2. The 'trans' list defines how each parameter transforms before multiplying each matrix 3. For each matrix, the coefficients from all parameters are multiplied together 4. The final K is the sum of all transformed matrices

Common use cases:

- **AR1 model**: K = rho * C + G (where rho is between -1 and 1) - **Matern (alpha=2)**: K = kappa^2 * C + G (where kappa > 0) - **Matern (alpha=4)**: K = kappa^4 * C + 2*kappa^2 * G + G * Cinv * G

Value

An object of class `ngme_operator`. The object stores the sparse precision matrix `K`, integration weights `h`, parameter vector `theta_K`, base `matrices`, transformation specification `trans`, and an `update_K` function that rebuilds `K` for new parameter values. It is intended for use as the latent `model` inside `f()`.

Examples

```
# AR1 model with rho = 0.5
ar1_obj <- ar1(1:10, rho = 0.5)
g <- name2fun("tanh", inv = TRUE)
generic_ar1 <- generic(
  theta_K = c(rho = g(0.5)),  # Transform from real scale
  # rho on C, nothing on G, sqrt(1 - rho^2) on the stationary (1,1) entry
  trans = list(rho = c("tanh", "null", "sech")),
  matrices = list(ar1_obj$C, ar1_obj$G, ar1_obj$E11),
  h = ar1_obj$h
)

# Matern model with kappa = 2
mesh <- fmesher::fm_mesh_1d(seq(0, 1, length.out = 10))
matern_obj <- matern(mesh)
log_kappa <- log(2)
generic_matern <- generic(
  theta_K = c(kappa = log_kappa),
  trans = list(kappa = c("exp2", "null")),  # exp(2*kappa) for C, nothing for G
  matrices = list(matern_obj$C, matern_obj$G),
  h = matern_obj$h
)

# Matern model with alpha = 4 and kappa = 2
set.seed(1)
mesh <- fmesher::fm_mesh_2d(cbind(x = runif(20), y = runif(20)))
matern_obj <- matern(mesh, alpha = 4)
```

```
C <- matern_obj$C
G <- matern_obj$G
Cinv <- C; diag(Cinv) <- 1 / Matrix::diag(C)

generic_matern4 <- generic(
  theta_K = c(kappa = log(2)),
  trans = list(kappa = c("exp4", "exp2", "null")),
  matrices = list(C, 2*G, G %*% Cinv %*% G),
  h = matern_obj$h
)
```

`generic_ns` *Non-stationary precision matrix operator with custom matrix combinations*

Description

Creates a flexible non-stationary precision matrix K by allowing both linear combinations and matrix multiplications with parameter-dependent diagonal matrices. This enables modeling spatially varying parameters through basis expansions.

Usage

```
generic_ns(
  theta_K,
  matrices,
  position,
  h,
  model = "generic_ns",
  B_theta_K = NULL,
  trans = NULL,
  mesh = NULL,
  zero_trace = FALSE,
  param_name = NULL,
  param_trans = NULL,
  ...
)
```

Arguments

`theta_K`	List of parameter vectors (in real scale)
`matrices`	List of fixed matrices to be used in the model
`position`	List of vectors specifying matrix combinations (required)
`h`	The integration weights vector
`model`	The model type name stored on the returned operator
`B_theta_K`	List of basis matrices for parameters, defaults to matrices of 1s if not provided

trans	List of transformations for each parameter (one transformation per parameter)
mesh	The mesh object
zero_trace	Whether the trace of K should be zero
param_name	Optional parameter names stored on the returned operator for diagnostics
param_trans	Optional parameter transformations stored on the returned operator for diagnostics
...	Additional arguments (ignored)

Details

The key distinction from 'generic' is that 'generic_ns': 1. Requires explicit position specification for matrix combinations 2. Converts parameters to diagonal matrices for multiplication with fixed matrices 3. Allows for basis expansions via B_theta_K

The 'generic_ns' operator constructs K through the following steps:

1. Create diagonal matrices for each parameter using basis expansions: D_param = diag(B_param * theta_param) 2. Combine parameter matrices with fixed matrices according to position specifications 3. Sum all the resulting combinations

The 'position' parameter defines how matrices are combined: - Each element of the list represents a term in the sum - Each vector contains indices of matrices to multiply (parameter matrices first, then fixed matrices) - For example: 'position = list(c(1, 3), c(2, 4))' with one parameter means: Multiply the parameter diagonal matrix (index 1) with fixed matrix 1 (index 3), then add parameter diagonal matrix 2 (index 2) multiplied by fixed matrix 2 (index 4)

Spatially-varying parameters can be created using basis matrices in B_theta_K.

Value

An object of class `ngme_operator`. The object stores the sparse non-stationary precision matrix `K`, integration weights `h`, the flattened parameter vector `theta_K`, basis matrices `B_theta_K`, parameter grouping metadata `param_map`, matrix combination rules `position`, and an update function for rebuilding `K`. It is intended for use as the latent `model` inside `f()`.

Examples

```
# Simple example with one parameter
n <- 5
A <- matrix(1, n, n)
B <- matrix(2, n, n)
alpha <- 0.4

# Create a simple generic_ns model: D_alpha * A + B
model <- generic_ns(
  theta_K = list(alpha = c(alpha)),
  matrices = list(A, B),
  position = list(c(1, 2), c(3)), # D_alpha * A + B
  h = rep(1, n)
)

# AR1 model representation: rho * C + G (without the stationary (1,1) entry,
```

```
# which is not a linear coefficient -- see ar1_obj$E11 and the "sech"
# transformation in name2fun())
ar1_obj <- ar1(1:10, rho = 0.5)
g <- name2fun("tanh", inv = TRUE)
generic_ar1 <- generic_ns(
  theta_K = list(x = c(g(0.5))), # Transform from real scale
  trans = list(x = "tanh"), # Apply tanh transformation
  matrices = list(ar1_obj$C, ar1_obj$G),
  position = list(c(1, 2), c(3)), # D_rho * C + G
  h = ar1_obj$h,
  mesh = 1:10
)

# Spatially varying Matern model
# Create a simple 1D mesh
mesh <- fmesher::fm_mesh_1d(seq(0, 1, length.out = 10))

# Create basis for spatially-varying kappa
n <- 10
B_kappa <- matrix(0, n, 2)
B_kappa[1:(n / 2), 1] <- 1 # First half of the domain
B_kappa[(n / 2 + 1):n, 2] <- 1 # Second half of the domain

# Create standard Matern components
matern_model <- matern(mesh)

# Create model with space-varying kappa
ns_model <- generic_ns(
  theta_K = list(kappa = c(log(1), log(2))), # Different values for each region
  matrices = list(matern_model$C, matern_model$G),
  B_theta_K = list(kappa = B_kappa),
  trans = list(kappa = "exp2"), # kappa^2 transformation for C
  h = matern_model$h,
  position = list(c(1, 2), c(3)), # D_kappa * C + G
  mesh = mesh
)
```

`get_data_from_formula` *Extracts design matrix from a formula and data.*

Description

This function takes a formula and a data frame, and returns a design matrix based on the right-hand side of the formula. If the formula is '~.', it treats all columns in the data as the design matrix. Otherwise, it constructs a design matrix without an intercept based on the specified formula.

Usage

```
get_data_from_formula(form, data)
```

Arguments

form	A formula object, e.g., '~ x1 + x2' or '~.'.
data	A data frame or matrix from which to extract the design matrix.

Value

A numeric matrix representing the design matrix.

Examples

```
data(mtcars)
# Extract a design matrix for 'mpg' and 'cyl'
X1 <- get_data_from_formula(~ mpg + cyl, mtcars)
# Extract a design matrix for all columns
X2 <- get_data_from_formula(~., mtcars)
# Extract a design matrix for 'hp'
X3 <- get_data_from_formula(~hp, mtcars)
```

---

get_parameter_distance

Calculate parameter distance from true values

Description

Compute ||theta - theta_hat|| distance for all parameters across iterations. This provides a single measure of convergence combining all parameters.

Usage

```
get_parameter_distance(
  trajectories,
  true_values,
  norm_type = "euclidean",
  chain_summary = "mean"
)
```

Arguments

trajectories	ngme_trajectories object from get_trace_trajectories()
true_values	named list or vector of true parameter values
norm_type	character, type of norm to use: "euclidean" (default), "manhattan", "max"
chain_summary	character, how to summarize across chains: "mean" (default), "median", "chain1"

Value

numeric vector of distances across iterations

get_trace_trajectories *Get trace trajectories from ngme fitting*

Description

Extract numerical trajectories of parameters during ngme optimization without creating plots. This function provides the underlying data used by traceplot().

Usage

```
get_trace_trajectories(ngme, name = "general", apply_transform = TRUE)
```

Arguments

ngme	ngme object
name	name of latent models, otherwise get fixed effects and measurement noise should be in names(ngme$models) or other
apply_transform	logical, whether to apply parameter transformations (default: TRUE)

Value

list containing:

trajectories named list of trajectory matrices for each parameter

parameter_names character vector of parameter names

transformations list of transformation functions used

n_chains number of chains

n_iterations number of iterations

get_trajectories *get the trajectories of parameters of the model*

Description

Get the trajectories of the parameters of the model

Usage

```
get_trajectories(ngme_object, model_name)
```

Arguments

ngme_object	ngme object
model_name	name of the model, if NULL, return the trajectories of the parameters of the measurement noise and fixed effects

Value

a list of trajectories, each element is a matrix of trajectories (n_samples * n_trajectories)

---

gig *The Generalised Inverse-Gaussian (GIG) Distribution*

Description

Density, distribution function, quantile function and random generation for the generalised inverse-Gaussian distribution with parameters p, a and b.

Usage

```
dgig(x, p, a, b, log = FALSE)

rgig(n, p, a, b, seed = ngme_random_seed())

pgig(q, p, a, b, lower.tail = TRUE, log.p = FALSE)

qgig(prob, p, a, b, lower.tail = TRUE, log.p = FALSE)
```

Arguments

x, q	vector of quantiles.
p	parameter p.
a, b	parameters a and b. Must be positive.
log, log.p	logical; if TRUE, probabilities/densities p are returned as $log(p)$.
n	number of observations.
seed	Positive integer seed for the sampler. Defaults to a draw from R's random number stream, so 'set.seed()' makes the result reproducible. A value of 0 seeds from the system clock, which is *not* reproducible.
lower.tail	logical; if TRUE, probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
prob	vector of probabilities.

Details

The generalised inverse-Gaussian distribution has density given by

$$f(x; p, a, b) = ((a/b)^{p/2})/(2K_p(\sqrt{ab}))x^{p-1} \exp\{-(a/2)x - (b/2)/x\},$$

where K_p is modified Bessel function of the second kind of order p, $x > 0$, $a, b > 0$ and $p \in \mathbb{R}$. See Jørgensen (1982) for further details.

Value

dgig gives the density, pgig gives the distribution function, qgig gives the quantile function, and rgig generates random deviates.

Invalid arguments will result in return value NaN, with a warning.

The length of the result is determined by n for rgig.

References

Jørgensen, Bent (1982). Statistical Properties of the Generalized Inverse Gaussian Distribution. Lecture Notes in Statistics. 9. New York–Berlin: Springer-Verlag. doi:10.1007/9781461256984

See Also

dnig, dig, digam

Examples

```
rgig(20, p = 1, a = 1, b = 1)
pgig(0.4, p = 1, a = 1, b = 1)
qgig(0.8, p = 1, a = 1, b = 1)
plot(function(x){dgig(x, p = 1, a = 1, b = 1)}, main =
"Generalised inverse-Gaussian density", ylab = "Probability density",
xlim = c(0,10))
```

---

group_cv *Leave-group-out cross-validation*

Description

Scores each fold against a posterior that never saw it, conditioning on the fitted parameters as [cross_validation()] does, but reaching that posterior by the cheapest route the model allows.

