Package: LaMa (via r-universe)

May 28, 2026

Type Package

Title Fast Numerical Maximum Likelihood Estimation for Latent Markov Models

Version 2.1.1

Description A variety of latent Markov models, including hidden Markov models, hidden semi-Markov models, state-space models and continuous-time variants can be formulated and estimated within the same framework via directly maximising the likelihood function using the so-called forward algorithm. Applied researchers often need custom models that standard software does not easily support. Writing tailored 'R' code offers flexibility but suffers from slow estimation speeds. We address these issues by providing easy-to-use functions (written in 'C++' for speed) for common tasks like the forward algorithm. These functions can be combined into custom models in a Lego-type approach, offering up to 10-20 times faster estimation via standard numerical optimisers. To aid in building fully custom likelihood functions, several vignettes are included that show how to simulate data from and estimate all the above model classes.

URL https://janolefi.github.io/LaMa/

License MIT + file LICENSE

Encoding UTF-8

Imports Rcpp, stats, utils, methods, Matrix, splines2, mgcv, MASS, numDeriv, RTMBdist (>= 1.0.4)

LinkingTo Rcpp, RcppArmadillo

Depends R (>= 3.5.0), RTMB (>= 1.9.0)

RoxygenNote 7.3.3

Suggests knitr, rmarkdown, testthat (>= 3.0.0), PHSMM, MSwM, scales

VignetteBuilder knitr

Config/testthat/edition 3

LazyData true

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

Date/Publication 2026-05-28 10:18:47 UTC

RemoteUrl https://github.com/janolefi/lama

RemoteRef HEAD

RemoteSha 7035273e0e481e3767ff7ec555b69be27651a1f9

Contents

`%sp%` *Sparsity-retaining matrix multiplication*

Description

Standard matrix multiplication destroys automatic sparsity detection by `RTMB` which is essential for models with high-dimensional random effects. This can be mitigated by changing to "plain" with `TapeConfig`, but this can make AD tape construction very slow. Here, we provide a different version that retains sparsity. It may be slightly slower than the standard method when constructing the AD tape.

Usage

```
A %sp% B
```

Arguments

A	matrix of dimension n x p
B	matrix of dimension p x m

Value

the matrix product of A and B, which is of dimension n x m

Examples

```
A <- matrix(1:6, nrow = 2, ncol = 3)
B <- matrix(7:12, nrow = 3, ncol = 2)
A %sp% B
```

---

calc_trackInd *Calculate the index of the first observation of each track based on an ID variable*

Description

Function to conveniently calculate the trackInd variable that is needed internally when fitting a model to longitudinal data with multiple tracks.

Usage

```
calc_trackInd(ID)
```

Arguments

ID	ID variable of track IDs that is of the same length as the data to be analysed

Value

A vector of indices of the first observation of each track which can be passed to the forward and forward_g to sum likelihood contributions of each track

Examples

```
uniqueID = c("Animal1", "Animal2", "Animal3")
ID = rep(uniqueID, c(100, 200, 300))
trackInd = calc_trackInd(ID)
```

`cosinor` *Trigonometric basis expansion*

Description

Builds a design matrix of `sin`/`cos` pairs for use in models with periodic predictors. Can be used directly or inside formulas passed to `make_matrices` (where expansion is handled automatically).

Usage

```
cosinor(x, period = 24)
```

Arguments

`x`	Numeric vector of the periodic variable.
`period`	Numeric vector of period lengths, e.g. `24` for a daily cycle with hourly data or `c(24, 12)` for a daily + semi-daily cycle.

Details

The resulting columns form the basis for linear predictors of the form

$$\eta_t = \beta_0 + \sum_k \left(\beta_{1k} \sin\!\left(\tfrac{2\pi x_t}{\text{period}_k}\right) + \beta_{2k} \cos\!\left(\tfrac{2\pi x_t}{\text{period}_k}\right) \right).$$

Value

A numeric matrix with `2 * length(period)` columns named `sin(2*pi*x/period)` / `cos(2*pi*x/period)`.

Examples

```
cosinor(1:24, period = 24)
cosinor(1:24, period = c(24, 12, 6))

## In model formulas (expand_cosinor handles the expansion):
form <- ~ x + temp * cosinor(hour, c(24, 12))
data <- data.frame(x = runif(24), temp = rnorm(24, 20), hour = 1:24)
modmat <- make_matrices(form, data = data)
```

ddwell *State dwell-time distributions of periodically inhomogeneous Markov chains*

Description

Computes the dwell-time distribution of a periodically inhomogeneous Markov chain for a given transition probability matrix.

Usage

```
ddwell(x, Gamma, time = NULL, state = NULL)
```

Arguments

x	vector of (non-negative) dwell times to compute the dwell-time distribution for
Gamma	array of L unique transition probability matrices of a periodically inhomogeneous Markov chain, with dimensions c(N,N,L), where N is the number of states and L is the cycle length
time	integer vector of time points in 1:L at which to compute the dwell-time distribution. If NULL, the overall dwell-time distribution is computed.
state	integer vector of state indices for which to compute the dwell-time distribution. If NULL, dwell-time distributions for all states are returned in a named list.

Details

For Markov chains whose transition probabilities vary only periodically, which is achieved for example by expressing the transition probability matrix as a periodic function of the time of day using tpm_p or cosinor, the probability distribution of time spent in a state can be computed analytically. This function computes said distribution, either for a specific time point (conditioning on transitioning into the state at that time point) or for the overall distribution (conditioning on transitioning into the state at any time point).

Value

either time-varying dwell-time distribution(s) if time is specified, or overall dwell-time distribution if time is NULL. If more than one state is specified, a named list over states is returned.

References

Koslik, J. O., Feldmann, C. C., Mews, S., Michels, R., & Langrock, R. (2023). Inference on the state process of periodically inhomogeneous hidden Markov models for animal behavior. arXiv preprint arXiv:2312.14583.

Examples

```
# setting parameters for trigonometric link
beta = matrix(c(-1, 2, -1, -2, 1, -1), nrow = 2, byrow = TRUE)
Gamma = tpm_p(beta = beta, degree = 1)

# at specific times and for specific state
ddwell(1:20, Gamma, time = 1:4, state = 1)
# results in 4x20 matrix

# or overall distribution for all states
ddwell(1:20, Gamma)
# results in list of length 2, each element is a vector of length 20
```

---

dgmrf2 *Reparametrised multivariate Gaussian distribution*

Description

Density function of the multivariate Gaussian distribution reparametrised in terms of its precision matrix (inverse variance). This implementation is particularly useful for defining the **joint log-likelihood** with penalised splines or i.i.d. random effects that have a multivariate Gaussian distribution with fixed precision/ penalty matrix λS. As S is fixed and only scaled by λ, it is more efficient to precompute the determinant of S (for the normalisation constant) and only scale the quadratic form by λ when multiple spline parameters/ random effects with different λ's but the same penalty matrix S are evaluated.

Usage

```
dgmrf2(x, mu = 0, S, lambda, logdetS = NULL, log = FALSE)
```

Arguments

`x`	density evaluation point, either a vector or a matrix
`mu`	mean parameter. Either scalar or vector
`S`	unscaled precision matrix
`lambda`	precision scaling parameter Can be a vector if `x` is a matrix. Then each row of `x` is evaluated with the corresponding `lambda`. This is benefitial from an efficiency perspective because the determinant of `S` is only computed once.
`logdetS`	Optional precomputed log determinant of the precision matrix `S`. If the precision matrix does not depend on parameters, it can be precomputed and passed to the function.
`log`	logical; if `TRUE`, densities are returned on the log scale.

Details

This implementation allows for automatic differentiation with RTMB.

Value

vector of density values

Examples

```
x = matrix(runif(30), nrow = 3)

# iid random effects
S = diag(10)
sigma = c(1, 2, 3) # random effect standard deviations
lambda = 1 / sigma^2
d = dgmrf2(x, 0, S, lambda)

# P-splines
L = diff(diag(10), diff = 2) # second-order difference matrix
S = t(L) %*% L
lambda = c(1,2,3)
d = dgmrf2(x, 0, S, lambda, log = TRUE)
```

---

forward *Forward algorithm to calculate the HMM log-likelihood*

Description

Calculates the log-likelihood of a sequence of observations under a hidden Markov model using the **forward algorithm** (Zucchini, MacDonald & Langrock, 2016).

Usage

```
forward(
  delta,
  Gamma,
  allprobs,
  trackID = NULL,
  logspace = FALSE,
  bw = NULL,
  report = TRUE,
  ad = NULL
)
```

Arguments

`delta`	initial distribution; either • a vector of length `nStates`, or • a matrix of dimension `c(nTracks, nStates)`, if `trackID` is provided.
`Gamma`	transition probability matrix; either • a matrix of dimension `c(nStates, nStates)`, or • an array of dimension `c(nStates, nStates, nTracks)` if `trackID` is provided, or • an array of dimension `c(nStates, nStates, nObs)` for time-varying transition probabilities, in which case `forward_g` is called internally.
`allprobs`	matrix of state-dependent probabilities or density values of dimension `c(nObs, nStates)`
`trackID`	optional vector of length `nObs` containing `nTracks` unique IDs that separate tracks (see 'Details').
`logspace`	logical; if `TRUE`, `allprobs` is assumed to be on the log-scale, improving numerical stability for small probabilities. Only supported with `RTMB`.
`bw`	optional positive integer specifying the bandwidth for a banded approximation of the forward algorithm, inducing a banded Hessian w.r.t. the observations. Defaults to `NULL` (exact algorithm). Approximation error decays geometrically in `bw`.
`report`	logical; if `TRUE` (default), `delta`, `Gamma`, `allprobs`, and `trackID` are reported from the fitted model. Requires `ad = TRUE`.
`ad`	logical; whether to use automatic differentiation. Determined automatically — for debugging only.

Details

If `trackID` is provided, the total log-likelihood is the sum of each track's likelihood contribution. In this case, `Gamma` can be

- a matrix (same transition probabilities for each track),
- an array of dimension `c(nStates, nStates, nTracks)` (track-specific transition probability matrix), or
- an array of dimension `c(nStates, nStates, nObs)` for time-varying transition probabilities, in which case `forward_g` is called internally.

Additionally, `delta` can be a vector (same initial distribution for each track) or a matrix of dimension `c(nTracks, nStates)` (track-specific initial distributions).

Note: When there are multiple tracks, for compatibility with downstream functions like `viterbi`, `stateprobs` or `pseudo_res`, `forward` should only be called **once** with a `trackID` argument.

Value

HMM log-likelihood for given data and parameters

References

Zucchini, W., MacDonald, I.L., & Langrock, R. (2016). *Hidden Markov Models for Time Series: An Introduction Using R* (2nd ed.). Chapman & Hall/CRC.

See Also

Other forward algorithms: forward_g(), forward_hsmm(), forward_ihsmm(), forward_p(), forward_phsmm()

Examples

```
## negative log likelihood function
nll = function(par, step) {
 # parameter transformations for unconstrained optimisation
 Gamma = tpm(par[1:2]) # multinomial logit link
 delta = stationary(Gamma) # stationary HMM
 mu = exp(par[3:4])
 sigma = exp(par[5:6])
 # calculate all state-dependent probabilities
 allprobs = matrix(1, length(step), 2)
 ind = which(!is.na(step))
 for(j in 1:2) allprobs[ind,j] = dgamma2(step[ind], mu[j], sigma[j])
 # simple forward algorithm to calculate log-likelihood
 -forward(delta, Gamma, allprobs)
}

## fitting an HMM to the trex data
par = c(-2,-2,            # initial tpm params (logit-scale)
        log(c(0.3, 2.5)), # initial means for step length (log-transformed)
        log(c(0.2, 1.5))) # initial sds for step length (log-transformed)
mod = nlm(nll, par, step = trex$step[1:1000])
```

---

forward_g *Forward algorithm with time-varying transition probability matrix*

Description

Calculates the log-likelihood of a sequence of observations under a hidden Markov model with time-varying transition probabilities using the **forward algorithm** (Zucchini, MacDonald & Langrock, 2016).

Usage

```
forward_g(
  delta,
  Gamma,
  allprobs,
  trackID = NULL,
  logspace = FALSE,
```

```
  bw = NULL,
  report = TRUE,
  ad = NULL
)
```

Arguments

`delta`	initial distribution; either • a vector of length `nStates`, or • a matrix of dimension `c(nTracks, nStates)`, if `trackID` is provided.
`Gamma`	array of transition probability matrices of dimension `c(nStates, nStates, nObs)`, where the first slice of each track is ignored as there is no transition into the start of a track. For a single track, an array of dimension `c(nStates, nStates, nObs-1)` is also accepted. This function also supports continuous-time HMMs, where each slice is a Markov semigroup $\Gamma(\Delta t) = \exp(Q\Delta t)$ for generator Q.
`allprobs`	matrix of state-dependent probabilities or density values of dimension `c(nObs, nStates)`
`trackID`	optional vector of length `nObs` containing `nTracks` unique IDs that separate tracks (see 'Details').
`logspace`	logical; if `TRUE`, `allprobs` is assumed to be on the log-scale, improving numerical stability for small probabilities. Only supported with `RTMB`.
`bw`	optional positive integer specifying the bandwidth for a banded approximation of the forward algorithm, inducing a banded Hessian w.r.t. the observations. Defaults to `NULL` (exact algorithm). Approximation error decays geometrically in `bw`.
`report`	logical; if `TRUE` (default), `delta`, `Gamma`, `allprobs`, and `trackID` are reported from the fitted model. Requires `ad = TRUE`.
`ad`	logical; whether to use automatic differentiation. Determined automatically — for debugging only.

Details

If `trackID` is provided, the total log-likelihood will be the sum of each track's likelihood contribution. In this case, `Gamma` must be an array of dimension `c(nStates, nStates, nObs)`, matching the number of rows of allprobs. For each track, the transition matrix at the beginning of the track will be ignored (as there is no transition between tracks). Additionally, `delta` can be a vector (same initial distribution for each track) or a matrix of dimension `c(nTracks, nStates)` (different initial distribution for each track).

Note: When there are multiple tracks, for compatibility with downstream functions like `viterbi_g`, `stateprobs_g` or `pseudo_res`, `forward_g` should only be called **once** with a `trackID` argument.

Value

HMM log-likelihood for given data and parameters

References

Zucchini, W., MacDonald, I.L., & Langrock, R. (2016). *Hidden Markov Models for Time Series: An Introduction Using R* (2nd ed.). Chapman & Hall/CRC.

See Also

Other forward algorithms: `forward()`, `forward_hsmm()`, `forward_ihsmm()`, `forward_p()`, `forward_phsmm()`

Examples

```
## Simple usage
Gamma = array(c(0.9, 0.2, 0.1, 0.8), dim = c(2,2,10))
delta = c(0.5, 0.5)
allprobs = matrix(0.5, 10, 2)
forward_g(delta, Gamma, allprobs)

## Full model fitting example
## negative log likelihood function
nll = function(par, step, Z) {
 # parameter transformations for unconstrained optimisation
 beta = matrix(par[1:6], nrow = 2)
 Gamma = tpm_g(Z, beta) # multinomial logit link for each time point
 delta = stationary(Gamma[,,1]) # stationary HMM
 mu = exp(par[7:8])
 sigma = exp(par[9:10])
 # calculate all state-dependent probabilities
 allprobs = matrix(1, length(step), 2)
 ind = which(!is.na(step))
 for(j in 1:2) allprobs[ind,j] = dgamma2(step[ind], mu[j], sigma[j])
 # simple forward algorithm to calculate log-likelihood
 -forward_g(delta, Gamma, allprobs)
}

## fitting an HMM to the trex data
par = c(-1.5,-1.5,        # initial tpm intercepts (logit-scale)
        rep(0, 4),        # initial tpm slopes
        log(c(0.3, 2.5)), # initial means for step length (log-transformed)
        log(c(0.2, 1.5))) # initial sds for step length (log-transformed)
mod = nlm(nll, par, step = trex$step[1:500], Z = cosinor(trex$tod[1:500]))
```

---

forward_hsmm — R*hrefhttps://www.taylorfrancis.com/books/mono/10.1201/b20790/hidden-markov-models-time-series-walter-zucchini-iain-macdonald-roland-langrockForward algorithm for homogeneous hidden semi-Markov models*

Description

Calculates the (approximate) log-likelihood of a sequence of observations under a homogeneous hidden semi-Markov model using a modified **forward algorithm**.

Usage

```
forward_hsmm(
  dm,
  omega,
  allprobs,
  trackID = NULL,
  delta = NULL,
  eps = 1e-10,
  report = TRUE
)
```

Arguments

`dm`	list of length `nStates` containing vectors of dwell-time probability mass functions (PMFs) for each state. The vector lengths correspond to the approximating state aggregate sizes, hence there should be little probablity mass not covered by these.
`omega`	matrix of dimension `c(nStates, nStates)` of conditional transition probabilites, also called embedded transition probability matrix. Contains the transition probabilities given that the current state is left. Hence, the diagonal elements need to be zero and the rows need to sum to one. Can be constructed using `tpm_emb`.
`allprobs`	matrix of state-dependent probabilities/ density values of dimension `c(nObs, nStates)` which will automatically be converted to the appropriate dimension.
`trackID`	optional vector of length `nObs` containing IDs If provided, the total log-likelihood will be the sum of each track's likelihood contribution. In this case, `dm` can be a nested list, where the top layer contains k `dm` lists as described above. `omega` can then also be an array of dimension `c(nStates, nStates, nTracks)` with one conditional transition probability matrix for each track. Furthermore, instead of a single vector `delta` corresponding to the initial distribution, a `delta` matrix of initial distributions, of dimension `c(nTracks, nStates)`, can be provided, such that each track starts with it's own initial distribution.
`delta`	optional vector of initial state probabilities of length `nStates` By default, the stationary distribution is computed (which is typically recommended).
`eps`	small value to avoid numerical issues in the approximating transition matrix construction. Usually, this should not be changed.
`report`	logical, indicating whether initial distribution, approximating transition probability matrix and `allprobs` matrix should be reported from the fitted model. Defaults to `TRUE`.

Details

Hidden semi-Markov models (HSMMs) are a flexible extension of HMMs, where the state duration distribution is explicitly modelled by a distribution on the positive integers. For direct numerical maximum likelihood estimation, HSMMs can be represented as HMMs on an enlarged state space (of size M) and with structured transition probabilities.

This function is designed to be used with automatic differentiation based on the `R` package `RTMB`. It will be very slow without it!

Value

HSMM log-likelihood for given data and parameters

References

Langrock, R., & Zucchini, W. (2011). Hidden Markov models with arbitrary state dwell-time distributions. Computational Statistics & Data Analysis, 55(1), 715-724.

Koslik, J. O. (2025). Hidden semi-Markov models with inhomogeneous state dwell-time distributions. Computational Statistics & Data Analysis, 209, 108171.

