Package: tidyabc (via r-universe)

May 23, 2026

Title Approximate Bayesian Computing with Tidy Data

Version 0.0.1

Description A flexible framework for Approximate Bayesian Computation (ABC) that integrates with the tidyverse. Define simulation models and summary statistics as standard R functions, use 'dist_fns' to represent prior and posterior distributions, and perform inference via rejection sampling, Sequential Monte Carlo (SMC), or Adaptive ABC. The package provides tools for diagnostics, visualization, and convergence assessment, enabling reproducible Bayesian inference for complex models with intractable likelihoods.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE, packages = ``pkgtools")

RoxygenNote 7.3.3.9007

Config/Needs/build terminological/pkgtools

Suggests progressr, testthat, lifecycle, withr, parallel, mirai (>= 2.5.1), future, rmarkdown

Config/testthat/edition 3

Config/Needs/website rmarkdown

Imports dplyr, furrr, ggplot2, knitr, pillar, purrr, rlang, scales, splines, stats, stringr, systemfonts, utils, carrier, magrittr, Matrix, cli, locfit, mvtnorm, patchwork, signal, tibble, tidyr, ragg, fitdistrplus

VignetteBuilder knitr

URL https://ai4ci.github.io/tidyabc, https://github.com/ai4ci/tidyabc

BugReports https://github.com/ai4ci/tidyabc/issues

Depends R (>= 3.5)

LazyData true

Config/pak/sysreqs libfontconfig1-dev libfreetype6-dev libfribidi-dev libharfbuzz-dev libicu-dev libjpeg-dev libpng-dev libtiff-dev libwebp-dev

Repository https://ai4ci.r-universe.dev

Date/Publication 2025-11-24 20:25:01 UTC

RemoteUrl https://github.com/ai4ci/tidyabc

RemoteRef 0.0.1

RemoteSha 6d673cfed37e9321fce896a079e64f928a6c67b8

Contents

`abc_adaptive` *Perform ABC sequential adaptive fitting*

Description

This function will execute a simulation for a random selection of parameters. Based on the `acceptance_rate` it will reject a proportion of the results. The remaining results are weighted (using a kernel with a tolerance equivalent to half the acceptance rate). Empirical distributions are fitted to weighted parameter particles and from these proposals are generated for further waves by fresh sampling. Waves are executed until a maximum is reached or the results converge sufficiently that the changes between waves are small. A relatively small number of simulations may be attempted with a high acceptance rate, over multiple waves.

Usage

```
abc_adaptive(
  obsdata,
  priors_list,
  sim_fn,
  scorer_fn,
  n_sims,
  acceptance_rate,
  ...,
  max_time = 5 * 60,
  converged_fn = default_termination_fn(),
  obsscores = NULL,
  distance_method = "euclidean",
  seed = NULL,
  knots = NULL,
  parallel = FALSE,
  max_recover = 3,
  allow_continue = interactive(),
  debug_errors = FALSE,
  kernel = "epanechnikov",
  distfit = c("empirical", "analytical"),
```

```
  bw = 0.1,
  widen_by = 1.05,
  scoreweights = NULL,
  use_proposal_correlation = TRUE,
  ess_limit = c(200, n_sims * acceptance_rate)
)
```

Arguments

obsdata	The observational data. The data in this will typically be a named list, but could be anything, e.g. dataframe. It is the reference data that the simulation model is aiming to replicate.
priors_list	a named list of priors specified as a abc_prior S3 object (see priors()), this can include derived values as unnamed 2-sided formulae, where the LHS of the formula will be assigned to the value of the RHS, plus optionally a set of constraints as one sided formulae where the RHS of the formulae will resolve to a boolean value.
sim_fn	a user defined function that takes a set of parameters named the same as priors_list. It must return a simulated data set in the same format as obsdata, or that can be compared to simdata by scorer_fn. This function must not refer to global parameters, and will be automatically crated with carrier.
scorer_fn	a user supplied function that matches the following signature scorer_fn(simdata, obsdata, ...), i.e. it takes data in the format of simdata paired with the original obsdata and returns a named list of component scores per simulation. This function can make use of the calculate_*() set of functions to compare components of the simulation to the original data. This function must not refer to global parameters, and will be automatically crated with carrier. If this is a purrr style function then .x will refer to simulation output and .y to original observation data.
n_sims	The number of simulations to run per wave (for SMC and Adaptive) or overall (for Rejection). For rejection sampling a large number is recommended, for the others sma
acceptance_rate	What proportion of simulations to keep in ABC rejection or hard ABC parts of the algorithms.
...	must be empty
max_time	the maximum time in seconds to spend in ABC waves before admitting defeat. This time may not be all used if the algorithm converges.
converged_fn	a function that takes a summary and per_param input and generates a logical indicator that the function has converged
obsscores	Summary scores for the observational data. This will be a named list, and is equivalent to the output of scorer_fn, on the observed data. If not given typically it will be assumed to be all zeros.
distance_method	what metric is used to combine simscores and obsscores. One of "euclidean", "normalised", "manhattan", or "mahalanobis".

seed	an optional random seed
knots	the number of knots to model the CDF with. Optional, and will be typically inferred from the data size. Small numbers tend to work better if we expect the distribution to be unimodal.
parallel	parallelise the simulation? If this is set to true then the simulation step will be parallelised using furrr. For this to make any difference it must have been set up with the following: future::plan(future::multisession, workers = parallel::detectCores()-2)
max_recover	if the effective sample size of SMC or adaptive algorithms drops below 200, the algorithm will retry the wave with double the sample size to try and recover the shape of the distribution, up to a maximum of max_recover times.
allow_continue	if SMC or adaptive algorithms have not converged after max_time allow the algorithm to interactively prompt the user to continue.
debug_errors	Errors that crop up in sim_fn during a simulation due to anomolous value combinations are hard to debug. If this flag is set, whenever a sim_fn or scorer_fn throws an error an interactive debugging session is started with the failing parameter combinations. This is not compatible with running in parallel.
kernel	one of "epanechnikov" (default), "uniform", "triangular", "biweight", or "gaussian". The kernel defines how the distance metric translates into the importance weight that decides whether a given simulation and associated parameters should be rejected or held for the next round. All kernels except gaussian have a hard cut-off outside of which the probability of acceptance of a particle is zero. Use of gaussian kernels can result in poor convergence.
distfit	one of "empirical" or "analytical" determines what kind of distribution the ABC adaptive algorithm will fit for the posteriors.
bw	for Adaptive ABC data distributions are smoothed before modelling empirical CDF. Over smoothing can reduce convergence rate, under-smoothing may result in noisy posterior estimates, and appearance of local modes.
widen_by	change the dispersion of the empirical proposal distribution in ABC adaptive, preserving the median. This is akin to a nonlinear, heteroscedastic random walk in the quantile space, and can help address over-fitting or local modes in the ABC adaptive waves. widen_by is an odds ratio and describes how much further from the median any given part of the distribution is after transformation. E.g. if the median of a distribution is zero, and the widen_by is 2 then the 0.75 quantile will move to the position of the 0.9 quantile. The distribution will stay within the support of the prior. This is by default 1.05 which allows for some additional variability in proposals.
scoreweights	A named vector with names matching output of scorer_fn that defines the importance of this component of the scoring in the overall distance and weighting of any given simulation. This can be used to assign more weight on certain parts of the model output. For euclidean and manhattan distance methods these weights multiply the output of scorer_fn directly. For the other 2 distance methods some degree of normalisation is done first on the first wave scores to make different components have approximately the same relevance to the overall score.

`use_proposal_correlation`
: When calculating the weight of a particle the proposal correlation structure is available, to help determine how unusual or otherwise a particle is.

`ess_limit`
: a numeric vector of length 2 which for ABC adaptive, defines the limits which rate at which the algorithm will converge in terms of effective sample size. If for example the algorithm is converging too quickly and some high weight particles are dominating then the ESS will drop below the lower limit. In this case more particles will be accepted to try and offset this. On the other hand if the algorithm is converging too slowly low probability particles in proposal space are not filtered out quickly enough and this can lead to too much importance being given to unlikely proposals and wide bi-modal peaked posteriors.

Details

Performs the ABC Adaptive algorithm. This iterative method refines parameter estimates across waves by fitting empirical proposal distributions to the weighted posterior samples from the previous wave. Unlike ABC-SMC, which uses a fixed perturbation kernel, `abc_adaptive` constructs a new proposal distribution $Q_t(\theta)$ at each wave t.

1. **Initialization (Wave 1):** Parameters $\theta^{(i)}$ are sampled from the prior $P(\theta)$. Simulations are run, summary statistics $S_s^{(i)}$ are computed, and distances $d^{(i)} = d(S_s^{(i)}, S_o)$ are calculated. A tolerance threshold ϵ_1 is set as the $\alpha =$ `acceptance_rate` quantile of these distances. Unnormalized weights $\tilde{w}_1^{(i)}$ are calculated using a kernel $K_{\epsilon_1}(d^{(i)})$.
2. **Subsequent Waves ($t > 1$):**
   - **Proposal Generation:** An empirical joint proposal distribution $Q_t(\theta)$ is constructed from the weighted posterior sample $\{(\theta_{t-1}^{(i)}, w_{t-1}^{(i)})\}$ of the previous wave. This is done by fitting marginal empirical distributions $Q_{t,j}(\theta_j)$ to each parameter θ_j, using the `empirical()` function with the prior $P_j(\theta_j)$ as a link to enforce support constraints. These marginals are assumed independent, but their weighted correlation matrix R_t is retained as an attribute and used to induce dependence in the MVN sampling space. New proposals $\theta_t^{(i)}$ are generated by:
     - Sampling a vector $Z \sim \mathcal{N}(0, R_t)$ in a correlated standard normal space.
     - Mapping each component Z_j to uniform space: $U_j = \Phi(Z_j)$.
     - Mapping to the parameter space using the empirical quantile functions: $\theta_{t,j}^{(i)} = Q_{t,j}^{-1}(U_j)$.
   - **Simulation and Weighting:** Simulations are run for the new proposals. Distances $d_t^{(i)}$ are computed and the tolerance ϵ_t is set as the α-quantile of the current wave's distances. The unnormalized weight for particle i in wave t is calculated as:

$$\tilde{w}_t^{(i)} = \frac{P(\theta_t^{(i)}) K_{\epsilon_t}(d_t^{(i)})}{Q_t(\theta_t^{(i)})}$$

   where $P(\theta_t^{(i)}) = \prod_j P_j(\theta_{t,j}^{(i)})$ is the prior density (assuming independence), K_{ϵ_t} is the ABC kernel, and $Q_t(\theta_t^{(i)}) = \prod_j Q_{t,j}(\theta_{t,j}^{(i)})$ is the empirical proposal density (also assuming independence for density calculation, consistent with the marginal fitting). The correlation structure is handled in the sampling process, and optionally in the density evaluation.

3. **Normalization:** Weights $w_t^{(i)}$ are normalized to sum to one. The algorithm includes a recovery mechanism: if the Effective Sample Size (ESS) falls below a threshold (e.g., 200), the acceptance rate is increased (i.e., ϵ_t is made larger) to accept more particles and improve the ESS.
4. **Termination:** The process repeats until a maximum time is reached or convergence criteria based on parameter stability and credible interval contraction are met.

Value

an S3 object of class `abc_fit` this contains the following:

- type: the type of ABC algorithm
- iterations: number of completed iterations
- converged: boolean - did the result meet convergence criteria
- waves: a list of dataframes of wave convergence metrics
- summary: a dataframe with the summary of the parameter fits after each wave.
- priors: the priors for the fit as a `abc_prior` S3 object
- posteriors: the final wave posteriors

Examples

```
fit = abc_adaptive(
  obsdata = example_obsdata(),
  priors_list = example_priors_list(),
  sim_fn = example_sim_fn,
  scorer_fn = example_scorer_fn,
  n_sims = 1000,
  acceptance_rate = 0.25,
  max_time = 5, # 5 seconds to fit within examples limit
  parallel = FALSE,
  allow_continue = FALSE
)

summary(fit)
```

`abc_fit` `abc_fit` *S3 class*

Description

A class holding the output of a single ABC model fitting, either as a single ABC rejection round or after a set of SMC waves.

Usage

```
new_abc_fit(type, iterations, converged, priors_list, wave_df, summ_df, sim_df)

## S3 method for class 'abc_fit'
format(x, ...)

## S3 method for class 'abc_fit'
summary(object, ..., truth = NULL)

tidy.abc_fit(x, ...)

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

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

Arguments

type	the type of ABC algorithm
iterations	number of completed iterations
converged	boolean - did the result meet convergence criteria
priors_list	a named list of priors specified as a abc_prior S3 object (see priors()), this can include derived values as unnamed 2-sided formulae, where the LHS of the formula will be assigned to the value of the RHS, plus optionally a set of constraints as one sided formulae where the RHS of the formulae will resolve to a boolean value.
wave_df	a list of dataframes of wave convergence metrics
summ_df	the summary of the parameter fits after each wave.
sim_df	the final wave posteriors
x	a abc_fit object as output by the abc_XXX functions
...	passed on to methods Named arguments passed on to plot.dist_fns_list mapping override default aesthetics with name, id or group steps resolution of the plot tail the minimum tail probability to plot plot_quantiles by default the quantiles of the distribution are plotted over the density sometimes this makes it hard to read smooth by default some additional smoothing is used to cover up small discontinuities in the PDF.
object	a abc_fit object as output by the abc_XXX functions
truth	a named numeric vector of known parameter values

Value

an S3 object of class `abc_fit` this contains the following:

- type: the type of ABC algorithm
- iterations: number of completed iterations
- converged: boolean - did the result meet convergence criteria
- waves: a list of dataframes of wave convergence metrics
- summary: a dataframe with the summary of the parameter fits after each wave.
- priors: the priors for the fit as a `abc_prior` S3 object
- posteriors: the final wave posteriors

Methods (by generic)

- `format(abc_fit)`: S3 format method
- `summary(abc_fit)`: S3 summary method
- `print(abc_fit)`: S3 print method
- `plot(abc_fit)`: S3 plot method

Functions

- `new_abc_fit()`: Create a `abc_fit` object
- `tidy.abc_fit()`: S3 summary method

`abc_prior` `abc_prior` *S3 class*

Description

`abc_prior` S3 class

Usage

```
new_abc_prior(.dists, .constraints = list(), .derived = list(), .cor = NULL)

as.abc_prior(x, ...)

is.abc_prior(x, ...)