Usage

```
group_cv(
  ngme,
  type = c("k-fold", "loo", "lpo", "custom"),
  seed = NULL,
  print = TRUE,
  N_sim = 5,
  n_gibbs_samples = 1000,
  n_burnin = 200,
  k = 5,
  percent = 0.2,
  times = 10,
  metric = NULL,
  transform = identity,
```

```
  test_idx = NULL,
  train_idx = NULL,
  keep_pred = FALSE,
  parallel = FALSE,
  thining_gap = 1,
  max_num_threads = NULL,
  cores_layer1 = 1,
  cores_layer2 = 1,
  merge_groups = FALSE,
  merged_group_name = NULL,
  data = NULL,
  chain_combine = c("param_mean", "predictive_average"),
  method = c("auto", "lowrank", "exact"),
  max_lowrank_group = 30L,
  n_score_draws = 4000L,
  n_q = 8L,
  crps_max = 200000L,
  num_threads = 1,
  chunk_cols = 256,
  n_chains = 1L,
  min_ess = 100,
  khat_threshold = 0.7,
  n_draw_out = 0L,
  r_eff = TRUE
)
```

Arguments

ngme	a fitted ‘ngme‘ object.
type	how to form the folds, as in [cross_validation()]: ‘"loo"‘ (one group per observation), ‘"k-fold"‘, ‘"lpo"‘ (leave-‘percent‘-out, repeated ‘times‘), or ‘"custom"‘ to supply the design directly.
seed	random seed. If ‘NULL‘, drawn from the current R stream.
print	logical; emit the advisory warnings (ignored arguments, a ‘min_ess‘ that will flag everything, a method that will degenerate). ‘FALSE‘ silences them, as it silences [cross_validation()]’s progress output. It does not affect what is returned.
N_sim	accepted as a synonym for ‘n_chains‘, which is the same quantity: ‘N_sim‘ independent runs of ‘n_gibbs_samples‘ draws each. An explicit ‘n_chains‘ takes precedence. It has an effect only where folds have their own chains – see ‘n_chains‘.
n_gibbs_samples	number of posterior draws to retain. **Ignored for a fully Gaussian model**, where the answer is exact from the factorization and one draw suffices; honouring a large value there would cost orders of magnitude more for the same number. What governs the error elsewhere is not this count but the r_eff-corrected effective sample size it yields, which is a fraction of it because the draws come from a Markov chain. Check the ‘ess‘ column rather than assuming. The default is higher than [cross_validation()]’s, and ‘n_burnin‘ likewise: that function

splits its budget over ‘N_sim‘ repeats of each fold, while here one chain carries all of it. This is the only default that differs between the two.

`n_burnin` number of burn-in sweeps.

`k` number of folds for ‘type = "k-fold"‘.

`percent` proportion held out per fold for ‘type = "lpo"‘.

`times` number of repetitions for ‘type = "lpo"‘.

`metric` **not supported.** A custom scoring function; this function reports a fixed set of scores. Supplying it is an error rather than a warning, because quietly returning different scores than were asked for would be worse.

`transform` optional function applied to the predictive and to the observation before scoring, so scores are reported on a different scale. It may take a second argument, the observation index, for a position-dependent transform. A named list of functions scores the same folds on each scale and returns one result per entry. ‘log_score‘ is only defined on the untransformed scale.

`test_idx` for ‘type = "custom"‘, a list of integer vectors – one per fold, holding the observations to be **scored**. As in [cross_validation()].

`train_idx` for ‘type = "custom"‘, a list of the same length holding the observations each fold may **condition on**. Optional: left out, each fold trains on everything it does not score.

`keep_pred` logical; if ‘TRUE‘, also return draws from each fold’s predictive, as [cross_validation()] does. A shorthand for ‘n_draw_out‘, which sets how many.

`parallel`, `cores_layer1`, `cores_layer2`, `thining_gap`, `chain_combine`
ignored. Parallelism happens inside C++ across folds and replicates – see ‘num_threads‘ – so there are no nested worker pools to configure, and there is one chain rather than several to thin or combine.

`max_num_threads`
accepted as a synonym for ‘num_threads‘.

`merge_groups`, `merged_group_name`, `data`
not supported. Paired splits for grouped data; use [cross_validation()].

`method` force a strategy instead of dispatching on model class. See Details. ‘"lowrank"‘ on a non-Gaussian latent field warns, because that is where the weights degenerate.

`max_lowrank_group`
largest group the low-rank route will accept under ‘method = "auto"‘ before deferring to per-fold chains.

`n_score_draws` draws used for CRPS and sCRPS when a ‘transform‘ is in force, where the scores have no closed form. Unused otherwise.

`n_q` number of prior draws of the held-out group’s measurement mixing variable taken per retained posterior draw. The reported ‘mean‘ and ‘sd‘ do not depend on this where the measurement family has closed-form mixing moments (normal, NIG, GAL) – there it is used only for the proper scores, which need the whole predictive mixture rather than its first two moments, and for the importance weights. Raise it when the measurement noise is strongly heavy tailed. Ignored for Gaussian measurement noise, where the mixing variable is the constant 1.

`crps_max` budget, in mixture-component pairs, for the 'E|X - X'|' term behind CRPS and sCRPS. The exact double sum is taken whenever it fits inside the budget; past that the term is estimated from this many sampled pairs, which is unbiased and draws on every component. It is what limits the precision of those two scores; the other scores do not use it.

`num_threads` number of OpenMP threads. How they are spent depends on the route: the low-rank one parallelises across replicates first and then across groups within a chain; the per-fold routes run replicates one after another and parallelise across **(fold, chain) pairs**.

`chunk_cols` bounds how many right-hand sides are handed to the solver at once, and so the peak memory of the dense solve. Larger is faster until the dense block stops fitting in cache.

`n_chains` run this many independent chains per fold, of 'n_gibbs_samples' draws each, started from the stored per-chain states of the fit where the object kept them. Their between-chain spread is reported as 'rhat', and is a real convergence signal precisely because the starts are independent; folds that have not settled are warned about, never dropped.

This applies to the routes that give each fold its own chain. The low-rank route draws one full-data chain that every fold shares, so there is nothing per-fold to replicate: 'n_chains' there is reported as ignored, and 'n_gibbs_samples' is the knob that governs its error.

`min_ess` groups whose r_eff-corrected effective sample size falls below this are reported as unreliable. **This is the trigger that matters.** Pareto k diagnoses the tail of the weight distribution; what actually fails on the low-rank route is how slowly the chain mixes, and only the corrected ESS sees that. Groups where the estimator is badly biased can leave k untouched while their corrected ESS collapses.

A flagged group is a group to re-score exactly, not one to throw draws at. Where the weights are degenerate the effective sample size does not grow with 'n_gibbs_samples': more draws can shrink the spread while the estimate concentrates on the wrong value. Low variance and large bias together is the dangerous failure mode, and it looks like convergence.

`khat_threshold` groups whose Pareto k exceeds this are also reported as unreliable. 0.7 is the usual PSIS cutoff. Keep it, but do not rely on it alone – see 'min_ess'.

`n_draw_out` if positive, also return 'n_draw_out' draws from each group's leave-group-out predictive, as a list of '|I| x n_draw_out' matrices in 'attr(x, "draws")'. The predictive is a weighted mixture of Gaussians, so these are exact draws from it rather than a re-run of any sampler. Useful for plotting it against what [cross_validation()] simulates.

`r_eff` logical; correct the effective sample size for the autocorrelation of the chain. Leave this on: the weight-based ESS alone assumes independent draws and will call a group reliable when it is not.

Details

It is meant as a drop-in for [cross_validation()]: it takes that function's arguments under the same names, so an existing call can be switched over by changing the function name alone. A few of

them describe machinery this function does not have and are either reported as ignored or refused outright.

‘method = "auto"‘ picks between three strategies:

* **Fully Gaussian fit** – closed form. The leave-group-out posterior is available exactly from the factorization already computed, so no sampling happens at all and ‘n_gibbs_samples‘ is ignored. * **Gaussian latent field, non-Gaussian measurement noise** – one full-data Gibbs chain serves every fold. Conditional on the mixing variables the latent posterior is Gaussian with precision ‘QQ‘, and dropping a group subtracts a rank-|I| term from it, so each fold costs a handful of triangular solves against the existing factor. Those draws target ‘p(. | y)‘ rather than ‘p(. | y_-I)‘, and are corrected by Pareto-smoothed importance weights ‘1 / p(y_I | y_-I, V)‘. * **Anything else** (a non-Gaussian latent field) – a real leave-group-out Gibbs chain per fold, run inside C++ and parallel across folds.

Only the second route approximates anything, and its error is diagnosed per group rather than assumed away.

As |I| grows both the linear algebra advantage and the importance weights degrade together, so ‘"auto"‘ hands groups larger than ‘max_lowrank_group‘ to the per-fold chains instead.

Like [cross_validation()], parameters are **not** re-estimated per fold: everything conditions on the fitted values in ‘ngme‘.

Value

An object of class ‘ngme_group_cv‘: a data frame with one row per fold, holding the fold’s identity (‘group‘, ‘size‘, ‘n_held_out‘), the predictive summary (‘mean‘, ‘sd‘), the scores (‘log_score‘, ‘MAE‘, ‘MSE‘, ‘neg.CRPS‘, ‘neg.sCRPS‘ – all lower-is-better despite the ‘neg.‘, which is inherited from [cross_validation()]), and the diagnostics (‘khat‘, ‘ess‘, ‘r_eff‘, ‘rhat‘, ‘exact‘, and the ‘low_ess‘ / ‘high_khat‘ / ‘unreliable‘ flags). A ‘summary‘ attribute carries the score means, and ‘print()‘ reports which folds are untrustworthy and why.

When ‘transform‘ is a named list, a list of such objects is returned, one per scale.

See Also

[cross_validation()], which re-scores by refitting each fold and covers models this does not (correlated measurement noise).

`ig` *The Inverse-Gaussian (IG) Distribution*

Description

Density, distribution function, quantile function and random generation for the inverse-Gaussian distribution with parameters `a` and `b`.

Usage

```
dig(x, a, b, log = FALSE)

rig(n, a, b, seed = ngme_random_seed())

pig(q, a, b, lower.tail = TRUE, log.p = FALSE)

qig(p, a, b, lower.tail = TRUE, log.p = FALSE)
```

Arguments

x, q	vector of quantiles.
a, b	parameters a and b. Must be positive.
log, log.p	logical; if TRUE, probabilities/densities p are returned as $log(p)$.
n	number of observations.
seed	Positive integer seed for the sampler. Defaults to a draw from R's random number stream, so 'set.seed()' makes the result reproducible. A value of 0 seeds from the system clock, which is *not* reproducible.
lower.tail	logical; if TRUE, probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
p	vector of probabilities.

Details

The inverse-Gaussian distribution has density given by

$$f(x; a, b) = \frac{\sqrt{b}}{\sqrt{2\pi x^3}} \exp(-\frac{a}{2}x - \frac{b}{2x} + \sqrt{ab}),$$

where $x > 0$ and $a, b > 0$. In this parameterization, $E(X) = \sqrt{b}/\sqrt{a}$. See Tweedie (1957a, 1957b) for further details.

Value

dig gives the density, pig gives the distribution function, qig gives the quantile function, and rig generates random deviates.

Invalid arguments will result in return value NaN, with a warning.

The length of the result is determined by n for rig.

References

Tweedie, M. C. K. (1957a). "Statistical Properties of Inverse Gaussian Distributions I". Annals of Mathematical Statistics. 28 (2): 362–377. doi:10.1214/aoms/1177706964

Tweedie, M. C. K. (1957b). "Statistical Properties of Inverse Gaussian Distributions II". Annals of Mathematical Statistics. 28 (3): 696–705. doi:10.1214/aoms/1177706881

See Also

dnig, dgig, digam

Examples

```
rig(100, a = 1, b = 1)
pig(0.4, a = 1, b = 1)
qig(0.8, a = 1, b = 1)
plot(function(x){dig(x, a = 1, b = 1)}, main =
"Inverse-Gaussian density", ylab = "Probability density",
xlim = c(0,10))
```

---

igam *The Inverse-Gamma (IGam) Distribution*

Description

Density, distribution function, quantile function and random generation for the inverse-Gamma distribution with parameters a and b.

Usage

```
digam(x, a, b, log = FALSE)

rigam(n, a, b)

pigam(q, a, b, lower.tail = TRUE, log.p = FALSE)

qigam(p, a, b, lower.tail = TRUE, log.p = FALSE)
```

Arguments

x, q	vector of quantiles.
a, b	parameters a and b. Must be positive.
log, log.p	logical; if TRUE, probabilities/densities p are returned as $log(p)$.
n	number of observations.
lower.tail	logical; if TRUE, probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
p	vector of probabilities.

Details

The inverse-Gamma distribution has density given by

$$f(x; a, b) = \frac{b^a}{\Gamma(a)} x^{-a-1} \exp(-\frac{b}{x}),$$

where $x > 0$ and $a, b > 0$.

Value

digam gives the density, pigam gives the distribution function, qigam gives the quantile function, and rigam generates random deviates.

Invalid arguments will result in return value NaN, with a warning.

The length of the result is determined by n for rig.

See Also

dnig, dgig

Examples

```
rigam(100, a = 1, b = 1)
pigam(0.4, a = 1, b = 1)
qigam(0.8, a = 1, b = 1)
plot(function(x){digam(x, a = 1, b = 1)}, main =
"Inverse-Gamma density", ylab = "Probability density",
xlim = c(0,10))
```

---

iid *ngme iid model specification*

Description

ngme iid model specification

Usage

```
iid(mesh = NULL)
```

Arguments

mesh	integer or factor, index to build the mesh

Value

ngme_operator object

`make_time_series_cv_index` *Create Time Series Cross-Validation Indices*

Description

Creates indices for time series cross-validation with options for expanding window or sliding window approaches. Supports both single-step and multi-step forecasting, and can handle replicated observations.