See Also

Other forward algorithms: `forward()`, `forward_g()`, `forward_ihsmm()`, `forward_p()`, `forward_phsmm()`

Examples

```
# currently no examples
```

`forward_ihsmm` R*hrefhttps://www.taylorfrancis.com/books/mono/10.1201/b20790/hidden-markov-models-time-series-walter-zucchini-iain-macdonald-roland-langrockForward algorithm for hidden semi-Markov models with inhomogeneous state durations and/ or conditional transition probabilities*

Description

Calculates the (approximate) log-likelihood of a sequence of observations under an inhomogeneous hidden semi-Markov model using a modified **forward algorithm**.

Usage

```
forward_ihsmm(
  dm,
  omega,
  allprobs,
  trackID = NULL,
  delta = NULL,
```

```
  startInd = NULL,
  eps = 1e-10,
  report = TRUE
)
```

Arguments

dm	list of length nStates containing matrices (or vectors) of dwell-time probability mass functions (PMFs) for each state. If the dwell-time PMFs are constant, the vectors are the PMF of the dwell-time distribution fixed in time. The vector lengths correspond to the approximating state aggregate sizes, hence there should be little probablity mass not covered by these. If the dwell-time PMFs are inhomogeneous, the matrices need to have n rows, where n is the number of observations. The number of columns again corresponds to the size of the approximating state aggregates. In the latter case, the first max(sapply(dm, ncol)) - 1 observations will not be used because the first approximating transition probability matrix needs to be computed based on the first max(sapply(dm, ncol)) covariate values (represented by dm).
omega	matrix of dimension c(nStates, nStates) or array of dimension c(nStates, nStates, nObs) of conditional transition probabilites, also called embedded transition probability matrix. It contains the transition probabilities given the current state is left. Hence, the diagonal elements need to be zero and the rows need to sum to one. Such a matrix can be constructed using tpm_emb and an array using tpm_emb_g.
allprobs	matrix of state-dependent probabilities/ density values of dimension c(nObs, nStates)
trackID	trackID optional vector of length nObs containing IDs If provided, the total log-likelihood will be the sum of each track's likelihood contribution. Instead of a single vector delta corresponding to the initial distribution, a delta matrix of initial distributions, of dimension c(nTracks, nStates), can be provided, such that each track starts with it's own initial distribution.
delta	optional vector of initial state probabilities of length N By default, instead of this, the stationary distribution is computed corresponding to the first approximating transition probability matrix of each track is computed. Contrary to the homogeneous case, this is not theoretically motivated but just for convenience.
startInd	optional integer index at which the forward algorithm starts. When approximating inhomogeneous HSMMs by inhomogeneous HMMs, the first transition probability matrix that can be constructed is at time max(sapply(dm, ncol)) (as it depends on the previous covariate values). Hence, when not provided, startInd is chosen to be max(sapply(dm, ncol)). Fixing startInd at a value **larger** than max(aggregate sizes) is useful when models with different aggregate sizes are fitted to the same data and are supposed to be compared. In that case it is important that all models use the same number of observations.

`eps`	small value to avoid numerical issues in the approximating transition matrix construction. Usually, this should not be changed.
`report`	logical, indicating whether initial distribution, approximating transition probability matrix and `allprobs` matrix should be reported from the fitted model. Defaults to `TRUE`.

Details

Hidden semi-Markov models (HSMMs) are a flexible extension of HMMs, where the state duration distribution is explicitly modelled by a distribution on the positive integers. This function can be used to fit HSMMs where the state-duration distribution and/ or the conditional transition probabilities vary with covariates. For direct numerical maximum likelihood estimation, HSMMs can be represented as HMMs on an enlarged state space (of size M) and with structured transition probabilities.

This function is designed to be used with automatic differentiation based on the `R` package `RTMB`. It will be very slow without it!

Value

HSMM log-likelihood for given data and parameters

References

Koslik, J. O. (2025). Hidden semi-Markov models with inhomogeneous state dwell-time distributions. Computational Statistics & Data Analysis, 209, 108171.

See Also

Other forward algorithms: `forward()`, `forward_g()`, `forward_hsmm()`, `forward_p()`, `forward_phsmm()`

Examples

```
# currently no examples
```

`forward_p` R*hrefhttps://www.taylorfrancis.com/books/mono/10.1201/b20790/hidden-markov-models-time-series-walter-zucchini-iain-macdonald-roland-langrockForward algorithm with for periodically varying transition probability matrices*

Description

Calculates the log-likelihood of a sequence of observations under a hidden Markov model with periodically varying transition probabilities using the **forward algorithm**.

Usage

```
forward_p(
  delta,
  Gamma,
  allprobs,
  tod,
  trackID = NULL,
  ad = NULL,
  report = TRUE,
  logspace = FALSE
)
```

Arguments

`delta`	initial or stationary distribution of length `nStates`, or matrix of dimension `c(nTracks, nStates)` for `nTracks` independent tracks, if `trackID` is provided
`Gamma`	array of transition probability matrices of dimension `c(nStates, nStates, L)`. Here we use the definition $\Pr(S_t = j \mid S_{t-1} = i) = \gamma_{ij}^{(t)}$ such that the transition probabilities between time point $t-1$ and t are an element of $\Gamma^{(t)}$.
`allprobs`	matrix of state-dependent probabilities/ density values of dimension `c(nObs, nStates)`
`tod`	(Integer valued) variable for cycle indexing in 1, ..., L, mapping the data index to a generalised time of day (length `nObs`) For half-hourly data L = 48. It could, however, also be day of year for daily data and L = 365.
`trackID`	optional vector of length `nObs` containing IDs If provided, the total log-likelihood will be the sum of each track's likelihood contribution. Instead of a single vector `delta` corresponding to the initial distribution, a `delta` matrix of initial distributions of dimension `c(nTracks, nObs)`, can be provided, such that each track starts with it's own initial distribution.
`ad`	optional logical, indicating whether automatic differentiation with `RTMB` should be used. By default, the function determines this itself.
`report`	logical, indicating whether `delta`, `Gamma`, `allprobs`, and potentially `trackID` should be reported from the fitted model. Defaults to `TRUE`, but only works if `ad = TRUE`, as it uses the `RTMB` package. **Caution:** When there are multiple tracks, for compatibility with downstream functions like `viterbi_p`, `stateprobs_p` or `pseudo_res`, `forward_p` should only be called **once** with a `trackID` argument.
`logspace`	logical, indicating whether the probabilities/ densities in the `allprobs` matrix are on log-scale. If so, internal computations are also done on log-scale which is numerically more robust when the entries are very small.

Details

When the transition probability matrix only varies periodically (e.g. as a function of time of day), there are only L unique matrices if L is the period length (e.g. $L = 24$ for hourly data and time-of-day variation). Thus, it is much more efficient to only calculate these L matrices and index them

by a time variable (e.g. time of day or day of year) instead of calculating such a matrix for each index in the data set (which would be redundant). This function allows for that by only expecting a transition probability matrix for each time point in a period and an integer valued $(1, \ldots, L)$ time variable that maps the data index to the according time.

Value

HMM log-likelihood for given data and parameters

See Also

Other forward algorithms: forward(), forward_g(), forward_hsmm(), forward_ihsmm(), forward_phsmm()

Examples

```
## negative log likelihood function
nll = function(par, step, tod) {
 # parameter transformations for unconstrained optimisation
 beta = matrix(par[1:6], nrow = 2)
 Gamma = tpm_p(1:24, beta = beta) # multinomial logit link for each time point
 delta = stationary_p(Gamma, tod[1]) # stationary HMM
 mu = exp(par[7:8])
 sigma = exp(par[9:10])
 # calculate all state-dependent probabilities
 allprobs = matrix(1, length(step), 2)
 ind = which(!is.na(step))
 for(j in 1:2) allprobs[ind,j] = dgamma2(step[ind], mu[j], sigma[j])
 # simple forward algorithm to calculate log-likelihood
 -forward_p(delta, Gamma, allprobs, tod)
}

## fitting an HMM to the nessi data
par = c(-2,-2,            # initial tpm intercepts (logit-scale)
        rep(0, 4),        # initial tpm slopes
        log(c(0.3, 2.5)), # initial means for step length (log-transformed)
        log(c(0.2, 1.5))) # initial sds for step length (log-transformed)
mod = nlm(nll, par, step = trex$step[1:500], tod = trex$tod[1:500])
```

---

forward_phsmm — R*hrefhttps://www.taylorfrancis.com/books/mono/10.1201/b20790/hidden-markov-models-time-series-walter-zucchini-iain-macdonald-roland-langrockForward algorithm for hidden semi-Markov models with periodically inhomogeneous state durations and/ or conditional transition probabilities*

Description

Hidden semi-Markov models (HSMMs) are a flexible extension of HMMs, where the state duration distribution is explicitly modelled by a distribution on the positive integers. This function can be used to fit HSMMs where the state-duration distribution and/ or the conditional transition probabilities vary with covariates. For direct numerical maximum likelihood estimation, HSMMs can be represented as HMMs on an enlarged state space (of size M) and with structured transition probabilities.

This function can be used to fit HSMMs where the state-duration distribution and/ or the conditional transition probabilities vary periodically. In the special case of periodic variation (as compared to arbitrary covariate influence), this version is to be preferred over `forward_ihsmm` because it computes the **correct periodically stationary distribution** and no observations are lost for the approximation.

This function is designed to be used with automatic differentiation based on the `R` package `RTMB`. It will be very slow without it!

Usage

```
forward_phsmm(
  dm,
  omega,
  allprobs,
  tod,
  trackID = NULL,
  delta = NULL,
  eps = 1e-10,
  report = TRUE
)
```

Arguments

`dm`	list of length `nStates` containing matrices (or vectors) of dwell-time probability mass functions (PMFs) for each state. If the dwell-time PMFs are constant, the vectors are the PMF of the dwell-time distribution fixed in time. The vector lengths correspond to the approximating state aggregate sizes, hence there should be little probablity mass not covered by these. If the dwell-time PMFs are inhomogeneous, the matrices need to have L rows, where L is the cycle length. The number of columns again correpond to the size of the approximating state aggregates.
`omega`	matrix of dimension `c(nStates, nStates)` or array of dimension `c(nStates, nStates, L)` of conditional transition probabilites, also called embedded transition probability matrix It contains the transition probabilities given the current state is left. Hence, the diagonal elements need to be zero and the rows need to sum to one. Such a matrix can be constructed using `tpm_emb` and an array using `tpm_emb_g`.
`allprobs`	matrix of state-dependent probabilities/ density values of dimension `c(nObs, nStates)`

`tod`	(Integer valued) variable for cycle indexing in 1, ..., L, mapping the data index to a generalised time of day (length `nObs`). For half-hourly data L = 48. It could, however, also be day of year for daily data and L = 365.
`trackID`	optional vector of length `nObs` containing IDs If provided, the total log-likelihood will be the sum of each track's likelihood contribution. Instead of a single vector `delta` corresponding to the initial distribution, a `delta` matrix of initial distributions, of dimension `c(nTracks, nStates)`, can be provided, such that each track starts with it's own initial distribution.
`delta`	Optional vector of initial state probabilities of length `nStates`. By default, instead of this, the stationary distribution is computed corresponding to the first approximating t.p.m. of each track is computed. Contrary to the homogeneous case, this is not theoretically motivated but just for convenience.
`eps`	small value to avoid numerical issues in the approximating transition matrix construction. Usually, this should not be changed.
`report`	logical, indicating whether initial distribution, approximating transition probability matrix and `allprobs` matrix should be reported from the fitted model. Defaults to `TRUE`.

Details

Calculates the (approximate) log-likelihood of a sequence of observations under a periodically inhomogeneous hidden semi-Markov model using a modified **forward algorithm**.

Value

HSMM log-likelihood for given data and parameters

References

Koslik, J. O. (2025). Hidden semi-Markov models with inhomogeneous state dwell-time distributions. Computational Statistics & Data Analysis, 209, 108171.

See Also

Other forward algorithms: `forward()`, `forward_g()`, `forward_hsmm()`, `forward_ihsmm()`, `forward_p()`

Examples

```
# currently no examples
```

forward_s R*hrefhttps://www.taylorfrancis.com/books/mono/10.1201/b20790/hidden-markov-models-time-series-walter-zucchini-iain-macdonald-roland-langrockForward algorithm for hidden semi-Markov models with homogeneous transition probability matrix*

Description

Hidden semi-Markov models (HSMMs) are a flexible extension of HMMs that can be approximated by HMMs on an enlarged state space (of size M) and with structured transition probabilities.

Usage

```
forward_s(delta, Gamma, allprobs, sizes)
```

Arguments

delta	initial or stationary distribution of length N, or matrix of dimension c(k,N) for k independent tracks, if trackID is provided
Gamma	transition probability matrix of dimension c(M,M)
allprobs	matrix of state-dependent probabilities/ density values of dimension c(n, N) which will automatically be converted to the appropriate dimension.
sizes	state aggregate sizes that are used for the approximation of the semi-Markov chain.

Value

log-likelihood for given data and parameters

Examples

```
## generating data from homogeneous 2-state HSMM
mu = c(0, 6)
lambda = c(6, 12)
omega = matrix(c(0,1,1,0), nrow = 2, byrow = TRUE)
# simulation
# for a 2-state HSMM the embedded chain always alternates between 1 and 2
s = rep(1:2, 100)
C = x = numeric(0)
for(t in 1:100){
  dt = rpois(1, lambda[s[t]])+1 # shifted Poisson
  C = c(C, rep(s[t], dt))
  x = c(x, rnorm(dt, mu[s[t]], 1.5)) # fixed sd 2 for both states
}

## negative log likelihood function
mllk = function(theta.star, x, sizes){
  # parameter transformations for unconstraint optimization
```

```
  omega = matrix(c(0,1,1,0), nrow = 2, byrow = TRUE) # omega fixed (2-states)
  lambda = exp(theta.star[1:2]) # dwell time means
  dm = list(dpois(1:sizes[1]-1, lambda[1]), dpois(1:sizes[2]-1, lambda[2]))
  Gamma = tpm_hsmm2(omega, dm)
  delta = stationary(Gamma) # stationary
  mu = theta.star[3:4]
  sigma = exp(theta.star[5:6])
  # calculate all state-dependent probabilities
  allprobs = matrix(1, length(x), 2)
  for(j in 1:2){ allprobs[,j] = dnorm(x, mu[j], sigma[j]) }
  # return negative for minimization
  -forward_s(delta, Gamma, allprobs, sizes)
}

## fitting an HSMM to the data
theta.star = c(log(5), log(10), 1, 4, log(2), log(2))
mod = nlm(mllk, theta.star, x = x, sizes = c(20, 30), stepmax = 5)
```

`forward_sp` R*hrefhttps://www.taylorfrancis.com/books/mono/10.1201/b20790/hidden-markov-models-time-series-walter-zucchini-iain-macdonald-roland-langrockForward algorithm for hidden semi-Markov models with periodically varying transition probability matrices*

Description

Hidden semi-Markov models (HSMMs) are a flexible extension of HMMs that can be approximated by HMMs on an enlarged state space (of size M) and with structured transition probabilities. Recently, this inference procedure has been generalised to allow either the dwell-time distributions or the conditional transition probabilities to depend on external covariates such as the time of day. This special case is implemented here. This function allows for that, by expecting a transition probability matrix for each time point in a period, and an integer valued $(1, \ldots, L)$ time variable that maps the data index to the according time.

Usage

```
forward_sp(delta, Gamma, allprobs, sizes, tod)
```

Arguments

`delta`	initial or stationary distribution of length N, or matrix of dimension c(k,N) for k independent tracks, if `trackID` is provided
`Gamma`	array of transition probability matrices of dimension c(M,M,L). Here we use the definition $\Pr(S_t = j \mid S_{t-1} = i) = \gamma_{ij}^{(t)}$ such that the transition probabilities between time point $t-1$ and t are an element of $\Gamma^{(t)}$.
`allprobs`	matrix of state-dependent probabilities/ density values of dimension c(n, N) which will automatically be converted to the appropriate dimension.

sizes	state aggregate sizes that are used for the approximation of the semi-Markov chain.
tod	(Integer valued) variable for cycle indexing in 1, ..., L, mapping the data index to a generalised time of day (length n). For half-hourly data L = 48. It could, however, also be day of year for daily data and L = 365.

Value

log-likelihood for given data and parameters

Examples

```
## generating data from homogeneous 2-state HSMM
mu = c(0, 6)
beta = matrix(c(log(4),log(6),-0.2,0.2,-0.1,0.4), nrow=2)
# time varying mean dwell time
Lambda = exp(cbind(1, cosinor(1:24))%*%t(beta))
omega = matrix(c(0,1,1,0), nrow = 2, byrow = TRUE)
# simulation
# for a 2-state HSMM the embedded chain always alternates between 1 and 2
s = rep(1:2, 100)
C = x = numeric(0)
tod = rep(1:24, 50) # time of day variable
time = 1
for(t in 1:100){
  dt = rpois(1, Lambda[tod[time], s[t]])+1 # dwell time depending on time of day
  time = time + dt
  C = c(C, rep(s[t], dt))
  x = c(x, rnorm(dt, mu[s[t]], 1.5)) # fixed sd 2 for both states
}
tod = tod[1:length(x)]

## negative log likelihood function
mllk = function(theta.star, x, sizes, tod){
  # parameter transformations for unconstraint optimization
  omega = matrix(c(0,1,1,0), nrow = 2, byrow = TRUE) # omega fixed (2-states)
  mu = theta.star[1:2]
  sigma = exp(theta.star[3:4])
  beta = matrix(theta.star[5:10], nrow=2)
  # time varying mean dwell time
  Lambda = exp(cbind(1, cosinor(1:24))%*%t(beta))
  dm = list()
  for(j in 1:2){
    dm[[j]] = sapply(1:sizes[j]-1, dpois, lambda = Lambda[,j])
  }
  Gamma = tpm_phsmm2(omega, dm)
  delta = stationary_p(Gamma, tod[1])
  # calculate all state-dependent probabilities
  allprobs = matrix(1, length(x), 2)
  for(j in 1:2){ allprobs[,j] = dnorm(x, mu[j], sigma[j]) }
  # return negative for minimization
  -forward_sp(delta, Gamma, allprobs, sizes, tod)
```

```
}

## fitting an HSMM to the data
theta.star = c(1, 4, log(2), log(2), # state-dependent parameters
                log(4), log(6), rep(0,4)) # state process parameters dm
# mod = nlm(mllk, theta.star, x = x, sizes = c(10, 15), tod = tod, stepmax = 5)
```

---

gamma2 *Reparametrised gamma distribution*

Description

Density, distribution function, quantile function and random generation for the gamma distribution reparametrised in terms of mean and standard deviation.