## S3 method for class 'abc_prior'
format(x, ...)

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

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

Arguments

.dists	distribution functions as a named list of S3 dist_fns objects
.constraints	a list of one sided formulae the result each of which should evaluate to a boolean when compared against the names of the priors and derived values.
.derived	a list of two sided formulae. The RHS refer to the priors, and the LHS as a name to derive.
.cor	(optional) a correlation matrix for the priors
x	an abc_prior S3 object
...	passed on to methods

Value

an S3 object of class abc_prior which contains

- a list of dist_fns
- a cor attribute describing their correlation
- a derived attribute describing derive values
- a constraints attribute listing the constraints
- a params attribute listing the names of the parameters

Methods (by generic)

- format(abc_prior): Format an abc_prior
- print(abc_prior): Print an abc_prior
- plot(abc_prior): Plot an abc_prior

Functions

- new_abc_prior(): Create a new prior
- as.abc_prior(): Create a prior from a named list of dist_fns
- is.abc_prior(): Test is an abc_prior

Unit tests

```
p = new_abc_prior(
  .dists = list(
    mean = as.dist_fns("norm",4,2),
    sd = as.dist_fns("gamma",2)
  ),
  .derived = list(
    shape ~ mean^2 / sd^2,
    rate ~ mean / sd^2
  ),
  .constraints = list(
```

```
    ~ mean > sd
  )
)

testthat::expect_equal(
  format(p),
  "Parameters: \n* mean: norm(mean = 4, sd = 2)\n* sd: gamma(shape = 2, rate = 1)\nConstraints:\n* mean >
)
```

---

abc_rejection *Perfom simple ABC rejection algorithm*

Description

This function will execute a simulation for a random selection of parameters and identify the best matching `acceptance_rate` percent, as defined by the summary distance metric. A large number of simulations and a low acceptance rate are best here.

Usage

```
abc_rejection(
  obsdata,
  priors_list,
  sim_fn,
  scorer_fn,
  n_sims,
  acceptance_rate,
  ...,
  converged_fn = default_termination_fn(),
  obsscores = NULL,
  distance_method = "euclidean",
  keep_simulations = FALSE,
  seed = NULL,
  parallel = FALSE,
  debug_errors = FALSE,
  kernel = "epanechnikov",
  scoreweights = NULL
)
```

Arguments

`obsdata`	The observational data. The data in this will typically be a named list, but could be anything, e.g. dataframe. It is the reference data that the simulation model is aiming to replicate.

priors_list — a named list of priors specified as a abc_prior S3 object (see priors()), this can include derived values as unnamed 2-sided formulae, where the LHS of the formula will be assigned to the value of the RHS, plus optionally a set of constraints as one sided formulae where the RHS of the formulae will resolve to a boolean value.

sim_fn — a user defined function that takes a set of parameters named the same as priors_list. It must return a simulated data set in the same format as obsdata, or that can be compared to simdata by scorer_fn. This function must not refer to global parameters, and will be automatically crated with carrier.

scorer_fn — a user supplied function that matches the following signature scorer_fn(simdata, obsdata, ...), i.e. it takes data in the format of simdata paired with the original obsdata and returns a named list of component scores per simulation. This function can make use of the calculate_*() set of functions to compare components of the simulation to the original data. This function must not refer to global parameters, and will be automatically crated with carrier. If this is a purrr style function then .x will refer to simulation output and .y to original observation data.

n_sims — The number of simulations to run per wave (for SMC and Adaptive) or overall (for Rejection). For rejection sampling a large number is recommended, for the others sma

acceptance_rate — What proportion of simulations to keep in ABC rejection or hard ABC parts of the algorithms.

... — must be empty

converged_fn — a function that takes a summary and per_param input and generates a logical indicator that the function has converged

obsscores — Summary scores for the observational data. This will be a named list, and is equivalent to the output of scorer_fn, on the observed data. If not given typically it will be assumed to be all zeros.

distance_method — what metric is used to combine simscores and obsscores. One of "euclidean", "normalised", "manhattan", or "mahalanobis".

keep_simulations — keep the individual simulation results in the output of an ABC workflow. This can have large implications for the size of the result. It may also not be what you want and it is probably worth considering resampling the posteriors rather than keeping the simulations.

seed — an optional random seed

parallel — parallelise the simulation? If this is set to true then the simulation step will be parallelised using furrr. For this to make any difference it must have been set up with the following: future::plan(future::multisession, workers = parallel::detectCores()-2)

debug_errors — Errors that crop up in sim_fn during a simulation due to anomolous value combinations are hard to debug. If this flag is set, whenever a sim_fn or scorer_fn throws an error an interactive debugging session is started with the failing parameter combinations. This is not compatible with running in parallel.

`kernel` one of `"epanechnikov"` (default), `"uniform"`, `"triangular"`, `"biweight"`, or `"gaussian"`. The kernel defines how the distance metric translates into the importance weight that decides whether a given simulation and associated parameters should be rejected or held for the next round. All kernels except `gaussian` have a hard cut-off outside of which the probability of acceptance of a particle is zero. Use of `gaussian` kernels can result in poor convergence.

`scoreweights` A named vector with names matching output of `scorer_fn` that defines the importance of this component of the scoring in the overall distance and weighting of any given simulation. This can be used to assign more weight on certain parts of the model output. For `euclidean` and `manhattan` distance methods these weights multiply the output of `scorer_fn` directly. For the other 2 distance methods some degree of normalisation is done first on the first wave scores to make different components have approximately the same relevance to the overall score.

Details

Parameters $\theta^{(i)}$ are sampled independently from the prior distribution $P(\theta)$ for $i = 1, \ldots, n_{\text{sims}}$. For each $\theta^{(i)}$, simulated data $D_s^{(i)} = M(\theta^{(i)})$ is generated via the simulator function `sim_fn`, and a vector of summary statistics $S_s^{(i)} = \texttt{scorer\_fn}(D_s^{(i)})$ is computed and compared to the observed summary statistics $S_o = \texttt{scorer\_fn}(D_o)$.

A distance metric $d^{(i)} = d(S_s^{(i)}, S_o)$ is computed. By default, this is the Euclidean distance:

$$d^{(i)} = \left\| W \circ (S_s^{(i)} - S_o) \right\|_2,$$

where W is a vector of optional summary statistic weights (`scoreweights`), and $\circ$ denotes elementwise multiplication. Other supported metrics include Manhattan (ℓ_1) and Mahalanobis distance (using the empirical covariance from the first wave).

The tolerance threshold ϵ is set to the $\alpha = \texttt{acceptance\_rate}$ quantile of the distances $\{d^{(i)}\}_{i=1}^{n_{\text{sims}}}$:

$$\epsilon = \text{quantile}\big(\{d^{(i)}\}, \alpha\big).$$

Unnormalized ABC weights are then assigned using a kernel function $K_\epsilon(\cdot)$:

$$\tilde{w}^{(i)} = K_\epsilon\big(d^{(i)}\big),$$

where $K_\epsilon(d)$ is one of the kernels defined in `kernels.R` (e.g., Epanechnikov: $K_\epsilon(d) = \frac{3}{4\epsilon}(1 - d^2/\epsilon^2)\,\mathbb{I}(d \le \epsilon)$). These weights are then transformed via a logistic ("expit") function and normalized to sum to one:

$$w^{(i)} = \frac{\text{expit}(\log \tilde{w}^{(i)})}{\sum_j \text{expit}(\log \tilde{w}^{(j)})} = \frac{\tilde{w}^{(i)}/(1 + \tilde{w}^{(i)})}{\sum_j \tilde{w}^{(j)}/(1 + \tilde{w}^{(j)})}.$$

The resulting weighted sample $\{(\theta^{(i)}, w^{(i)})\}$ approximates the ABC posterior distribution $P_\epsilon(\theta \mid D_o)$.

Value

an S3 object of class `abc_fit` this contains the following:

- type: the type of ABC algorithm
- iterations: number of completed iterations
- converged: boolean - did the result meet convergence criteria
- waves: a list of dataframes of wave convergence metrics
- summary: a dataframe with the summary of the parameter fits after each wave.
- priors: the priors for the fit as a `abc_prior` S3 object
- posteriors: the final wave posteriors

Examples

```
fit = abc_rejection(
  example_obsdata(),
  example_priors_list(),
  example_sim_fn,
  example_scorer_fn,
  n_sims = 10000,
  acceptance_rate = 0.01
)

summary(fit)
```

---

abc_smc *Perform ABC sequential Monte Carlo fitting*

Description

This function will execute a simulation for a random selection of parameters. Based on the `acceptance_rate` it will reject a proportion of the results. The remaining results are weighted (using a kernel with a tolerance equivalent to half the acceptance rate). Weighted parameter particles generate proposals for further waves but a particle perturbation. Waves are executed until a maximum is reached or the results converge sufficiently that the changes between waves are small. A relatively small number of simulations may be attempted with a high acceptance rate, over multiple waves.

Usage

```
abc_smc(
  obsdata,
  priors_list,
  sim_fn,
  scorer_fn,
  n_sims,
  acceptance_rate,
```

```
  ...,
  max_time = 5 * 60,
  converged_fn = default_termination_fn(),
  obsscores = NULL,
  distance_method = "euclidean",
  seed = NULL,
  parallel = FALSE,
  allow_continue = interactive(),
  debug_errors = FALSE,
  kernel = "epanechnikov",
  scoreweights = NULL
)
```

Arguments

obsdata	The observational data. The data in this will typically be a named list, but could be anything, e.g. dataframe. It is the reference data that the simulation model is aiming to replicate.
priors_list	a named list of priors specified as a abc_prior S3 object (see priors()), this can include derived values as unnamed 2-sided formulae, where the LHS of the formula will be assigned to the value of the RHS, plus optionally a set of constraints as one sided formulae where the RHS of the formulae will resolve to a boolean value.
sim_fn	a user defined function that takes a set of parameters named the same as priors_list. It must return a simulated data set in the same format as obsdata, or that can be compared to simdata by scorer_fn. This function must not refer to global parameters, and will be automatically crated with carrier.
scorer_fn	a user supplied function that matches the following signature scorer_fn(simdata, obsdata, ...), i.e. it takes data in the format of simdata paired with the original obsdata and returns a named list of component scores per simulation. This function can make use of the calculate_*() set of functions to compare components of the simulation to the original data. This function must not refer to global parameters, and will be automatically crated with carrier. If this is a purrr style function then .x will refer to simulation output and .y to original observation data.
n_sims	The number of simulations to run per wave (for SMC and Adaptive) or overall (for Rejection). For rejection sampling a large number is recommended, for the others sma
acceptance_rate	What proportion of simulations to keep in ABC rejection or hard ABC parts of the algorithms.
...	must be empty
max_time	the maximum time in seconds to spend in ABC waves before admitting defeat. This time may not be all used if the algorithm converges.
converged_fn	a function that takes a summary and per_param input and generates a logical indicator that the function has converged

obsscores — Summary scores for the observational data. This will be a named list, and is equivalent to the output of scorer_fn, on the observed data. If not given typically it will be assumed to be all zeros.

distance_method — what metric is used to combine simscores and obsscores. One of "euclidean", "normalised", "manhattan", or "mahalanobis".

seed — an optional random seed

parallel — parallelise the simulation? If this is set to true then the simulation step will be parallelised using furrr. For this to make any difference it must have been set up with the following: future::plan(future::multisession, workers = parallel::detectCores()-2)

allow_continue — if SMC or adaptive algorithms have not converged after max_time allow the algorithm to interactively prompt the user to continue.

debug_errors — Errors that crop up in sim_fn during a simulation due to anomolous value combinations are hard to debug. If this flag is set, whenever a sim_fn or scorer_fn throws an error an interactive debugging session is started with the failing parameter combinations. This is not compatible with running in parallel.

kernel — one of "epanechnikov" (default), "uniform", "triangular", "biweight", or "gaussian". The kernel defines how the distance metric translates into the importance weight that decides whether a given simulation and associated parameters should be rejected or held for the next round. All kernels except gaussian have a hard cut-off outside of which the probability of acceptance of a particle is zero. Use of gaussian kernels can result in poor convergence.

scoreweights — A named vector with names matching output of scorer_fn that defines the importance of this component of the scoring in the overall distance and weighting of any given simulation. This can be used to assign more weight on certain parts of the model output. For euclidean and manhattan distance methods these weights multiply the output of scorer_fn directly. For the other 2 distance methods some degree of normalisation is done first on the first wave scores to make different components have approximately the same relevance to the overall score.

Details

Performs the ABC Sequential Monte Carlo (SMC) algorithm. This iterative method refines parameter estimates across multiple waves.