Usage

```
make_time_series_cv_index(
  time_idx,
  train_length = NULL,
  test_length = 1,
  replicate = time_idx,
  gap = 0
)
```

Arguments

`time_idx`	A numeric vector of time indices in ascending order
`train_length`	An integer specifying the fixed length of training sets. If NULL (default), an expanding window approach is used.
`test_length`	An integer specifying the number of observations to include in each test set. Default is 1 (single-step forecasting).
`replicate`	An optional vector of the same length as time_idx, indicating which observations belong to the same replicate group. When provided, ensures that all observations with the same replicate value are either entirely in the training set or entirely in the test set.
`gap`	An integer specifying the gap between the training set and test set. Default is 0 (no gap, test set starts immediately after training set). For example, gap=1 means skip one time point between training and test (useful for 2-step ahead forecasting).

Details

Time series cross-validation requires respecting the temporal order of observations. This function implements two common approaches:

1. Expanding window (when train_length = NULL): The training set grows with each fold, starting with a minimal set and expanding to include all but the test data.

2. Sliding window (when train_length is specified): Uses a fixed-length window that slides through the time series, maintaining the same training size across folds.

The test_length parameter allows for multi-step forecasting evaluation.

When replicate is provided, the function ensures that all observations with the same replicate value are kept together, either all in the training set or all in the test set. This is useful for scenarios where multiple observations at the same time point should be treated as a group.

The gap parameter creates a separation between training and test sets, which is useful for multi-step ahead forecasting validation.

Value

A list with two components:

train A list of numeric vectors, where each vector contains the time indices for training in that fold

test A list of numeric vectors, where each vector contains the time indices for testing in that fold

Examples

```
# Expanding window approach with single-step forecasting
cv_expanding <- make_time_series_cv_index(1:10)

# Sliding window approach with window size 3 and single-step forecasting
cv_sliding <- make_time_series_cv_index(1:10, train_length = 3)

# Sliding window with multi-step forecasting (predict 2 steps ahead)
cv_multistep <- make_time_series_cv_index(1:10, train_length = 3, test_length = 2)

# Working with replicates
time_idx <- c(1, 1, 1, 2, 2, 3, 3)
replicates <- c(1, 1, 1, 2, 2, 3, 3)
cv_with_replicates <- make_time_series_cv_index(time_idx, replicate = replicates)

# 2-step ahead forecasting with a gap
cv_with_gap <- make_time_series_cv_index(1:10, gap = 1)
```

---

matern *ngme Matern SPDE model specification*

Description

ngme Matern SPDE model specification

Usage

```
matern(
  mesh = NULL,
  alpha = 2,
  fix_alpha = TRUE,
  kappa = NULL,
  theta_kappa = NULL,
```

```
  rational_order = 2,
  B_kappa = NULL
)
```

Arguments

mesh	an fmesher::fm_mesh_2d object, mesh for build the SPDE model
alpha	SPDE smoothness parameter (alpha = 2beta) 2 or 4 for integer case, otherwise for fractional case
fix_alpha	if FALSE, enable fractional modeling
kappa	the range parameter, for the stationary model, it is the range parameter
theta_kappa	kappa = exp(B_kappa * theta_kappa), for the non-stationary model, it is the range parameter
rational_order	order of rational approximation for fractional operators (default: 2)
B_kappa	bases for theta_kappa, ignore if use the stationary model

Value

ngme_operator object

---

mean_list *taking mean over a list of nested lists*

Description

taking mean over a list of nested lists

Usage

```
mean_list(lls, weights = NULL)
```

Arguments

lls	a list
weights	weights of each list

Value

a list of nested lists

Examples

```
ls <- list(
  list(a = 1, b = 2, t = "nig", ll = list(a = 1, b = 2, w = "ab")),
  list(a = 3, b = 5, t = "nig", ll = list(a = 1, b = 6, w = "ab")),
  list(a = 5, b = 5, t = "nig", ll = list(a = 4, b = 2, w = "ab"))
)
mean_list(ls)
```

`merge_noise` *Merge 2 noise into 1 noise*

Description

Merge 2 noise into 1 noise

Usage

```
merge_noise(noise1, noise2)
```

Arguments

`noise1`	noise 1
`noise2`	noise 2

Value

merged noise

`merge_replicates` *Merge model of replicates into model of 1 replicate given train_idx and test_idx, the merged model contains all the information of train_idx from different replicates.*

Description

Merge model of replicates into model of 1 replicate given train_idx and test_idx, the merged model contains all the information of train_idx from different replicates.

Usage

```
merge_replicates(ngme, train_idx, test_idx)
```

Arguments

`ngme`	a ngme object
`train_idx`	a vector of indices of train data
`test_idx`	a vector of indices of test data

momentum *Momentum SGD optimization*

Description

Momentum SGD optimization

Usage

```
momentum(stepsize = 0.05, beta1 = 0.9, beta2 = 1 - beta1)
```

Arguments

stepsize	stepsize for SGD
beta1	beta1 for momentum
beta2	beta2 for momentum

Details

The update rule for momentum is:

$$v_t = \beta_1 v_{t-1} + \beta_2 g_t$$
$$x_{t+1} = x_t - \text{stepsize} * v_t$$

Value

a list of control variables for optimization (used in `control_opt` function)

name2fun *Convert transformation name to function*

Description

This function converts a transformation name to the corresponding function. Convertion is from original scale to real scale by default. Available transformations:

- `exp4`: $exp(4x)$, inverse is $log(x)/4$
- `exp2`: $exp(2x)$, inverse is $log(x)/2$
- `tanh`: Hyperbolic tangent transformation used for AR1 parameter, uses ar1_th2a and ar1_a2th
- `sech`: $sqrt(1 - tanh(x)^2)$ for the `tanh` above, i.e. the AR(1) stationary standard deviation $sqrt(1 - rho^2)$. Not invertible, so it has no inverse
- `identity`: Identity function, no transformation
- `exp`: $exp(x)$, inverse is $log(x)$
- `sqrt`: $sqrt(x)$, inverse is x^2
- `square`: x^2, inverse is $sqrt(x)$
- `log`: $log(x)$, inverse is $exp(x)$

Usage

```
name2fun(trans, inv = FALSE)
```

Arguments

trans	Character string specifying the transformation type from original scale to real scale
inv	Logical, if TRUE returns the transformation function from real scale to original scale

Value

A function that applies the specified transformation

---

ngme *Fit an additive linear mixed effect model over replicates*

Description

ngme function performs an analysis of non-gaussian additive models. It does the maximum likelihood estimation via stochastic gradient descent. The prediction of unknown location can be performed by leaving the response variable to be NA. The likelihood is specified by family. The model estimation control can be setted in control using control_opt() function, see ?control_opt for details. See ngme_model_types() for available models.

Usage

```
ngme(
  formula,
  data,
  family = "normal",
  control_opt = NULL,
  control_ngme = NULL,
  group = NULL,
  replicate = NULL,
  start = NULL,
  prior_beta = NULL,
  debug = FALSE
)
```

Arguments

formula	formula
data	a dataframe or a list providing data (Only response variable can contain NA value, NA value in other columns will cause problem)
family	likelihood type, same as measurement noise specification. It can be provided as:

1. a string, e.g., "normal", "nig", "t".
2. an ngme noise object, e.g., noise_normal(), noise_nig(mu = 0, sigma = 1, nu = 1), noise_t(nu=5).

control_opt	control for optimizer. by default it is control_opt(). See ?control_opt for details.
control_ngme	control for ngme model. by default it is control_ngme(). See ?control_ngme for details.
group	factor, used for bivariate model, indicating which group the observation belongs to
replicate	factor, used for divide data into different replicates
start	starting ngme object (usually object from last fit)
prior_beta	prior specification for fixed effects ('beta'), created by prior_*() or priors(...).
debug	toggle debug mode

Value

random effects (for different replicate) + models(fixed effects, measuremnt noise, and latent process)

Examples

```
ngme(
  formula = Y ~ x1 + f(
    x2,
    model = ar1(rho = 0.5),
    noise = noise_nig()
  ) + f(x1,
    model = rw1(),
    noise = noise_normal(),
  ),
  family = noise_normal(sd = 0.5),
  data = data.frame(Y = 1:5, x1 = 2:6, x2 = 3:7),
  control_opt = control_opt(
    estimation = FALSE
  )
)
```

---

ngme_as_sparse *Convert sparse matrix into sparse dgCMatrix*

Description

Convert sparse matrix into sparse dgCMatrix

Usage

```
ngme_as_sparse(G)
```

Arguments

G	matrix

Value

sparse dgCMatrix

---

ngme_batch_ci *Batch-Means Confidence Intervals from an ngme Fit*

Description

Compute Xi-style batch-means confidence intervals directly from trajectories stored in an ‘ngme‘ object.

Usage

```
ngme_batch_ci(
  ngme,
  name = "all",
  level = 0.95,
  alpha = 0.501,
  M = NULL,
  N = NULL,
  apply_transform = TRUE,
  drop_burnin = TRUE,
  burnin_iter = 0
)
```

Arguments

ngme	fitted ‘ngme‘ object with ‘store_traj = TRUE‘ during optimization.
name	either ‘"all"‘ (default) to use all parameters jointly, a latent model name, or ‘"general"‘ for measurement-noise/fixed-effect block.
level	confidence level.
alpha	stepsize decay exponent in $(1/2, 1)$.
M	number of retained batches (excluding burn-in batch 0).
N	decorrelation constant in the batch boundary formula.
apply_transform	logical; if ‘TRUE‘, inference is first computed in the raw optimization space and then mapped to user scale via post-hoc Delta method (Post-delta).
drop_burnin	logical; if ‘TRUE‘, discard batch 0.
burnin_iter	non-negative integer. Explicitly discard the first ‘burnin_iter‘ optimization iterations before Xi-style batching.

Value

Same structure as [batch_means_ci()] with extra metadata fields ‘name‘ and ‘apply_transform‘.

---

ngme_cov_matrix *variance of the data or the latent field*

Description

Compute the variance of the data or the latent field

Usage

```
ngme_cov_matrix(ngme_object, model_name = "data", replicate = 1)
```

Arguments

ngme_object	ngme_model
model_name	if model_name = "data", then return the covariance matrix of the data (without measurement noise) if the model_name is the name or index of the latent, then return the covariance matrix of the latent field
replicate	which replicate (default = 1)

Value

a data.frame of posterior samples (mesh_size * n_post_samples)

---

ngme_make_mesh_repls *ngme make mesh for different replicates*

Description

Make different mesh for different replicates

Usage

```
ngme_make_mesh_repls(data, map, replicate, mesh_type = "regular")
```

Arguments

data	provide the data.frame
map	provide the map to make mesh, i.g. ~x+y, x and y will be extracted from data to make a 2d mesh
replicate	provide the replicate information, i.g. ~id
mesh_type	type of mesh, "regular" means use all the point from same replicate to make a mesh

Value

a list of mesh of length of different replicates

`ngme_model_types` *Show ngme model types*

Description

Show ngme model types

Usage

```
ngme_model_types()
```

Value

available types for models

`ngme_noise` *ngme noise specification*

Description

Function for specifying ngme noise. Please use `noise_nig` and `noise_normal` for simpler usage. Use `ngme_noise_types()` to check all the available types.