Usage

```
dgamma2(x, mean = 1, sd = 1, log = FALSE)

pgamma2(q, mean = 1, sd = 1, lower.tail = TRUE, log.p = FALSE)

qgamma2(p, mean = 1, sd = 1, lower.tail = TRUE, log.p = FALSE)

rgamma2(n, mean = 1, sd = 1)
```

Arguments

x, q	vector of quantiles
mean	mean parameter, must be positive scalar.
sd	standard deviation parameter, must be positive scalar.
log, log.p	logical; if TRUE, probabilities/ densities p are returned as $\log(p)$.
lower.tail	logical; if TRUE, probabilities are $P[X <= x]$, otherwise, $P[X > x]$.
p	vector of probabilities
n	number of observations. If length(n) > 1, the length is taken to be the number required.

Details

This implementation allows for automatic differentiation with RTMB.

Value

dgamma2 gives the density, pgamma2 gives the distribution function, qgamma2 gives the quantile function, and rgamma2 generates random deviates.

Examples

```
x = rgamma2(1)
d = dgamma2(x)
p = pgamma2(x)
q = qgamma2(p)
```

`gdeterminant` *Computes generalised determinant*

Description

Computes generalised determinant

Usage

```
gdeterminant(x, eps = NULL, log = TRUE)
```

Arguments

`x`	symmetric matrix
`eps`	eigenvalues smaller than this will be treated as zero
`log`	logical. If `TRUE`, the log-determinant is returned. If `FALSE`, the determinant is returned.

Value

generalised log-determinant of `x`

`generator` *Build the generator matrix of a continuous-time Markov chain*

Description

This function builds the **infinitesimal generator matrix** for a **continuous-time Markov chain** from an unconstrained parameter vector.

Usage

```
generator(
  beta,
  Z = NULL,
  Eta = NULL,
  byrow = FALSE,
  report = TRUE,
  param = NULL
)
```

Arguments

beta	parameters; either • a vector of length nStates * (nStates-1), or • a matrix of dimension c(nStates * (nStates-1), p+1) if design matrix Z is also provided.
Z	optional covariate design matrix with or without intercept column, i.e. of dimension c(nObs, p) or c(nObs, p+1). If provided, beta needs to be a matrix of dimension c(nStates * (nStates-1), p+1).
Eta	optional pre-calculated matrix of linear predictors of dimension c(nObs, nStates * (nStates-1)). If provided, Z and beta will be ignored.
byrow	logical indicating if the generator matrix should be filled by row
report	logical, indicating whether the generator matrix Q should be reported from the fitted model. Defaults to TRUE, but only works if when automatic differentiation with RTMB is used.
param	depricated, please use argument beta instead.

Details

Off-diagonal entries are calculated as $\exp(\beta_i)$ to ensure positivity. The diagonal entries are then set to the negative row sums, which is required for generator matrices.

If a design matrix Z or a matrix of linear predictors Eta is provided, the function will automatically call generator_g to build the generator matrix based on the design matrix and coefficient matrix. In that case, the argument beta needs to be a matrix of coefficients of dimension c(nStates * (nStates-1), p+1), where the first column contains the intercepts.

Value

infinitesimal generator matrix of dimension c(nStates, nStates) or array of such matrices of dimension c(nStates, nStates, nObs) if Z or Eta is provided.

See Also

Other transition probability matrix functions: generator_g(), tpm(), tpm_ct(), tpm_emb(), tpm_emb_g(), tpm_g(), tpm_g2(), tpm_p()

Examples

```
# 2 states: 2 free off-diagonal elements
generator(rep(-1, 2))
# 3 states: 6 free off-diagonal elements
generator(rep(-2, 6))
```

generator_g *Build generator matrices of a continuous-time Markov chain*

Description

This function builds **infinitesimal generator matrices** for a **continuous-time Markov chain** based on a design matrix and coefficient matrix.

Usage

```
generator_g(Z, beta, Eta = NULL, byrow = FALSE, report = TRUE)
```

Arguments

Z	Covariate design matrix with or without intercept column, i.e. of dimension c(nObs, p) or c(nObs, p+1). If not provided, intercept column is added automatically.
beta	Matrix of coefficients for the off-diagonal elements of the generator matrix of dimension c(nStates * (nStates-1), p+1). First columns contains the intercepts.
Eta	optional pre-calculated matrix of linear predictors of dimension c(nObs, nStates * (nStates-1)). If provided, no Z and beta are necessary and will be ignored.
byrow	logical indicating if the generator matrices should be filled by row
report	logical, indicating whether the generator matrices Q should be reported from the fitted model. Defaults to TRUE, but only works if when automatic differentiation with RTMB is used.

Details

Off-diagonal entries are calculated as $\exp(Z\beta_i)$ to ensure positivity. The diagonal entries are then set to the negative row sums, which is required for generator matrices.

Value

array of infinitesimal generator matrices of dimension c(nStates, nStates, nObs)

See Also

Other transition probability matrix functions: generator(), tpm(), tpm_ct(), tpm_emb(), tpm_emb_g(), tpm_g(), tpm_g2(), tpm_p()

Examples

```
# 2 states: 2 free off-diagonal elements
generator(rep(-1, 2))
# 3 states: 6 free off-diagonal elements
generator(rep(-2, 6))
```

LaMaColors *Generate a colour-blind-friendly palette*

Description

Generate a colour-blind-friendly palette

Usage

```
LaMaColors(ncolors)
```

Arguments

ncolors	Number of colours to return; at most 10

Value

A colour palette of length ncolors

Examples

```
cb <- LaMaColors(3)
```

logLik.LaMaModel *Extract log-likelihood from LaMaModel object*

Description

Extract log-likelihood from LaMaModel object

Usage

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

Arguments

object	A model fitted using RTMB and obtained via report(obj) of class "LaMaModel"
...	Additional arguments (not used)

Value

An object of class "logLik"

logLik.qremlModel *Extract log-likelihood from qremlModel object*

Description

Extract log-likelihood from qremlModel object

Usage

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

Arguments

object	A fitted model of class "qremlModel"
...	Additional arguments (not used)

Value

An object of class "logLik"

make_matrices *Build the design and the penalty matrix for models involving penalised splines based on a formula and a data set*

Description

Build the design and the penalty matrix for models involving penalised splines based on a formula and a data set

Usage

```
make_matrices(formula, data, knots = NULL)
```

Arguments

formula	formula as used in mgcv. Formulas can be right-side only, or contain a response variable, which is just extracted for naming. Can also be a list of formulas, which are then processed separately. In that case, both a named list of right-side only formulas or a list of formulas with response variables can be provided.
data	data frame containing all the variables on the right side of the formula(s)

knots — optional list containing user specified knot values for each covariate to be used for basis construction. For most bases the user simply supplies the knots to be used, which must match up with the k value supplied (note that the number of knots is not always just k). See mgcv documentation for more details.
If formula is a list, this needs to be a named (based on the response variables) list over such lists.

Value

a list of class LaMa_matrices containing:

- Z — design matrix (or list of such matrices if formula is a list))
- S — list of penalty matrices (with names based on the response terms of the formulas as well as the smooth terms and covariates). For tensorproduct smooths, corresponding entries are themselves lists, containing the d marginal penalty matrices if d is the dimension of the tensor product)
- pardim — list of parameter dimensions (fixed and penalised separately) for each formula, for ease of setting up initial parameters
- coef — list of coefficient vectors filled with zeros of the correct length for each formula, for ease of setting up initial parameters
- data — the data frame used for the model(s)
- gam — unfitted mgcv::gam object used for construction of Z and S (or list of such objects if formula is a list)
- gam0 — fitted mgcv::gam which is used internally to create prediction design matrices (or list of such objects if formula is a list)
- knots — knot list used in the basis construction (or named list over such lists if formula is a list)

See Also

predict.LaMa_matrices for prediction design matrix construction based on the model matrices object created by this function.

Examples

```
data = data.frame(x = runif(100),
                  y = runif(100),
                  g = factor(rep(1:10, each = 10)))

# unvariate thin plate regression spline
modmat = make_matrices(~ s(x), data)
# univariate P-spline
modmat = make_matrices(~ s(x, bs = "ps"), data)
# adding random intercept
modmat = make_matrices(~ s(g, bs = "re") + s(x, bs = "ps"), data)
# tensorproduct of x and y
modmat = make_matrices(~ s(x) + s(y) + ti(x,y), data)
# multiple formulas at once
modmat = make_matrices(list(mu ~ s(x) + y, sigma ~ s(g, bs = "re")), data = data)
```

`make_matrices_dens` *Build a standardised P-Spline design matrix and the associated P-Spline penalty matrix*

Description

This function builds the B-spline design matrix for a given data vector. Importantly, the B-spline basis functions are normalised such that the integral of each basis function is 1, hence this basis can be used for spline-based density estimation, when the basis functions are weighted by non-negative weights summing to one.

Usage

```
make_matrices_dens(
  x,
  k,
  type = "real",
  degree = 3,
  knots = NULL,
  diff_order = 2,
  pow = 0.5,
  npoints = 10000
)
```

Arguments

`x`	data vector
`k`	number of basis functions
`type`	type of the data, either `"real"` for data on the reals, `"positive"` for data on the positive reals or `"circular"` for circular data like angles.
`degree`	degree of the B-spline basis functions, defaults to cubic B-splines
`knots`	optional vector of knots (including the boundary knots) to be used for basis construction. If not provided, the knots are placed equidistantly for `"real"` and `"circular"` and using polynomial spacing for `"positive"`. For `"real"` and `"positive"` `k - degree + 1` knots are needed, for `"circular"` `k + 1` knots are needed. # @param quantile logical, if `TRUE` use quantile-based knot spacing (instead of equidistant or polynomial)
`diff_order`	order of differencing used for the P-Spline penalty matrix for each data stream. Defaults to second-order differences.
`pow`	power for polynomial knot spacing
`npoints`	number of points used in the numerical integration for normalizing the B-spline basis functions Such non-equidistant knot spacing is only used for `type = "positive"`.

Value

list containing the design matrix Z, the penalty matrix S, the prediction design matrix Z_predict, the prediction grid xseq, and details for the basis expansion.

Examples

```
set.seed(1)
# real-valued
x <- rnorm(100)
modmat <- make_matrices_dens(x, k = 20)
# positive-continuouos
x <- rgamma2(100, mean = 5, sd = 2)
modmat <- make_matrices_dens(x, k = 20, type = "positive")
# circular
x <- rvm(100, mu = 0, kappa = 2)
modmat <- make_matrices_dens(x, k = 20, type = "circular")
# bounded in an interval
x <- rbeta(100, 1, 2)
modmat <- make_matrices_dens(x, k = 20)
```

---

make_matrices_old *Build the design and the penalty matrix for models involving penalised splines based on a formula and a data set*

Description

Build the design and the penalty matrix for models involving penalised splines based on a formula and a data set

Usage

```
make_matrices_old(formula, data, knots = NULL)
```

Arguments

formula	right side of a formula as used in mgcv
data	data frame containing the variables in the formula
knots	optional list containing user specified knot values to be used for basis construction For most bases the user simply supplies the knots to be used, which must match up with the k value supplied (note that the number of knots is not always just k). See mgcv documentation for more details.

Value

a list containing the design matrix Z, a (potentially nested) list of penalty matrices S, the formula, the data, the knots, and the original mod object returned by mgcv. Note that for tensorproduct smooths, the corresponding list entry is itself a list, containing the d marginal penalty matrices if d is the dimension of the tensor product.

Examples

```
data = data.frame(x = runif(100),
                  y = runif(100),
                  g = factor(rep(1:10, each = 10)))

# unvariate thin plate regression spline
modmat = make_matrices(~ s(x), data)
# univariate P-spline
modmat = make_matrices(~ s(x, bs = "ps"), data)
# adding random intercept
modmat = make_matrices(~ s(g, bs = "re") + s(x, bs = "ps"), data)
# tensorproduct of x and y
modmat = make_matrices(~ s(x) + s(y) + ti(x,y), data)
```

MCreport *Sample parameters from approximate Gaussian posterior distribution*

Description

Efficient Monte Carlo sampling of parameters (and REPORTed quantities) from the approximate posterior of an (R)TMB model. See sdreport for details on posterior variance-covariance in random effects models.

Usage

```
MCreport(
  obj,
  nSamples = 1000,
  include_random_pars = TRUE,
  report = FALSE,
  Q = NULL,
  ...
)
```

Arguments

obj	Optimised RTMB object generated by MakeADFun
nSamples	Number of samples to draw
include_random_pars	Logical; Should random parameters be included in the output?
report	Logical; Should REPORTed quantities be sampled as well? Defaults to FALSE because this may be slow depending on your model.
Q	Optional precalculated sparse precision matrix returned by sdreport(..., getJointPrecision = TRUE)$jointPrecision. If not provided, computed internally using sdreport. Only used for models with random effects.
...	For internal use only

Details

Caution: This has nothing to do with Bayesian posterior sampling. It is simple Monte Carlo sampling from a Gaussian distribution with mean at the MLE and covariance given by the inverse of the Hessian (for fixed effects models) or the joint precision matrix (for random effects models). This is a common approach to get an approximate idea of parameter uncertainty around the MLE, but it relies on large-sample asymptotics.

Sampling random effects from their posterior as compared to calculating marginal standard deviations like `sdreport` is particularly useful for multidimensional random effects (e.g. for locations x and y) where pointwise confidence intervals (e.g. along a path) based on standard deviations are not possible.

Value

A list structured like the original parameter list used in the MakeADFun call (potentially including additional `REPORT`ed quantities). Each entry is a list with `nSamples` entries.

Examples

```
step <- trex$step[1:1000] # subsetting trex data
N <- 2                    # 2 states

# custom likelihood
nll <- function(par) {
  getAll(par)
  Gamma <- tpm(eta)
  delta <- stationary(Gamma)
  mu <- exp(log_mu); REPORT(mu)
  sigma <- exp(log_sigma); REPORT(sigma)
  allprobs <- matrix(1, length(step), N)
  for(j in 1:N) allprobs[,j] <- dgamma2(step, mu[j], sigma[j])
  -forward(delta, Gamma, allprobs)
}

# initial parameters in named list
par0 <- list(eta = rep(-2,2),
             log_mu = log(c(0.3, 1)),
             log_sigma = log(c(0.2, 0.7)))

# constructing AD object
obj <- MakeADFun(nll, par0, silent = TRUE)

# optimising
opt <- nlminb(obj$par, obj$fn, obj$gr)

# sampling from distribution of the MLE
samples <- MCreport(obj, nSamples = 10, report = TRUE)
```

minmax *AD-compatible minimum and maximum functions*

Description

These functions compute the parallel minimum/ maximum of two vector-valued inputs and are compatible with automatic differentiation using RTMB.

Usage

```
min2(x, y)

max2(x, y)
```

Arguments

x	first vector
y	second vector

Value

min2 returns the parallel minimum and max2 the parallel maximum of x and y

Examples

```
x <- c(1, 4, 8, 2)
y <- c(2, 5, 3, 7)
min2(x, y)
max2(x, y)
```

minmax0_smooth *Smooth approximations to max(x, 0) and min(x, 0)*

Description

Smooth approximations to max(x, 0) and min(x, 0)

Usage

```
max0_smooth(x, rho = 20)

min0_smooth(x, rho = 20)
```

Arguments

x	a vector of values
rho	smoothing parameter, larger values lead to closer approximation

Value

the approximate maximum or minimum of x and 0

Examples

```
x <- seq(-1, 1, by = 0.1)
min0_smooth(x)
max0_smooth(x)
```

---

nessi *Loch Ness Monster Acceleration Data*

Description

A small group of researchers managed to put an accelerometer on the Loch Ness Monster and collected data for a few days. Now we have a data set of the overall dynamic body acceleration (ODBA) of the creature.

Usage

```
nessi
```

Format

A data frame with 5.000 rows and 3 variables:

ODBA overall dynamci body acceleration

logODBA logarithm of overall dynamic body acceleration

state hidden state variable

Source

Generated for example purposes.

penalty *Computes penalty based on quadratic form*

Description

This function computes quadratic penalties of the form

$$0.5 \sum_i \lambda_i b_i^T S_i b_i,$$

with smoothing parameters λ_i, coefficient vectors b_i, and fixed penalty matrices S_i.

It is intended to be used inside the **penalised negative log-likelihood function** when fitting models with penalised splines or simple random effects via **quasi restricted maximum likelihood** (qREML) with the `qreml` function. For `qreml` to work, the likelihood function needs to be compatible with the `RTMB` R package to enable automatic differentiation.

Usage

```
penalty(re_coef, S, lambda)
```

Arguments

`re_coef`	coefficient vector/ matrix or list of coefficient vectors/ matrices Each list entry corresponds to a different smooth/ random effect with its own associated penalty matrix in `S`. When several smooths/ random effects of the same kind are present, it is convenient to pass them as a matrix, where each row corresponds to one smooth/ random effect. This way all rows can use the same penalty matrix.
`S`	fixed penalty matrix or list of penalty matrices matching the structure of `re_coef` and also the dimension of the individuals smooths/ random effects
`lambda`	penalty strength parameter vector that has a length corresponding to the **total number** of random effects/ spline coefficients in `re_coef` E.g. if `re_coef` contains one vector and one matrix with 4 rows, then `lambda` needs to be of length 5.

Details

Caution: The formatting of `re_coef` needs to match the structure of the parameter list in your penalised negative log-likelihood function, i.e. you cannot have two random effect vectors of different names (different list elements in the parameter list), combine them into a matrix inside your likelihood and pass the matrix to `penalty`. If these are seperate random effects, each with its own name, they need to be passed as a list to `penalty`. Moreover, the ordering of `re_coef` needs to match the character vector `random` specified in `qreml`.

Value

returns the penalty value and reports to `qreml`.

References

Koslik, J. O. (2024). Efficient smoothness selection for nonparametric Markov-switching models via quasi restricted maximum likelihood. arXiv preprint arXiv:2411.11498.