1. **Initialization (Wave 1):** Parameters $\theta^{(i)}$ are sampled from the prior $P(\theta)$. Simulations $D_s^{(i)} = M(\theta^{(i)})$ are run, summaries $S_s^{(i)}$ are computed, and distances $d^{(i)} = d(S_s^{(i)}, S_o)$ are calculated. A tolerance threshold ϵ_1 is set as the $\alpha =$ acceptance_rate quantile of these initial distances. Unnormalized weights $\tilde{w}_1^{(i)}$ are calculated using a kernel $K_{\epsilon_1}(d^{(i)})$.
2. **Subsequent Waves ($t > 1$):**
   - **Proposal Generation:** A particle $\theta_{t-1}^{(j)}$ is selected from the previous wave's accepted particles with probability proportional to its weight $w_{t-1}^{(j)}$. The particle is then perturbed in a transformed MVN space using a multivariate normal kernel with covariance $\Sigma_t = \frac{\kappa_t^2}{d}\text{Cov}_{w_{t-1}}(\theta_{t-1})$, where $\text{Cov}_{w_{t-1}}$ is the weighted covariance from wave $t-1$, d is

the parameter dimension, and κ_t is the `kernel_t` parameter. The new proposal $\theta_t^{(i)}$ is generated as:

$$\theta_t^{(i)} = \theta_{t-1}^{(j)} + \zeta, \quad \zeta \sim \mathcal{N}(0, \Sigma_t)$$

This proposal is mapped back to the original parameter space using the prior's copula transformation (from MVN space defined by prior CDFs).

- **Simulation and Weighting:** Simulations $D_s^{(i)} = M(\theta_t^{(i)})$ are run for the new proposals. Distances $d_t^{(i)}$ are computed. The tolerance ϵ_t is set as the α-quantile of distances from the *current* wave's simulations. The unnormalized weight for particle i in wave t is calculated as:

$$\tilde{w}_t^{(i)} = \frac{P(\theta_t^{(i)}) K_{\epsilon_t}(d_t^{(i)})}{q_t(\theta_t^{(i)})}$$

where $P(\theta_t^{(i)})$ is the prior density, K_{ϵ_t} is the ABC kernel, and $q_t(\theta_t^{(i)})$ is the proposal density from the previous wave's weighted particles (calculated using the perturbation kernel). This proposal density is computed as a weighted sum:

$$q_t(\theta_t^{(i)}) = \sum_j w_{t-1}^{(j)} \phi(\theta_t^{(i)}; \theta_{t-1}^{(j)}, \Sigma_t)$$

where $\phi(\cdot; \mu, \Sigma)$ is the PDF of a multivariate normal with mean μ and covariance Σ.

3. **Normalization:** Weights $w_t^{(i)}$ are normalized to sum to one. Particles with negligible weights are typically filtered out.
4. **Termination:** The process repeats until a maximum number of waves or time is reached, or convergence criteria are met based on changes in parameter estimates or effective sample size (ESS).

Value

an S3 object of class `abc_fit` this contains the following:

- type: the type of ABC algorithm
- iterations: number of completed iterations
- converged: boolean - did the result meet convergence criteria
- waves: a list of dataframes of wave convergence metrics
- summary: a dataframe with the summary of the parameter fits after each wave.
- priors: the priors for the fit as a `abc_prior` S3 object
- posteriors: the final wave posteriors

Examples

```
fit = abc_smc(
  obsdata = example_obsdata(),
  priors_list = example_priors_list(),
  sim_fn = example_sim_fn,
  scorer_fn = example_scorer_fn,
  n_sims = 1000,
  acceptance_rate = 0.25,
```

```
  max_time = 5, # 5 seconds to fit within examples limit
  parallel = FALSE,
  allow_continue = FALSE
)

summary(fit)
```

`as.dist_fns.character` *Create a* `dist_fns` *S3 object*

Description

A class wrapping a single (or set) of parametrised distributions and provides access to the quantile, cumulative probability and random functions of that specific distribution. Parametrisation is handled on construction.

Usage

```
## S3 method for class 'character'
as.dist_fns(x, ...)

## S3 method for class '`function`'
as.dist_fns(x, ...)

## S3 method for class 'fitdist'
as.dist_fns(x, ...)

as.dist_fns(x, ...)
```

Arguments

`x`	a `dist_fns` S3 object
`...`	passed onto methods

Details

`dist_fns` and `dist_fns_list` objects support `$` access for fields and `@` access for attributes. `dist_fns_lists` can be made with the `c()` or `rep()` functions, or with the `purrr` style map functions, and they support subsetting. Individual `dist_fns` members of `dist_fns_lists` can be accessed with `[[`.

Value

a `dist_fns` S3 object

Methods (by class)

- `as.dist_fns(character)`: Construct a distribution by name
- `as.dist_fns(`function`)`: From a statistical function
- `as.dist_fns(fitdist)`: From a `fitdistrplus::fitdist` output

Unit tests

```
pois = as.dist_fns("pois",lambda = 8)
n = as.dist_fns("norm",mean=4)
```

`as.link_fns.character` *Create a* `link_fns` *S3 object*

Description

The link function class allows forwards and backwards transformation. Link functions can be defined by name or using a statistical distribution in which case the forward link is a logit of the cumulative probability and the reverse is the quantile of the expit.

Usage

```
## S3 method for class 'character'
as.link_fns(x, ...)

## S3 method for class 'dist_fns'
as.link_fns(x, ...)

## S3 method for class 'family'
as.link_fns(x, ...)

## S3 method for class 'numeric'
as.link_fns(x, ..., na.rm = TRUE)

as.link_fns(x, ...)
```

Arguments

`x`	a range of values that for the support
`...`	ignored
`na.rm`	remove NAs when estimating mean and sd for data driven link functions

Details

A `link_fns` S3 object encapsulates a monotonic transformation function h, its inverse h^{-1}, and their derivatives h' and $(h^{-1})'$. It also defines the support (domain) $[a, b]$ of the original space and the range $[h(a), h(b)]$ of the transformed space.

The function dispatches based on the input `x`:

- `character`: Selects standard links (e.g., "log", "logit", "probit", "identity"). For example, "log" defines $h(x) = \log(x)$ with support $(0, \infty)$.
- `dist_fns`: Defines the link as the logit of the CDF and the quantile of the expit: $h(x) = \text{logit}(F(x))$, $h^{-1}(z) = F^{-1}(\text{expit}(z))$, where F and F^{-1} are the CDF and quantile functions from the `dist_fns` object. The support is determined by the quantile function's range (e.g., $[Q(0), Q(1)]$).
- `family` (from `stats`): Uses the link function and its inverse from the GLM family object.
- `numeric`: If length 2, interprets as a support range $[a, b]$ and creates a logit-like transformation mapping this range to $(-\infty, \infty)$: $h(x) = \text{logit}(\frac{x-a}{b-a})$. If all values are finite and length > 2, creates a standardization link: $h(x) = \frac{x-\mu}{\sigma}$, where μ and σ are the mean and standard deviation of the input vector.

Value

a `link_fns` S3 object

Methods (by class)

- `as.link_fns(character)`: Link function from name
- `as.link_fns(dist_fns)`: Link function from name
- `as.link_fns(family)`: Link function from name
- `as.link_fns(numeric)`: Link function from support vector

Unit tests

```
links = c("ident", "log", "logit", "probit", "cloglog", "neginv", "inv2")
test = seq(0.1,0.9,0.1) # within support of all links
for (l in links) {
  lfn = as.link_fns(l)
  t = lfn$trans(test)
  i = lfn$inv(t)
  testthat::expect_equal(i,test)
}
```

c.dist_fns *Concatenate a* dist_fns *S3 object or* dist_fns_list*s*

Description

Concatenate a dist_fns S3 object or dist_fns_lists

Usage

```
## S3 method for class 'dist_fns'
c(...)
```

Arguments

...	some of dist_fns, dist_fns_lists or lists of dist_fns

Value

a dist_fns_list

c.link_fns *Concatenate a* link_fns *S3 object or* link_fns_list*s*

Description

Concatenate a link_fns S3 object or link_fns_lists

Usage

```
## S3 method for class 'link_fns'
c(...)
```

Arguments

...	some of link_fns, link_fns_lists or lists of link_fns

Value

a link_fns_list

`calculate_rmse` *Generate a function to calculate a Root Mean Squared Error (RMSE)*

Description

This function takes reference data in the form, for example of count data, and returns a crated function to calculate the mean squared error from simulated data to the observed data.

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2}$$

where x_i are the simulated data points, y_i are the observed data points, and N is the number of data points. Both input vectors must be of the same length. Missing values (`NA`) are removed pairwise before calculation. Returns `NA` if the input vectors are not of equal length.

Usage

```
calculate_rmse(sim, obs)
```

Arguments

`sim`	A vector of simulated counts
`obs`	A vector of observed counts

Value

The square root of the average squared distance between simulation and observation. Simulation and observation must be the same length.

Unit tests

```
# zero if no distance
testthat::expect_equal(calculate_rmse(0:10, 0:10), 0)

testthat::expect_equal(
  calculate_rmse(rep(5, 11), 0:10),
  3.16227766016838
)

withr::with_seed(100, {
   ref = rnorm(1000)
   cmp1 = rnorm(1000)
   cmp2 = rnorm(1000, sd=2)
})
```

```
breaks = seq(-2,2,length.out=8)
nref = table(cut(ref,breaks))
ncmp1 = table(cut(cmp1,breaks))
ncmp2 = table(cut(cmp2,breaks))

tmp1 = calculate_rmse(ncmp1, nref)
tmp2 = calculate_rmse(ncmp2, nref)

testthat::expect_equal(tmp1, 10.3578817470424)
testthat::expect_equal(tmp2, 68.8403951179829)
```

Examples

```
# example case counts from an exponential growth process
sim = table(floor(rexpgrowth(1000, 0.05, 40, 0)))
obs = table(floor(rexpgrowth(1000, 0.075, 40, 0)))
obs2 = table(floor(rexpgrowth(1000, 0.05, 40, 0)))

# obs is a different distribution to sim (larger growth)
calculate_rmse(sim, obs)

# obs2 is from the same distribution as sim so the RMSE should be lower:
calculate_rmse(sim, obs2)
```

---

calculate_wasserstein *Calculate a Wasserstein distance*

Description

This function takes simulation and observed data and calculates a normalised Wasserstein distance.

Usage

```
calculate_wasserstein(sim, obs, debias = FALSE, bootstraps = 1)
```

Arguments

sim	A vector of simulated data points
obs	A vector of observed data points
debias	Should the simulations be shifted to match the mean of the observed data
bootstraps	Randomly resample from the simulated data points to match the observed size this many times and combine the output by averaging. The alternative, when this is 1 (the default) matches the sizes by selecting and/or repeating the simulated data points in order (deterministically)

Details

In the comparison unequal lengths of the data can be accommodated. The simulated data is sorted and linearly interpolated to the same length as the observed data before the comparison.

$$W = \frac{1}{N_{obs} \cdot \sigma_{obs}} \sum_{i=1}^{N_{obs}} |\hat{x}_i - y_i|$$

where y_i are the ordered observed data points, $\hat{x}_i$ are the simulated data points after matching size (potentially via interpolation), debiasing (optional), and sorting, N_{obs} is the number of observed points, and σ_{obs} is the standard deviation of the observed data (used for normalisation).

Size matching via linear interpolation:

Let x be the sorted simulated data of length N_{sim}, and N_{obs} be the target length.

Indices idx are generated as $idx = \frac{(i-1)\cdot(N_{sim}-1)}{N_{obs}-1} + 1$ for $i = 1, ..., N_{obs}$. Then $\hat{x}_i$ is calculated as:

$$\hat{x}_i = (1 - p_i) \cdot x_{\lfloor idx_i \rfloor} + p_i \cdot x_{\lceil idx_i \rceil}$$

where $p_i = idx_i - \lfloor idx_i \rfloor$. If $\lfloor idx_i \rfloor = \lceil idx_i \rceil$, then $\hat{x}_i = x_{\lfloor idx_i \rfloor}$.

For bootstrapping (`bootstraps > 1`), the process is repeated `bootstraps` times with random sampling, and the final distance is the average of the results.

Value

a length normalised wasserstein distance. This is the average distance an individual simulated data point must be shifted to match the observed data normalised by the average distance of the observed data from the mean.

Unit tests

```
testthat::expect_equal(calculate_wasserstein(0:10, 10:0), 0)

# zero if no distance
testthat::expect_equal(calculate_wasserstein(0:10, 0:10), 0)
testthat::expect_equal(calculate_wasserstein(10:0, 0:10), 0)

# normalised so that all mass at mean = 1
ref = mean(abs(0:10-5))/sd(0:10)
testthat::expect_equal(calculate_wasserstein(rep(5, 11), 0:10), ref)

# smaller sample recycled and normalises to same value
testthat::expect_equal(calculate_wasserstein(rep(5, 5), 0:10), ref)

# should be ((0+1+0+1+0+0+0+1+0+1+0) / 11) / ((5+4+3+2+1+0+1+2+3+4+5) / 11) = 0.1333...
testthat::expect_equal(
  calculate_wasserstein(c(0, 0, 2, 2, 4, 5, 6, 8, 8, 10, 10), 0:10),
```

```
  0.109640488937369
)

withr::with_seed(100, {
   ref = rnorm(1000)
   cmp1 = rnorm(1000)
   cmp2 = rnorm(1000, sd=2)
   cmp3 = rnorm(100)
})

testthat::expect_equal(
  calculate_wasserstein(cmp1, ref),
  0.0458673541615467
)
testthat::expect_equal(
  calculate_wasserstein(cmp2, ref),
  0.835125114099775
)
testthat::expect_equal(
  calculate_wasserstein(cmp3, ref),
  0.180171816429487
)

tmp = withr::with_seed(100, {
 calculate_wasserstein(cmp1,ref,bootstraps=10)
})
testthat::expect_equal(tmp, 0.0678606864134826)


gen = function(n, mean=0, sd=1) {
  sample(pnorm(seq(-1,1,length.out = n - n
}

# there should be approximately zero
testthat::expect_equal(calculate_wasserstein(gen(1000), gen(1000)), 0)
testthat::expect_equal(calculate_wasserstein(gen(1000), gen(100)), 0)
testthat::expect_equal(
  calculate_wasserstein(gen(100), gen(1000)),
  0,tolerance = 0.01
)
testthat::expect_equal(
  calculate_wasserstein(gen(200), gen(1000)),
  0,tolerance = 0.01
)

# these should be approximately equal:
tmp2 = max(abs(diff(c(
calculate_wasserstein(gen(100,0.1), gen(1000)),
```

```
calculate_wasserstein(gen(200,0.1), gen(1000)),
calculate_wasserstein(gen(1000,0.1), gen(1000)),
calculate_wasserstein(gen(1000, 0.1), gen(200)),
calculate_wasserstein(gen(1000, 0.1), gen(100))
))))

testthat::expect_equal(tmp2, 0, tolerance = 0.01)
```

Examples

```
# example case timeseries from an exponential growth process
sim = rexpgrowth(1000, 0.05, 40, 0)
obs = rexpgrowth(1000, 0.075, 40, 0)
obs2 = rexpgrowth(1000, 0.05, 40, 0)

# obs is a different distribution to sim (larger growth)
calculate_wasserstein(sim, obs)

# obs2 is from the same distribution as sim so the distance should be lower:
calculate_wasserstein(sim, obs2)
```

---

dbeta2 *The Beta Distribution*

Description

Density, distribution function, quantile function and random generation for the Beta distribution with parameters shape1 and shape2 (and optional non-centrality parameter ncp).