Usage

```
ngme_noise(
  noise_type,
  mu = 0,
  sigma = 1,
  nu = 1,
  B_mu = NULL,
  theta_mu = NULL,
  B_sigma = NULL,
  theta_sigma = NULL,
  B_nu = NULL,
  theta_nu = NULL,
  theta_sigma_normal = NULL,
  B_sigma_normal = NULL,
  fix_theta_mu = FALSE,
  fix_theta_sigma = FALSE,
  fix_rho = FALSE,
  fix_theta_sigma_normal = FALSE,
```

```
  fix_theta_nu = FALSE,
  V = NULL,
  fix_V = FALSE,
  single_V = FALSE,
  share_V = FALSE,
  corr_measurement = FALSE,
  index_corr = NULL,
  map_corr = NULL,
  nu_lower_bound = 0,
  rho = double(0),
  prior = NULL,
  ...
)

noise_normal(
  sigma = NULL,
  theta_sigma = NULL,
  B_sigma = matrix(1),
  corr_measurement = FALSE,
  index_corr = NULL,
  ...
)

noise_nig(
  mu = NULL,
  sigma = NULL,
  nu = NULL,
  V = NULL,
  theta_mu = NULL,
  theta_sigma = NULL,
  theta_nu = NULL,
  nu_lower_bound = 0,
  B_mu = matrix(1),
  B_sigma = matrix(1),
  B_nu = matrix(1),
  corr_measurement = FALSE,
  index_corr = NULL,
  ...
)

noise_gal(
  mu = NULL,
  sigma = NULL,
  nu = NULL,
  V = NULL,
  theta_mu = NULL,
  theta_sigma = NULL,
  theta_nu = NULL,
```

```
  nu_lower_bound = 0.01,
  B_mu = matrix(1),
  B_sigma = matrix(1),
  B_nu = matrix(1),
  corr_measurement = FALSE,
  index_corr = NULL,
  ...
)

noise_skew_t(
  mu = NULL,
  sigma = NULL,
  nu = NULL,
  theta_mu = NULL,
  theta_sigma = NULL,
  theta_nu = NULL,
  nu_lower_bound = 0.01,
  B_mu = matrix(1),
  B_sigma = matrix(1),
  B_nu = matrix(1),
  corr_measurement = FALSE,
  index_corr = NULL,
  ...
)

noise_t(
  nu = NULL,
  theta_nu = NULL,
  nu_lower_bound = 0,
  B_nu = matrix(1),
  corr_measurement = FALSE,
  index_corr = NULL,
  ...
)

noise_normal_nig(
  sigma_normal = NULL,
  mu = NULL,
  sigma_nig = NULL,
  nu = NULL,
  V = NULL,
  theta_mu = NULL,
  theta_sigma_nig = NULL,
  theta_sigma_normal = NULL,
  theta_nu = NULL,
  B_mu = matrix(1),
  B_sigma_nig = matrix(1),
  B_sigma_normal = matrix(1),
```

```
  B_nu = matrix(1),
  corr_measurement = FALSE,
  index_corr = NULL,
  ...
)
```

Arguments

noise_type	type of noise, "normal", "nig", "gal", "t", "skew_t"
mu	specify the NIG noise parameter mu, see ?nig
sigma	specify the noise parameter sigma, see ?nig
nu	specify the noise parameter nu, see ?nig and ?gal
B_mu	Basis matrix for mu (if non-stationary)
theta_mu	specify a non-stationary noise using theta_mu
B_sigma	Basis matrix for sigma (if non-stationary)
theta_sigma	specify a non-stationary noise using theta_sigma
B_nu	Basis matrix for nu (if non-stationary)
theta_nu	specify a non-stationary noise using theta_nu
theta_sigma_normal	for normal nosie with nig noise sharing same parameter
B_sigma_normal	for normal nosie with nig noise sharing same parameter
fix_theta_mu	fix the parameter of theta_mu
fix_theta_sigma	fix the parameter of theta_sigma, can be a single logical value or a vector of logical values with length equal to length(theta_sigma)
fix_rho	fix the parameter of rho
fix_theta_sigma_normal	fix the parameter of sigma_normal, used in noise_normal_nig()
fix_theta_nu	fix the parameter of nu
V	start value for V
fix_V	fix the sampling of V gives Y|W ~ N(mean * sigma, sigma^2)
single_V	TRUE if V is a single number
share_V	used only for bivariate model
corr_measurement	TRUE if we use correlated measurement noise
index_corr	used when corr_measurement=TRUE, indicate which observation has correlation
map_corr	1d, 2d, or formula, used when corr_measurement=TRUE, specify use which covariate to infer the index_corr.
nu_lower_bound	specify the lower bound of parameter nu; effective parametrization is $nu = nu_lower_bound + \exp(B_\nu \theta_\nu)$ so theta_nu remains unconstrained. Default 0.

rho	used when corr_measurement=TRUE, starting point for correlation
prior	prior specification created by `priors(...)`. Supported keys are `mu`, `sigma`, and `nu`. Any key left unset gets a weakly-informative normal prior on the coefficient scale, *except* that for a stationary `nig` or `normal_nig` latent noise `f()` replaces the `nu` prior with the penalised-complexity prior `prior_pc_nu`, $1/\nu \sim \mathrm{Exp}(\lambda)$ with $\lambda = -\log(\alpha)/U$ calibrated from $\Pr(1/\nu > U) = \alpha$. The `prior_nu` field of a bare noise object is therefore only a placeholder until it is passed to `f()`; supplying `nu` here suppresses that substitution.
...	additional arguments
sigma_normal	for normal nosie with nig noise sharing same parameter
sigma_nig	similar to sigma_normal
theta_sigma_nig	similar to theta_sigma_normal
B_sigma_nig	similar to B_sigma_nig

Details

The parameterization is given in `?nig` and `?gal`. Moreover, for specifying non-stationary mu and sigma, nu

$$\mu = B_\mu \theta_\mu,$$

and

$$\sigma = \exp(B_\sigma \theta_\sigma),$$
$$\nu = \nu_{\text{lower}} + \exp(B_\nu \theta_\nu).$$

Value

a list of specification of noise

Examples

```
noise_normal(sigma = 2)
noise_nig(mu = 1, sigma = 2, nu = 1)
noise_gal(mu = 1, sigma = 2, nu = 1)
noise_skew_t(mu = 0, sigma = 1, nu = 5)
noise_t(nu = 5)
```

---

ngme_noise_types *Show ngme noise types*

Description

Show ngme noise types

Usage

```
ngme_noise_types()
```

Value

available types for noise

---

ngme_optimizers *List supported optimizers*

Description

This function returns a list of supported optimizers in the ngme package. The optimizers are categorized into three groups:

- **Gradient descent**: "sgd", "sgld", "momentum", "adaptive_gd"
- **Adaptive learning rate**: "adagrad", "rmsprop", "adam", "adamW"
- **Preconditioner**: "precond_sgd", "bfgs"

Usage

```
ngme_optimizers()
```

Value

a character vector of supported optimizers

---

ngme_parse_formula *Parse the formula for ngme function*

Description

Parse the formula for ngme function

Usage

```
ngme_parse_formula(
  fm,
  data,
  control_ngme,
  noise,
  group,
  replicate,
  prior_beta,
  standardize
)
```

Arguments

fm	formula
data	data.frame
control_ngme	control_ngme
noise	noise
group	group factor
replicate	replicate vector
prior_beta	prior specification for fixed effects
standardize	logical, whether fixed effects are standardized

Value

a list (replicate) of ngme_replicate models

---

ngme_post_samples *posterior samples of different latent models*

Description

Extract the posterior samples of different latent models

Usage

```
ngme_post_samples(ngme_object, model_name = 1, type = "W", replicate = 1)
```

Arguments

ngme_object	ngme object
model_name	name of the model, or index of the model
type	type of samples, "W" or "V"
replicate	which replicate

Value

a data.frame of posterior samples (mesh_size * n_post_samples)

ngme_prior_types *Show ngme priors*

Description

Show ngme priors

Usage

```
ngme_prior_types()
```

Value

available types of priors

ngme_result *Access the result of a ngme fitted model*

Description

Access the result of a ngme fitted model

Usage

```
ngme_result(ngme_object, model = NULL, transformed = TRUE)
```

Arguments

ngme_object	a ngme fitted model object
model	latent model name to filter by (e.g., "my_ar1", "my_matern", "my_rw1", etc.), "data" for fixed effects and measurement noise, if NULL, return all models
transformed	logical, if TRUE (default) return transformed parameters, if FALSE return raw parameters

Value

a list of parameters for the specified model, or all models if model is NULL

Examples

```
# Fit a simple AR(1) model
set.seed(1)
Y <- 1:10
n_obs <- length(Y)
x1 <- rnorm(n_obs)
x2 <- runif(n_obs)

ngme_out <- ngme(
  Y ~ x1 + x2 + f(
    1:n_obs,
    name = "my_ar",
    model = ar1(rho = 0.5),
    noise = noise_nig(mu = 2, sigma = 3, nu = 1)
  ),
  data = data.frame(x1 = x1, x2 = x2, Y = Y),
  control_opt = control_opt(estimation = FALSE)
)

# Get all model parameters (transformed)
all_params <- ngme_result(ngme_out)
# Returns: list(my_ar = list(rho = 0.5, mu = 2, sigma = 3, nu = 1),
#               data = list(fixed_effects = c(...), sigma = 0.5))

# Get parameters for specific latent model
ar_params <- ngme_result(ngme_out, model = "my_ar")
# Returns: list(rho = 0.5, mu = 2, sigma = 3, nu = 1)

# Get raw (untransformed) parameters
ar_raw <- ngme_result(ngme_out, model = "my_ar", transformed = FALSE)
# Returns: list(theta_rho = 1.099, mu = 2, sigma = 3, nu = 1)

# Get fixed effects and measurement noise
data_params <- ngme_result(ngme_out, model = "data")
# Returns: list(fixed_effects = c(...), sigma = 0.5, ...)

# For models with multiple latent processes
ngme_out2 <- ngme(
  Y ~ x1 + x2 + f(
    1:n_obs,
    name = "my_ar",
    model = ar1(rho = 0.5),
    noise = noise_nig(mu = 2, sigma = 3, nu = 1)
  ) + f(
    1:n_obs,
    name = "my_ou",
    model = ou(),
    noise = noise_normal(sigma = 1)
  ),
  data = data.frame(x1 = x1, x2 = x2, Y = Y),
  control_opt = control_opt(estimation = FALSE)
)
```

```
# Get all models
all_models <- ngme_result(ngme_out2)
# Returns: list(my_ar = list(...), my_ou = list(...), data = list(...))

# Get specific model
ou_params <- ngme_result(ngme_out2, model = "my_ou")
# Returns: list(theta_K1 = 0.5, sigma = 1)
```

---

ngme_sgld_ci *Quantile Confidence Intervals from SGLD Samples*

Description

Compute parameter-wise quantile confidence intervals from posterior-like SGLD samples returned by [ngme_sgld_samples()].

Usage

```
ngme_sgld_ci(samples, lower = 0.025, upper = 0.975)
```

Arguments

samples	data.frame (or list of data.frames) returned by [ngme_sgld_samples()].
lower	lower quantile probability (e.g. 0.025).
upper	upper quantile probability (e.g. 0.975).

Value

A list with posterior-like point estimates (‘estimates‘), quantile intervals (‘ci‘), and sample covariance matrix (‘covariance‘). Class: ‘ngme_sgld_ci‘.

---

ngme_sgld_samples *Extract Posterior-like Samples from Stored SGLD Trajectories*

Description

Build posterior-like samples from optimizer trajectories by dropping an initial burn-in segment and applying thinning.

Usage

```
ngme_sgld_samples(
  ngme,
  name = "all",
  burnin_iter = 0,
  thinning = 1,
  apply_transform = TRUE,
  combine_chains = TRUE
)
```

Arguments

ngme	fitted ‘ngme‘ object with ‘store_traj = TRUE‘.
name	parameter block to extract: ‘"all"‘ (default), latent model name, or ‘"general"‘.
burnin_iter	non-negative integer. Number of initial iterations to discard before sampling.
thinning	positive integer thinning interval.
apply_transform	logical; apply parameter transforms to user scale.
combine_chains	logical; if ‘TRUE‘, return one combined data.frame, otherwise return one data.frame per chain.

Value

A data.frame (or list of data.frames when ‘combine_chains = FALSE‘) with columns ‘.chain‘, ‘.draw‘, ‘.iter‘, and one column per parameter.

---

ngme_ts_make_A *Make observation matrix for time series*

Description

Make observation matrix for time series

Usage

```
ngme_ts_make_A(loc, replicate = NULL, range = c(min(loc), max(loc)))
```

Arguments

loc	integers (after sorting, no gaps > 1)
replicate	indicating replicate measure at same location
range	range for the mesh by default range=(min(loc), max(loc))

Value

A matrix (length(loc) * length(unique(loc)))

Examples

```
ngme_ts_make_A(c(1, 2, 2), replicate = c(1, 1, 2))
ngme_ts_make_A(c(1, 2, 2), range = c(1, 5))
```

`ngme_update` *Update ngme2 to the latest stable version*

Description

Update ngme2 to the latest stable version

Usage

```
ngme_update()
```

Value

void

`nig` *The Normal Inverse-Gaussian (NIG) Distribution*

Description

Density, distribution function, quantile function and random generation for the normal inverse-Gaussian distribution with parameters `p`, `a` and `b`.

Usage

```
dnig(x, delta, mu, nu, sigma, h = NULL, log = FALSE)

rnig(n, delta, mu, nu, sigma, h = NULL, seed = ngme_random_seed())

pnig(q, delta, mu, nu, sigma, h = NULL, lower.tail = TRUE, log.p = FALSE)

qnig(p, delta, mu, nu, sigma, h = NULL, lower.tail = TRUE, log.p = FALSE)
```

Arguments

`x`, `q`	vector of quantiles.
`delta`	A numeric value for the location parameter.
`mu`	A numeric value for the shift parameter.
`nu`	A numeric value for the shape parameter.
`sigma`	A numeric value for the scaling parameter.

h	A numeric value for the additional parameter, see details.
log, log.p	logical; if TRUE, probabilities/densities p are returned as $log(p)$.
n	number of observations.
seed	Positive integer seed for the sampler. Defaults to a draw from R's random number stream, so 'set.seed()' makes the result reproducible. A value of 0 seeds from the system clock, which is *not* reproducible.
lower.tail	logical; if TRUE, probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
p	vector of probabilities.