See Also

qreml for the **qREML** algorithm

Examples

```
# Example with a single random effect
re = rep(0, 5)
S = diag(5)
lambda = 1
penalty(re, S, lambda)

# Example with two random effects,
# where one element contains two random effects of similar structure
re = list(matrix(0, 2, 5), rep(0, 4))
S = list(diag(5), diag(4))
lambda = c(1,1,2) # length = total number of random effects
penalty(re, S, lambda)

# Full model-fitting example

data = trex[1:1000,] # subset

# initial parameter list
par = list(logmu = log(c(0.3, 1)), # step mean
           logsigma = log(c(0.2, 0.7)), # step sd
           beta0 = c(-2,-2), # state process intercept
           betaspline = matrix(rep(0, 18), nrow = 2)) # state process spline coefs

# data object with initial penalty strength lambda
dat = list(step = data$step, # step length
           tod = data$tod, # time of day covariate
           N = 2, # number of states
           lambda = rep(10,2)) # initial penalty strength

# building model matrices
modmat = make_matrices(~ s(tod, bs = "cp"),
                       data = data.frame(tod = 1:24),
                       knots = list(tod = c(0,24))) # wrapping points
dat$Z = modmat$Z # spline design matrix
dat$S = modmat$S # penalty matrix

# penalised negative log-likelihood function
pnll = function(par) {
  getAll(par, dat) # makes everything contained available without $
  Gamma = tpm_g(Z, cbind(beta0, betaspline)) # transition probabilities
  delta = stationary_p(Gamma, t = 1) # initial distribution
  mu = exp(logmu) # step mean
```

```
  sigma = exp(logsigma) # step sd
  # calculating all state-dependent densities
  allprobs = matrix(1, nrow = length(step), ncol = N)
  ind = which(!is.na(step)) # only for non-NA obs.
  for(j in 1:N) allprobs[ind,j] = dgamma2(step[ind],mu[j],sigma[j])
  -forward_g(delta, Gamma[,,tod], allprobs) +
      penalty(betaspline, S, lambda) # this does all the penalization work
}

# model fitting
mod = qreml(pnll, par, dat, random = "betaspline")
```

---

penalty_uni *Penalty approximation of unimodality constraints for univariates smooths*

Description

Penalty approximation of unimodality constraints for univariates smooths

Usage

```
penalty_uni(coef, m, kappa = 1000, concave = TRUE, rho = 20)
```

Arguments

coef	coefficient vector of matrix on which to apply the unimodality penalty
m	vector of indices for the position of the coefficient mode. If coef is a vector, must be of length 1. Otherwise, must be of length equal to nrow(coef)
kappa	global scaling factor for the penalty
concave	logical; if TRUE (default), the penalty enforces increasing until the mode then decreasing. If the coefficients should decrease until the mode, then increase, set concave = FALSE.
rho	control parameter for smooth approximation to min(x, 0) used internally. For large values, gets closer to true minimum function but less stable.

Value

a numeric value of the penalty for the given coefficients

Examples

```
## coefficient vector
coef <- c(1, 2, 3, 2, 1)
# mode at position 3
penalty_uni(coef, m = 3) # basically zero
#' # mode at position 2
```

```
penalty_uni(coef, m = 2) # large positive penalty

## coefficient matrix
coef <- rbind(coef, coef)
m <- c(1, 4)
penalty_uni(coef, m)
```

`penalty2` *Computes generalised quadratic-form penalties*

Description

This function computes a quadratic penalty of the form

$$0.5 \sum_i \lambda_i b^T S_i b,$$

with smoothing parameters λ_i, coefficient vector b, and fixed penalty matrices S_i. This generalises the `penalty` by allowing subsets of the coefficient vector b to be penalised multiple times with different smoothing parameters, which is necessary for **tensor products**, **functional random effects** or **adaptive smoothing**.

It is intended to be used inside the **penalised negative log-likelihood function** when fitting models with penalised splines or simple random effects via **quasi restricted maximum likelihood** (qREML) with the `qreml` function. For `qreml` to work, the likelihood function needs to be compatible with the `RTMB` R package to enable automatic differentiation.

Usage

```
penalty2(re_coef, S, lambda)
```

Arguments

`re_coef`	list of coefficient vectors/ matrices Each list entry corresponds to a different smooth/ random effect with its own associated penalty matrix or penalty-matrix list in `S`. When several smooths/ random effects of the same kind are present, it is convenient to pass them as a matrix, where each row corresponds to one smooth/ random effect. This way all rows can use the same penalty matrix.
`S`	list of fixed penalty matrices matching the structure of `re_coef`. This means if `re_coef` is of length 3, then `S` needs to be a list of length 3. Each entry needs to be either a penalty matrix, matching the dimension of the corresponding entry in `re_coef`, or a list with multiple penalty matrices for tensor products.
`lambda`	penalty strength parameter vector that has a length corresponding to the provided `re_coef` and `S`. Specifically, for entries with one penalty matrix, `nrow(re_coef[[i]])` parameters are needed. For entries with k penalty matrices, k * `nrow(re_coef[[i]])` parameters are needed.

E.g. if `re_coef[[1]]` is a vector and `re_coef[[2]]` a matrix with 4 rows, `S[[1]]` is a list of length 2 and `S[[2]]` is a matrix, then `lambda` needs to be of length 1 * 2 + 4 = 6.

Details

Caution: The formatting of `re_coef` needs to match the structure of the parameter list in your penalised negative log-likelihood function, i.e. you cannot have two random effect vectors of different names (different list elements in the parameter list), combine them into a matrix inside your likelihood and pass the matrix to `penalty`. If these are seperate random effects, each with its own name, they need to be passed as a list to `penalty`. Moreover, the ordering of `re_coef` needs to match the character vector `random` specified in `qreml`.

Value

returns the penalty value and reports to `qreml`.

See Also

`qreml` for the **qREML** algorithm

Examples

```
# Example with a single random effect
re = rep(0, 5)
S = diag(5)
lambda = 1
penalty(re, S, lambda)

# Example with two random effects,
# where one element contains two random effects of similar structure
re = list(matrix(0, 2, 5), rep(0, 4))
S = list(diag(5), diag(4))
lambda = c(1,1,2) # length = total number of random effects
penalty(re, S, lambda)

# Full model-fitting example

data = trex[1:1000,] # subset

# initial parameter list
par = list(logmu = log(c(0.3, 1)), # step mean
           logsigma = log(c(0.2, 0.7)), # step sd
           beta0 = c(-2,-2), # state process intercept
           betaspline = matrix(rep(0, 18), nrow = 2)) # state process spline coefs

# data object with initial penalty strength lambda
dat = list(step = data$step, # step length
           tod = data$tod, # time of day covariate
           N = 2, # number of states
           lambda = rep(10,2)) # initial penalty strength
```

```
# building model matrices
modmat = make_matrices(~ s(tod, bs = "cp"),
                       data = data.frame(tod = 1:24),
                       knots = list(tod = c(0,24))) # wrapping points
dat$Z = modmat$Z # spline design matrix
dat$S = modmat$S # penalty matrix

# penalised negative log-likelihood function
pnll = function(par) {
  getAll(par, dat) # makes everything contained available without $
  Gamma = tpm_g(Z, cbind(beta0, betaspline)) # transition probabilities
  delta = stationary_p(Gamma, t = 1) # initial distribution
  mu = exp(logmu) # step mean
  sigma = exp(logsigma) # step sd
  # calculating all state-dependent densities
  allprobs = matrix(1, nrow = length(step), ncol = N)
  ind = which(!is.na(step)) # only for non-NA obs.
  for(j in 1:N) allprobs[ind,j] = dgamma2(step[ind],mu[j],sigma[j])
  -forward_g(delta, Gamma[,,tod], allprobs) +
      penalty(betaspline, S, lambda) # this does all the penalisation work
}

# model fitting
mod = qreml(pnll, par, dat, random = "betaspline")
```

`plot.LaMaResiduals` *Plot pseudo-residuals*

Description

Plot pseudo-residuals computed by `pseudo_res`.

Usage

```
## S3 method for class 'LaMaResiduals'
plot(
  x,
  col = "darkblue",
  lwd = 1.5,
  main = NULL,
  breaks = "Sturges",
  axis.lab = list(qq = c("Theoretical quantiles", "Sample quantiles"), hist =
    c("Pseudo-residuals", "Density"), acf = c("Lag", "ACF")),
  ...
)
```

Arguments

x	pseudo-residuals as returned by pseudo_res
col	character, color for the QQ-line (and density curve if histogram = TRUE)
lwd	numeric, line width for the QQ-line (and density curve if histogram = TRUE)
main	optional character vector of main titles for the plots of length 2 (or 3 if histogram = TRUE)
breaks	breaks argument passed to hist
axis.lab	labels used for the x and y axis of each plot (named list)
...	currently ignored. For method consistency

Value

NULL, plots the pseudo-residuals in a 3-panel layout

Examples

```
## pseudo-residuals for the trex data
step = trex$step[1:1000]
angle = trex$angle[1:1000]

nll = function(par){
  getAll(par)
  Gamma = tpm(logitGamma)
  delta = stationary(Gamma)
  mu = exp(logMu); REPORT(mu)
  sigma = exp(logSigma); REPORT(sigma)
  kappa = exp(logKappa); REPORT(kappa)
  allprobs = matrix(1, length(step), 2)
  ind = which(!is.na(step) & !is.na(angle))
  for(j in 1:2) {
    allprobs[ind,j] = dgamma2(step[ind], mu[j], sigma[j]) *
         dvm(angle[ind], 0, kappa[j])
  }
  -forward(delta, Gamma, allprobs)
}

par = list(logitGamma = c(-2,-2),
           logMu = log(c(0.3, 2.5)),
           logSigma = log(c(0.3, 0.5)),
           logKappa = log(c(0.2, 1)))

obj = MakeADFun(nll, par, silent = TRUE)
opt = nlminb(obj$par, obj$fn, obj$gr)

mod = report(obj)

pres_step = pseudo_res(step,      # observations
                       "gamma2",  # family that is used
                       list(mean = mod$mu, sd = mod$sigma), # the family's parameters
```

```
                        mod = mod) # model object
pres_angle = pseudo_res(angle,
                        "vm",
                        list(mu = 0, kappa = mod$kappa),
                        mod = mod)

# separate plots
plot(pres_step)
plot(pres_angle)

# together
par(mfrow = c(2,3))
plot(pres_step, main = c("", "Step Length", ""),
     axis.lab = list(hist = c("Step residuals", "Density")))
plot(pres_angle, main = c("", "Turning Angle", ""),
     axis.lab = list(hist = c("Angle residuals", "Density")))
```

`pred_matrix` *Build the prediction design matrix based on new data and model_matrices object created by* `make_matrices`

Description

Build the prediction design matrix based on new data and model_matrices object created by `make_matrices`

Usage

```
pred_matrix(model_matrices, newdata, what = NULL, exclude = NULL)
```

Arguments

`model_matrices`	model_matrices object as returned from `make_matrices`
`newdata`	data frame containing the variables in the formula and new data for which to evaluate the basis
`what`	optional character string specifying which formula to use for prediction, if `object` contains multiple formulas. If `NULL`, the first formula is used.
`exclude`	optional vector of terms to set to zero in the predicted design matrix. Useful for predicting main effects only when e.g. `sd(...,bs="re")` terms are present. See `mgcv::predict.gam` for more details.

Value

prediction design matrix for `newdata` with the same basis as used for `model_matrices`

Examples

```
# single formula
modmat = make_matrices(~ s(x), data.frame(x = 1:10))
Z_p = pred_matrix(modmat, data.frame(x = 1:10 - 0.5))
# with multiple formulas
modmat = make_matrices(list(mu ~ s(x), sigma ~ s(x, bs = "ps")), data = data.frame(x = 1:10))
Z_p = pred_matrix(modmat, data.frame(x = 1:10 - 0.5), what = "mu")
# nested formula list
form = list(stream1 = list(mu ~ s(x), sigma ~ s(x, bs = "ps")))
modmat = make_matrices(form, data = data.frame(x = 1:10))
Z_p = pred_matrix(modmat, data.frame(x = 1:10 - 0.5), what = c("stream1", "mu"))
```

---

predict.LaMa_matrices *Build the prediction design matrix based on new data and model_matrices object created by* make_matrices

---

Description

Build the prediction design matrix based on new data and model_matrices object created by make_matrices

Usage

```
## S3 method for class 'LaMa_matrices'
predict(object, newdata, what = NULL, ...)
```

Arguments

object	model matrices object as returned from make_matrices
newdata	data frame containing the variables in the formula and new data for which to evaluate the basis
what	optional character string specifying which formula to use for prediction if object contains multiple formulas.
...	for method consistency only

Value

prediction design matrix for newdata with the same basis as used for model_matrices

See Also

make_matrices for creating objects of class LaMa_matrices which can be used for prediction by this function.

Examples

```
# single formula
modmat = make_matrices(~ s(x), data.frame(x = 1:10))
Z_p = predict(modmat, data.frame(x = 1:10 - 0.5))
# with multiple formulas
modmat = make_matrices(list(mu ~ s(x), sigma ~ s(x, bs = "ps")), data = data.frame(x = 1:10))
Z_p = predict(modmat, data.frame(x = 1:10 - 0.5), what = "mu")
# nested formula list
form = list(stream1 = list(mu ~ s(x), sigma ~ s(x, bs = "ps")))
modmat = make_matrices(form, data = data.frame(x = 1:10))
Z_p = predict(modmat, data.frame(x = 1:10 - 0.5), what = c("stream1", "mu"))
```

---

process_hid_formulas *Process and standardise formulas for the state process of hidden Markov models*

Description

Process and standardise formulas for the state process of hidden Markov models

Usage

```
process_hid_formulas(formulas, nStates, ref = NULL)
```

Arguments

formulas	formulas for the transition process of a hidden Markov model, either as a single formula, a list of formulas, or a matrix.
nStates	number of states of the Markov chain
ref	optional vector of reference categories for each state, defaults to 1:nStates. If provided, must be of length nStates and contain valid state indices. If a formula matrix is provided, this cannot be specified because reference categries are specified by one "." entry in each row.

Value

named list of formulas of length nStates * (nStates - 1), where each formula corresponds to a transition from state i to state j, excluding transitions from i to ref[i].

Examples

```
# single formula for all non-reference category elements
formulas = process_hid_formulas(~ s(x), nStates = 3)
# now a list of length 6 with names tr.ij, not including reference categories

# different reference categories
formulas = process_hid_formulas(~ s(x), nStates = 3, ref = c(1,1,1))
```

```
# different formulas for different entries (and only for 2 of 6)
formulas = list(tr.12 ~ s(x), tr.23 ~ s(y))
formulas = process_hid_formulas(formulas, nStates = 3, ref = c(1,1,1))
# also a list of length 6, remaining entries filled with tr.ij ~ 1

# matrix input with reference categories
formulas = matrix(c(".", "~ s(x)", "~ s(y)",
                    "~ g", ".", "~ I(x^2)",
                    "~ y", "~ 1", "."),
                    nrow = 3, byrow = TRUE)
# dots define reference categories
formulas = process_hid_formulas(formulas, nStates = 3)
```

---

pseudo_res *Calculate pseudo-residuals*

Description

For HMMs, pseudo-residuals are used to assess the overall goodness-of-fit of the model. These are based on the cumulative distribution function (CDF)

$$F_{X_t}(x_t) = F(x_t \mid x_1, \ldots, x_{t-1})$$

and can be used to quantify whether an observation is extreme relative to its model-implied distribution.

This function calculates such residuals via the probability integral transform, based on the local state probabilities obtained by `stateprobs` or `stateprobs_g` and the respective parametric family.

Usage

```
pseudo_res(
  obs,
  dist,
  par,
  stateprobs = NULL,
  mod = NULL,
  normal = TRUE,
  discrete = NULL,
  randomise = TRUE
)
```

Arguments

`obs`	vector of continuous-valued observations (of length `nObs`)
`dist`	character string specifying which parametric CDF to use (e.g., `"norm"` for normal or `"pois"` for Poisson) or CDF function to evaluate directly. If a discrete CDF function is passed, the `discrete` argument needs to be set to `TRUE` because this cannot determined automatically.

par	named parameter list for the parametric CDF Names need to correspond to the parameter names in the specified distribution (e.g. `list(mean = c(1,2), sd = c(1,1))` for a normal distribution and 2 states). This argument is as flexible as the parametric distribution allows. For example you can have a matrix of parameters with one row for each observation and one column for each state.
stateprobs	matrix of local state probabilities for each observation (of dimension c(nObs, nStates)) as computed by `stateprobs`, `stateprobs_g` or `stateprobs_p`
mod	optional model object containing required quantities When using automatic differentiation either with `RTMB::MakeADFun` or `qreml` and include `forward`, `forward_g` or `forward_p` in your likelihood function, the objects needed for state decoding are automatically reported after model fitting. Hence, you can pass the model object obtained from running `report()` or from `qreml` directly to this function and avoid calculating local state proabilities manually. In this case, a call should look like `pseudo_res(obs, "norm", par, mod = mod)`.
normal	logical, if `TRUE`, returns Gaussian pseudo residuals These will be approximately standard normally distributed if the model is correct.
discrete	logical, if `TRUE`, computes discrete pseudo residuals (which slightly differ in their definition) By default, will be determined using `dist` argument, but only works for standard discrete distributions. When used with a special discrete distribution, set to `TRUE` manually.
randomise	for discrete pseudo residuals only. Logical, if `TRUE`, return randomised pseudo residuals. Recommended for discrete observations.

Details

Pseudo-residuals are based on the probability integral transform. If the cumulative distribution function (CDF) F of a random variable X is known, then $F(X)$ is uniformly distributed on $[0, 1]$. Applying the standard normal quantile function Φ^{-1} then gives a residual that is approximately standard normal under the model.

For discrete-valued observations, the CDF is a step function, so the residual is not uniquely defined. One can compute a "lower" and "upper" pseudo-residual corresponding to $F(X-1)$ and $F(X)$, respectively. If `randomise = TRUE`, a value is drawn uniformly between the lower and upper bounds. In this case, the resulting pseudo-residuals are again approximately standard normal after applying the standard normal quantile function.

Value

vector of pseudo residuals of class `LaMaResiduals` or list containing `lower`, `upper`, and `mean` if discrete residuals are not randomised.

See Also

`plot.LaMaResiduals` for plotting pseudo-residuals.