Usage

```
dbeta2(x, prob, kappa, log = FALSE)
```

Arguments

x	vector of quantiles
prob	the mean probability (vectorised)
kappa	a coefficient of variation. where 0 is no variability and 1 is maximally variability (vectorised)
log	logical; if TRUE, probabilities p are given as log(p).

Value

dbeta gives the density, pbeta the distribution function, qbeta the quantile function, and rbeta generates random deviates.

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

The length of the result is determined by n for rbeta, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

stats::dbeta()

Examples

```
dbeta2(c(0.25,0.5,0.75), 0.5, 0.25)
```

---

dcgamma *Density: gamma distribution constrained to have mean > sd*

Description

The following conversion describes the parameters mean and kappa

Usage

```
dcgamma(x, mean, kappa = 1/mean, log = FALSE)
```

Arguments

x	vector of quantiles
mean	the mean value on the true scale (vectorised)
kappa	a coefficient of variation. where 0 is no variability and 1 is maximally variability (vectorised)
log	logical; if TRUE, probabilities p are given as log(p).

Details

$$\text{shape: } \alpha = \frac{1}{\kappa} \text{rate: } \beta = \frac{1}{\mu \times \kappa} \text{scale: } \sigma = \mu \times \kappa$$

Value

dgamma gives the density, pgamma gives the distribution function, qgamma gives the quantile function, and rgamma generates random deviates.

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

The length of the result is determined by n for rgamma, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

stats::dgamma()

Examples

```
dcgamma(seq(0,4,0.25), 2, 0.5)
```

---

default_termination_fn

Set up default convergence criteria for SMC and adaptive ABC

Description

Convergence is assessed on firstly whether the central estimate of the parameters being assessed is stable, and not changing from one wave to the next, and secondly if the 95 percent credible interval is stable between waves. If the parameter central estimate is stationary but the credible intervals are still dropping then continuing simulation may get better estimates of confidence.

Usage

```
default_termination_fn(stability = 0.01, confidence = 0.1)
```

Arguments

stability	how close do sequential estimates need to be before declaring them as a good set of parameter estimates. This is in the units of the parameter. If this is given as a single number it applies to all parameters equally, alternatively a named vector can be used to set parameter specific cutoffs.
confidence	how stable do the 95% confidence intervals need to be before we are happy with the parameter estimates. This is in the scale of the parameters, but represents a change in IQR from wave to wave. If this is given as a single number it applies to all parameters equally, alternatively a named vector can be used to set parameter specific cutoffs.

Value

a function that specifies the convergence.

Examples

```
# A more permissive definition of convergence has
# less strict stability criteria (sequential estimates varying by less than 5%)
# and confidence intervals not changing by more than 1 unit between waves.)
check = default_termination_fn(0.05, 1.0)


fit = example_smc_fit()

# This is performed as an integral part of the SMC and adaptive
# fitting and is here only for example
last_wave_metrics = utils::tail(fit$waves,1)
converged = check(
  wave = 0,
  summary = last_wave_metrics$summary[[1]],
  per_param = last_wave_metrics$per_param[[1]]
)

if (isTRUE(converged)) print("Converged (permissive definition)")
```

---

dgamma2 *The Gamma Distribution*

Description

Density, distribution function, quantile function and random generation for the Gamma distribution with parameters shape and scale.

Usage

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

Arguments

x	vector of quantiles
mean	the mean value on the true scale (vectorised)
sd	the standard deviation on the true scale (vectorised)
log	logical; if TRUE, probabilities p are given as log(p).

Value

dgamma gives the density, pgamma gives the distribution function, qgamma gives the quantile function, and rgamma generates random deviates.

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

The length of the result is determined by n for rgamma, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

stats::dgamma()

Examples

```
dgamma2(seq(0,4,0.25), 2, 1)
```

---

dist_fns *Create an empty* dist_fns_list

Description

Boilerplate S3 class operation

Usage

```
dist_fns()
```

Value

an empty dist_fns_list

---

dlnorm2 *The Log Normal Distribution*

Description

Density, distribution function, quantile function and random generation for the log normal distribution whose logarithm has mean equal to meanlog and standard deviation equal to sdlog.

Usage

```
dlnorm2(x, mean = 1, sd = sqrt(exp(1) - 1), log = FALSE)
```

Arguments

x	vector of quantiles
mean	the mean value on the true scale (vectorised)
sd	the standard deviation on the true scale (vectorised)
log	logical; if TRUE, probabilities p are given as log(p).

Details

The log normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-(\log(x)-\mu)^2/2\sigma^2}$$

where μ and σ are the mean and standard deviation of the logarithm. The mean is $E(X) = exp(\mu + 1/2\sigma^2)$, the median is $med(X) = exp(\mu)$, and the variance $Var(X) = exp(2\mu+\sigma^2)(exp(\sigma^2)-1)$ and hence the coefficient of variation is $\sqrt{exp(\sigma^2) - 1}$ which is approximately σ when that is small (e.g., $\sigma < 1/2$).

Value

`dlnorm` gives the density, `plnorm` gives the distribution function, `qlnorm` gives the quantile function, and `rlnorm` generates random deviates.

The length of the result is determined by `n` for `rlnorm`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-plnorm(t, r, lower = FALSE, log = TRUE)`.

Source

`dlnorm` is calculated from the definition (in 'Details'). `[pqr]lnorm` are based on the relationship to the normal.

Consequently, they model a single point mass at `exp(meanlog)` for the boundary case `sdlog = 0`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 14. Wiley, New York.

See Also

Distributions for other standard distributions, including `dnorm` for the normal distribution.

Examples

```
dlnorm2(seq(0,4,0.25), 2, 1)
```

dlogitnorm *Logit-normal distribution*

Description

The logit-normal distribution has a support of 0 to 1.

Usage

```
dlogitnorm(x, meanlogit = 0, sdlogit = 1, log = FALSE)
```

Arguments

x	vector of quantiles (0<x<1)
meanlogit	the mean on the logit scale
sdlogit	the sd on the logit scale
log	logical; if TRUE, probabilities p are given as log(p).

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
dlogitnorm(seq(0.1,0.9,0.1), 0, 1)
```

dlogitnorm2 *Logit-normal distribution*

Description

The logit-normal distribution has a support of 0 to 1.

Usage

```
dlogitnorm2(x, prob.0.5 = 0.5, kappa = 1 - exp(-1), log = FALSE)
```

Arguments

x	vector of quantiles (0<x<1)
prob.0.5	the median on the true scale
kappa	a dispersion parameter from 0 (none) to 1 maximum dispersion
log	logical; if TRUE, probabilities p are given as log(p).

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
q = seq(0.1,0.9,0.1)
eps = sqrt(.Machine$double.eps)
dlogitnorm2(q, 0.75, 0.2)

(plogitnorm2(q+eps, 0.75, 0.2) -
  plogitnorm2(q-eps, 0.75, 0.2)) / (2*eps)
```

---

dnbinom2 *The Negative Binomial Distribution*

Description

Density, distribution function, quantile function and random generation for the negative binomial distribution with parameters `size` and `prob`.

Usage

```
dnbinom2(x, mean, sd = sqrt(mean), log = FALSE)
```

Arguments

`x`	vector of (non-negative integer) quantiles.
`mean`	the mean value on the true scale (vectorised)
`sd`	the standard deviation on the true scale (vectorised)
`log`	logical; if TRUE, probabilities p are given as log(p).

Details

The negative binomial distribution with `size` $= n$ and `prob` $= p$ has density

$$p(x) = \frac{\Gamma(x+n)}{\Gamma(n)x!}p^n(1-p)^x$$

for $x = 0, 1, 2, \ldots$, $n > 0$ and $0 < p \leq 1$.

This represents the number of failures which occur in a sequence of Bernoulli trials before a target number of successes is reached. The mean is $\mu = n(1-p)/p$ and variance $n(1-p)/p^2$.

A negative binomial distribution can also arise as a mixture of Poisson distributions with mean distributed as a gamma distribution (see pgamma) with scale parameter `(1 - prob)/prob` and shape parameter `size`. (This definition allows non-integer values of `size`.)

An alternative parametrization (often used in ecology) is by the *mean* `mu` (see above), and `size`, the *dispersion parameter*, where `prob = size/(size+mu)`. The variance is `mu + mu^2/size` in this parametrization.

If an element of x is not integer, the result of dnbinom is zero, with a warning.

The case size == 0 is the distribution concentrated at zero. This is the limiting distribution for size approaching zero, even if mu rather than prob is held constant. Notice though, that the mean of the limit distribution is 0, whatever the value of mu.

The quantile is defined as the smallest value x such that $F(x) \geq p$, where F is the distribution function.

Value

dnbinom gives the density, pnbinom gives the distribution function, qnbinom gives the quantile function, and rnbinom generates random deviates.

Invalid size or prob will result in return value NaN, with a warning.

The length of the result is determined by n for rnbinom, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

rnbinom returns a vector of type integer unless generated values exceed the maximum representable integer when double values are returned.

Source

dnbinom computes via binomial probabilities, using code contributed by Catherine Loader (see dbinom).

pnbinom uses pbeta.

qnbinom uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

rnbinom uses the derivation as a gamma mixture of Poisson distributions, see

Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer-Verlag, New York. Page 480.

See Also

Distributions for standard distributions, including dbinom for the binomial, dpois for the Poisson and dgeom for the geometric distribution, which is a special case of the negative binomial.

Examples

```
dnbinom2(0:5, 2, sqrt(2))
```

dnull *Null distributions always returns NA*

Description

Null distributions always returns NA

Usage

```
dnull(x, ...)
```

Arguments

x	vector of quantiles
...	not used

Examples

```
dnull(c(0.25,0.5,0.75), 0.5, 0.25)
```

dwedge *Wedge distribution*

Description

The wedge distribution has a support of 0 to 1 and has a linear probability density function over that support.

Usage

```
dwedge(x, a, lower.tail = TRUE, log = FALSE)
```

Arguments

x	vector of quantiles
a	a gradient from -2 (left skewed) to 2 (right skewed)
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
log	logical; if TRUE, probabilities p are given as log(p).

Details

The rwedge can be combined with quantile functions to skew standard distributions, or introduce correlation or down weight certain parts of the distribution.

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
pwedge(seq(0,1,0.1), a=1)
dwedge(seq(0,1,0.1), a=1)
qwedge(c(0.25,0.5,0.75), a=-1)

stats::cor(
  stats::qnorm(rwedge(1000, a=2)),
  stats::qnorm(rwedge(1000, a=-2))
)
```

`empirical` *Fit a piecewise logit transformed linear model to cumulative data*

Description

This creates statistical distribution functions fitting to data in a transformed space. Inputs can either be weighted data observations (`x`,`w` pairs) of a CDF (`x`,`p` pairs). X axis transformation is specified in the `link` parameter and is either something like "log", "logit", etc or can also be specified as a statistical distribution, or even a length 2 numeric vector defining support.

Usage