Details

The normal inverse-Gaussian distribution has density given by

$$f(x;\delta,\mu,\sigma,\nu) = \frac{e^{\nu+\mu(x-\delta)/\sigma^2}\sqrt{\nu\mu^2/\sigma^2+\nu^2}}{\pi\sqrt{\nu\sigma^2+(x-\delta)^2}} K_1(\sqrt{(\nu\sigma^2+(x-\delta)^2)(\mu^2/\sigma^4+\nu/\sigma^2)}),$$

where K_p is modified Bessel function of the second kind of order p, $x > 0$, $\nu > 0$ and $\mu, \delta, \sigma \in \mathbb{R}$. See Barndorff-Nielsen (1977, 1978 and 1997) for further details.

The additional parameter h is used when

$$V \sim IG(\nu, \nu h^2)$$

. By the infinite divisibility,

$$\frac{1}{h}V \sim IG(\nu h, \nu h)$$

. Then

$$\delta + \mu V + \sigma\sqrt{V}Z$$

has the distribution of

$$NIG(\delta = -\mu h, \mu = \mu h, \sigma = \sigma\sqrt{h}, \nu = \nu h).$$

Value

dnig gives the density, pnig gives the distribution function, qnig gives the quantile function, and rnig generates random deviates.

Invalid arguments will result in return value NaN, with a warning.

The length of the result is determined by n for rnig.

References

Barndorff-Nielsen, O. (1977) Exponentially decreasing distributions for the logarithm of particle size. Proceedings of the Royal Society of London.

Series A, Mathematical and Physical Sciences. The Royal Society. 353, 401–409. doi:10.1098/rspa.1977.0041

Barndorff-Nielsen, O. (1978) Hyperbolic Distributions and Distributions on Hyperbolae, Scandinavian Journal of Statistics. 5, 151–157.

Barndorff-Nielsen, O. (1997) Normal Inverse Gaussian Distributions and Stochastic Volatility Modelling, Scandinavian Journal of Statistics. 24, 1-13. doi:10.1111/14679469.00045

See Also

dgig, dig, digam

Examples

```
rnig(100, delta = 0, mu = 5, sigma = 1, nu = 1)
pnig(0.4, delta = 0, mu = 5, sigma = 1, nu = 1)
qnig(0.8, delta = 0, mu = 5, sigma = 1, nu = 1)
plot(function(x){dnig(x, delta = 0, mu = 5, sigma = 1, nu = 1)}, main =
"Normal inverse-Gaussian density", ylab = "Probability density",
xlim = c(0,10))
```

---

noise_moments *Moments of stationary NIG and GAL noise*

Description

Compute the mean, variance, skewness, kurtosis, and excess kurtosis for stationary NIG and GAL noise. The distribution is evaluated through the normal mean-variance mixture

$$X = \delta + \mu V + \sigma\sqrt{V}Z,$$

where $Z \sim N(0, 1)$. For the package's stationary noise parameterization, `delta = -mu`, so the mean is zero.

Usage

```
noise_moments(noise, ...)

## S3 method for class 'ngme_noise'
noise_moments(noise, ...)

noise_nig_moments(delta = -mu, mu = 0, sigma = 1, nu = 1)

noise_gal_moments(delta = -mu, mu = 0, sigma = 1, nu = 1)
```

Arguments

noise	An ngme_noise object created by noise_nig() or noise_gal().
...	Currently unused.
delta	Location parameter. Defaults to -mu, matching the stationary ngme noise convention.
mu	Shift/skewness parameter in the mixture mean.
sigma	Positive Gaussian scale parameter.
nu	Positive mixing-distribution shape parameter.

Value

A named numeric vector with entries `skewness`, `kurtosis`, `excess_kurtosis`, `variance`, `sd`, `central_moment3`, and `central_moment4`.

Examples

```
noise_nig_moments(mu = 1, sigma = 2, nu = 3)
noise_gal_moments(mu = 1, sigma = 2, nu = 3)
noise_moments(noise_nig(mu = 1, sigma = 2, nu = 3))
```

---

openmp_test *Test OpenMP availability and report the number of threads.*

Description

This function checks if OpenMP is available in the current R environment and, if so, reports the number of OpenMP threads detected. If OpenMP is not available, it reports a corresponding message. It relies on an internal or external function 'get_openmp_threads()' which is assumed to return the number of threads or 0 if unavailable.

Usage

```
openmp_test()
```

Value

Invisibly returns the detected number of OpenMP threads. A value of zero means OpenMP is not available.

---

ou *Ornstein-Uhlenbeck Process Model*

Description

Implements the exact discrete-time representation of the continuous Ornstein-Uhlenbeck (OU) process using a matrix formulation $\mathbf{KX} = \boldsymbol{\delta}$.

Usage

```
ou(mesh = NULL, theta = 1)
```

Arguments

mesh	numerical vector or 'inla.mesh.1d' object giving the ordered time locations. Must be strictly increasing.
theta	positive mean-reversion rate (stiffness parameter). Internally stored on the log scale so optimization remains unconstrained.

Details

The OU process is defined by the stochastic differential equation (SDE):

$$dX_t = -\theta X_t dt + \sigma dW_t$$

where $\theta > 0$ is the mean-reversion rate (stiffness) and σ is the volatility (diffusion coefficient).

Exact Discretization

Unlike Euler-Maruyama approximations, the OU process has an exact solution between time points t_i and t_{i+1} with step size Δt_i:

$$X_{i+1} = X_i e^{-\theta \Delta t_i} + \epsilon_i$$

This is an AR(1) form with autoregressive coefficient:

$$\rho_i = e^{-\theta \Delta t_i}$$

and Gaussian noise:

$$\epsilon_i \sim \mathcal{N}\left(0, \frac{\sigma^2}{2\theta}(1 - \rho_i^2)\right)$$

Matrix Representation

The precision matrix $\mathbf{K}$ for the linear system $\mathbf{KX} = \boldsymbol{\delta}$ is constructed as:

$$\mathbf{K} = \begin{bmatrix} \sqrt{1-\rho_1^2} & 0 & 0 & \cdots & 0 \\ -\rho_1 & 1 & 0 & \cdots & 0 \\ 0 & -\rho_2 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & -\rho_{n-1} & 1 \end{bmatrix}$$

where $\rho_i = \exp(-\theta \Delta t_i)$ with $\Delta t_i = t_{i+1} - t_i$.

Why the $\sqrt{1-\rho_1^2}$ term? The first row ensures that X_1 is drawn from the stationary distribution $\mathcal{N}(0, \sigma^2/(2\theta))$. This scaling factor matches the variance of the first component of $\boldsymbol{\delta}$ with the transition noise variance in subsequent steps.

Non-uniform Mesh

For non-uniform meshes, each ρ_i is computed locally based on the varying time step Δt_i, allowing the model to naturally handle irregularly spaced observations.

The h vector

The h vector represents integration weights for the mass matrix. For the OU model, it is set as:

$$h = [\Delta t_1, \Delta t_2, \ldots, \Delta t_{n-1}, \Delta t_{n-1}]$$

where the last element is duplicated. This ensures proper weighting in the finite element formulation.

The $\boldsymbol{\delta}$ vector

In the equation $\mathbf{KX} = \boldsymbol{\delta}$, the vector $\boldsymbol{\delta}$ contains independent identically distributed (i.i.d.) random variables with unit variance. The precision matrix $\mathbf{K}$ is constructed such that the resulting process $\mathbf{X}$ has the correct OU covariance structure.

Value

An 'ngme_operator' object containing:

- 'K': the precision matrix
- 'h': integration weights
- 'theta_K': log-transformed theta for unconstrained optimization
- 'mesh': the mesh object

References

Uhlenbeck, G. E., & Ornstein, L. S. (1930). On the theory of the Brownian motion. Physical review, 36(5), 823.

Examples

```
# Uniform mesh
mesh_uniform <- seq(0, 10, by = 0.5)
ou_uniform <- ou(mesh = mesh_uniform, theta = 0.5)
print(ou_uniform)

# Non-uniform mesh
mesh_nonuniform <- c(0, 1, 2.5, 3.5, 5, 8, 11)
ou_nonunif <- ou(mesh = mesh_nonuniform, theta = 0.8)
print(ou_nonunif)
```

---

plot.ngme_noise *Plot the density of one or more stationary noise objects*

Description

This function plots the probability density function for one or more stationary noise objects (e.g., NIG, GAL, or normal noise). Multiple noise objects can be compared on the same plot.

Usage

```
## S3 method for class 'ngme_noise'
plot(x = NULL, ...)
```

Arguments

x	An ngme_noise object (required).
...	Additional ngme_noise objects to plot, or plotting parameters such as xlim. Named arguments will be used as legend labels.

Value

A ggplot object showing the density curves for the provided noise objects.

Examples

```
plot(noise_nig(mu = 1, sigma = 2, nu = 1))
plot(n1 = noise_nig(mu = 0, sigma = 1, nu = 1), n2 = noise_nig(mu = 1, sigma = 1.5, nu = 0.5))
```

---

plot.parameter_distance

Plot method for parameter_distance

Description

Plot method for parameter_distance

Usage

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

Arguments

x	parameter_distance object
...	additional arguments for ggplot

Value

A ggplot object showing the parameter-distance trajectory over optimization iterations. The y-axis is the stored distance $||theta - \hat{theta}||$; plot annotations report the norm and across-chain summary used when x was computed.

pm25_quarterly_2022 *Quarterly PM2.5 monitor averages over CONUS in 2022*

Description

A processed example dataset containing quarterly fine particulate matter (PM2.5) averages for monitoring sites in the contiguous United States. The data are derived from the US EPA Air Quality System and AirData pre-generated monitor files and aggregated to site-quarter means.

Usage

```
data("pm25_quarterly_2022")
```

Format

A data frame with 2640 rows and 9 columns.

site_id EPA monitor site identifier.

state State name.

county County name.

lat Monitor latitude in decimal degrees.

lon Monitor longitude in decimal degrees.

quarter Calendar quarter, coded as `Q1`, `Q2`, `Q3`, or `Q4`.

pm25_q Quarterly average PM2.5 concentration in micrograms per cubic meter.

n_days Number of daily monitor summaries contributing to the quarterly average.

replicate Integer replicate index corresponding to quarter.

Source

US EPA Air Quality System Data Mart and EPA AirData pre-generated files for PM2.5 monitor summaries. https://www.epa.gov/aqs https://aqs.epa.gov/aqsweb/airdata/download_files.html

poly_decay *Polynomial schedule helper*

Description

Polynomial schedule helper

Usage

```
poly_decay(alpha = 0.501, t0 = 1, burnin_iter = 0)
```

Arguments

alpha	polynomial exponent in $(1/2, 1)$.
t0	non-negative schedule offset.
burnin_iter	non-negative integer. Initial iterations without polynomial schedule scaling.

Details

Convenience helper that enables polynomial schedule and disables checkpoint-based decay.

Value

a `stepsize_control()` object.

---

posterior_plot *Plot Posterior Distributions from SGLD Samples*

Description

Visualize marginal posterior distributions of parameters extracted from [ngme_sgld_samples()] or summarized by [ngme_sgld_ci()].