Examples

```
## continuous-valued observations
obs = rnorm(100)
stateprobs = matrix(0.5, nrow = 100, ncol = 2)
par = list(mean = c(1,2), sd = c(1,1))
pres = pseudo_res(obs, "norm", par, stateprobs)

## discrete-valued observations
obs = rpois(100, lambda = 1)
par = list(lambda = c(1,2))
pres = pseudo_res(obs, "pois", par, stateprobs)

## custom CDF function
obs = rnbinom(100, size = 1, prob = 0.5)
par = list(size = c(0.5, 2), prob = c(0.4, 0.6))
pres = pseudo_res(obs, pnbinom, par, stateprobs,
                  discrete = TRUE)
# if discrete CDF function is passed, 'discrete' needs to be set to TRUE

## no CDF function available, only density (artificial example)
obs = rnorm(100)
par = list(mean = c(1,2), sd = c(1,1))
# construct CDF using numerical integration
cdf = function(x, mean, sd) integrate(dnorm, -Inf, x, mean = mean, sd = sd)$value
pres = pseudo_res(obs, cdf, par, stateprobs)


### Full model fit example ###
step = trex$step[1:200]

nll = function(par){
  getAll(par)
  Gamma = tpm(logitGamma)
  delta = stationary(Gamma)
  mu = exp(logMu); REPORT(mu)
  sigma = exp(logSigma); REPORT(sigma)
  allprobs = matrix(1, length(step), 2)
  ind = which(!is.na(step))
  for(j in 1:2) allprobs[ind,j] = dgamma2(step[ind], mu[j], sigma[j])
  -forward(delta, Gamma, allprobs)
}

par = list(logitGamma = c(-2,-2),
           logMu = log(c(0.3, 2.5)),
           logSigma = log(c(0.3, 0.5)))

obj = MakeADFun(nll, par, silent = TRUE)
opt = nlminb(obj$par, obj$fn, obj$gr)

mod = obj$report()

pres = pseudo_res(step,      # observation sequence
```

```
                  "gamma2",  # parametric family that was used
                  list(mean = mod$mu, sd = mod$sigma), # parameters for that family
                  mod = mod) # model object

plot(pres)
```

`qreml` *Quasi restricted maximum likelihood (qREML) algorithm for models with penalised splines or simple i.i.d. random effects*

Description

This algorithm can be used very flexibly to fit statistical models that involve **penalised splines** or simple **i.i.d. random effects**, i.e. that have penalties of the form

$$0.5 \sum_i \lambda_i b_i^T S_i b_i,$$

with smoothing parameters λ_i, coefficient vectors b_i, and fixed penalty matrices S_i.

The **qREML** algorithm is typically much faster than REML or marginal ML using the full Laplace approximation method, but may be slightly less accurate regarding the estimation of the penalty strength parameters.

Under the hood, `qreml` uses the R package `RTMB` for automatic differentiation in the inner optimisation. The user has to specify the **penalised negative log-likelihood function** `pnll` structured as dictated by `RTMB` and use the `penalty` function to compute the quadratic-form penalty inside the likelihood.

Usage

```
qreml(
  pnll,
  par,
  dat,
  random,
  map = NULL,
  silent = 1,
  psname = "lambda",
  alpha = 0.3,
  smoothing = 1,
  maxiter = 100,
  tol = 1e-04,
  method = "BFGS",
  control = list(),
  conv_crit = "relchange",
  spHess = FALSE,
  joint_unc = FALSE,
  saveall = FALSE
)
```

Arguments

pnll	penalised negative log-likelihood function that is structured as dictated by RTMB and uses the penalty function from LaMa to compute the penalty Needs to be a function of the named list of initial parameters par only.
par	named list of initial parameters The random effects/ spline coefficients can be vectors or matrices, the latter summarising several random effects of the same structure, each one being a row in the matrix.
dat	initial data list that contains the data used in the likelihood function, hyperparameters, and the **initial penalty strength** vector If the initial penalty strength vector is **not** called lambda, the name it has in dat needs to be specified using the psname argument below. Its length needs to match the to the total number of random effects.
random	vector of names of the random effects/ penalised parameters in par **Caution:** The ordering of random needs to match the order of the random effects passed to penalty inside the likelihood function.
map	optional map argument, containing factor vectors to indicate parameter sharing or fixing. Needs to be a named list for a subset of fixed effect parameters or penalty strength parameters. For example, if the model has four penalty strength parameters, map[[psname]] could be factor(c(NA, 1, 1, 2)) to fix the first penalty strength parameter, estimate the second and third jointly, and estimate the fourth separately.
silent	integer silencing level: 0 corresponds to full printing of inner and outer iterations, 1 to printing of outer iterations only, and 2 to no printing.
psname	optional name given to the penalty strength parameter in dat. Defaults to "lambda".
alpha	optional hyperparamater for exponential smoothing of the penalty strengths. For larger values smoother convergence is to be expected but the algorithm may need more iterations.
smoothing	optional scaling factor for the final penalty strength parameters Increasing this beyond one will lead to a smoother final model. Can be an integer or a vector of length equal to the length of the penalty strength parameter.
maxiter	maximum number of iterations in the outer optimisation over the penalty strength parameters.
tol	Convergence tolerance for the penalty strength parameters.
method	optimisation method to be used by optim. Defaults to "BFGS", but might be changed to "L-BFGS-B" for high-dimensional settings.
control	list of control parameters for optim to use in the inner optimisation. Here, optim uses the BFGS method which cannot be changed. We advise against changing the default values of reltol and maxit as this can decrease the accuracy of the Laplace approximation.
conv_crit	character, convergence criterion for the penalty strength parameters. Can be "relchange" (default) or "gradient".

spHess	logical, if TRUE, sparse AD Hessian is used in each outer iteration. If your Hessian is large and sparse (many cross derivatives are 0), this will speed up the computations a lot. If your Hessian is dense, this will slow down the computations slightly and might require significantly more memory.
joint_unc	logical, if TRUE, joint RTMB object is returned allowing for joint uncertainty quantification
saveall	logical, if TRUE, then all model objects from each iteration are saved in the final model object.

Value

model object of class 'qremlModel'. This is a list containing:

...	everything that is reported inside pnll using RTMB::REPORT(). When using forward, tpm_g, etc., this may involve automatically reported objects.
obj	RTMB AD object containing the final conditional model fit
psname	final penalty strength parameter vector
all_psname	list of all penalty strength parameter vectors over the iterations
par	named estimated parameter list in the same structure as the initial par. Note that the name par is not fixed but depends on the original name of your par list.
relist_par	function to convert the estimated parameter vector to the estimated parameter list. This is useful for uncertainty quantification based on sampling from a multivariate normal distribution.
par_vec	estimated parameter vector
llk	unpenalised log-likelihood at the optimum
n_fixpar	number of fixed, i.e. unpenalised, parameters
edf	overall effective number of parameters
all_edf	list of effective number of parameters for each smooth
Hessian_condtional	final Hessian of the conditional penalised fit
obj_joint	if joint_unc = TRUE, joint RTMB object for joint uncertainty quantification in model and penalty parameters.

References

Koslik, J. O. (2024). Efficient smoothness selection for nonparametric Markov-switching models via quasi restricted maximum likelihood. arXiv preprint arXiv:2411.11498.

See Also

penalty and penalty2 to compute the penalty inside the likelihood function

Examples

```
data = trex[1:1000,] # subset

# initial parameter list
par = list(logmu = log(c(0.3, 1)), # step mean
           logsigma = log(c(0.2, 0.7)), # step sd
           beta0 = c(-2,-2), # state process intercept
           betaspline = matrix(rep(0, 18), nrow = 2)) # state process spline coefs

# data object with initial penalty strength lambda
dat = list(step = data$step, # step length
           tod = data$tod, # time of day covariate
           N = 2, # number of states
           lambda = rep(10,2)) # initial penalty strength

# building model matrices
modmat = make_matrices(~ s(tod, bs = "cp"),
                       data = data.frame(tod = 1:24),
                       knots = list(tod = c(0,24))) # wrapping points
dat$Z = modmat$Z # spline design matrix
dat$S = modmat$S # penalty matrix

# penalised negative log-likelihood function
pnll = function(par) {
  getAll(par, dat) # makes everything contained available without $
  Gamma = tpm_g(Z, cbind(beta0, betaspline), ad = TRUE) # transition probabilities
  delta = stationary_p(Gamma, t = 1, ad = TRUE) # initial distribution
  mu = exp(logmu) # step mean
  sigma = exp(logsigma) # step sd
  # calculating all state-dependent densities
  allprobs = matrix(1, nrow = length(step), ncol = N)
  ind = which(!is.na(step)) # only for non-NA obs.
  for(j in 1:N) allprobs[ind,j] = dgamma2(step[ind],mu[j],sigma[j])
  -forward_g(delta, Gamma[,,tod], allprobs) +
      penalty(betaspline, S, lambda) # this does all the penalization work
}

# model fitting
mod = qreml(pnll, par, dat, random = "betaspline", silent = 2)
```

`qreml_old` *Quasi restricted maximum likelihood (qREML) algorithm for models with penalised splines or simple i.i.d. random effects*

Description

This algorithm can be used very flexibly to fit statistical models that involve **penalised splines** or simple **i.i.d. random effects**, i.e. that have penalties of the form

$$0.5 \sum_i \lambda_i b_i^T S_i b_i,$$

with smoothing parameters λ_i, coefficient vectors b_i, and fixed penalty matrices S_i.

The **qREML** algorithm is typically much faster than REML or marginal ML using the full Laplace approximation method, but may be slightly less accurate regarding the estimation of the penalty strength parameters.

Under the hood, `qreml` uses the R package `RTMB` for automatic differentiation in the inner optimisation. The user has to specify the **penalised negative log-likelihood function** `pnll` structured as dictated by `RTMB` and use the `penalty` function to compute the quadratic-form penalty inside the likelihood.

Usage

```
qreml_old(
  pnll,
  par,
  dat,
  random,
  map = NULL,
  psname = "lambda",
  alpha = 0.25,
  smoothing = 1,
  maxiter = 100,
  tol = 1e-04,
  control = list(reltol = 1e-10, maxit = 1000),
  silent = 1,
  joint_unc = TRUE,
  saveall = FALSE
)
```

Arguments

`pnll`	penalised negative log-likelihood function that is structured as dictated by `RTMB` and uses the `penalty` function from `LaMa` to compute the penalty Needs to be a function of the named list of initial parameters `par` only.
`par`	named list of initial parameters The random effects/ spline coefficients can be vectors or matrices, the latter summarising several random effects of the same structure, each one being a row in the matrix.
`dat`	initial data list that contains the data used in the likelihood function, hyperparameters, and the **initial penalty strength** vector If the initial penalty strength vector is **not** called `lambda`, the name it has in `dat` needs to be specified using the `psname` argument below. Its length needs to match the to the total number of random effects.
`random`	vector of names of the random effects/ penalised parameters in `par` **Caution:** The ordering of `random` needs to match the order of the random effects passed to `penalty` inside the likelihood function.
`map`	optional map argument, containing factor vectors to indicate parameter sharing or fixing.

	Needs to be a named list for a subset of fixed effect parameters or penalty strength parameters. For example, if the model has four penalty strength parameters, `map[[psname]]` could be `factor(c(NA, 1, 1, 2))` to fix the first penalty strength parameter, estimate the second and third jointly, and estimate the fourth separately.
`psname`	optional name given to the penalty strength parameter in `dat`. Defaults to `"lambda"`.
`alpha`	optional hyperparamater for exponential smoothing of the penalty strengths. For larger values smoother convergence is to be expected but the algorithm may need more iterations.
`smoothing`	optional scaling factor for the final penalty strength parameters Increasing this beyond one will lead to a smoother final model. Can be an integer or a vector of length equal to the length of the penalty strength parameter.
`maxiter`	maximum number of iterations in the outer optimisation over the penalty strength parameters.
`tol`	Convergence tolerance for the penalty strength parameters.
`control`	list of control parameters for `optim` to use in the inner optimisation. Here, `optim` uses the `BFGS` method which cannot be changed. We advise against changing the default values of `reltol` and `maxit` as this can decrease the accuracy of the Laplace approximation.
`silent`	integer silencing level: 0 corresponds to full printing of inner and outer iterations, 1 to printing of outer iterations only, and 2 to no printing.
`joint_unc`	logical, if `TRUE`, joint `RTMB` object is returned allowing for joint uncertainty quantification
`saveall`	logical, if `TRUE`, then all model objects from each iteration are saved in the final model object. # @param epsilon vector of two values specifying the cycling detection parameters. If the relative change of the new penalty strength to the previous one is larger than `epsilon[1]` but the change to the one before is smaller than `epsilon[2]`, the algorithm will average the two last values to prevent cycling.

Value

model object of class 'qremlModel'. This is a list containing:

`...`	everything that is reported inside `pnll` using `RTMB::REPORT()`. When using `forward`, `tpm_g`, etc., this may involve automatically reported objects.
`obj`	`RTMB` AD object containing the final conditional model fit
`psname`	final penalty strength parameter vector
`all_psname`	list of all penalty strength parameter vectors over the iterations
`par`	named estimated parameter list in the same structure as the initial `par`. Note that the name `par` is not fixed but depends on the original name of your `par` list.
`relist_par`	function to convert the estimated parameter vector to the estimated parameter list. This is useful for uncertainty quantification based on sampling from a multivariate normal distribution.

par_vec	estimated parameter vector
llk	unpenalised log-likelihood at the optimum
n_fixpar	number of fixed, i.e. unpenalised, parameters
edf	overall effective number of parameters
all_edf	list of effective number of parameters for each smooth
Hessian_condtional	final Hessian of the conditional penalised fit
obj_joint	if joint_unc = TRUE, joint RTMB object for joint uncertainty quantification in model and penalty parameters.

References

Koslik, J. O. (2024). Efficient smoothness selection for nonparametric Markov-switching models via quasi restricted maximum likelihood. arXiv preprint arXiv:2411.11498.

See Also

penalty to compute the penalty inside the likelihood function

Examples

```
# no example
```

---

report *Get reported quantities from and* RTMB *object and return a* LaMaModel

---

Description

Having fitted a latent Markov model using automatic differentiation via RTMB, this function calls RTMB's obj$report() and does some additional processing. This then yields estimated parameters on their natural scale, allows for convenient calculation of AIC and BIC, state-decoding, and residual calculation.

Usage

```
report(obj)
```

Arguments

obj	Optimised RTMB object

Value

A model object of class "LaMaModel" containing a list with the reported quantities from the RTMB object, along estimated parameters and other quantities.

Examples

```
data <- trex[1:200,]

# initial parameters and observations
par <- list(
  log_mu = log(c(0.3, 1)),      # initial means for step length (log-transformed)
  log_sigma = log(c(0.2, 0.7)), # initial sds for step length (log-transformed)
  eta = rep(-2, 2)              # initial t.p.m. parameters (on logit scale)
)
dat <- list(
  step = data$step,             # hourly step lengths
  nStates = 2                   # number of hidden states
)

# likelihood function
nll <- function(par) {
  getAll(par, dat)
  Gamma <- tpm(eta)
  delta <- stationary(Gamma)
  mu <- exp(log_mu); REPORT(mu)
  sigma <- exp(log_sigma); REPORT(sigma)
  allprobs <- matrix(1, length(step), nStates)
  ind <- which(!is.na(step))
  for(j in 1:nStates) {
    allprobs[ind,j] <- dgamma2(step[ind], mu[j], sigma[j])
  }
  -forward(delta, Gamma, allprobs)
}

# automatic differentiation and optimisation
obj <- MakeADFun(nll, par, silent = TRUE)
opt <- nlminb(obj$par, obj$fn, obj$gr)

### reporting ###
mod <- report(obj)

# estimated parameters
mod$par

# estimated quantities on natural scale
mod$mu
mod$sigma
mod$Gamma

# information criteria
AIC(mod)
BIC(mod)

# state decoding
states <- viterbi(mod = mod)   # global decoding
probs <- stateprobs(mod = mod) # local decoding
```

```
# residual calculation
pres <- pseudo_res(data$step, # observation sequence
                   "gamma2",  # distribution family
                   list(mean = mod$mu, sd = mod$sigma), # parameters for that family
                   mod = mod) # model object
```

`sdreport_outer` *Report uncertainty of the estimated smoothing parameters or variances*

Description

Computes standard deviations for the smoothing parameters of a model object returned by `qreml` using the delta method.

Usage

```
sdreport_outer(mod, invert = FALSE)
```

Arguments

`mod`	model objects as returned by `qreml`
`invert`	optional logical; if `TRUE`, the inverse smoothing paramaters (variances) are returned along with the transformed standard deviations obtained via the delta method.

Details

The computations are based on the approximate gradient of the restricted log likelihood. The outer Hessian is computed by finite differencing of this gradient. If the inverse smoothing parameters are requested, the standard deviations are transformed to the variances using the delta method.

Value

list containing `report` matrix summarising parameters and standard deviations as well as the outer `Hessian` matrix.

Examples

```
## no examples
```

sdreportMC *Monte Carlo version of* sdreport

Description

After optimisation of an AD model, sdreportMC can be used to calculate samples of confidence intervals of all model parameters and REPORT()ed quantities including nonlinear functions of random effects and parameters.

Usage

```
sdreportMC(
  obj,
  what,
  nSamples = 1000,
  Hessian = NULL,
  CI = FALSE,
  probs = c(0.025, 0.975)
)
```

Arguments

obj	object returned by MakeADFun() after optimisation or model of class qremlModel as returned by qreml.
what	vector of strings with names of parameters and/or REPORT()ed quantities to be reported
nSamples	number of samples to draw from the multivariate normal distribution of the MLE
Hessian	optional Hessian matrix. If not provided, it will be computed from the object
CI	logical. If TRUE, only confidence intervals instead of samples will be returned
probs	vector of probabilities for the confidence intervals (ignored if no CIs are computed)

Details

This function simply samples from the approximate multivariate normal distribution of the maximum likelihood estimate (MLE) of the parameters

$$\hat{\theta} \sim N(\theta, H^{-1}),$$

where H is the Hessian matrix of the negative log-likelihood function at the MLE. It then returns either the sampled parameters or REPORT()ed transformations of them. If CI = TRUE, it does not return the samples but directly computes confidence intervals.

If you are interested in several quantities, calling sdreportMC once with a vector what will generally be faster than calling it several times with single elements of what.

Value

named list corresponding to the elements of `what`. Each element has the structure of the corresponding quantity with an additional dimension added for the samples. For example, if a quantity is a vector, the list contains a matrix. If a quantity is a matrix, the list contains an array. If quantity is an array, the list contains an array with one extra dimension.

Examples

```
# fitting an HMM to the trex data and running sdreportMC
## negative log-likelihood function
nll = function(par) {
  getAll(par, dat) # makes everything contained available without $
  Gamma = tpm(eta) # computes transition probability matrix from unconstrained eta
  delta = stationary(Gamma) # computes stationary distribution
  # exponentiating because all parameters strictly positive
  mu = exp(logmu)
  sigma = exp(logsigma)
  kappa = exp(logkappa)
  # reporting statements for sdreportMC
  REPORT(mu)
  REPORT(sigma)
  REPORT(kappa)
  # calculating all state-dependent densities
  allprobs = matrix(1, nrow = length(step), ncol = N)
  ind = which(!is.na(step) & !is.na(angle)) # only for non-NA obs.
  for(j in 1:N){
    allprobs[ind,j] = dgamma2(step[ind],mu[j],sigma[j])*dvm(angle[ind],0,kappa[j])
  }
  -forward(delta, Gamma, allprobs) # simple forward algorithm
}

## initial parameter list
par = list(
 logmu = log(c(0.3, 1)),       # initial means for step length (log-transformed)
  logsigma = log(c(0.2, 0.7)), # initial sds for step length (log-transformed)
  logkappa = log(c(0.2, 0.7)), # initial concentration for turning angle (log-transformed)
  eta = rep(-2, 2)             # initial t.p.m. parameters (on logit scale)
)
## data and hyperparameters
dat = list(
  step = trex$step[1:500],   # hourly step lengths
  angle = trex$angle[1:500], # hourly turning angles
  N = 2
)

## creating AD function
obj = MakeADFun(nll, par, silent = TRUE) # creating the objective function

## optimising
opt = nlminb(obj$par, obj$fn, obj$gr) # optimization

## running sdreportMC
```

```
# `mu` has report statement, `delta` is automatically reported by `forward()`
sdrMC = sdreportMC(obj,
                   what = c("mu", "delta"),
                   nSamples = 50)
dim(sdrMC$delta)
# now a matrix with 50 samples (rows)
```

---

skewnorm *Skew normal distribution*

Description

Density, distribution function, quantile function and random generation for the skew normal distribution.