```
empirical(
  x,
  ...,
  w = NULL,
  p = NULL,
  link = "ident",
  fit_spline = !is.null(knots),
  knots = NULL,
  name = NULL
)
```

Arguments

`x`	either a vector of samples from a distribution `X` or cut-offs for cumulative probabilities when combined with `p`
`...`	Named arguments passed on to `.logit_z_interpolation` `bw` a bandwidth expressed in terms of the probability width, or proportion of observations. Named arguments passed on to `empirical_cdf`

`smooth` fits the empirical distribution with a pair of splines for CDF and quantile function, creating a mostly smooth density. This smoothness comes at the price of potential over-fitting and will produce small differences between `p` and `q` functions such that `x=p(q(x))` is no longer exactly true. Setting this to false will replace this with a piecewise linear fit that is not smooth in the density, but is exact in forward and reverse transformation.

`...` not used

`w` for data fits, a vector the same length as `x` giving the importance weight of each observation. This does not need to be normalised. There must be some non zero weights, and all must be finite.

`p` for CDF fits, a vector the same length as `x` giving `P(X <= x)`.

`link` a link function. Either as name, or a `link_fns` S3 object. In the latter case this could be derived from a statistical distribution by `as.link_fns(<dist_fns>)`. This supports the use of a prior to define the support of the empirical function, and is designed to prevent tail truncation. Support for the updated quantile function will be the same as the provided prior.

`fit_spline` for distributions from data should we fit a spline to reduce memory usage and speed up sampling? If `knots` is given this is assumed to be true, set this to TRUE to allow knots to be determined by the data.

`knots` for spline fitting from data how many points do we use to model the cdf? By default this will be estimated from the data. I recommend an uneven number, without a lot of data this will tend to overfit 9 knots for 1000 samples seems OK, max 7 for 250, 5 for 100. Less is usually more.

`name` a label for this distribution. If not given one will be generated from the distribution and parameters. This can be used as part of plotting.

Details

If the empirical distribution if from data it is fully transformed to a `logit-z` space where the cumulative weighted data is interpolated with a kernel weighted linear model.

In the case of CDF data the empirical distribution is fitted with a piecewise linear or monotonically increasing spline fit to data in `Q-Q` space. The spline is bounded by 0 and 1 in both dimensions.

This function imputes tails of distributions within the constraints of the link functions. Given perfect data as samples or as quantiles it should well approximate the tail.

If `p` is provided, data is treated as CDF points (x_i, p_i). The function calls `empirical_cdf(x, p, ...)` internally. This involves:

1. Transformation: x values are mapped via a link function T to a standardized scale $q_x = T(x)$.
2. Interpolation: Monotonic splines (or piecewise linear functions if `smooth=FALSE`) are fitted between (q_x, p) pairs in Q-Q space, yielding functions $F_{cdf}(q_x)$ and $F_{qf}(p)$.
3. Tail Extrapolation: The fit is extended to $(0, 0)$ and $(1, 1)$ if necessary.
4. Final Functions: $P(X \leq x) = F_{cdf}(T(x))$ and $Q(p) = T^{-1}(F_{qf}(p))$.

If `w` is provided (or `p` is NULL), data is treated as weighted samples (x_i, w_i). The function calls `empirical_data(x, w, ...)` internally. This involves:

1. Transformation: x values are mapped via a link function T to $x_T = T(x)$.
2. Standardization: Values are standardized $x_Z = (x_T - \mu_{w,T})/\sigma_{w,T}$.
3. Weighted CDF: Empirical CDF $y = P(X_T \leq x_T)$ is calculated from w.
4. Logit Transformation: $y_L = \text{logit}(y)$.
5. Local Fitting: `locfit` is used to fit models between x_Z and y_L.
6. Final Functions: Composed from fitted models and the inverse link T^{-1}.

If `fit_spline=TRUE` (or `knots` is specified) when fitting from data, the resulting `empirical_data` fit is re-interpolated using `empirical_cdf` at quantiles defined by `knots`.

Value

a `dist_fns` S3 object containing functions `p()` for CDF, `q()` for quantile, and `r()` for a RNG, and `d()` for density. The density function may be discontinuous.

Unit tests

```
# from samples:
withr::with_seed(123,{
 e2 = empirical(rnorm(10000))
 testthat::expect_equal(e2$p(-5:5), pnorm(-5:5), tolerance=0.01)
 testthat::expect_equal(e2$q(seq(0,1,0.1)), qnorm(seq(0,1,0.1)), tolerance=0.05)
})

p2 = seq(0,1,0.1)
testthat::expect_equal( e2$p(e2$q(p2)), p2, tolerance = 0.001)

# updating a prior, with a horribly skewed gamma distribution
# not a great fit but not great data
withr::with_seed(124,{
  data = rgamma(500,1)
  e4 = empirical(data, link=as.dist_fns("unif",0,10))
  if (interactive()) plot(e4)+ggplot2::geom_function(fun = ~ dgamma(.x, 1))
  testthat::expect_equal(mean(e4$r(10000)), 1, tolerance=0.1)

  e5 = empirical(data,link="log")
  testthat::expect_equal(mean(e5$r(10000)), 1, tolerance=0.1)
  if (interactive()) plot(e5)+ggplot2::geom_function(fun = ~ dgamma(.x, 1))
})

withr::with_seed(123,{
  data = c(rnorm(200,4),rnorm(200,7))
  weights = c(rep(0.1,200), rep(0.3,200))
  e6 = empirical(x=data,w = weights)
  plot(e6)
  testthat::expect_equal(
```

```
    format(e6),
    "empirical; Median (IQR) 6.57 [5.2 — 7.43]"
  )
})

# Construct a normal using a sequence and density as weight.
e7 = empirical(x=seq(-10,10,length.out=1000),w=dnorm(seq(-10,10,length.out=1000)),knots = 20)
testthat::expect_equal(e7$p(-5:5), pnorm(-5:5), tolerance=0.01)
```

Examples

```
# A random sample from a distribution:
sample = rgamma2(1000, mean=5, sd=2)

# fit direct from data
fit = empirical(sample, link="log")
plot(fit)+ggplot2::geom_function(fun= ~ dgamma2(.x, mean=5, sd=2))

# suppose we only have quantiles
p = seq(0.1,0.9, 0.1)
quantiles = quantile(sample, p)

# fit from quantiles:
fit2 = empirical(x=quantiles,p=p, link="log")
plot(fit2)+ggplot2::geom_function(fun= ~ dgamma2(.x, mean=5, sd=2))

# fit weighted data
samples = seq(0,10,0.1)
weights = dgamma2(samples, mean=5, sd=2)
fit3 = empirical(x=samples, w=weights, link="log")
plot(fit3)+ggplot2::geom_function(fun= ~ dgamma2(.x, mean=5, sd=2))
```

---

empirical_cdf *Fit a piecewise logit transformed linear model to a CDF*

Description

This fits a CDF and quantile function to cumulative probabilities in a transformed space. X value transformation is specified in the `link` parameter and is either something like "log", "logit", etc or can also be specified as the logit transformed cdf and quantile function from a statistical distribution.

Usage

```
empirical_cdf(x, p, link = "ident", smooth = TRUE, name = NULL, ...)
```

Arguments

x	either a vector of samples from a distribution X or cut-offs for cumulative probabilities when combined with p
p	for CDF fits, a vector the same length as x giving P(X <= x).
link	a link function. Either as name, or a link_fns S3 object. In the latter case this could be derived from a statistical distribution by as.link_fns(<dist_fns>). This supports the use of a prior to define the support of the empirical function, and is designed to prevent tail truncation. Support for the updated quantile function will be the same as the provided prior.
smooth	fits the empirical distribution with a pair of splines for CDF and quantile function, creating a mostly smooth density. This smoothness comes at the price of potential over-fitting and will produce small differences between p and q functions such that x=p(q(x)) is no longer exactly true. Setting this to false will replace this with a piecewise linear fit that is not smooth in the density, but is exact in forward and reverse transformation.
name	a label for this distribution. If not given one will be generated from the distribution and parameters. This can be used as part of plotting.
...	not used

Details

The empirical distribution fitted is a piecewise linear or monotonically increasing spline fit to CDF in transformed X and logit Y space. The end points are linearly interpolated in this space to the tail_pth quantile. The function can fit data provided as x, P(X<=x) pairs.

Constructs an empirical distribution from CDF points (x_i, p_i) by fitting monotonic splines in a transformed quantile–quantile (Q–Q) space. The input x is first mapped to a standardized probability scale via a link-dependent transformation $q_x = T(x)$. The resulting pairs (q_x, p) are then used to build two monotonic interpolation functions:

$$p = F_{\text{cdf}}(q_x), \quad q_x = F_{\text{qf}}(p)$$

where F_{cdf} and F_{qf} are constructed as follows:

- If smooth = FALSE, both are piecewise linear interpolants (via stats::approx).
- If smooth = TRUE, both are strictly monotonic cubic splines fitted using the "monoH.FC" method from stats::splinefun, converted to polynomial spline form (polySpline). Monotonicity is enforced by requiring q_x and p to be strictly increasing after tie-breaking perturbations.

Tail extrapolation is applied by linearly extending the first and last segments in Q–Q space to the points $(0, 0)$ and $(1, 1)$ if not already included. The final distribution functions are:

$$P(X \leq x) = F_{\text{cdf}}(T(x)), \quad Q(p) = T^{-1}(F_{\text{qf}}(p))$$

where T and T^{-1} are derived from the link argument.

This function imputes tails of distributions. Given perfect data as samples or as quantiles it should recover the tail

Value

a dist_fns S3 object containing functions p() for CDF, q() for quantile, and r() for a RNG, and d() for density. The density function may be discontinuous.

Unit tests

```
#from cdf:
xs = c(2,3,6,9)
ps = c(0.1,0.4,0.6,0.95)
e = empirical_cdf(xs, ps, link="log")

testthat::expect_equal(e$p(xs), ps)
testthat::expect_equal(e$q(ps), xs)

# quantiles:
p = c(0.025,0.05,0.10,0.25,0.5,0.75,0.9,0.95,0.975)
q = stats::qgamma(p, shape=2)
shape2_gamma = as.dist_fns(pgamma, shape=2)
gemp = empirical_cdf(q,p,link = shape2_gamma)
withr::with_seed(123, {
  testthat::expect_equal(mean(gemp$r(100000)),2, tolerance=0.01)
  testthat::expect_equal(sd(gemp$r(100000)), sqrt(2), tolerance=0.01)
})

# With perfect input can recover the underlying distribution including tails:
tmp = empirical_cdf(x=seq(0.01,0.99,0.01),p=seq(0.01,0.99,0.01),link = as.dist_fns(punif,0, 1), knots
testthat::expect_equal(
  tmp$q(c(0.01, 0.1, 0.25, 0.75, 0.9, 0.99)),
  c(0.01, 0.1, 0.25, 0.75, 0.9, 0.99),
  tolerance = 0.002
)

# bimodal with log link and end defined
tmp3 = empirical_cdf(x = 1:7, c(0.1,0.3,0.4,0.4,0.5,0.9,1),link="log")
testthat::expect_equal(
  tmp3$p(-1:8),
  c(0, 0, 0.1, 0.3, 0.4, 0.4, 0.5, 0.9, 1, 1),
  tolerance = 0.01
)

testthat::expect_equal(
  tmp3$q(seq(0, 1, 0.2)),
  c(0, 1.63476839034733, 3, 5.05332769159444, 5.51891960240613, 7)
)
```

Examples

```
# A random sample from a distribution:
sample = rgamma2(1000, mean=5, sd=2)

# suppose we only have quantiles
p = seq(0.1,0.9, 0.1)
quantiles = quantile(sample, p)

# fit from quantiles:
fit2 = empirical(x=quantiles,p=p, link="log")
plot(fit2)+ggplot2::geom_function(fun= ~ dgamma2(.x, mean=5, sd=2))
```

`empirical_data` *Fit a piecewise logit transformed linear model to weighted data*

Description

This fits a CDF and quantile function to ranked data in a transformed space. X value transformation is specified in the `link` parameter and is either something like "log", "logit", etc.

Usage

```
empirical_data(
  x,
  w = NULL,
  link = "identity",
  ...,
  name = NULL,
  bw = wbw.nrd(x, w)^sqrt(2)
)
```

Arguments

`x`	either a vector of samples from a distribution `X` or cut-offs for cumulative probabilities when combined with `p`
`w`	for data fits, a vector the same length as `x` giving the importance weight of each observation. This does not need to be normalised. There must be some non zero weights, and all must be finite.
`link`	a link function. Either as name, or a `link_fns` S3 object. In the latter case this could be derived from a statistical distribution by `as.link_fns(<dist_fns>)`. This supports the use of a prior to define the support of the empirical function, and is designed to prevent tail truncation. Support for the updated quantile function will be the same as the provided prior.
`...`	Named arguments passed on to `.logit_z_interpolation` `bw` a bandwidth expressed in terms of the probability width, or proportion of observations.

Named arguments passed on to empirical_cdf

`smooth` fits the empirical distribution with a pair of splines for CDF and quantile function, creating a mostly smooth density. This smoothness comes at the price of potential over-fitting and will produce small differences between `p` and `q` functions such that `x=p(q(x))` is no longer exactly true. Setting this to false will replace this with a piecewise linear fit that is not smooth in the density, but is exact in forward and reverse transformation.

`...` not used

`name` a label for this distribution. If not given one will be generated from the distribution and parameters. This can be used as part of plotting.

`bw` a bandwidth expressed in terms of the probability width, or proportion of observations.

Details

The empirical distribution fitted is a piecewise linear in z transformed X and logit Y space. The evaluation points are linearly interpolated in this space given a bandwidth for interpolation.

1. Link Transformation: Input data `x` is transformed using the specified link function: $x_T = T(x)$, where T is the transformation defined by the `link` argument.
2. Standardization (Z-space): Transformed values x_T are standardized: $x_Z = \frac{x_T - \mu_{w,T}}{\sigma_{w,T}}$, where $\mu_{w,T}$ and $\sigma_{w,T}$ are the weighted mean and standard deviation of x_T.
3. Weighted Empirical CDF: The cumulative weights are calculated to form probabilities $y = P(X_T \leq x_T)$.
4. Logit Transformation: CDF probabilities are transformed: $y_L = \text{logit}(y)$.
5. Local Fitting (via `.logit_z_locfit`): Local likelihood models (using `locfit`) are fitted between x_Z and y_L to represent the CDF, its derivative (density), and the inverse (quantile) function in the transformed space.
6. Function Construction: The final `p`, `q`, `r`, and `d` functions are constructed by composing the fitted models from step 5 with the inverse link transformation T^{-1}. For example, the final CDF is $P(X \leq q) = F_{fitted}(\text{logit}^{-1}(T(q)))$, and the final quantile function is $Q(p) = T^{-1}(Q_{fitted}(p))$, where Q_{fitted} is the quantile function derived from the fitted models in Z-space.

This function imputes tails of distributions. Given perfect data as samples or as quantiles it should recover the tail

Value

a `dist_fns` S3 object that function that contains statistical distribution functions for this data.

Unit tests

```
# from samples:
withr::with_seed(123,{
```

```
 e2 = empirical_data(rnorm(10000), bw=0.1)
 testthat::expect_equal(e2$p(-5:5), pnorm(-5:5), tolerance=0.01)
 testthat::expect_equal(e2$d(-5:5), dnorm(-5:5), tolerance=0.05)
testthat::expect_equal(e2$q(seq(0,1,0.1)), qnorm(seq(0,1,0.1)), tolerance=0.025)
})