Usage

```
posterior_plot(
  x,
  parameters = NULL,
  type = c("density", "histogram"),
  by_chain = FALSE,
  show_estimate = TRUE,
  show_interval = TRUE,
  lower = NULL,
  upper = NULL,
  bins = 30,
  ncol = NULL,
  ...
)

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

Arguments

x	a data.frame (or list of data.frames) returned by [ngme_sgld_samples()], or an object of class ‘ngme_sgld_ci‘.
parameters	optional character vector of parameter names to plot. Defaults to all parameter columns.

type	character; ‘"density"‘ (default) or ‘"histogram"‘.
by_chain	logical; if ‘TRUE‘, color distributions by chain when a ‘.chain‘ column is available.
show_estimate	logical; add a vertical line at the posterior mean.
show_interval	logical; add vertical lines for the equal-tail interval.
lower	lower quantile probability used for intervals. If ‘x‘ is an ‘ngme_sgld_ci‘ object, the stored value is used by default.
upper	upper quantile probability used for intervals. If ‘x‘ is an ‘ngme_sgld_ci‘ object, the stored value is used by default.
bins	number of bins for histograms.
ncol	number of facet columns. If ‘NULL‘, use 2 columns for multiple parameters and 1 for a single parameter.
...	unused.

Value

A faceted ggplot object.

`precision_matrix_multivariate`
Compute the precision matrix for multivariate model

Description

Compute the precision matrix for multivariate model

Usage

```
precision_matrix_multivariate(
  p,
  operator_list,
  rho,
  theta = NULL,
  Q = NULL,
  scale = NULL
)
```

Arguments

p	dimension, should be integer and greater than 1
operator_list	a list of ngme_operator object (length should be p)
rho	vector with the p(p-1)/2 correlation parameters rho_11, rho_21, rho_22, ... rho_p1, rho_p2, ... rho_p(p-1)
theta	parameter for Q matrix (length of 1 when p=2, length of 3 when p=3)
Q	orthogonal matrix of dim p*p (provide when p > 3)
scale	A vector of length p with constants to multiply each operator matrix with

Details

The general model is defined as $D diag(L_1, ..., L_p) x = M$. D is the dependence matrix, it is paramterized by $D = Q(theta) * D_l(cor_mat)$, where Q is the orthogonal matrix, and D_l is matrix controls the cross-correlation. See the section 2.2 of Bolin and Wallin (2020) for exact parameterization of Dependence matrix.

Value

the precision matrix of the multivariate model

References

Bolin, D. and Wallin, J. (2020), Multivariate type G Matérn stochastic partial differential equation random fields. J. R. Stat. Soc. B, 82: 215-239. https://doi.org/10.1111/rssb.12351

Examples

```
rho <- c(-0.5, 0.5, -0.25) # correlation parameters
operator_list <- list(ar1(1:5, rho = 0.4), ar1(1:5, rho = 0.5), ar1(1:5, rho = 0.6))
precision_matrix_multivariate(3, operator_list, rho, theta = c(1, 2, 3))
```

---

precision_matrix_multivariate_spde

Compute the precision matrix for multivariate spde Matern model

---

Description

Compute the precision matrix for multivariate spde Matern model

Usage

```
precision_matrix_multivariate_spde(
  p,
  mesh,
  rho,
  alpha_list = NULL,
  theta_K_list = NULL,
  variance_list = NULL,
  B_K_list = NULL,
  theta = NULL,
  Q = NULL
)
```

Arguments

p	dimension, should be integer and greater than 1
mesh	an fmesher::fm_mesh_2d object, mesh for build the SPDE model
rho	vector with the p(p-1)/2 correlation parameters rho_11, rho_21, rho_22, ... rho_p1, rho_p2, ... rho_p(p-1)
alpha_list	a list of SPDE smoothness parameter
theta_K_list	a list (length is p) of theta_K
variance_list	If provided, it should be a vector of length p, where the kth element corresponds to a desired variance of the kth field. The kth operator is then scaled by a constant c so that this variance is achieved in the stationary case (default no scaling)
B_K_list	a list (length is p) of B_K (non-stationary case)
theta	parameter for Q matrix (length of 1 when p=2, length of 3 when p=3)
Q	orthogonal matrix of dim p*p (provide when p > 3)

Details

The general model is defined as $D diag(L_1, ..., L_p) x = M$. D is the dependence matrix, it is paramterized by $D = Q(theta) * D_l(cor_mat)$, where Q is the orthogonal matrix, and D_l is matrix controls the cross-correlation. See the section 2.2 of Bolin and Wallin (2020) for exact parameterization of Dependence matrix.

Value

the precision matrix of the multivariate model

References

Bolin, D. and Wallin, J. (2020), Multivariate type G Matérn stochastic partial differential equation random fields. J. R. Stat. Soc. B, 82: 215-239. https://doi.org/10.1111/rssb.12351

Examples

```
library(fmesher)
library(fields)
# Define mesh
x <- seq(from = 0, to = 1, length.out = 40)
mesh <- fm_rcdt_2d_inla(lattice = fm_lattice_2d(x, x), extend = FALSE)
# Set parameters
p <- 3 # number of fields
rho <- c(-0.5, 0.5, -0.25) # correlation parameters
log_kappa <- list(2, 2, 2) # log(kappa)
variances <- list(1, 1, 1) # set marginal variances to 1
alpha <- list(2, 2, 2) # smoothness parameters
# Compute precision
Q <- precision_matrix_multivariate_spde(p,
  mesh = mesh, rho = rho,
  alpha = alpha, theta_K_list = log_kappa,
  variance_list = variances
```

```
)
# Plot the cross covariances
A <- as.vector(fm_basis(mesh, loc = matrix(c(0.5, 0.5), 1, 2)))
Sigma <- as.vector(solve(Q, c(A, rep(0, 2 * mesh$n))))
r11 <- Sigma[1:mesh$n]
r12 <- Sigma[(mesh$n + 1):(2 * mesh$n)]
r13 <- Sigma[(2 * mesh$n + 1):(3 * mesh$n)]
Sigma <- as.vector(solve(Q, c(rep(0, mesh$n), A, rep(0, mesh$n))))
r22 <- Sigma[(mesh$n + 1):(2 * mesh$n)]
r23 <- Sigma[(2 * mesh$n + 1):(3 * mesh$n)]
Sigma <- as.vector(solve(Q, v <- c(rep(0, 2 * mesh$n), A)))
r33 <- Sigma[(2 * mesh$n + 1):(3 * mesh$n)]

proj <- fm_evaluator(mesh)

oldpar <- par(no.readonly = TRUE)
par(mfrow = c(3, 3))
image.plot(fm_evaluate(proj, r11), main = "Cov(X_1(s0),X_1(s)")
plot.new()
plot.new()
image.plot(fm_evaluate(proj, r12), main = "Cov(X_1(s0),X_2(s)")
image.plot(fm_evaluate(proj, r22), main = "Cov(X_2(s0),X_2(s)")
plot.new()
image.plot(fm_evaluate(proj, r13), main = "Cov(X_1(s0),X_3(s)")
image.plot(fm_evaluate(proj, r23), main = "Cov(X_2(s0),X_3(s)")
image.plot(fm_evaluate(proj, r33), main = "Cov(X_3(s0),X_3(s)")
par(oldpar)
```

`precond_sgd` *Preconditioner SGD optimization*

Description

Preconditioner SGD optimization

Usage

```
precond_sgd(stepsize = 1, numerical_eps = 1e-05)
```

Arguments

`stepsize`	stepsize for SGD
`numerical_eps`	numerical, the gap used for estimate preconditioner, default is 1e-5

Details

Preconditioner SGD is using fisher information matrix as preconditioner (natural gradient descent).

Value

a list of control variables for optimization (used in `control_opt` function)

`predict.ngme` *Predict function of ngme2 predict using ngme after estimation*

Description

Predict function of ngme2 predict using ngme after estimation

Usage

```
## S3 method for class 'ngme'
predict(
  object,
  map,
  data = NULL,
  type = "lp",
  group = NULL,
  estimator = c("mean", "sd", "0.05q", "0.95q", "median", "mode"),
  sampling_size = 500,
  burnin_size = 100,
  seed = ngme_random_seed(),
  replicate = NULL,
  train_idx = NULL,
  chain_combine = c("param_mean", "predictive_average"),
  return_samples = FALSE,
  ...
)
```

Arguments

`object`	a ngme object
`map`	a named list (or dataframe) of the locations to make the prediction
`data`	a data.frame or matrix of covariates (used for fixed effects) names(loc) corresponding to the name each latent model vector or matrix (n * 2) for spatial coords
`type`	what type of prediction, c("fe", "lp", <model_name>) "fe" is fixed effect prediction <model_name> is prediction of a specific model "lp" is linear predictor (including fixed effect and all sub-models) "response" is the linear predictor plus a fresh measurement-noise draw
`group`	which filed to predict (used for bivariate model, should be of same length as map)
`estimator`	what type of estimator. Options include: - "mean", "median", "mode", "sd": standard estimators - "0.XXXq": any quantile specified as probability (e.g., "0.025q", "0.5q", "0.975q")

sampling_size	size of posterior sampling
burnin_size	size of posterior burnin
seed	random seed. Defaults to a seed drawn from the current R random number stream, so set.seed() makes the result reproducible.
replicate	which replicate to predict for. A replicate LEVEL (the name used in the fit, e.g. ‘2007‘ or ‘"site_a"‘) or a positive integer position in ‘object$replicates‘. Defaults to the first replicate.
train_idx	optional vector of training indices to use for posterior sampling. If provided, only these indices from the original data will be used for training, similar to cross-validation. If NULL, uses all original training data.
chain_combine	how to combine multiple optimization chains: • "param_mean": default behavior using the fitted object parameters. • "predictive_average": run prediction for each optimization chain and average predictions across chains.
return_samples	logical; when ‘TRUE‘, attach sample draws for the requested output in ‘attr(ret, "samples")‘. For ‘type = "response"‘, the attached samples are response predictive draws.
...	additional arguments (currently unused)

Value

a list of outputs contains estimation of operator paramters, noise parameters

---

print.ngme *Print an ngme model*

Description

Print an ngme model

Usage

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

Arguments

x	ngme model object
...	...

Value

Invisibly returns x, an ngme object. The method is called for its side effect of printing a model summary.

print.ngme_model *Print ngme model*

Description

Print ngme model

Usage

```
## S3 method for class 'ngme_model'
print(x, padding = 0, ...)
```

Arguments

x	ngme model object
padding	number of white space padding in front
...	...

Value

a list (model specifications)

print.ngme_noise *Print ngme noise*

Description

Print ngme noise

Usage

```
## S3 method for class 'ngme_noise'
print(x, padding = 0, prefix = "Noise type", model_type = NULL, ...)
```

Arguments

x	noise object
padding	number of white space padding in front
prefix	prefix
model_type	model type
...	...

Value

a list (noise specifications)

print.ngme_operator *Print ngme operator*

Description

Print ngme operator

Usage

```
## S3 method for class 'ngme_operator'
print(x, padding = 0, prefix = "Model type", ...)
```

Arguments

x	ngme operator object
padding	number of white space padding in front
prefix	prefix string
...	...

Value

a list (operator specifications)

print.ngme_replicate *Print ngme object*

Description

Print ngme object

Usage

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

Arguments

x	ngme object
...	ignored

Value

Invisibly returns `x`, an `ngme_replicate` object. The method is called for its side effect of printing a replicate summary.

print.ngme_trajectories

Print method for ngme_trajectories

Description

Print method for ngme_trajectories

Usage

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

Arguments

x	ngme_trajectories object
...	additional arguments

Value

Invisibly returns x, an ngme_trajectories object. The method is called for its side effect of printing a summary of stored parameter trajectory matrices, including parameter names, chain count, and iteration count.

print.noise_kld_comparison

Print method for noise_kld_comparison

Description

Print method for noise_kld_comparison

Usage

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

Arguments

x	noise_kld_comparison object
...	additional arguments

Value

Invisibly returns x, a noise_kld_comparison object. The method is called for its side effect of printing the reference noise object, the Kullback-Leibler divergence values for each comparison noise, and the closest comparison.

print.parameter_distance *Print method for parameter_distance*

Description

Print method for parameter_distance

Usage

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

Arguments

x	parameter_distance object
...	additional arguments

Value

Invisibly returns x, a numeric vector with class parameter_distance. The method is called for its side effect of printing the norm type, chain summary, number of iterations, final distance, and true parameter values.

prior_half_cauchy *Prior Half-Cauchy*

Description

Prior Half-Cauchy

Usage

```
prior_half_cauchy(scale = 1, target = "coef")
```

Arguments

scale	half-Cauchy scale
target	apply prior on coefficient scale (‘"coef"‘) or field scale (‘"field"‘)

Value

prior specification

prior_inv_exponential *Prior Inverse-Exponential*

Description

Prior induced by $\kappa = 1/\nu \sim \text{Exp}(\lambda)$, giving $p(\nu) = \lambda \exp(-\lambda/\nu)\nu^{-2}$ for $\nu > 0$. Internally this prior is applied to $\nu = \text{lower} + \exp(\theta)$.

Usage

```
prior_inv_exponential(lambda = 1, lower = 0, target = "coef")

prior_inv_exp(lambda = 1, lower = 0, target = "coef")
```

Arguments

lambda	exponential rate on $\kappa = 1/\nu$
lower	lower shift used in $\nu = \text{lower} + \exp(\theta)$
target	apply prior on coefficient scale (‘"coef"‘) or field scale (‘"field"‘)

Value

prior specification

prior_none *Prior None*

Description

Prior None

Usage

```
prior_none(target = "coef")
```

Arguments

target	apply prior on coefficient scale (‘"coef"‘) or field scale (‘"field"‘)

Value

prior specification

prior_normal *Prior Normal*

Description

Prior Normal

Usage