Usage

```
dskewnorm(x, xi = 0, omega = 1, alpha = 0, log = FALSE)

pskewnorm(q, xi = 0, omega = 1, alpha = 0, ...)

qskewnorm(p, xi = 0, omega = 1, alpha = 0, ...)

rskewnorm(n, xi = 0, omega = 1, alpha = 0)
```

Arguments

x, q	vector of quantiles
xi	location parameter
omega	scale parameter, must be positive.
alpha	skewness parameter, +/- Inf is allowed.
log	logical; if TRUE, probabilities/ densities p are returned as $\log(p)$.
...	additional parameters to be passed to the sn package functions for pskewnorm and qskewnorm.
p	vector of probabilities
n	number of observations. If length(n) > 1, the length is taken to be the number required.

Details

This implementation of dskewnorm allows for automatic differentiation with RTMB while the other functions are imported from the sn package.

Value

dskewnorm gives the density, pskewnorm gives the distribution function, qskewnorm gives the quantile function, and rskewnorm generates random deviates.

Examples

```
x = rskewnorm(1)
d = dskewnorm(x)
p = pskewnorm(x)
q = qskewnorm(p)
```

`smooth_dens_construct` *Build the design and penalty matrices for smooth density estimation*

Description

This high-level function can be used to prepare objects needed to estimate mixture models of smooth densities using P-Splines.

Usage

```
smooth_dens_construct(
  data,
  par,
  type = "real",
  k = 25,
  knots = NULL,
  degree = 3,
  diff_order = 2
)
```

Arguments

`data`	named data frame of 1 or multiple data streams
`par`	nested named list of initial means and sds/concentrations for each data stream
`type`	vector of length 1 or number of data streams containing the type of each data stream, either `"real"` for data on the reals, `"positive"` for data on the positive reals or `"circular"` for angular data.
`k`	vector of length 1 or number of data streams containing the number of basis functions for each data stream
`knots`	optional list of knots vectors (including the boundary knots) to be used for basis construction. If not provided, the knots are placed equidistantly for `"real"` and `"circular"` and using polynomial spacing for `"positive"`. For `"real"` and `"positive"` `k - degree + 1` knots are needed, for `"circular"` `k + 1` knots are needed.
`degree`	degree of the B-spline basis functions for each data stream, defaults to cubic B-splines
`diff_order`	order of differencing used for the P-Spline penalty matrix for each data stream. Defaults to second-order differences.

Details

Under the hood, make_matrices_dens is used for the actual construction of the design and penalty matrices.

You can provide one or multiple data streams of different types (real, positive, circular) and specify initial means and standard deviations/ concentrations for each data stream. This information is then converted into suitable spline coefficients. smooth_dens_construct then constructs the design and penalty matrices for standardised B-splines basis functions (integrating to one) for each data stream. For types "real" and "circular" the knots are placed equidistant in the range of the data, for type "positive" the knots are placed using polynomial spacing.

Value

a nested list containing the design matrices Z, the penalty matrices S, the initial coefficients coef the prediction design matrices Z_predict, the prediction grids xseq, and details for the basis expansion for each data stream.

Examples

```
## 3 data streams, each with one distribution
# normal data with mean 0 and sd 1
x1 = rnorm(100, mean = 0, sd = 1)
# gamma data with mean 5 and sd 3
x2 = rgamma2(100, mean = 5, sd = 3)
# circular data
x3 = rvm(100, mu = 0, kappa = 2)

data = data.frame(x1 = x1, x2 = x2, x3 = x3)

par = list(x1 = list(mean = 0, sd = 1),
           x2 = list(mean = 5, sd = 3),
           x3 = list(mean = 0, concentration = 2))

SmoothDens = smooth_dens_construct(data,
                                   par,
                                   type = c("real", "positive", "circular"))

# extracting objects for x1
Z1 = SmoothDens$Z$x1
S1 = SmoothDens$S$x1
coefs1 = SmoothDens$coef$x1

## one data stream, but mixture of two distributions
# normal data with mean 0 and sd 1
x = rnorm(100, mean = 0, sd = 1)
data = data.frame(x = x)

# now parameters for mixture of two normals
par = list(x = list(mean = c(0, 5), sd = c(1,1)))

SmoothDens = smooth_dens_construct(data, par = par)
```

```
# extracting objects
Z = SmoothDens$Z$x
S = SmoothDens$S$x
coefs = SmoothDens$coef$x
```

`stateprobs` *Calculate conditional local state probabilities in HMMs*

Description

Computes

$$\Pr(\text{State}_t = j \mid X_1, ..., X_T)$$

for a given HMM.

Usage

```
stateprobs(
  delta,
  Gamma,
  allprobs,
  trackID = NULL,
  mod = NULL,
  forecast = FALSE
)
```

Arguments

`delta`	initial distribution; either • a vector of length `nStates`, or • a matrix of dimension `c(nTracks, nStates)` if `trackID` is provided
`Gamma`	transition probability matrix; either • a matrix of dimension `c(nStates, nStates)`, • an array of dimension `c(nStates, nStates, nTracks)` if `trackID` is provided, or • an array of dimension `c(nStates, nStates, nObs)` for time-varying transition probabilities, in which case `stateprobs_g` is called internally
`allprobs`	matrix of state-dependent probabilities or density values of dimension `c(nObs, nStates)`
`trackID`	optional vector of length `nObs` containing `nTracks` unique IDs that separate tracks
`mod`	optional model object containing `delta`, `Gamma`, `allprobs`, and optionally `trackID`. When using `RTMB::MakeADFun` or `qreml` with `forward` in the likelihood, these are reported automatically after model fitting and the object returned by `RTMB::report()` or `qreml` can be passed directly.
`forecast`	logical, indicating if forecast probabilities $\Pr(\text{State}_t = j \mid X_1, ..., X_{t-1})$ should be calculated instead.

Value

matrix of conditional state probabilities of dimension `c(nObs, nStates)`

See Also

Other decoding functions: `stateprobs_g()`, `stateprobs_p()`, `viterbi()`, `viterbi_g()`, `viterbi_p()`

Examples

```
Gamma = tpm(c(-1,-2))
delta = stationary(Gamma)
allprobs = matrix(runif(10), nrow = 10, ncol = 2)

probs = stateprobs(delta, Gamma, allprobs)
```

---

stateprobs_g *Calculate conditional local state probabilities for inhomogeneous HMMs*

Description

Computes

$$\Pr(\text{State}_t = j \mid X_1, ..., X_T)$$

for inhomogeneous HMMs

Usage

```
stateprobs_g(
  delta,
  Gamma,
  allprobs,
  trackID = NULL,
  mod = NULL,
  forecast = FALSE
)
```

Arguments

`delta`	initial distribution; either • a vector of length `nStates`, or • a matrix of dimension `c(nTracks, nStates)` if `trackID` is provided
`Gamma`	array of transition probability matrices of dimension `c(nStates, nStates, nObs)`, where the first slice of each track is ignored as there is no transition into the start of a track. For a single track, an array of dimension `c(nStates, nStates, nObs-1)` is also accepted.

allprobs	matrix of state-dependent probabilities or density values of dimension c(nObs, nStates)
trackID	optional vector of length nObs containing nTracks unique IDs that separate tracks
mod	optional model object containing delta, Gamma, allprobs, and optionally trackID. When using RTMB::MakeADFun or qreml with forward_g in the likelihood, these are reported automatically after model fitting and the object returned by RTMB::report() or qreml can be passed directly.
forecast	logical, indicating if forecast probabilities $\Pr(\text{State}_t = j \mid X_1, ..., X_{t-1})$ should be calculated instead.

Value

matrix of conditional state probabilities of dimension c(nObs, nStates)

See Also

Other decoding functions: stateprobs(), stateprobs_p(), viterbi(), viterbi_g(), viterbi_p()

Examples

```
Gamma = tpm_g(runif(10), matrix(c(-1,-1,1,-2), nrow = 2, byrow = TRUE))
delta = c(0.5, 0.5)
allprobs = matrix(runif(20), nrow = 10, ncol = 2)

probs = stateprobs_g(delta, Gamma[,,-1], allprobs)
```

---

stateprobs_p *Calculate conditional local state probabilities for periodically inhomogeneous HMMs*

Description

Computes

$$\Pr(S_t = j \mid X_1, ..., X_T)$$

for periodically inhomogeneous HMMs

Usage

```
stateprobs_p(
  delta,
  Gamma,
  allprobs,
  tod,
  trackID = NULL,
  mod = NULL,
  forecast = FALSE
)
```

Arguments

`delta`	initial or stationary distribution of length N, or matrix of dimension c(k,N) for k independent tracks, if `trackID` is provided This could e.g. be the periodically stationary distribution (for each track) as computed by `stationary_p`.
`Gamma`	array of transition probability matrices for each time point in the cycle of dimension c(N,N,L), where L is the length of the cycle.
`allprobs`	matrix of state-dependent probabilities/ density values of dimension c(n, N)
`tod`	(Integer valued) variable for cycle indexing in 1, ..., L, mapping the data index to a generalised time of day (length n). For half-hourly data L = 48. It could, however, also be day of year for daily data and L = 365.
`trackID`	optional vector of k track IDs, if multiple tracks need to be decoded separately
`mod`	optional model object containing initial distribution `delta`, transition probability matrix `Gamma`, matrix of state-dependent probabilities `allprobs`, and potentially a `trackID` variable If you are using automatic differentiation either with `RTMB::MakeADFun` or `qreml` and include `forward_p` in your likelihood function, the objects needed for state decoding are automatically reported after model fitting. Hence, you can pass the model object obtained from running `RTMB::report()` or from `qreml` directly to this function.
`forecast`	logical, indicating if forecast probabilities $\Pr(S_t = j \mid X_1, ..., X_{t-1})$ should be calculated instead.

Value

matrix of conditional state probabilities of dimension c(n,N)

See Also

Other decoding functions: `stateprobs()`, `stateprobs_g()`, `viterbi()`, `viterbi_g()`, `viterbi_p()`

Examples

```
L = 24
beta = matrix(c(-1, 2, -1, -2, 1, -1), nrow = 2, byrow = TRUE)
Gamma = tpm_p(1:L, L, beta, degree = 1)
delta = stationary_p(Gamma, 1)
allprobs = matrix(runif(200), nrow = 100, ncol = 2)
tod = rep(1:24, 5)[1:100]

probs = stateprobs_p(delta, Gamma, allprobs, tod)
```

`stationary` *Compute the stationary distribution of a homogeneous Markov chain*

Description

A homogeneous, finite state Markov chain that is irreducible and aperiodic converges to a unique stationary distribution, here called δ. As it is stationary, this distribution satisfies

$$\delta\Gamma = \delta,$$

subject to $\sum_{j=1}^{N} \delta_j = 1$, where Γ is the transition probability matrix. This function solves the linear system of equations above.

Usage

```
stationary(Gamma)
```

Arguments

`Gamma`	transition probability matrix of dimension `c(N,N)` or array of such matrices of dimension `c(N,N,nTracks)` if the stationary distribution should be computed for several matrices at once

Value

either a single stationary distribution of the Markov chain (vector of length `N`) or a matrix of stationary distributions of dimension `c(nTracks,N)` with one stationary distribution in each row

See Also

`tpm` to create a transition probabilty matrix using the multinomial logistic link (softmax)

Other stationary distribution functions: `stationary_ct()`, `stationary_p()`

Examples

```
# single matrix
Gamma = tpm(c(rep(-2,3), rep(-3,3)))
delta = stationary(Gamma)
# multiple matrices
Gamma = array(Gamma, dim = c(3,3,10))
Delta = stationary(Gamma)
```

`stationary_ct` *Compute the stationary distribution of a continuous-time Markov chain*

Description

A well-behaved continuous-time Markov chain converges to a unique stationary distribution, here called π. This distribution satisfies

$$\pi Q = 0,$$

subject to $\sum_{j=1}^{N} \pi_j = 1$, where Q is the infinitesimal generator of the Markov chain. This function solves the linear system of equations above for a given generator matrix.

Usage

```
stationary_ct(Q)

stationary_cont(Q)
```

Arguments

`Q`	infinitesimal generator matrix of dimension `c(N,N)` or array of such matrices of dimension `c(N,N,nTracks)` if the stationary distribution should be computed for several matrices at once

Value

either a single stationary distribution of the continuous-time Markov chain (vector of length `N`) or a matrix of stationary distributions of dimension `c(nTracks,N)` with one stationary distribution in each row

See Also

`generator` to create a generator matrix

Other stationary distribution functions: `stationary()`, `stationary_p()`

Examples

```
# single matrix
Q = generator(c(-2,-2))
Pi = stationary_ct(Q)
# multiple matrices
Q = array(Q, dim = c(2,2,10))
Pi = stationary_ct(Q)
```

`stationary_p` *Periodically stationary distribution of a periodically inhomogeneous Markov chain*

Description

Computes the periodically stationary distribution of a periodically inhomogeneous Markov chain.

Usage

```
stationary_p(Gamma, t = NULL, ad = NULL)
```

Arguments

`Gamma`	array of transition probability matrices of dimension c(N,N,L)
`t`	integer index of the time point in the cycle, for which to calculate the stationary distribution If `t` is not provided, the function calculates all stationary distributions for each time point in the cycle.
`ad`	optional logical, indicating whether automatic differentiation with `RTMB` should be used. By default, the function determines this itself.

Details

If the transition probability matrix of an inhomogeneous Markov chain varies only periodically (with period length L), it converges to a so-called periodically stationary distribution. This happens, because the thinned Markov chain, which has a full cycle as each time step, has homogeneous transition probability matrix

$$\Gamma_t = \Gamma^{(t)}\Gamma^{(t+1)}\dots\Gamma^{(t+L-1)}$$

for all $t = 1, \dots, L$. The stationary distribution for time t satifies $\delta^{(t)}\Gamma_t = \delta^{(t)}$.

This function calculates said periodically stationary distribution.

Value

either the periodically stationary distribution at time t or all periodically stationary distributions.

References

Koslik, J. O., Feldmann, C. C., Mews, S., Michels, R., & Langrock, R. (2023). Inference on the state process of periodically inhomogeneous hidden Markov models for animal behavior. arXiv preprint arXiv:2312.14583.

See Also

`tpm_p` and `tpm_g` to create multiple transition matrices based on a cyclic variable or design matrix

Other stationary distribution functions: `stationary()`, `stationary_ct()`

Examples

```
# setting parameters for trigonometric link
beta = matrix(c(-1, 2, -1, -2, 1, -1), nrow = 2, byrow = TRUE)
Gamma = tpm_p(beta = beta, degree = 1)
# periodically stationary distribution for specific time point
delta = stationary_p(Gamma, 4)

# all periodically stationary distributions
Delta = stationary_p(Gamma)
```

`stationary_p_sparse` *Sparse version of* `stationary_p`

Description

This is function computes the periodically stationary distribution of a Markov chain given a list of L **sparse** transition probability matrices. Compatible with automatic differentiation by `RTMB`

Usage

```
stationary_p_sparse(Gamma, t = NULL)
```

Arguments

`Gamma`	sist of length L containing sparse transition probability matrices for one cycle.
`t`	integer index of the time point in the cycle, for which to calculate the stationary distribution If t is not provided, the function calculates all stationary distributions for each time point in the cycle.

Value

either the periodically stationary distribution at time t or all periodically stationary distributions.

Examples

```
## periodic HSMM example (here the approximating tpm is sparse)
N = 2 # number of states
L = 24 # cycle length
# time-varying mean dwell times
Z = cosinor(1:L, period = L) # trigonometric basis functions design matrix
beta = matrix(c(2, 2, 0.1, -0.1, -0.2, 0.2), nrow = 2)
Lambda = exp(cbind(1, Z) %*% t(beta))
sizes = c(20, 20) # approximating chain with 40 states
# state dwell-time distributions
dm = lapply(1:N, function(i) sapply(1:sizes[i]-1, dpois, lambda = Lambda[,i]))
omega = matrix(c(0,1,1,0), nrow = N, byrow = TRUE) # embedded t.p.m.

# calculating extended-state-space t.p.m.s
Gamma = tpm_phsmm(omega, dm)
```

```
# Periodically stationary distribution for specific time point
delta = stationary_p_sparse(Gamma, 4)

# All periodically stationary distributions
Delta = stationary_p_sparse(Gamma)
```

`stationary_sparse` *Sparse version of* `stationary`

Description

This is function computes the stationary distribution of a Markov chain with a given **sparse** transition probability matrix. Compatible with automatic differentiation by `RTMB`

Usage

```
stationary_sparse(Gamma)
```

Arguments

`Gamma`	sparse transition probability matrix of dimension c(N,N)

Value

stationary distribution of the Markov chain with the given transition probability matrix

Examples

```
## HSMM example (here the approximating tpm is sparse)
# building the t.p.m. of the embedded Markov chain
omega = matrix(c(0,1,1,0), nrow = 2, byrow = TRUE)
# defining state aggregate sizes
sizes = c(20, 30)
# defining state dwell-time distributions
lambda = c(5, 11)
dm = list(dpois(1:sizes[1]-1, lambda[1]), dpois(1:sizes[2]-1, lambda[2]))
# calculating extended-state-space t.p.m.
Gamma = tpm_hsmm(omega, dm)
delta = stationary_sparse(Gamma)
```

summary.qremlModel *Summary method for* qremlModel *objects*

Description

Prints a summary of a model object created by qreml.

Usage

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

Arguments

object	qremlModel object created by qreml
...	additional arguments

Value

prints a summary of the model object

Examples

```
# no examples
```

tpm *Build the transition probability matrix from unconstrained parameter vector*

Description

Markov chains are parametrised in terms of a transition probability matrix Γ, for which each row contains a conditional probability distribution of the next state given the current state. Hence, each row has entries between 0 and 1 that need to sum to one.

For numerical optimisation, we parameterise in terms of unconstrained parameters, thus this function computes said matrix from an unconstrained parameter vector via the inverse multinomial logistic link (also known as softmax) applied to each row.