# Construct a normal using a sequence and density as weight.
e7 = empirical_data(
  x=seq(-10,10,length.out=1000),
  w=dnorm(seq(-10,10,length.out=1000))
)
plot(e7)+ggplot2::geom_function(fun = dnorm)
testthat::expect_equal(e7$p(-5:5), pnorm(-5:5), tolerance=0.01)
```

Examples

```
# A random sample from a distribution:
sample = rgamma2(1000, mean=5, sd=2)

# fit direct from data
fit = empirical(sample, link="log")
plot(fit)+ggplot2::geom_function(fun= ~ dgamma2(.x, mean=5, sd=2))

# fit weighted data
samples = seq(0,10,0.1)
weights = dgamma2(samples, mean=5, sd=2)
fit3 = empirical(x=samples, w=weights, link="log")
plot(fit3)+ggplot2::geom_function(fun= ~ dgamma2(.x, mean=5, sd=2))
```

---

fixed_wave_termination_fn

Run the SMC or adaptive algorithm for a set number of waves

Description

Run the SMC or adaptive algorithm for a set number of waves

Usage

```
fixed_wave_termination_fn(max_wave)
```

Arguments

max_wave	the number of waves to run

Value

A function that will test for completion

Examples

```
# Declare converged after 3 waves:
converged_fn = fixed_wave_termination_fn(3)
```

`format.dist_fns` *Format a* `dist_fns` *S3 object*

Description

Format a `dist_fns` S3 object

Usage

```
## S3 method for class 'dist_fns'
format(x, ..., digits = 3)
```

Arguments

`x`	a `dist_fns` S3 object
`...`	passed onto methods
`digits`	the number of significant digits

Value

a character value

`format.link_fns` *Format a* `link_fns` *S3 object*

Description

Format a `link_fns` S3 object

Usage

```
## S3 method for class 'link_fns'
format(x, ...)
```

Arguments

`x`	a `link_fns` S3 object
`...`	passed onto methods

Value

a character value

---

is.dist_fns *Check if this is a* dist_fns *S3 object*

Description

Check if this is a dist_fns S3 object

Usage

```
is.dist_fns(x)
```

Arguments

x	anything

Value

TRUE or FALSE

---

is.dist_fns_list *Check if this is a* dist_fns_list *S3 object*

Description

Check if this is a dist_fns_list S3 object

Usage

```
is.dist_fns_list(x)
```

Arguments

x	anything

Value

TRUE or FALSE

is.link_fns *Check if this is a* link_fns *S3 object*

Description

Check if this is a link_fns S3 object

Usage

```
is.link_fns(x)
```

Arguments

x	anything

Value

TRUE or FALSE

is.link_fns_list *Check if this is a* link_fns_list *S3 object*

Description

Check if this is a link_fns_list S3 object

Usage

```
is.link_fns_list(x)
```

Arguments

x	anything

Value

TRUE or FALSE

kurtosis *Calculate the excess kurtosis of a set of data*

Description

Calculate the excess kurtosis of a set of data

Usage

```
kurtosis(x, na.rm = FALSE, excess = TRUE)
```

Arguments

x	a vector of observations
na.rm	remove NAs?
excess	if false calculates raw kurtosis rather than excess

Value

the excess kurtosis

Examples

```
kurtosis(stats::rnorm(1000))
kurtosis(stats::rpois(1000, 2)) # leptokurtic > 0 (usually)
kurtosis(stats::runif(1000)) # platykurtic: < 0

kurtosis(stats::rlnorm(1000))
```

link_fns *Create an empty* link_fns_list

Description

Boilerplate S3 class operation

Usage

```
link_fns()
```

Value

an empty link_fns_list

`map_dist_fns` *Apply a function to each element of a vector returning a* `dist_fns_list`

Description

Analogous to `purrr::map_dbl()`

Usage

```
map_dist_fns(.x, .f, ..., .progress = FALSE)
```

Arguments

`.x`	A list or atomic vector.
`.f`	a function to apply that returns a `dist_fns` S3 object (usually an `as.dist_fns()` call)

`...`	Additional arguments passed on to the mapped function.
	We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:
	
	`# Instead of`
	`x |> map(f, 1, 2, collapse = ",")`
	`# do:`
	`x |> map(\(x) f(x, 1, 2, collapse = ","))`
	
	This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

`.progress`	Whether to show a progress bar. Use `TRUE` to turn on a basic progress bar, use a string to give it a name, or see progress_bars for more details.

Value

a `dist_fns_list`

See Also

`purrr::map()`

map_link_fns *Apply a function to each element of a vector returning a* link_fns_list

Description

Analogous to purrr::map_dbl()

Usage

```
map_link_fns(.x, .f, ..., .progress = FALSE)
```

Arguments

.x	A list or atomic vector.
.f	a function to apply that returns a link_fns S3 object (usually an as.link_fns() call)

...	Additional arguments passed on to the mapped function.
	We now generally recommend against using ... to pass additional (constant) arguments to .f. Instead use a shorthand anonymous function:
	
	`# Instead of`
	`x |> map(f, 1, 2, collapse = ",")`
	`# do:`
	`x |> map(\(x) f(x, 1, 2, collapse = ","))`
	
	This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

.progress	Whether to show a progress bar. Use TRUE to turn on a basic progress bar, use a string to give it a name, or see progress_bars for more details.

Value

a link_fns_list

See Also

purrr::map()

map2_dist_fns *Map over two inputs returning a* dist_fns_list

Description

Analogous to purrr::map2_dbl()

Usage

```
map2_dist_fns(.x, .y, .f, ..., .progress = FALSE)
```

Arguments

.x, .y	A pair of vectors, usually the same length. If not, a vector of length 1 will be recycled to the length of the other.
.f	a function to apply to each .x, .y pair that returns a dist_fns S3 object (usually an as.dist_fns() call)

...	Additional arguments passed on to the mapped function.
	We now generally recommend against using ... to pass additional (constant) arguments to .f. Instead use a shorthand anonymous function:
	
	`# Instead of`
	`x |> map(f, 1, 2, collapse = ",")`
	`# do:`
	`x |> map(\(x) f(x, 1, 2, collapse = ","))`
	
	This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

.progress	Whether to show a progress bar. Use TRUE to turn on a basic progress bar, use a string to give it a name, or see progress_bars for more details.

Value

a dist_fns_list

See Also

purrr::map2()

`map2_link_fns` *Map over two inputs returning a* `link_fns_list`

Description

Analogous to `purrr::map2_dbl()`

Usage

```
map2_link_fns(.x, .y, .f, ..., .progress = FALSE)
```

Arguments

`.x, .y`	A pair of vectors, usually the same length. If not, a vector of length 1 will be recycled to the length of the other.
`.f`	a function to apply to each `.x`, `.y` pair that returns a `link_fns` S3 object (usually an `as.link_fns()` call)

`...`	Additional arguments passed on to the mapped function.
	We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:
	
	`# Instead of`
	`x |> map(f, 1, 2, collapse = ",")`
	`# do:`
	`x |> map(\(x) f(x, 1, 2, collapse = ","))`
	
	This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

`.progress`	Whether to show a progress bar. Use `TRUE` to turn on a basic progress bar, use a string to give it a name, or see progress_bars for more details.

Value

a `link_fns_list`

See Also

`purrr::map2()`

`mixture` *Construct a mixture distribution*

Description

Constructs a mixture distribution from a list of component distributions dist_i with corresponding weights w_i. The CDF F_{mix} of the mixture is a weighted sum of the component CDFs:

$$F_{\text{mix}}(x) = \sum_{i=1}^{k} w_i \cdot F_i(x)$$

where F_i is the CDF of the i-th component distribution and $\sum w_i = 1$. The implementation first evaluates the weighted CDF on a grid of x values (including tail points defined by `tail_p` and potentially knot points from empirical components). The resulting $(x, F_{\text{mix}}(x))$ pairs are then used as input to `empirical_cdf` to create the final smooth or piecewise linear `dist_fns` object representing the mixture distribution.

Usage

```
mixture(dists, weights = 1, steps = 200, tail_p = 1e-04, ..., name = "mixture")
```

Arguments

`dists`	a `dist_fn_list` of distribution functions
`weights`	a vector of weights
`steps`	the number of points that the mixture distribution is evaluated at to construct the empirical mixture
`tail_p`	the support fo the tail of the distribution
`...`	Named arguments passed on to `empirical_cdf`
	`smooth` fits the empirical distribution with a pair of splines for CDF and quantile function, creating a mostly smooth density. This smoothness comes at the price of potential over-fitting and will produce small differences between `p` and `q` functions such that `x=p(q(x))` is no longer exactly true. Setting this to false will replace this with a piecewise linear fit that is not smooth in the density, but is exact in forward and reverse transformation.
`name`	a name for the mixture

Value

a `dist_fn` of the mixture distribution

Unit tests

```
dists = c(
  as.dist_fns("norm", mean=-1),
  as.dist_fns("norm", mean=1),
  as.dist_fns("gamma", shape=2)
)
weights = c(1,1,0.3)

mix = mixture(dists,weights)
testthat::expect_equal(
  format(mix),
  "mixture; Median (IQR) 0.289 [-0.886 — 1.31]"
)
```

Examples

```
dists = c(
  as.dist_fns("norm", mean=-1),
  as.dist_fns("norm", mean=1),
  as.dist_fns("gamma", shape=2)
)
weights = c(1,1,0.3)

mix = mixture(dists,weights)
plot(c(dists,mix))+ggplot2::facet_wrap(~name)
```

---

pbeta2 *The Beta Distribution*

Description

Density, distribution function, quantile function and random generation for the Beta distribution with parameters shape1 and shape2 (and optional non-centrality parameter ncp).

Usage

```
pbeta2(q, prob, kappa, lower.tail = TRUE, log.p = FALSE)
```

Arguments

q	vector of quantiles
prob	the mean probability (vectorised)

kappa	a coefficient of variation. where 0 is no variability and 1 is maximally variability (vectorised)
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
log.p	logical; if TRUE, probabilities p are given as log(p).

Value

dbeta gives the density, pbeta the distribution function, qbeta the quantile function, and rbeta generates random deviates.

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

The length of the result is determined by n for rbeta, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

stats::pbeta()

Examples

```
pbeta2(c(0.25,0.5,0.75), 0.5, 0.25)
```

---

pcgamma *Cumulative probability: gamma distribution constrained to have mean > sd*

Description

The following conversion describes the parameters mean and kappa

Usage

```
pcgamma(q, mean, kappa = 1/mean, lower.tail = TRUE, log.p = FALSE)
```

Arguments

q	vector of quantiles
mean	the mean value on the true scale (vectorised)
kappa	a coefficient of variation. where 0 is no variability and 1 is maximally variability (vectorised)
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
log.p	logical; if TRUE, probabilities p are given as log(p).

Details

$$\text{shape:}\alpha = \frac{1}{\kappa} \text{rate:}\beta = \frac{1}{\mu \times \kappa} \text{scale:}\sigma = \mu \times \kappa$$

Value

dgamma gives the density, pgamma gives the distribution function, qgamma gives the quantile function, and rgamma generates random deviates.

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

The length of the result is determined by n for rgamma, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

stats::pgamma()

Examples

```
pcgamma(seq(0,4,0.25), 2, 0.5)
```

---

pgamma2 *The Gamma Distribution*

Description

Density, distribution function, quantile function and random generation for the Gamma distribution with parameters shape and scale.

Usage

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

Arguments

q	vector of quantiles
mean	the mean value on the true scale (vectorised)
sd	the standard deviation on the true scale (vectorised)
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
log.p	logical; if TRUE, probabilities p are given as log(p).

Value

dgamma gives the density, pgamma gives the distribution function, qgamma gives the quantile function, and rgamma generates random deviates.

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

The length of the result is determined by n for rgamma, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

stats::pgamma()

Examples

```
pgamma2(seq(0,4,0.25), 2, 1)
```

---

plnorm2 *The Log Normal Distribution*

Description

Density, distribution function, quantile function and random generation for the log normal distribution whose logarithm has mean equal to meanlog and standard deviation equal to sdlog.

Usage

```
plnorm2(q, mean = 1, sd = sqrt(exp(1) - 1), lower.tail = TRUE, log.p = FALSE)
```

Arguments

q	vector of quantiles
mean	the mean value on the true scale (vectorised)
sd	the standard deviation on the true scale (vectorised)
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
log.p	logical; if TRUE, probabilities p are given as log(p).

Details

The log normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-(\log(x)-\mu)^2/2\sigma^2}$$

where μ and σ are the mean and standard deviation of the logarithm. The mean is $E(X) = exp(\mu + 1/2\sigma^2)$, the median is $med(X) = exp(\mu)$, and the variance $Var(X) = exp(2\mu+\sigma^2)(exp(\sigma^2)-1)$ and hence the coefficient of variation is $\sqrt{exp(\sigma^2) - 1}$ which is approximately σ when that is small (e.g., $\sigma < 1/2$).

Value

`dlnorm` gives the density, `plnorm` gives the distribution function, `qlnorm` gives the quantile function, and `rlnorm` generates random deviates.

The length of the result is determined by `n` for `rlnorm`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-plnorm(t, r, lower = FALSE, log = TRUE)`.

Source

`dlnorm` is calculated from the definition (in 'Details'). `[pqr]lnorm` are based on the relationship to the normal.

Consequently, they model a single point mass at `exp(meanlog)` for the boundary case `sdlog = 0`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 14. Wiley, New York.

See Also

Distributions for other standard distributions, including `dnorm` for the normal distribution.

Examples

```
plnorm2(seq(0,4,0.25), 2, 1)
```

`plogitnorm` *Logit-normal distribution*

Description

The logit-normal distribution has a support of 0 to 1.

Usage

```
plogitnorm(q, meanlogit = 0, sdlogit = 1, lower.tail = TRUE, log.p = FALSE)
```

Arguments

q	vector of quantiles (0<q<1)
meanlogit	the mean on the logit scale
sdlogit	the sd on the logit scale
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
log.p	logical; if TRUE, probabilities p are given as log(p).

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
plogitnorm(seq(0.1,0.9,0.1), 0, 1)
```

---

plogitnorm2 *Logit-normal distribution*

Description

The logit-normal distribution has a support of 0 to 1.