```
prior_normal(mean = 0, sd = 1, target = "coef")
```

Arguments

mean	prior mean
sd	prior standard deviation
target	apply prior on coefficient scale ('"coef"') or field scale ('"field"')

Value

prior specification

prior_pc_nu *PC Prior for the NIG Flexibility Parameter*

Description

Penalised-complexity prior for the flexibility parameter $\eta = 1/\nu$ of NIG driving noise, following Cabral, Bolin and Rue (2023). The distance to the Gaussian base model is proportional to η (their Corollary 3.1.1), so the PC prior is $\eta \sim \mathrm{Exp}(\lambda)$, which is exactly prior_inv_exponential on ν.

The rate is calibrated as in their Section 3.3, from the statement $\Pr(\eta > U) = \alpha$, giving $\lambda = -\log(\alpha)/U$.

For symmetric noise the excess kurtosis of a unit-h increment is 3η, which is the easiest way to read the defaults: $U = 2.5$ is an excess kurtosis of 7.5 (equivalently $\nu = 0.4$), and $\alpha = 0.01$ puts 1% of the prior mass above it. That gives $\lambda \approx 1.84$, a mild contraction: the prior median is $\nu \approx 2.7$ (excess kurtosis 1.1), and the prior is close to neutral over the range of ν these models typically occupy.

The defaults are read from getOption("ngme2.pc_nu_U") and getOption("ngme2.pc_nu_alpha"), so the package-wide default used by f can be changed without passing a prior to every model.

Usage

```
prior_pc_nu(
  U = getOption("ngme2.pc_nu_U", 2.5),
  alpha = getOption("ngme2.pc_nu_alpha", 0.01),
  lower = 0,
  target = "coef"
)
```

Arguments

U	upper value $U > 0$ for $\eta = 1/\nu$ in the calibration statement $\Pr(\eta > U) = \alpha$
alpha	prior probability $\alpha \in (0, 1)$ of exceeding U
lower	lower shift used in $\nu = \text{lower} + \exp(\theta)$
target	apply prior on coefficient scale (‘"coef"‘) or field scale (‘"field"‘)

Value

prior specification

References

Cabral, R., Bolin, D., and Rue, H. (2023). Controlling the Flexibility of Non-Gaussian Processes Through Shrinkage Priors. *Bayesian Analysis*, 18(4), 1223-1246. doi:10.1214/22BA1342

See Also

prior_inv_exponential

---

prior_pc_sd *Prior PC-SD*

Description

Prior PC-SD

Usage

```
prior_pc_sd(u, alpha, target = "coef")
```

Arguments

u	tail event threshold
alpha	tail probability
target	apply prior on coefficient scale (‘"coef"‘) or field scale (‘"field"‘)

Value

prior specification

priors *Prior Container*

Description

Prior Container

Usage

```
priors(...)
```

Arguments

...	named prior specifications

Value

named prior container

re *ngme random effect model*

Description

ngme random effect model

Usage

```
re(map, theta_K = NULL, ...)
```

Arguments

map	numerical vector, covariates to build index for the process (can be formula, provided data)
theta_K	initial value for theta_K (build covariance matrix)
...	currently ignored

Value

ngme_operator object

rmsprop — *Root Mean Square Propagation (RMSProp) SGD optimization*

Description

Root Mean Square Propagation (RMSProp) SGD optimization

Usage

```
rmsprop(stepsize = 0.05, beta1 = 0.9, epsilon = 1e-08)
```

Arguments

stepsize	stepsize for SGD
beta1	beta1 for momentum
epsilon	epsilon for numerical stability

Details

The update rule for RMSProp is:

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t^2$$

$$x_{t+1} = x_t - \text{stepsize} * \frac{g_t}{\sqrt{v_t} + \epsilon}$$

Value

a list of control variables for optimization (used in `control_opt` function)

rw1 — *Random Walk Model of Order 1 (RW1)*

Description

Constructs a first-order random walk model for spatial or temporal processes. The RW1 model assumes that first-order differences $\Delta W_i = W_i - W_{i-1}$ are independent and identically distributed Gaussian variables.

Usage

```
rw1(mesh = NULL, cyclic = FALSE, constr = TRUE)
```

Arguments

`mesh`	numerical vector or inla.mesh.1d object, locations to build the mesh. For numerical vectors, assumes equally spaced locations.
`cyclic`	logical, whether the mesh is circular. If TRUE, the first and last locations are treated as neighbors. Cannot be FALSE when constr = TRUE.
`constr`	logical, whether to enforce the sum-to-zero constraint $\sum_{i=1}^{n} h_i W_i = 0$. If FALSE, fixes the first element $W_1 = 0$. Must be TRUE for cyclic models.

Details

The RW1 model is defined by the precision matrix K that penalizes first-order differences. The model has different structures depending on the constraints:

Non-cyclic, constrained (default): The first row of K enforces the constraint $\sum_{i=1}^{n} h_i W_i = 0$, where h_i are the mesh weights.

Non-cyclic, unconstrained: The first element is fixed at 0 ($W_1 = 0$).

Cyclic: Treats the domain as circular, connecting the first and last locations as neighbors. The constraint $\sum_{i=1}^{n} h_i W_i = 0$ is always enforced. The precision matrix is expanded to size $(n+1) \times (n+1)$ to handle the constraint properly.

Value

An ‘ngme_operator‘ object containing the precision matrix and related components for the RW1 model.

Examples

```
# Non-cyclic constrained RW1 (default)
rw1_default <- rw1(1:5)
print(rw1_default$K)

# Non-cyclic unconstrained RW1 (fixes first element to 0)
rw1_fixed <- rw1(1:5, constr = FALSE)
print(rw1_fixed$K)

# Cyclic RW1 (connects first and last locations)
rw1_cyclic <- rw1(1:5, cyclic = TRUE)
print(rw1_cyclic$K)

# Using with unequally spaced locations
locations <- c(0, 1, 3, 6, 10)
rw1_unequal <- rw1(locations)
print(rw1_unequal$K)
```

`rw2` *Random Walk Model of Order 2 (RW2)*

Description

Constructs a second-order random walk model for spatial or temporal processes. The RW2 model assumes that second-order differences $\Delta^2 W_i = W_i - 2W_{i-1} + W_{i-2}$ are independent and identically distributed Gaussian variables.

Usage

```
rw2(mesh = NULL, cyclic = FALSE, constr = TRUE)
```

Arguments

`mesh`	numerical vector or inla.mesh.1d object, locations to build the mesh. For numerical vectors, assumes equally spaced locations. Must have at least 3 locations.
`cyclic`	logical, whether the mesh is circular. If TRUE, the first and last locations are treated as neighbors with second-order differences computed across the boundary.
`constr`	logical, whether to enforce the sum-to-zero constraint $\sum_{i=1}^{n} h_i W_i = 0$ and the linear trend constraint $\sum_{i=1}^{n} (\sum_{j=1}^{i} h_j - h_1) \cdot W_i = 0$. If FALSE, fixes the first and second elements $W_1 = 0$ and $W_2 = 0$.

Details

The RW2 model is defined by the precision matrix K that penalizes second-order differences, making it smoother than RW1. The model enforces constraints to ensure identifiability:

Non-cyclic (default): The first two rows of K enforce constraints: the first row implements $\sum_{i=1}^{n} h_i W_i = 0$ (sum-to-zero), and the second row implements $\sum_{i=1}^{n} h_i \cdot i \cdot W_i = 0$ (removes linear trend). The remaining rows penalize second-order differences.

Cyclic: Treats the domain as circular, connecting the first and last locations as neighbors. The constraint $\sum_{i=1}^{n} h_i W_i = 0$ is enforced. The precision matrix is expanded to size $(n+1) \times (n+1)$ to handle the constraint and circular structure properly.

Value

An ‘ngme_operator‘ object containing the precision matrix and related components for the RW2 model.

Examples

```
# Non-cyclic RW2 with constraints (default)
rw2_default <- rw2(1:6)
print(rw2_default$K)

# Cyclic RW2 (connects first and last locations)
```

```
rw2_cyclic <- rw2(1:6, cyclic = TRUE)
print(rw2_cyclic$K)

# Using with unequally spaced locations
locations <- c(0, 1, 3, 6, 10, 15)
rw2_unequal <- rw2(locations)
print(rw2_unequal$K)
```

---

sgd *Vanilla SGD optimization*

Description

Simple stochastic gradient descent without momentum or preconditioning.

Usage

```
sgd(stepsize = 0.001)
```

Arguments

stepsize	stepsize for SGD

Value

a list of control variables for optimization (used in `control_opt` function)

---

sgld *Stochastic Gradient Langevin Dynamics (SGLD) optimization*

Description

Stochastic Gradient Langevin Dynamics (SGLD) optimization

Usage

```
sgld(stepsize = 0.001, temperature = 1)
```

Arguments

stepsize	base stepsize
temperature	non-negative Langevin temperature

Details

SGLD adds Gaussian noise to vanilla SGD updates:

$$x_{t+1} = x_t - \eta_t \nabla U(x_t) + \sqrt{2T\eta_t}\,\xi_t, \quad \xi_t \sim \mathcal{N}(0, I)$$

where T is `temperature`. The implementation applies this component-wise using the current effective stepsize.

Value

a list of control variables for optimization (used in `control_opt` function)

`simulate.ngme` *Simulate from a ngme object (possibly with replicates)*

Description

Simulate from a ngme object (possibly with replicates)

Usage

```
## S3 method for class 'ngme'
simulate(object, nsim = 1, seed = NULL, ...)
```

Arguments

`object`	ngme object
`nsim`	number of simulations
`seed`	random seed. If `NULL` (the default), a seed is drawn from the current R random number stream, so `set.seed()` makes the result reproducible.
`...`	optional arguments. Supported names are `posterior` (whether to simulate from posterior sampling of latent fields), `m_noise` (whether to add the measurement noise) and `n_burnin` (Gibbs sweeps discarded before the draw is taken, used only when `posterior = TRUE`; defaults to 100).

Value

a realization of ngme object

simulate.ngme_model *Simulate latent process with noise*

Description

Simulate latent process with noise

Usage

```
## S3 method for class 'ngme_model'
simulate(object, nsim = 1, seed = NULL, ...)
```

Arguments

object	ngme model specified by f() function
nsim	number of simulations
seed	random seed. If NULL (the default), a seed is drawn from the current R random number stream, so set.seed() makes the result reproducible.
...	ignored

Value

a realization of latent model

Examples

```
simulate(f(1:10, model = ar1(rho = 0.4), noise = noise_nig()))
simulate(f(rnorm(10), model = rw1(), noise = noise_normal()))
simulate(f(1:10, model = ar1(rho = 0.4), noise = noise_t(nu = 5)))
```

simulate.ngme_noise *Simulate ngme noise object*

Description

Simulate ngme noise object

Usage

```
## S3 method for class 'ngme_noise'
simulate(object, nsim = 1, seed = NULL, h = NULL, ...)
```

Arguments

object	ngme noise object
nsim	number of simulations
seed	random seed. If NULL (the default), a seed is drawn from the current R random number stream, so set.seed() makes the result reproducible.
h	should be of same length as nsim
...	ignored

Value

data.frame (each col is a realization)

---

spacetime *Ngme space-time non-separable model specification*

Description

Implements a non-separable space-time model based on the advection-diffusion SPDE with first-order derivative in time. The model combines temporal and spatial components through a finite difference method (implicit Euler) for temporal discretization and finite element method (continuous Galerkin) for spatial discretization. When advection dominates diffusion, the "Streamline Diffusion" stabilization technique is applied.

Usage

```
spacetime(
  mesh,
  lambda = 1,
  alpha = 2,
  cc = 1,
  kappa = 1,
  fix_gamma = FALSE,
  theta_gamma_x = 0,
  theta_gamma_y = 0,
  shared_theta_gamma = FALSE,
  B_gamma_x = matrix(1, nrow = mesh[[2]]$n, ncol = 1),
  B_gamma_y = matrix(1, nrow = mesh[[2]]$n, ncol = 1),
  B_gamma_x_list = NULL,
  B_gamma_y_list = NULL,
  stabilization = TRUE,
  stationary_init = TRUE,
  cc_variance_free = TRUE
)
```

Arguments

`mesh`	A list of two objects: • mesh_t - The temporal mesh • mesh_s - The spatial mesh
`lambda`	The spatial damping parameter.
`alpha`	2 or 4, SPDE smoothness parameter.
`cc`	Parameter c in the SPDE.
`kappa`	Kappa parameter from Matern SPDE.
`fix_gamma`	TRUE if fix gamma (advection term), FALSE if estimate gamma.
`theta_gamma_x`	The x component of the advection term: `gamma_x = B_gamma_x %*% theta_gamma_x`.
`theta_gamma_y`	The y component of the advection term: `gamma_y = B_gamma_y %*% theta_gamma_y`.
`shared_theta_gamma`	TRUE if share the same theta_gamma for all time nodes. (theta_gamma_x and theta_gamma_y will be the same)
`B_gamma_x`	The design matrix for the x component of the advection term.
`B_gamma_y`	The design matrix for the y component of the advection term.
`B_gamma_x_list`	A list of design matrices for the x component of the advection term on every time node, length(B_gamma_x_list) == nt-1.
`B_gamma_y_list`	A list of design matrices for the y component of the advection term on every time node, length(B_gamma_y_list) == nt-1.
`stabilization`	TRUE if using a stabilization term (for implicit Euler).
`stationary_init`	use a stationary initial distribution for the first time slice (default TRUE).
`cc_variance_free`	Scale the operator with cc so that it affects the total variance less. Default TRUE.

Details

The model is particularly useful for large space-time datasets in environmental science, offering computationally efficient methods for parameter estimation, kriging prediction, and conditional simulations.

For details, see doi:10.1016/j.spasta.2024.100847

Value

ngme_operator object.

stepsize_control *Unified stepsize control*

Description

Unified stepsize control

Usage