Usage

```
tpm(
  beta,
  Z = NULL,
  Eta = NULL,
  byrow = FALSE,
  ref = NULL,
  ad = NULL,
  report = TRUE,
  param = NULL
)
```

Arguments

beta	parameters; either • a vector of length nStates * (nStates-1), or • a matrix of dimension c(nStates * (nStates-1), p+1) if design matrix Z is also provided.
Z	optional covariate design matrix with or without intercept column, i.e. of dimension c(nObs, p) or c(nObs, p+1). If provided, beta needs to be a matrix of dimension c(nStates * (nStates-1), p+1).
Eta	optional pre-calculated matrix of linear predictors of dimension c(nObs, nStates * (nStates-1)). If provided, Z and beta will be ignored.
byrow	logical indicating if each transition probability matrix should be filled by row. Defaults to FALSE, but should be set to TRUE if one wants to work with a matrix of beta parameters returned by popular HMM packages like moveHMM, momentuHMM, or hmmTMB.
ref	optional integer vector of length nStates giving, for each row, the column index of the reference state (its predictor is fixed to 0). Defaults to the diagonal (ref = 1:nStates).
ad	logical; whether to use automatic differentiation. Determined automatically — for debugging only.
report	logical; if TRUE (default), delta, Gamma, allprobs, and trackID are reported from the fitted model. Requires ad = TRUE.
param	depricated, please use argument beta instead.

Value

Transition probability matrix of dimension c(nStates, nStates) or array of such matrices of dimension c(nStates, nStates, nObs) if Z or Eta is provided.

See Also

Other transition probability matrix functions: generator(), generator_g(), tpm_ct(), tpm_emb(), tpm_emb_g(), tpm_g(), tpm_g2(), tpm_p()

Examples

```
## homogeneous Markov chain
# 2 states: 2 = 2*(2-1) free off-diagonal elements
par <- rep(-2, 2)
Gamma <- tpm(par)
# 3 states: 6 = 3*(3-1) free off-diagonal elements
par <- rep(-3, 6)
Gamma <- tpm(par)
# 4 states: 12 = 4*(4-1) free off-diagonal elements
par <- rep(-4, 12)
Gamma <- tpm(par)

## inhomogeneous Markov chain
# t.p.m. depends on covariates
z1 <- runif(100); z2 <- runif(100) # 2 covariates
Z <- cbind(1, z1, z2) # design matrix
beta0 <- c(-2, -2); beta1 = c(1, -2); beta2 = c(2, -1) # coefficients for intercept and covariates
beta <- cbind(beta0, beta1, beta2) # coefficient matrix; with intercepts!
Gamma <- tpm(beta, Z) # array with 100 slices
```

---

tpm_ct *Calculate continuous time transition probabilities*

Description

A continuous-time Markov chain is described by an infinitesimal generator matrix Q. When observing data at time points $t_1, \dots, t_n$ the transition probabilites between t_i and t_{i+1} are caluclated as

$$\Gamma(\Delta t_i) = \exp(Q\Delta t_i),$$

where $\exp()$ is the matrix exponential. The mapping $\Gamma(\Delta t)$ is also called the **Markov semigroup**. This function calculates all transition matrices based on a given generator and time differences.

Usage

```
tpm_ct(Q, timediff, rates = NULL, ad = NULL, report = TRUE)

tpm_cont(Q, timediff, rates = NULL, ad = NULL, report = TRUE)
```

Arguments

`Q`	infinitesimal generator matrix of the continuous-time Markov chain of dimension `c(nStates, nStates)`
`timediff`	time differences between observations of length `nObs-1` when based on `nObs` observations
`rates`	optional vector of state-dependent rates for MM(M)PP fitting. For the MM(M)PP likelihood, the matrices needed in the forward algorithm are $\exp((Q-\Lambda)\Delta t)$, where Λ is a diagonal matrix with the state-dependent rates on the diagonal.

ad	optional logical, indicating whether automatic differentiation with RTMB should be used. By default, the function determines this itself.
report	logical, indicating whether Q should be reported from the fitted model. Defaults to TRUE, but only works if ad = TRUE.

Value

array of continuous-time transition matrices of dimension c(nStates, nStates, nObs-1)

See Also

Other transition probability matrix functions: generator(), generator_g(), tpm(), tpm_emb(), tpm_emb_g(), tpm_g(), tpm_g2(), tpm_p()

Examples

```
# building a Q matrix for a 3-state cont.-time Markov chain
Q = generator(rep(-2, 6))

# draw random time differences
timediff = rexp(100, 10)

# compute all transition matrices
Gamma = tpm_cont(Q, timediff)
```

---

tpm_emb *Build the embedded transition probability matrix of an HSMM from unconstrained parameter vector*

Description

Hidden semi-Markov models are defined in terms of state durations and an **embedded** transition probability matrix that contains the conditional transition probabilities given that the **current state is left**. This matrix necessarily has diagonal entries all equal to zero as self-transitions are impossible.

This function builds such an embedded/ conditional transition probability matrix from an unconstrained parameter vector. For each row of the matrix, the inverse multinomial logistic link is applied.

For a matrix of dimension c(nStates, nStates), the number of free off-diagonal elements is nStates * (nStates-2), hence also the length of param. This means, for 2 states, the function needs to be called without any arguments, for 3-states with a vector of length 3, for 4 states with a vector of length 8, etc.

Compatible with automatic differentiation by RTMB

Usage

```
tpm_emb(param = NULL)
```

Arguments

param	unconstrained parameter vector of length nStates * (nStates-2) If the function is called without param, it will return the conditional transition probability matrix for a 2-state HSMM, which is fixed with 0 diagonal entries and off-diagonal entries equal to 1.

Value

embedded/ conditional transition probability matrix of dimension c(nStates, nStates)

See Also

Other transition probability matrix functions: generator(), generator_g(), tpm(), tpm_ct(), tpm_emb_g(), tpm_g(), tpm_g2(), tpm_p()

Examples

```
# 2 states: no free off-diagonal elements
omega = tpm_emb()

# 3 states: 3 free off-diagonal elements
param = rep(0, 3)
omega = tpm_emb(param)

# 4 states: 8 free off-diagonal elements
param = rep(0, 8)
omega = tpm_emb(param)
```

---

tpm_emb_g *Build all embedded transition probability matrices of an inhomogeneous HSMM*

Description

Hidden semi-Markov models are defined in terms of state durations and an **embedded** transition probability matrix that contains the conditional transition probabilities given that the **current state is left**. This matrix necessarily has diagonal entries all equal to zero as self-transitions are impossible. We can allow this matrix to vary with covariates, which is the purpose of this function.

It builds all embedded/ conditional transition probability matrices based on a design and parameter matrix. For each row of the matrix, the inverse multinomial logistic link is applied.

For a matrix of dimension c(nStates, nStates), the number of free off-diagonal elements is nStates * (nStates-2) which determines the number of rows of the parameter matrix.

Compatible with automatic differentiation by RTMB

Usage

```
tpm_emb_g(Z, beta, report = TRUE)
```

Arguments

Z	covariate design matrix with or without intercept column, i.e. of dimension c(nObs, p) or c(nObs, p+1) If Z has only p columns, an intercept column of ones will be added automatically.
beta	matrix of coefficients for the off-diagonal elements of the embedded transition probability matrix Needs to be of dimension c(nStates * (nStates-2), p+1), where the first column contains the intercepts. p can be 0, in which case the model is homogeneous.
report	logical, indicating whether the coefficient matrix beta should be reported from the fitted model. Defaults to TRUE.

Value

array of embedded/ conditional transition probability matrices of dimension c(nStates, nStates, nObs)

See Also

Other transition probability matrix functions: generator(), generator_g(), tpm(), tpm_ct(), tpm_emb(), tpm_g(), tpm_g2(), tpm_p()

Examples

```
## parameter matrix for 3-state HSMM
beta = matrix(c(rep(0, 3), -0.2, 0.2, 0.1), nrow = 3)
# no intercept
Z = rnorm(100)
omega = tpm_emb_g(Z, beta)
# intercept
Z = cbind(1, Z)
omega = tpm_emb_g(Z, beta)
```

---

tpm_g *Build all transition probability matrices of an inhomogeneous HMM*

Description

In an HMM, we often model the influence of covariates on the state process by linking them to the transition probabiltiy matrix. Most commonly, this is done by specifying a linear predictor

$$\eta_{ij}^{(t)} = \beta_0^{(ij)} + \beta_1^{(ij)} z_{t1} + \cdots + \beta_p^{(ij)} z_{tp}$$

for each off-diagonal element ($i \neq j$) of the transition probability matrix and then applying the inverse multinomial logistic link (also known as softmax) to each row. This function efficiently calculates all transition probabilty matrices for a given design matrix Z and parameter matrix beta.

Usage

```
tpm_g(
  Z,
  beta,
  Eta = NULL,
  byrow = FALSE,
  ref = NULL,
  ad = NULL,
  report = TRUE,
  sparse = FALSE
)
```

Arguments

Z	Covariate design matrix with or without intercept column, i.e. of dimension c(nObs, p) or c(nObs, p+1). If not provided, intercept column is added automatically.
beta	Matrix of coefficients for the off-diagonal elements of the transition probability matrix of dimension c(nStates * (nStates-1), p+1). First columns contains the intercepts.
Eta	optional pre-calculated matrix of linear predictors of dimension c(nObs, nStates * (nStates-1)). If provided, no Z and beta are necessary and will be ignored.
byrow	logical indicating if each transition probability matrix should be filled by row. Defaults to FALSE, but should be set to TRUE if one wants to work with a matrix of beta parameters returned by popular HMM packages like moveHMM, momentuHMM, or hmmTMB.
ref	optional integer vector of length nStates giving, for each row, the column index of the reference state (its predictor is fixed to 0). Defaults to the diagonal (ref = 1:nStates).
ad	logical; whether to use automatic differentiation. Determined automatically — for debugging only.
report	logical; if TRUE (default), delta, Gamma, allprobs, and trackID are reported from the fitted model. Requires ad = TRUE.
sparse	logical, indicating whether sparsity in the rows of Z should be exploited.

Value

array of transition probability matrices of dimension c(nStates, nStates, nObs)

See Also

Other transition probability matrix functions: generator(), generator_g(), tpm(), tpm_ct(), tpm_emb(), tpm_emb_g(), tpm_g2(), tpm_p()

Examples

```
## inhomogeneous Markov chain
# t.p.m. depends on covariates
z1 <- runif(100); z2 <- runif(100) # 2 covariates
Z <- cbind(1, z1, z2) # design matrix
beta0 <- c(-2, -2); beta1 = c(1, -2); beta2 = c(2, -1) # coefficients for intercept and covariates
beta <- cbind(beta0, beta1, beta2) # coefficient matrix; with intercepts!
#' Gamma <- tpm(beta, Z) # array with 100 slices
```

`tpm_g2` *Build all transition probability matrices of an inhomogeneous HMM*

Description

In an HMM, we often model the influence of covariates on the state process by linking them to the transition probabiltiy matrix. Most commonly, this is done by specifying linear predictors

$$\eta_{ij}^{(t)} = \beta_0^{(ij)} + \beta_1^{(ij)} z_{t1} + \cdots + \beta_p^{(ij)} z_{tp}$$

for each off-diagonal element ($i \neq j$) of the transition probability matrix and then applying the inverse multinomial logistic link (also known as softmax) to each row. This function efficiently calculates all transition probabilty matrices for a given design matrix `Z` and parameter matrix `beta`.

Usage

```
tpm_g2(Z, beta, byrow = FALSE, ad = NULL, report = TRUE, ref = NULL)
```

Arguments

`Z`	covariate design matrix with or without intercept column, i.e. of dimension c(n, p) or c(n, p+1) If `Z` has only p columns, an intercept column of ones will be added automatically. Can also be a list of N*(N-1) design matrices with different number of columns but the same number of rows. In that case, no intercept column will be added.
`beta`	matrix of coefficients for the off-diagonal elements of the transition probability matrix Needs to be of dimension c(N*(N-1), p+1), where the first column contains the intercepts. If `Z` is a list, `beta` can also be a list of length N*(N-1) with each entry being a vector or a (long) matrix of coefficients, each matching the dimension of the corresponding entry in `Z`.
`byrow`	logical indicating if each transition probability matrix should be filled by row Defaults to `FALSE`, but should be set to `TRUE` if one wants to work with a matrix of beta parameters returned by popular HMM packages like `moveHMM`, `momentuHMM`, or `hmmTMB`.

ad	optional logical, indicating whether automatic differentiation with RTMB should be used. By default, the function determines this itself.
report	logical, indicating whether the coefficient matrix beta should be reported from the fitted model. Defaults to TRUE, but only works if ad = TRUE.
ref	optional vector of length N with the reference state indices for each column of the transition probability matrix. Each row in the transition matrix corresponds to a multinomial regression, hence one state needs to be the reference category. Defaults to off-diagonal elements (ref = 1:N).

Value

array of transition probability matrices of dimension c(N,N,n)

See Also

Other transition probability matrix functions: generator(), generator_g(), tpm(), tpm_ct(), tpm_emb(), tpm_emb_g(), tpm_g(), tpm_p()

Examples

```
Z = matrix(runif(200), ncol = 2)
beta = matrix(c(-1, 1, 2, -2, 1, -2), nrow = 2, byrow = TRUE)
Gamma = tpm_g(Z, beta)
```

---

tpm_hsmm *Builds the transition probability matrix of an HSMM-approximating HMM*

Description

Hidden semi-Markov models (HSMMs) are a flexible extension of HMMs, where the state duration distribution is explicitly modelled. For direct numerical maximum likelihood estimation, HSMMs can be represented as HMMs on an enlarged state space (of size M) and with structured transition probabilities.

This function computes the transition matrix to approximate a given HSMM by an HMM with a larger state space.

Usage

```
tpm_hsmm(omega, dm, Fm = NULL, sparse = TRUE, eps = 1e-10)
```

Arguments

omega	embedded transition probability matrix of dimension c(nStates, nStates) as computed by tpm_emb.
dm	state dwell-time distributions arranged in a list of length nStates. Each list element needs to be a vector of length N_i, where N_i is the state aggregate size.
Fm	optional list of length nStates containing the cumulative distribution functions of the dwell-time distributions.
sparse	logical, indicating whether the output should be a **sparse** matrix. Defaults to TRUE.
eps	rounding value: If an entry of the transition probabily matrix is smaller, than it is rounded to zero. Usually, this should not be changed.

Value

extended-state-space transition probability matrix of the approximating HMM

Examples

```
# building the t.p.m. of the embedded Markov chain
omega = matrix(c(0,1,1,0), nrow = 2, byrow = TRUE)
# defining state aggregate sizes
sizes = c(20, 30)
# defining state dwell-time distributions
lambda = c(5, 11)
dm = list(dpois(1:sizes[1]-1, lambda[1]), dpois(1:sizes[2]-1, lambda[2]))
# calculating extended-state-space t.p.m.
Gamma = tpm_hsmm(omega, dm)
```

---

tpm_hsmm2 *Build the transition probability matrix of an HSMM-approximating HMM*

Description

Hidden semi-Markov models (HSMMs) are a flexible extension of HMMs. For direct numerical maximum likelihood estimation, HSMMs can be represented as HMMs on an enlarged state space (of size M) and with structured transition probabilities. This function computes the transition matrix of an HSMM.

Usage

```
tpm_hsmm2(omega, dm, eps = 1e-10)
```

Arguments

omega	embedded transition probability matrix of dimension c(N,N)
dm	state dwell-time distributions arranged in a list of length(N). Each list element needs to be a vector of length N_i, where N_i is the state aggregate size.
eps	rounding value: If an entry of the transition probabily matrix is smaller, than it is rounded to zero.

Value

extended-state-space transition probability matrix of the approximating HMM

Examples

```
# building the t.p.m. of the embedded Markov chain
omega = matrix(c(0,1,1,0), nrow = 2, byrow = TRUE)
# defining state aggregate sizes
sizes = c(20, 30)
# defining state dwell-time distributions
lambda = c(5, 11)
dm = list(dpois(1:sizes[1]-1, lambda[1]), dpois(1:sizes[2]-1, lambda[2]))
# calculating extended-state-space t.p.m.
Gamma = tpm_hsmm(omega, dm)
```

---

tpm_ihsmm *Builds all transition probability matrices of an inhomogeneous-HSMM-approximating HMM*

Description

Hidden semi-Markov models (HSMMs) are a flexible extension of HMMs. For direct numerical maximum likelihood estimation, HSMMs can be represented as HMMs on an enlarged state space (of size M) and with structured transition probabilities.

This function computes the transition matrices of a periodically inhomogeneos HSMMs.

Usage

```
tpm_ihsmm(omega, dm, eps = 1e-10)
```

Arguments

omega	embedded transition probability matrix Either a matrix of dimension `c(nStates, nStates)` for homogeneous conditional transition probabilities (as computed by `tpm_emb`), or an array of dimension `c(nStates, nStates, nObs)` for inhomogeneous conditional transition probabilities (as computed by `tpm_emb_g`).

dm	state dwell-time distributions arranged in a list of length nStates Each list element needs to be a matrix of dimension c(n, N_i), where each row t is the (approximate) probability mass function of state i at time t.
eps	rounding value: If an entry of the transition probabily matrix is smaller, than it is rounded to zero. Usually, this should not be changed.

Value

list of dimension length `n - max(sapply(dm, ncol))`, containing sparse extended-state-space transition probability matrices for each time point (except the first `max(sapply(dm, ncol)) - 1`).

Examples

```
N = 2
# time-varying mean dwell times
n = 100
z = runif(n)
beta = matrix(c(2, 2, 0.1, -0.1), nrow = 2)
Lambda = exp(cbind(1, z) %*% t(beta))
sizes = c(15, 15) # approximating chain with 30 states
# state dwell-time distributions
dm = lapply(1:N, function(i) sapply(1:sizes[i]-1, dpois, lambda = Lambda[,i]))

## homogeneous conditional transition probabilites
# diagonal elements are zero, rowsums are one
omega = matrix(c(0,1,1,0), nrow = N, byrow = TRUE)

# calculating extended-state-space t.p.m.s
Gamma = tpm_ihsmm(omega, dm)

## inhomogeneous conditional transition probabilites
# omega can be an array
omega = array(omega, dim = c(N,N,n))

# calculating extended-state-space t.p.m.s
Gamma = tpm_ihsmm(omega, dm)
```

---

tpm_p *Build all transition probability matrices of a periodically inhomogeneous HMM*

Description

Given a periodically varying variable such as time of day or day of year and the associated cycle length, this function calculates the transition probability matrices by applying the inverse multinomial logistic link (also known as softmax) to linear predictors of the form

$$\eta_{ij}^{(t)} = \beta_0^{(ij)} + \sum_{k=1}^{K} \left(\beta_{1k}^{(ij)} \sin(\frac{2\pi kt}{L}) + \beta_{2k}^{(ij)} \cos(\frac{2\pi kt}{L}) \right)$$

for the off-diagonal elements ($i \neq j$) of the transition probability matrix. This is relevant for modeling e.g. diurnal variation and the flexibility can be increased by adding smaller frequencies (i.e. increasing K).