Usage

```
plogitnorm2(
  q,
  prob.0.5 = 0.5,
  kappa = 1 - exp(-1),
  lower.tail = TRUE,
  log.p = FALSE
)
```

Arguments

q	vector of quantiles (0<q<1)
prob.0.5	the median on the true scale
kappa	a dispersion parameter from 0 (none) to 1 maximum dispersion
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
log.p	logical; if TRUE, probabilities p are given as log(p).

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
plogitnorm2(seq(0.1,0.9,0.1), 0.75, 0.2)
plogitnorm2(qlogitnorm2(seq(0.1,0.9,0.1), 0.75, 0.2), 0.75, 0.2)
```

`plot.dist_fns` *Plot a* `dist_fns` *S3 object*

Description

Plot a `dist_fns` S3 object

Usage

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

Arguments

`x`	a `dist_fns` S3 object

`...`	Named arguments passed on to `plot.dist_fns_list`
	`x` a `dist_fns_list`
	`...` passed to `ggplot2::geom_step`,`ggplot2::geom_rect` or `ggplot2::geom_area`,
	`mapping` override default aesthetics with `name`, `id` or `group`
	`steps` resolution of the plot
	`tail` the minimum tail probability to plot
	`plot_quantiles` by default the quantiles of the distribution are plotted over the density sometimes this makes it hard to read
	`smooth` by default some additional smoothing is used to cover up small discontinuities in the PDF.

Value

a ggplot

`plot.dist_fns_list` *Plot a* `dist_fns_list` *S3 object*

Description

Plot a smoothed version of the PDFs of a set of `dist_fns`. These are `ggplots` that can be facetted by `names`, `id` or `group`

Usage

```
## S3 method for class 'dist_fns_list'
plot(
  x,
  ...,
  mapping = .gg_check_for_aes(...),
  steps = 200,
  tail = 0.001,
  plot_quantiles = TRUE,
  smooth = TRUE
)
```

Arguments

`x`	a `dist_fns_list`
`...`	passed to `ggplot2::geom_step`,`ggplot2::geom_rect` or `ggplot2::geom_area`,
`mapping`	override default aesthetics with `name`, `id` or `group`
`steps`	resolution of the plot
`tail`	the minimum tail probability to plot
`plot_quantiles`	by default the quantiles of the distribution are plotted over the density sometimes this makes it hard to read
`smooth`	by default some additional smoothing is used to cover up small discontinuities in the PDF.

Value

a ggplot

plot_convergence *Plot convergence metrics by wave for SMC and adaptive ABC*

Description

Plot convergence metrics by wave for SMC and adaptive ABC

Usage

```
plot_convergence(fit)
```

Arguments

fit	A S3 abc_fit object as output by the abc_XXX functions

Value

a patchwork plot of convergence metrics

Examples

```
plot_convergence(
  example_adaptive_fit()
)
```

plot_correlations *A parameter posterior correlation plot*

Description

A parameter posterior correlation plot

Usage

```
plot_correlations(posteriors_df, truth = NULL)
```

Arguments

posteriors_df	a dataframe of posteriors that have been selected by ABC this may include columns for scores, weight and/or simulation outputs (abc_component_score, abc_summary_distance, abc_weight, abc_simulation) as well as columns matching the priors input specification.
truth	a named numeric vector of known parameter values

Value

a patchwork of ggplots including density and 2d scatters for each combination of posteriors.

Examples

```
p = plot_correlations(
  example_adaptive_fit(),
  example_truth()
)

p & ggplot2::theme(
  axis.title.y = ggplot2::element_text(angle=70,vjust=0.5),
  axis.title.x = ggplot2::element_text(angle=20,hjust=0.5)
)
```

---

plot_evolution *Plot the evolution of the density function by wave for SMC and adaptive ABC*

Description

Plot the evolution of the density function by wave for SMC and adaptive ABC

Usage

```
plot_evolution(fit, truth = NULL, ..., what = c("posteriors", "proposals"))
```

Arguments

fit	A S3 abc_fit object as output by the abc_XXX functions
truth	a named numeric vector of known parameter values
...	passed on to methods Named arguments passed on to plot.dist_fns_list mapping override default aesthetics with name, id or group steps resolution of the plot tail the minimum tail probability to plot plot_quantiles by default the quantiles of the distribution are plotted over the density sometimes this makes it hard to read smooth by default some additional smoothing is used to cover up small discontinuities in the PDF.
what	plot posterior densities or proposal distributions?

Value

a plot of the density functions by wave

Examples

```
plot_evolution(
  example_adaptive_fit(),
  example_truth()
)
```

`plot_simulations` *Spaghetti plot of resampled posterior fits*

Description

Spaghetti plot of resampled posterior fits

Usage

```
plot_simulations(obsdata, fit, sim_fn, method = "auto", n = 200, ...)
```

Arguments

`obsdata`	The observational data. The data in this will typically be a named list, but could be anything, e.g. dataframe. It is the reference data that the simulation model is aiming to replicate.
`fit`	A S3 `abc_fit` object as output by the `abc_XXX` functions
`sim_fn`	a user defined function that takes a set of parameters named the same as `priors_list`. It must return a simulated data set in the same format as `obsdata`, or that can be compared to `simdata` by `scorer_fn`. This function must not refer to global parameters, and will be automatically crated with `carrier`.
`method`	one of `"auto"`, `"count"`, or `"density"`, or a named vector one for each component of `obsdata`.
`n`	the number of simulations to plot
`...`	Named arguments passed on to `posterior_resample` `n_resamples` the number of resamples for each parameter combination. `max_samples` the maximum total number of resamples to pick.

Value

a patchwork of `ggplots`

Examples

```
plot_simulations(
  example_obsdata(),
  example_adaptive_fit(),
  example_sim_fn
)
```

`pmap_dist_fns` *Map over multiple inputs returning a* `dist_fns_list`

Description

Analogous to `purrr::pmap_dbl()`

Usage

```
pmap_dist_fns(.l, .f, ..., .progress = FALSE)
```

Arguments

`.l`	A list of vectors. The length of `.l` determines the number of arguments that `.f` will be called with. Arguments will be supply by position if unnamed, and by name if named. Vectors of length 1 will be recycled to any length; all other elements must be have the same length. A data frame is an important special case of `.l`. It will cause `.f` to be called once for each row.
`.f`	a function to apply to each `.l` item (usually an `as.dist_fns()` call)

`...`	Additional arguments passed on to the mapped function.
	We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:
	`# Instead of`
	`x |> map(f, 1, 2, collapse = ",")`
	`# do:`
	`x |> map(\(x) f(x, 1, 2, collapse = ","))`
	This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

`.progress`	Whether to show a progress bar. Use `TRUE` to turn on a basic progress bar, use a string to give it a name, or see progress_bars for more details.

Value

a `dist_fns_list`

See Also

`purrr::map()`

pmap_link_fns *Map over multiple inputs returning a* link_fns_list

Description

Analogous to purrr::pmap_dbl()

Usage

```
pmap_link_fns(.l, .f, ..., .progress = FALSE)
```

Arguments

.l	A list of vectors. The length of .l determines the number of arguments that .f will be called with. Arguments will be supply by position if unnamed, and by name if named. Vectors of length 1 will be recycled to any length; all other elements must be have the same length. A data frame is an important special case of .l. It will cause .f to be called once for each row.
.f	a function to apply to each .l item (usually an as.link_fns() call)

...	Additional arguments passed on to the mapped function.
	We now generally recommend against using ... to pass additional (constant) arguments to .f. Instead use a shorthand anonymous function:
	`# Instead of`
	`x |> map(f, 1, 2, collapse = ",")`
	`# do:`
	`x |> map(\(x) f(x, 1, 2, collapse = ","))`
	This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

.progress	Whether to show a progress bar. Use TRUE to turn on a basic progress bar, use a string to give it a name, or see progress_bars for more details.

Value

a link_fns_list

See Also

purrr::map()

pnbinom2 *The Negative Binomial Distribution*

Description

Density, distribution function, quantile function and random generation for the negative binomial distribution with parameters `size` and `prob`.

Usage

```
pnbinom2(q, mean, sd = sqrt(mean), lower.tail = TRUE, log.p = FALSE)
```

Arguments

`q`	vector of quantiles.
`mean`	the mean value on the true scale (vectorised)
`sd`	the standard deviation on the true scale (vectorised)
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
`log.p`	logical; if TRUE, probabilities p are given as log(p).

Details

The negative binomial distribution with `size` $= n$ and `prob` $= p$ has density

$$p(x) = \frac{\Gamma(x+n)}{\Gamma(n)x!}p^n(1-p)^x$$

for $x = 0, 1, 2, \ldots$, $n > 0$ and $0 < p \leq 1$.

This represents the number of failures which occur in a sequence of Bernoulli trials before a target number of successes is reached. The mean is $\mu = n(1-p)/p$ and variance $n(1-p)/p^2$.

A negative binomial distribution can also arise as a mixture of Poisson distributions with mean distributed as a gamma distribution (see `pgamma`) with scale parameter `(1 - prob)/prob` and shape parameter `size`. (This definition allows non-integer values of `size`.)

An alternative parametrization (often used in ecology) is by the *mean* `mu` (see above), and `size`, the *dispersion parameter*, where `prob = size/(size+mu)`. The variance is `mu + mu^2/size` in this parametrization.

If an element of `x` is not integer, the result of `dnbinom` is zero, with a warning.

The case `size == 0` is the distribution concentrated at zero. This is the limiting distribution for `size` approaching zero, even if `mu` rather than `prob` is held constant. Notice though, that the mean of the limit distribution is 0, whatever the value of `mu`.

The quantile is defined as the smallest value x such that $F(x) \geq p$, where F is the distribution function.

Value

dnbinom gives the density, pnbinom gives the distribution function, qnbinom gives the quantile function, and rnbinom generates random deviates.

Invalid size or prob will result in return value NaN, with a warning.

The length of the result is determined by n for rnbinom, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

rnbinom returns a vector of type integer unless generated values exceed the maximum representable integer when double values are returned.

Source

dnbinom computes via binomial probabilities, using code contributed by Catherine Loader (see dbinom).

pnbinom uses pbeta.

qnbinom uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

rnbinom uses the derivation as a gamma mixture of Poisson distributions, see

Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer-Verlag, New York. Page 480.

See Also

Distributions for standard distributions, including dbinom for the binomial, dpois for the Poisson and dgeom for the geometric distribution, which is a special case of the negative binomial.

Examples

```
pnbinom2(0:5, 2, sqrt(2))
```

---

pnull *Null distributions always returns NA*

Description

Null distributions always returns NA

Usage

```
pnull(q, ...)
```

Arguments

q	vector of quantiles
...	not used

Examples

```
pnull(c(0.25,0.5,0.75), 0.5, 0.25)
```

`posterior_distance_metrics` *Generate a set of metrics from component scores*

Description

The component scores are summary statistics output by the user supplied `scorer_fn` as a named list. These can be variable in scale and location and various options exist for combining them. They may need to be weighted by scale as well as importance to get a model that works well. Such weights can be input into the ABC algortihms using the `scoreweights` parameter. This function helps provide diagnostics for calibrating the `scoreweights` parameter.

Usage

```
posterior_distance_metrics(posteriors_df, obsscores = NULL, keep_data = FALSE)
```

Arguments

`posteriors_df`	a dataframe of posteriors that have been selected by ABC this may include columns for scores, weight and/or simulation outputs (`abc_component_score`, `abc_summary_distance`, `abc_weight`, `abc_simulation`) as well as columns matching the `priors` input specification.
`obsscores`	Summary scores for the observational data. This will be a named list, and is equivalent to the output of `scorer_fn`, on the observed data. If not given typically it will be assumed to be all zeros.
`keep_data`	mainly for internal use this flag gives you the component scores as a matrix

Details

Given a list of component scores $S_s^{(i)} = (s_1^{(i)}, \ldots, s_m^{(i)})$ from n simulations and observed scores $S_o = (s_1^{(o)}, \ldots, s_m^{(o)})$, this function calculates:

- The mean μ_j and standard deviation σ_j of each score component j:

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} s_j^{(i)}, \quad \sigma_j = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(s_j^{(i)} - \mu_j)^2}$$

- The covariance matrix Σ of the score components.
- The root mean squared difference (RMSD) between simulated and observed scores for each component:

$$\text{RMSD}_j = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(s_j^{(i)} - s_j^{(o)})^2}$$

- A recommended vector of `scoreweights` w_j, calculated as the ratio of the component's standard deviation to its RMSD, normalized to sum to 1:

$$w_j = \frac{\sigma_j / \text{RMSD}_j}{\sum_{k=1}^{m} (\sigma_k / \text{RMSD}_k)}$$

  These weights help balance the influence of different summary statistics in the overall distance metric, especially when using Euclidean distance.

Value

a list containing the following items:

- `obsscores`: the input reference scores for each component
- `means`, `sds`: the means and sds of each score component
- `cov`: the covariance matrix for the scores
- `mad`: the mean absolute differences between the `simscores` and the `obsscores`
- `rmsd`: the root mean squared differences between the `simscores` and the `obsscores`
- `scoreweights`: a the `sds` divided by the `rmsd`. This weight should mean that the weights of the individual summary scores have similar influence in the overall `abc_summary_distance` output once combined during SMC and adaptive waves, especially if euclidean distances are involved.
- `simscores`: (if `keep_data`) a matrix of all the scores from these input simulation posteriors
- `deltascores`: (if `keep_data`) a matrix of the differences between `simscores` and `obsscores`.

Examples

```
fit = example_rejection_fit()
metrics = posterior_distance_metrics(fit$posteriors)