```
stepsize_control(
  schedule = stepsize_schedule(method = "constant"),
  decay = stepsize_decay(method = "none")
)
```

Arguments

schedule	schedule component. Either an object from `stepsize_schedule()` or a method string accepted by `stepsize_schedule()`.
decay	decay component. Either an object from `stepsize_decay()` or a method string accepted by `stepsize_decay()`.

Details

Bundle an iteration schedule (`stepsize_schedule`) and an optional checkpoint-based decay rule (`stepsize_decay`) into one object for `control_opt()`.

Value

a bundled stepsize-control object for `control_opt()`.

stepsize_decay *Stepsize decay schedule*

Description

Stepsize decay schedule

Usage

```
stepsize_decay(
  method = c("none", "grad_norm_plateau", "trend"),
  patience = 3,
  gamma = 0.5,
  min_delta = 0,
  warmup = 0,
  min_stepsize = 0
)
```

Arguments

method	decay strategy. "none" disables decay; "grad_norm_plateau" decays when mean grad.norm() across chains fails to decrease for a number of epochs.
patience	number of consecutive epochs without grad.norm() improvement before decaying stepsize.
gamma	decay factor applied when triggered (0 < gamma < 1).
min_delta	minimum required decrease in grad.norm() to be counted as improvement.
warmup	number of initial epochs to skip decay checks.
min_stepsize	lower bound for stepsize after decay (absolute value).

Details

Define a stepsize decay strategy for use in `control_opt()`. Currently supports the plateau-based schedule using grad.norm().

Value

a list of control variables for stepsize decay (used in `control_opt`).

`stepsize_schedule` *Stepsize schedule*

Description

Stepsize schedule

Usage

```
stepsize_schedule(
  method = c("constant", "poly"),
  alpha = 0.501,
  t0 = 1,
  burnin_iter = 0
)
```

Arguments

method	schedule type. "constant" keeps $\eta_t = \eta_0$ and "poly" uses polynomial decay.
alpha	polynomial exponent used when `method = "poly"`. Must satisfy $1/2 < \alpha < 1$.
t0	non-negative offset in iteration index.
burnin_iter	non-negative integer. During these initial iterations, schedule scaling is fixed to 1. Afterward, polynomial decay is applied with reset local time index.

Details

Define an iteration-dependent stepsize schedule for use in `control_opt()`. This schedule is independent of `stepsize_decay()` and applies a direct multiplicative scaling by iteration index:

$$\eta_t = \eta_0 (t + t_0)^{-\alpha}$$

.

Value

a list of control variables for stepsize schedule (used in `control_opt`).

`summary.ngme` *Summary of ngme fit result*

Description

Summary of ngme fit result

Usage

```
## S3 method for class 'ngme'
summary(object, name = NULL, ...)
```

Arguments

`object`	an object of class ngme
`name`	name of the latent model to be summarized (if NULL, will print all)
`...`	other arguments

Value

a list of summary

summary.ngme_batch_ci *Summary for Batch-Means CI Results*

Description

Summarize an object of class ‘ngme_batch_ci‘, including parameter means, standard errors, confidence intervals, and covariance matrix.

Usage

```
## S3 method for class 'ngme_batch_ci'
summary(object, digits = 4, ...)
```

Arguments

object	an object of class ‘ngme_batch_ci‘.
digits	number of digits used for printing.
...	currently unused.

Value

A list with summary table and covariance matrix.

tp *ngme tensor-product model specification*

Description

Given 2 operator (first and second), build a tensor-product operator based on K = K_first x K_second (here x is Kronecker product)

Usage

```
tp(first, second)
```

Arguments

first	ngme_operator, left side of kronecker model (usually a temporal or iid model)
second	ngme_operator, right side of kronecker model (ususally a temporal or spatial model)

Value

a list of specification of model

`traceplot` *Trace plot of ngme fitting*

Description

Trace plot of ngme fitting

Usage

```
traceplot(
  ngme,
  name = "all",
  moving_window = 1,
  hline = NULL,
  combine = TRUE,
  ncol = NULL
)
```

Arguments

`ngme`	ngme object
`name`	name of latent models, otherwise plot fixed effects and measurement noise. Use `"all"` to plot all latent models plus data parameters in one figure. should be in names(ngme$models) or other
`moving_window`	moving window for the traceplot
`hline`	vector, add hline to each plot
`combine`	bool, if TRUE, return a single faceted ggplot; otherwise return a list of ggplot objects and print them one by one.
`ncol`	number of facet columns when `combine = TRUE`. If `NULL` (default), use 2 columns for multi-parameter plots and 1 for single-parameter plots.

Value

A ggplot object when `combine = TRUE`; otherwise a list of ggplot objects. The mean parameter trajectories are stored in the `avg_lines` attribute of the returned object.

var1 *ngme VAR(1) bivariate model specification (Cayley re-parameterization)*

Description

Builds a Vector Autoregressive order-1 (VAR(1)) operator for bivariate time-series data. The model follows:

$$Y_t = A\,Y_{t-1} + \varepsilon_t, \quad A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

Usage

```
var1(mesh = NULL, p1 = 0, p2 = 1, p3 = 0, p4 = 1)
```

Arguments

mesh	Integer vector or `inla.mesh.1d` object for the time mesh (length T). Pass `NULL` for deferred construction.
p1	Unconstrained skew-symmetric parameter ($\in \mathbb{R}$); controls the rotation part of the Cayley mapping. Default `0`.
p2, p3, p4	Unconstrained lower-triangular Cholesky parameters ($\in \mathbb{R}$); control the contraction (positive-definite) part. Defaults `p2 = 1`, `p3 = 0`, `p4 = 1`.

Value

An `ngme_operator` (or `ngme_operator_def` when `mesh = NULL`) suitable for use inside `f()`.

Cayley re-parameterization

To guarantee stationarity ($\rho(A) < 1$) at every SGD step, the 2×2 coefficient matrix A is obtained via the *Cayley transform*:

$$A = (I + S)(I - S)^{-1}$$

where S is constructed from four unconstrained real parameters $(p_1, p_2, p_3, p_4) \in \mathbb{R}^4$ as $S = J - R$:

Skew-symmetric part (rotation): $J = \begin{pmatrix} 0 & p_1 \\ -p_1 & 0 \end{pmatrix}$

Positive-definite part (contraction): $R = LL^\top + \varepsilon I$, where $L = \begin{pmatrix} p_2 & 0 \\ p_3 & p_4 \end{pmatrix}$ and $\varepsilon = 10^{-5}$.

Because $S + S^\top = -2R$ is negative definite, all eigenvalues of S have strictly negative real parts, and the Cayley transform then guarantees $|\lambda_i(A)| < 1$ for every i.

The default values $(p_1, p_2, p_3, p_4) = (0, 1, 0, 1)$ give $A \approx 0$ (no auto-regression at initialization).

Precision operator

The 2T x 2T precision operator is:

$$K = M_0 + a_{11}\, M_{11} + a_{22}\, M_{22} + a_{12}\, M_{12} + a_{21}\, M_{21}$$

where the M_{ij} are fixed sparse block matrices constructed from the $T \times T$ first-order difference matrix C_T.

Examples

```
set.seed(1)
n_obs <- 10
dat <- data.frame(
  y     = rnorm(2 * n_obs),
  idx   = rep(seq_len(n_obs), 2),
  group = factor(rep(c("y1", "y2"), each = n_obs))
)

fit <- ngme(
  y ~ 0 + f(idx, model = var1(mesh = 1:n_obs), group = group,
            noise = noise_nig()),
  data    = dat,
  family  = noise_normal(sigma = 0.01, fix_sigma = TRUE),
  control_opt = control_opt(estimation = FALSE)
)
print(fit)
```

Index

∗ **datasets**
argo_float, 9
cienaga, 18
cienaga.border, 19
pm25_quarterly_2022, 87

adagrad, 5
adam, 6
adamW, 7
adaptive_gd, 8
ar, 8
ar1, 9
argo_float, 9
arma, 10
arma11, 11

batch_decay, 12
batch_means_ci, 12
batch_means_estimator, 13
bfgs, 14
bv, 15
bv_matern, 16

calibrate_inv_exp_lambda *(calibrate_inv_exp_lambda_driven_nig)*, 17
calibrate_inv_exp_lambda_driven_nig, 17
cienaga, 18
cienaga.border, 19
compare_noise_kld, 19
compute_index_corr_from_map, 20
compute_log_like, 20
compute_ngme_CI *(compute_ngme_ci)*, 21
compute_ngme_ci, 21
compute_ngme_sgld_samples, 23
compute_score_given_pred, 24
control_ngme, 25
control_opt, 26
control_opt_batch_ci, 36
create_paired_cv_splits, 37
cross_validation, 38

dgal *(gal)*, 42
dgig, *43*, *58*, *59*, *82*
dgig *(gig)*, 51
dig, *43*, *52*, *82*
dig *(ig)*, 56
digam, *43*, *52*, *58*, *82*
digam *(igam)*, 58
dnig, *52*, *58*, *59*
dnig *(nig)*, 80

f, 40, *100*

gal, 42
generic, 44
generic_ns, 46
get_data_from_formula, 48
get_parameter_distance, 49
get_trace_trajectories, 50
get_trajectories, 50
gig, 51
group_cv, 52

ig, 56
igam, 58
iid, 59

make_time_series_cv_index, 60
matern, 61
mean_list, 62
merge_noise, 63
merge_replicates, 63
momentum, 64

name2fun, 64
ngme, 65
ngme_as_sparse, 66
ngme_batch_ci, 67
ngme_cov_matrix, 68

ngme_make_mesh_repls, 68
ngme_model_types, 69
ngme_noise, 69
ngme_noise_types, 73
ngme_optimizers, 74
ngme_parse_formula, 74
ngme_post_samples, 75
ngme_prior_types, 76
ngme_result, 76
ngme_sgld_ci, 78
ngme_sgld_samples, 78
ngme_ts_make_A, 79
ngme_update, 80
nig, 80
noise_gal *(*ngme_noise*)*, 69
noise_gal_moments *(*noise_moments*)*, 82
noise_moments, 82
noise_nig *(*ngme_noise*)*, 69
noise_nig_moments *(*noise_moments*)*, 82
noise_normal *(*ngme_noise*)*, 69
noise_normal_nig *(*ngme_noise*)*, 69
noise_skew_t *(*ngme_noise*)*, 69
noise_t *(*ngme_noise*)*, 69

openmp_test, 83
ou, 83

pgal *(*gal*)*, 42
pgig *(*gig*)*, 51
pig *(*ig*)*, 56
pigam *(*igam*)*, 58
plot.ngme_noise, 85
plot.ngme_sgld_ci *(*posterior_plot*)*, 88
plot.parameter_distance, 86
pm25_quarterly_2022, 87
pnig *(*nig*)*, 80
poly_decay, 87
posterior_plot, 88
precision_matrix_multivariate, 89
precision_matrix_multivariate_spde, 90
precond_sgd, 92
predict.ngme, 93
print.ngme, 94
print.ngme_model, 95
print.ngme_noise, 95
print.ngme_operator, 96
print.ngme_replicate, 96
print.ngme_trajectories, 97
print.noise_kld_comparison, 97
print.parameter_distance, 98
prior_half_cauchy, 98
prior_inv_exp *(*prior_inv_exponential*)*, 99
prior_inv_exponential, 99, *100, 101*
prior_none, 99
prior_normal, 100
prior_pc_nu, *41*, *73*, 100
prior_pc_sd, 101
priors, 102

qgal *(*gal*)*, 42
qgig *(*gig*)*, 51
qig *(*ig*)*, 56
qigam *(*igam*)*, 58
qnig *(*nig*)*, 80

re, 102
rgal *(*gal*)*, 42
rgig *(*gig*)*, 51
rig *(*ig*)*, 56
rigam *(*igam*)*, 58
rmsprop, 103
rnig *(*nig*)*, 80
rw1, 103
rw2, 105

sgd, 106
sgld, 106
simulate.ngme, 107
simulate.ngme_model, 108
simulate.ngme_noise, 108
spacetime, 109
stepsize_control, 111
stepsize_decay, 111
stepsize_schedule, 112
summary.ngme, 113
summary.ngme_batch_ci, 114

tp, 114
traceplot, 115

var1, 116