Usage

```
tpm_p(
  tod = 1:24,
  L = 24,
  beta,
  degree = 1,
  Z = NULL,
  byrow = FALSE,
  ad = NULL,
  report = TRUE
)
```

Arguments

tod	equidistant sequence of a cyclic variable For time of day and e.g. half-hourly data, this could be 1, ..., L and L = 48, or 0.5, 1, 1.5, ..., 24 and L = 24.
L	length of one full cycle, on the scale of tod
beta	matrix of coefficients for the off-diagonal elements of the transition probability matrix Needs to be of dimension `c(nStates*(nStates-1), 2*degree+1)`, where the first column contains the intercepts.
degree	degree of the trigonometric link function For each additional degree, one sine and one cosine frequency are added.
Z	pre-calculated design matrix (excluding intercept column) Defaults to `NULL` if trigonometric link should be calculated. From an efficiency perspective, `Z` should be pre-calculated within the likelihood function, as the basis expansion should not be redundantly calculated. This can be done by using `trigBasisExp`.
byrow	logical indicating if each transition probability matrix should be filled by row Defaults to `FALSE`, but should be set to `TRUE` if one wants to work with a matrix of `beta` parameters returned by popular HMM packages like `moveHMM`, `momentuHMM`, or `hmmTMB`.
ad	optional logical, indicating whether automatic differentiation with RTMB should be used. By default, the function determines this itself.
report	logical, indicating whether the coefficient matrix `beta` should be reported from the fitted model. Defaults to `TRUE`, but only works if `ad = TRUE`.

Details

Note that using this function inside the negative log-likelihood function is convenient, but it performs the basis expansion into sine and cosine terms each time it is called. As these do not change

during the optimisation, using tpm_g with a pre-calculated (by trigBasisExp) design matrix would be more efficient.

Value

array of transition probability matrices of dimension c(nStates, nStates, length(tod))

See Also

Other transition probability matrix functions: generator(), generator_g(), tpm(), tpm_ct(), tpm_emb(), tpm_emb_g(), tpm_g(), tpm_g2()

Examples

```
# hourly data
tod = seq(1, 24, by = 1)
L = 24
beta = matrix(c(-1, 2, -1, -2, 1, -1), nrow = 2, byrow = TRUE)
Gamma = tpm_p(tod, L, beta, degree = 1)

# half-hourly data
## integer tod sequence
tod = seq(1, 48, by = 1)
L = 48
beta = matrix(c(-1, 2, -1, -2, 1, -1), nrow = 2, byrow = TRUE)
Gamma1 = tpm_p(tod, L, beta, degree = 1)

## equivalent specification
tod = seq(0.5, 24, by = 0.5)
L = 24
beta = matrix(c(-1, 2, -1, -2, 1, -1), nrow = 2, byrow = TRUE)
Gamma2 = tpm_p(tod, L, beta, degree = 1)

all(Gamma1 == Gamma2) # same result
```

---

tpm_phsmm *Builds all transition probability matrices of an periodic-HSMM-approximating HMM*

Description

Hidden semi-Markov models (HSMMs) are a flexible extension of HMMs. For direct numerical maximum likelihood estimation, HSMMs can be represented as HMMs on an enlarged state space (of size M) and with structured transition probabilities.

This function computes the transition matrices of a periodically inhomogeneos HSMMs.

Usage

```
tpm_phsmm(omega, dm, eps = 1e-10)
```

Arguments

omega	embedded transition probability matrix Either a matrix of dimension c(nStates, nStates) for homogeneous conditional transition probabilities (as computed by tpm_emb), or an array of dimension c(nStates, nStates, L) for inhomogeneous conditional transition probabilities (as computed by tpm_emb_g).
dm	state dwell-time distributions arranged in a list of length nStates Each list element needs to be a matrix of dimension c(L, N_i), where each row t is the (approximate) probability mass function of state i at time t.
eps	rounding value: If an entry of the transition probabily matrix is smaller, than it is rounded to zero. Usually, this should not be changed.

Value

list of dimension length L, containing sparse extended-state-space transition probability matrices of the approximating HMM for each time point of the cycle.

Examples

```
N = 2 # number of states
L = 24 # cycle length
# time-varying mean dwell times
Z = cosinor(1:L, period = L) # trigonometric basis functions design matrix
beta = matrix(c(2, 2, 0.1, -0.1, -0.2, 0.2), nrow = 2)
Lambda = exp(cbind(1, Z) %*% t(beta))
sizes = c(20, 20) # approximating chain with 40 states
# state dwell-time distributions
dm = lapply(1:N, function(i) sapply(1:sizes[i]-1, dpois, lambda = Lambda[,i]))

## homogeneous conditional transition probabilites
# diagonal elements are zero, rowsums are one
omega = matrix(c(0,1,1,0), nrow = N, byrow = TRUE)

# calculating extended-state-space t.p.m.s
Gamma = tpm_phsmm(omega, dm)

## inhomogeneous conditional transition probabilites
# omega can be an array
omega = array(omega, dim = c(N,N,L))

# calculating extended-state-space t.p.m.s
Gamma = tpm_phsmm(omega, dm)
```

---

tpm_phsmm2 *Build all transition probability matrices of an periodic-HSMM-approximating HMM*

Description

Hidden semi-Markov models (HSMMs) are a flexible extension of HMMs. For direct numerical maximum likelihood estimation, HSMMs can be represented as HMMs on an enlarged state space (of size M) and with structured transition probabilities. This function computes the transition matrices of a periodically inhomogeneos HSMMs.

Usage

```
tpm_phsmm2(omega, dm, eps = 1e-10)
```

Arguments

omega	embedded transition probability matrix Either a matrix of dimension c(N,N) for homogeneous conditional transition probabilities, or an array of dimension c(N,N,L) for inhomogeneous conditional transition probabilities.
dm	state dwell-time distributions arranged in a list of length(N) Each list element needs to be a matrix of dimension c(L, N_i), where each row t is the (approximate) probability mass function of state i at time t.
eps	rounding value: If an entry of the transition probabily matrix is smaller, than it is rounded to zero.

Value

array of dimension c(N,N,L), containing the extended-state-space transition probability matrices of the approximating HMM for each time point of the cycle.

Examples

```
N = 3
L = 24
# time-varying mean dwell times
Lambda = exp(matrix(rnorm(L*N, 2, 0.5), nrow = L))
sizes = c(25, 25, 25) # approximating chain with 75 states
# state dwell-time distributions
dm = list()
for(i in 1:3){
  dmi = matrix(nrow = L, ncol = sizes[i])
  for(t in 1:L){
    dmi[t,] = dpois(1:sizes[i]-1, Lambda[t,i])
  }
  dm[[i]] = dmi
}

## homogeneous conditional transition probabilites
# diagonal elements are zero, rowsums are one
omega = matrix(c(0,0.5,0.5,0.2,0,0.8,0.7,0.3,0), nrow = N, byrow = TRUE)

# calculating extended-state-space t.p.m.s
Gamma = tpm_phsmm(omega, dm)
```

```
## inhomogeneous conditional transition probabilites
# omega can be an array
omega = array(rep(omega,L), dim = c(N,N,L))
omega[1,,4] = c(0, 0.2, 0.8) # small change for inhomogeneity

# calculating extended-state-space t.p.m.s
Gamma = tpm_phsmm(omega, dm)
```

`tpm_thinned` *Compute the transition probability matrix of a thinned periodically inhomogeneous Markov chain.*

Description

If the transition probability matrix of an inhomogeneous Markov chain varies only periodically (with period length L), it converges to a so-called periodically stationary distribution. This happens, because the thinned Markov chain, which has a full cycle as each time step, has homogeneous transition probability matrix

$$\Gamma_t = \Gamma^{(t)}\Gamma^{(t+1)}\dots\Gamma^{(t+L-1)}$$

for all $t = 1, \dots, L$. This function calculates the matrix above efficiently as a preliminery step to calculating the periodically stationary distribution.

Usage

```
tpm_thinned(Gamma, t)
```

Arguments

`Gamma`	array of transition probability matrices of dimension `c(nStates, nStates, L)`.
`t`	integer index of the time point in the cycle, for which to calculate the thinned transition probility matrix

Value

thinned transition probabilty matrix of dimension `c(nStates, nStates)`

Examples

```
# setting parameters for trigonometric link
beta = matrix(c(-1, -2, 2, -1, 2, -4), nrow = 2, byrow = TRUE)
# calculating periodically varying t.p.m. array (of length 24 here)
Gamma = tpm_p(beta = beta)
# calculating t.p.m. of thinned Markov chain
tpm_thinned(Gamma, 4)
```

trex *T-Rex Movement Data*

Description

Hourly step lengths and turning angles of a Tyrannosaurus rex, living 66 million years ago.

Usage

```
trex
```

Format

A data frame with 10.000 rows and 4 variables:

tod time of day variable ranging from 1 to 24

step hourly step lengths in kilometres

angle hourly turning angles in radians

state hidden state variable

Source

Generated for example purposes.

trigBasisExp *Compute the design matrix for a trigonometric basis expansion*

Description

Given a periodically varying variable such as time of day or day of year and the associated cycle length, this function performs a basis expansion to efficiently calculate a linear predictor of the form

$$\eta^{(t)} = \beta_0 + \sum_{k=1}^{K} \big(\beta_{1k} \sin(\frac{2\pi kt}{L}) + \beta_{2k} \cos(\frac{2\pi kt}{L})\big).$$

This is relevant for modeling e.g. diurnal variation and the flexibility can be increased by adding smaller frequencies (i.e. increasing K).

Usage

```
trigBasisExp(tod, L = 24, degree = 1)
```

Arguments

tod	equidistant sequence of a cyclic variable For time of day and e.g. half-hourly data, this could be 1, ..., L and L = 48, or 0.5, 1, 1.5, ..., 24 and L = 24.
L	length of one cycle on the scale of the time variable. For time of day, this would be 24.
degree	degree K of the trigonometric link above. Increasing K increases the flexibility.

Value

design matrix (without intercept column), ordered as sin1, cos1, sin2, cos2, ...

Examples

```
# no examples
```

---

viterbi *Viterbi algorithm for state decoding in HMMs*

Description

The Viterbi algorithm decodes the most probable state sequence of an HMM.

Usage

```
viterbi(delta, Gamma, allprobs, trackID = NULL, mod = NULL)
```

Arguments

delta	initial distribution; either • a vector of length nStates, or • a matrix of dimension c(nTracks, nStates) if trackID is provided
Gamma	transition probability matrix; either • a matrix of dimension c(nStates, nStates), • an array of dimension c(nStates, nStates, nTracks) if trackID is provided, or • an array of dimension c(nStates, nStates, nObs) for time-varying transition probabilities, in which case viterbi_g is called internally
allprobs	matrix of state-dependent probabilities or density values of dimension c(nObs, nStates)
trackID	optional vector of length nObs containing nTracks unique IDs that separate tracks
mod	optional model object containing delta, Gamma, allprobs, and optionally trackID. When using RTMB::MakeADFun or qreml with forward in the likelihood, these are reported automatically after model fitting and the object returned by RTMB::report() or qreml can be passed directly.

Value

vector of decoded states of length nObs

See Also

Other decoding functions: stateprobs(), stateprobs_g(), stateprobs_p(), viterbi_g(), viterbi_p()

Examples

```
delta = c(0.5, 0.5)
Gamma = matrix(c(0.9, 0.1, 0.2, 0.8), nrow = 2, byrow = TRUE)
allprobs = matrix(runif(200), nrow = 100, ncol = 2)
states = viterbi(delta, Gamma, allprobs)
```

---

viterbi_g *Viterbi algorithm for state decoding in inhomogeneous HMMs*

Description

The Viterbi algorithm decodes the most probable state sequence of an HMM.

Usage

```
viterbi_g(delta, Gamma, allprobs, trackID = NULL, mod = NULL)
```

Arguments

delta	initial distribution; either • a vector of length nStates, or • a matrix of dimension c(nTracks, nStates) if trackID is provided
Gamma	array of transition probability matrices of dimension c(nStates, nStates, nObs), where the first slice of each track is ignored as there is no transition into the start of a track. For a single track, an array of dimension c(nStates, nStates, nObs-1) is also accepted.
allprobs	matrix of state-dependent probabilities or density values of dimension c(nObs, nStates)
trackID	optional vector of length nObs containing nTracks unique IDs that separate tracks
mod	optional model object containing delta, Gamma, allprobs, and optionally trackID. When using RTMB::MakeADFun or qreml with forward_g in the likelihood, these are reported automatically after model fitting and the object returned by RTMB::report() or qreml can be passed directly.

Value

vector of decoded states of length nObs

See Also

Other decoding functions: stateprobs(), stateprobs_g(), stateprobs_p(), viterbi(), viterbi_p()

Examples

```
delta = c(0.5, 0.5)
Gamma = tpm_g(runif(10), matrix(c(-2,-2,1,-1), nrow = 2))
allprobs = matrix(runif(20), nrow = 10, ncol = 2)
states = viterbi_g(delta, Gamma[,,-1], allprobs)
```

---

viterbi_p *Viterbi algorithm for state decoding in periodically inhomogeneous HMMs*

Description

The Viterbi algorithm allows one to decode the most probable state sequence of an HMM.

Usage

```
viterbi_p(delta, Gamma, allprobs, tod, trackID = NULL, mod = NULL)
```

Arguments

delta	initial distribution of length N, or matrix of dimension c(k,N) for k independent tracks, if trackID is provided This could e.g. be the periodically stationary distribution (for each track).
Gamma	array of transition probability matrices for each time point in the cycle of dimension c(N,N,L), where L is the length of the cycle
allprobs	matrix of state-dependent probabilities/ density values of dimension c(n, N)
tod	(Integer valued) variable for cycle indexing in 1, ..., L, mapping the data index to a generalised time of day (length n) For half-hourly data L = 48. It could, however, also be day of year for daily data and L = 365.
trackID	optional vector of k track IDs, if multiple tracks need to be decoded separately
mod	optional model object containing initial distribution delta, transition probability matrix Gamma, matrix of state-dependent probabilities allprobs, and potentially a trackID variable If you are using automatic differentiation either with RTMB::MakeADFun or qreml and include forward_p in your likelihood function, the objects needed for state decoding are automatically reported after model fitting. Hence, you can pass the model object obtained from running RTMB::report() or from qreml directly to this function.

Value

vector of decoded states of length n

See Also

Other decoding functions: stateprobs(), stateprobs_g(), stateprobs_p(), viterbi(), viterbi_g()

Examples

```
delta = c(0.5, 0.5)
beta = matrix(c(-2, 1, -1,
                -2, -1, 1), nrow = 2, byrow = TRUE)
Gamma = tpm_p(1:24, 24, beta)

tod = rep(1:24, 5)
n = length(tod)

allprobs = matrix(runif(2*n), nrow = n, ncol = 2)
states = viterbi_p(delta, Gamma, allprobs, tod)
```

---

vm *Von Mises distribution*

Description

Density, distribution function and random generation for the von Mises distribution.

Usage

```
dvm(x, mu = 0, kappa = 1, log = FALSE)

pvm(q, mu = 0, kappa = 1, from = NULL, tol = 1e-20)

rvm(n, mu = 0, kappa = 1, wrap = TRUE)
```

Arguments

x, q	vector of angles measured in radians at which to evaluate the density function.
mu	mean direction of the distribution measured in radians.
kappa	non-negative numeric value for the concentration parameter of the distribution.
log	logical; if TRUE, densities are returned on the log scale.
from	value from which the integration for CDF starts. If NULL, is set to mu - pi.
tol	the precision in evaluating the distribution function
n	number of observations. If length(n) > 1, the length is taken to be the number required.
wrap	logical; if TRUE, generated angles are wrapped to the interval [-pi, pi].

Details

This implementation of dvm allows for automatic differentiation with RTMB. rvm and pvm are simply wrappers of the corresponding functions from circular.

Value

dvm gives the density, pvm gives the distribution function, and rvm generates random deviates.

Examples

```
set.seed(1)
x = rvm(10, 0, 1)
d = dvm(x, 0, 1)
p = pvm(x, 0, 1)
```

---

wrpcauchy *Wrapped Cauchy distribution*

Description

Density and random generation for the wrapped Cauchy distribution.

Usage

```
dwrpcauchy(x, mu = 0, rho, log = FALSE)

rwrpcauchy(n, mu = 0, rho, wrap = TRUE)
```

Arguments

x	vector of angles measured in radians at which to evaluate the density function.
mu	mean direction of the distribution measured in radians.
rho	concentration parameter of the distribution, must be in the interval from 0 to 1.
log	logical; if TRUE, densities are returned on the log scale.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
wrap	logical; if TRUE, generated angles are wrapped to the interval [-pi, pi].

Details

This implementation of dwrpcauchy allows for automatic differentiation with RTMB. rwrpcauchy is simply a wrapper for rwrappedcauchy imported from circular.

Value

dwrpcauchy gives the density and rwrpcauchy generates random deviates.

Examples

```
set.seed(1)
x = rwrpcauchy(10, 0, 0.3)
d = dwrpcauchy(x, 0, 0.3)
```

`zero_inflate` *Zero-inflated density constructer*

Description

Constructs a zero-inflated density function from a given probability density function

Usage

```
zero_inflate(dist, discrete = NULL)
```

Arguments

`dist`	either a probability density function or a probability mass function
`discrete`	logical; if `TRUE`, the density for `x = 0` will be `zeroprob + (1-zeroprob) * dist(0, ...)`. Otherwise it will just be `zeroprob`. In standard cases, this will be determined automatically. For non-standard cases, set this to `TRUE` or `FALSE` depending on the type of `dist`. See details.

Details

The definition of zero-inflation is different for discrete and continuous distributions. For discrete distributions with p.m.f. f and zero-inflation probability p, we have

$$\Pr(X = 0) = p + (1 - p) \cdot f(0),$$

and

$$\Pr(X = x) = (1 - p) \cdot f(x), \quad x > 0.$$

For continuous distributions with p.d.f. f, we have

$$f_{\text{zinfl}}(x) = p \cdot \delta_0(x) + (1 - p) \cdot f(x),$$

where δ_0 is the Dirac delta function at zero.

Value

zero-inflated density function with first argument `x`, second argument `zeroprob`, and additional arguments `...` that will be passed to `dist`.

Examples

```
dzinorm <- zero_inflate(dnorm)
dzinorm(c(NA, 0, 2), 0.5, mean = 1, sd = 1)

zipois <- zero_inflate(dpois)
zipois(c(NA, 0, 1), 0.5, 1)
```

Index