# other elements available but this is the most important and tells you what
# the relative sizes of the component scores from `scorer_fn` in this sample.
# If this is a sample from the prior then this gives us a way to judge the
# most appropriate relative weighting of each component:
metrics$scoreweights
```

---

posterior_fit_analytical

Fit analytical distribution to posterior samples for generating more waves

---

Description

This function allows "updating" of the prior with a posterior distribution from the same family as the prior with updated parameters.

Usage

```
posterior_fit_analytical(posteriors_df, priors_list)
```

Arguments

`posteriors_df`	a dataframe of posteriors that have been selected by ABC this may include columns for scores, weight and/or simulation outputs (`abc_component_score`, `abc_summary_distance`, `abc_weight`, `abc_simulation`) as well as columns matching the `priors` input specification.
`priors_list`	a named list of priors specified as a `abc_prior` S3 object (see `priors()`), this can include derived values as unnamed 2-sided formulae, where the LHS of the formula will be assigned to the value of the RHS, plus optionally a set of constraints as one sided formulae where the RHS of the formulae will resolve to a boolean value.

Details

This takes weighted posterior samples $\{(\theta^{(i)}, w^{(i)})\}$ for each parameter θ_j and fits an analytical distribution function $Q_j(\theta_j)$ that approximates the posterior marginal for that parameter.

The resulting empirical distribution Q_j is a `dist_fns` object that includes the support constraints from the original prior. The set of all fitted marginal distributions $\{Q_j\}$ forms the new proposal list for the next wave.

Additionally, the weighted covariance matrix R of the samples in the MVN space (defined by the prior copula) is calculated:

$$R = \text{Cov}_w(\Phi^{-1}(P_1(\theta_1)), \ldots, \Phi^{-1}(P_d(\theta_d)))$$

where Φ^{-1} is the quantile function of the standard normal. This covariance matrix is stored as an attribute (`"cor"`) of the returned proposal list and is used to induce correlation structure when sampling new proposals from the empirical distributions in subsequent waves.

Value

an `abc_prior` S3 object approximating the distribution of the posterior samples from the same family as the prior.

Examples

```
fit = example_smc_fit()
proposals = posterior_fit_analytical(fit$posteriors, fit$priors)

proposals
```

posterior_fit_empirical *Fit empirical distribution to posterior samples for generating more waves*

Description

This function allows "updating" of the prior with an empirical posterior distribution which will retain the bounds of the prior, and is effectively a spline based transform of the prior distribution, based on the density of data in prior space. This gives a clean density when data is close to a prior distribution limit and work better than a standard density.

Usage

```
posterior_fit_empirical(
  posteriors_df,
  priors_list,
  knots = NULL,
  bw = 0.1,
  widen_by = 1
)
```

Arguments

posteriors_df	a dataframe of posteriors that have been selected by ABC this may include columns for scores, weight and/or simulation outputs (abc_component_score, abc_summary_distance, abc_weight, abc_simulation) as well as columns matching the priors input specification.
priors_list	a named list of priors specified as a abc_prior S3 object (see priors()), this can include derived values as unnamed 2-sided formulae, where the LHS of the formula will be assigned to the value of the RHS, plus optionally a set of constraints as one sided formulae where the RHS of the formulae will resolve to a boolean value.
knots	the number of knots to model the CDF with. Optional, and will be typically inferred from the data size. Small numbers tend to work better if we expect the distribution to be unimodal.
bw	for Adaptive ABC data distributions are smoothed before modelling empirical CDF. Over smoothing can reduce convergence rate, under-smoothing may result in noisy posterior estimates, and appearance of local modes.
widen_by	change the dispersion of the empirical proposal distribution in ABC adaptive, preserving the median. This is akin to a nonlinear, heteroscedastic random walk in the quantile space, and can help address over-fitting or local modes in the ABC adaptive waves. widen_by is an odds ratio and describes how much further from the median any given part of the distribution is after transformation. E.g. if the median of a distribution is zero, and the widen_by is 2 then the 0.75 quantile will move to the position of the 0.9 quantile. The distribution will stay within

the support of the prior. This is by default 1.05 which allows for some additional variability in proposals.

Details

This takes weighted posterior samples $\{(\theta^{(i)}, w^{(i)})\}$ for each parameter θ_j and constructs an empirical distribution function $Q_j(\theta_j)$ that approximates the posterior marginal for that parameter.

For each parameter θ_j, the empirical distribution Q_j is fitted using the `empirical()` function. The fitting is performed on the weighted samples $(\theta_j^{(i)}, w^{(i)})$. A key feature is the use of a link function $h_j(\cdot)$ during the fitting process. This link function is typically taken from the original prior distribution $P_j(\theta_j)$ (i.e., $h_j = P_j$). This ensures that the fitted empirical distribution Q_j respects the support constraints defined by the prior (e.g., positive values for a log-normal prior). The fitting effectively occurs in the transformed space defined by the link: $h_j(\theta_j^{(i)})$.

The resulting empirical distribution Q_j is a `dist_fns` object that includes the support constraints from the original prior. The set of all fitted marginal distributions $\{Q_j\}$ forms the new proposal list for the next wave.

Additionally, the weighted covariance matrix R of the samples in the MVN space (defined by the prior copula) is calculated:

$$R = \text{Cov}_w(\Phi^{-1}(P_1(\theta_1)), \ldots, \Phi^{-1}(P_d(\theta_d)))$$

where Φ^{-1} is the quantile function of the standard normal. This covariance matrix is stored as an attribute (`"cor"`) of the returned proposal list and is used to induce correlation structure when sampling new proposals from the empirical distributions in subsequent waves.

Value

an `abc_prior` S3 object approximating the distribution of the posterior samples, and adhering to the support of the provided priors.

Examples

```
fit = example_smc_fit()
proposals = posterior_fit_empirical(fit$posteriors, fit$priors)

proposals
```

---

posterior_resample *Generate a set of samples from selected posteriors*

Description

Once an ABC model fitting is complete the simulation data is generally only one possible realisation of the parameters, which has been selected for closeness. To properly compare the output with the observed data we need a set of posterior re-samples, which are selected from posteriors according to importance.

Usage

```
posterior_resample(
  posteriors_df,
  sim_fn,
  n_resamples = 1,
  seed = NULL,
  parallel = FALSE,
  max_samples = 200
)
```

Arguments

posteriors_df	a dataframe of posteriors that have been selected by ABC this may include columns for scores, weight and/or simulation outputs (abc_component_score, abc_summary_distance, abc_weight, abc_simulation) as well as columns matching the priors input specification.
sim_fn	a user defined function that takes a set of parameters named the same as priors_list. It must return a simulated data set in the same format as obsdata, or that can be compared to simdata by scorer_fn. This function must not refer to global parameters, and will be automatically crated with carrier.
n_resamples	the number of resamples for each parameter combination.
seed	an optional random seed
parallel	parallelise the simulation? If this is set to true then the simulation step will be parallelised using furrr. For this to make any difference it must have been set up with the following: future::plan(future::multisession, workers = parallel::detectCores()-2)
max_samples	the maximum total number of resamples to pick.

Value

a dataframe of the posteriors with an abc_sim list column containing the output of sim_fn called with the parameters in that row.

Examples

```
fit = example_adaptive_fit()

sample_df = posterior_resample(
  fit$posteriors,
  sim_fn = example_sim_fn
)

# the fitted simulations are in the `abc_sim` column
sim1 = sample_df$abc_sim[[1]]

sim1 %>% lapply(head, 10)
```

`posterior_summarise` *Calculate a basket of summaries from a weighted list of posterior samples*

Description

Calculate a basket of summaries from a weighted list of posterior samples

Usage

```
posterior_summarise(
  posteriors_df,
  priors_list,
  p = c(0.025, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.975)
)
```

Arguments

`posteriors_df`	a dataframe of posteriors that have been selected by ABC this may include columns for scores, weight and/or simulation outputs (`abc_component_score`, `abc_summary_distance`, `abc_weight`, `abc_simulation`) as well as columns matching the `priors` input specification.
`priors_list`	a named list of priors specified as a `abc_prior` S3 object (see `priors()`), this can include derived values as unnamed 2-sided formulae, where the LHS of the formula will be assigned to the value of the RHS, plus optionally a set of constraints as one sided formulae where the RHS of the formulae will resolve to a boolean value.
`p`	a `progressr` progress bar

Value

a dataframe indexed by parameter with useful summary metrics.

Examples

```
fit = example_adaptive_fit()
summ = posterior_summarise(fit$posteriors, fit$priors)

summ %>% dplyr::glimpse()
```

priors *Construct a set of priors*

Description

abc_prior S3 objects are used to hold the specification of prior and intermediate proposal distributions. They are inputs to the main abc_...() workflow functions.

Usage

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

Arguments

...	a list of formulae. Two sided will be interpreted as distribution or derived value specifications. One sided as constraints between parameters. A distribution is specified as the name of the family of statistical distributions and their parameters: e.g.: x ~ norm(mean=3,sd=2). The name will be matched to the first hit on the search path.

Value

an S3 object of class abc_prior which contains

- a list of dist_fns
- a cor attribute describing their correlation
- a derived attribute describing derive values
- a constraints attribute listing the constraints
- a params attribute listing the names of the parameters

Examples

```
p = priors(
  mean ~ tidyabc::rgamma2(4,2),
  sd ~ gamma2(2,1),
  shape ~ mean^2 / sd^2,
  rate ~ mean / sd^2,
  ~ mean > sd
)

print(p)

# Plot methods are also provided:
if (interactive()) plot(p)

# constraints:
p@constraints
```

pwedge *Wedge distribution*

Description

The wedge distribution has a support of 0 to 1 and has a linear probability density function over that support.

Usage

```
pwedge(q, a, log.p = FALSE)
```

Arguments

q	vector of quantiles
a	a gradient from -2 (left skewed) to 2 (right skewed)
log.p	logical; if TRUE, probabilities p are given as log(p).

Details

The rwedge can be combined with quantile functions to skew standard distributions, or introduce correlation or down weight certain parts of the distribution.

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
pwedge(seq(0,1,0.1), a=1)
dwedge(seq(0,1,0.1), a=1)
qwedge(c(0.25,0.5,0.75), a=-1)

stats::cor(
  stats::qnorm(rwedge(1000, a=2)),
  stats::qnorm(rwedge(1000, a=-2))
)
```

qbeta2 *The Beta Distribution*

Description

Density, distribution function, quantile function and random generation for the Beta distribution with parameters shape1 and shape2 (and optional non-centrality parameter ncp).

Usage

```
qbeta2(p, prob, kappa, lower.tail = TRUE, log.p = FALSE)
```

Arguments

p	vector of probabilities
prob	the mean probability (vectorised)
kappa	a coefficient of variation. where 0 is no variability and 1 is maximally variability (vectorised)
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
log.p	logical; if TRUE, probabilities p are given as log(p).

Value

dbeta gives the density, pbeta the distribution function, qbeta the quantile function, and rbeta generates random deviates.

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

The length of the result is determined by n for rbeta, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

stats::qbeta()

Examples

```
qbeta2(c(0.25,0.5,0.75), 0.5, 0.25)
```

qcgamma *Quantile: gamma distribution constrained to have mean > sd*

Description

The following conversion describes the parameters mean and kappa

Usage

```
qcgamma(p, mean, kappa = 1/mean, lower.tail = TRUE, log.p = FALSE)
```

Arguments

p	vector of probabilities
mean	the mean value on the true scale (vectorised)
kappa	a coefficient of variation. where 0 is no variability and 1 is maximally variability (vectorised)
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
log.p	logical; if TRUE, probabilities p are given as log(p).

Details

$$\text{shape:}\alpha = \frac{1}{\kappa} \text{rate:}\beta = \frac{1}{\mu \times \kappa} \text{scale:}\sigma = \mu \times \kappa$$

Value

dgamma gives the density, pgamma gives the distribution function, qgamma gives the quantile function, and rgamma generates random deviates.

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

The length of the result is determined by n for rgamma, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

stats::qgamma()

Examples

```
qcgamma(c(0.25,0.5,0.75), 2, 0.5)
```

qgamma2 *The Gamma Distribution*

Description

Density, distribution function, quantile function and random generation for the Gamma distribution with parameters shape and scale.

Usage

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

Arguments

p	vector of probabilities
mean	the mean value on the true scale (vectorised)
sd	the standard deviation on the true scale (vectorised)
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
log.p	logical; if TRUE, probabilities p are given as log(p).

Value

dgamma gives the density, pgamma gives the distribution function, qgamma gives the quantile function, and rgamma generates random deviates.

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

The length of the result is determined by n for rgamma, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

stats::qgamma()

Examples

```
qgamma2(c(0.25,0.5,0.75), 2, 1)
```

qlnorm2 *The Log Normal Distribution*

Description

Density, distribution function, quantile function and random generation for the log normal distribution whose logarithm has mean equal to `meanlog` and standard deviation equal to `sdlog`.

Usage

```
qlnorm2(p, mean = 1, sd = sqrt(exp(1) - 1), lower.tail = TRUE, log.p = FALSE)
```

Arguments

`p`	vector of probabilities.
`mean`	the mean value on the true scale (vectorised)
`sd`	the standard deviation on the true scale (vectorised)
`lower.tail`	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
`log.p`	logical; if TRUE, probabilities p are given as log(p).

Details

The log normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-(\log(x)-\mu)^2/2\sigma^2}$$

where μ and σ are the mean and standard deviation of the logarithm. The mean is $E(X) = exp(\mu + 1/2\sigma^2)$, the median is $med(X) = exp(\mu)$, and the variance $Var(X) = exp(2\mu+\sigma^2)(exp(\sigma^2)-1)$ and hence the coefficient of variation is $\sqrt{exp(\sigma^2) - 1}$ which is approximately σ when that is small (e.g., $\sigma < 1/2$).

Value

`dlnorm` gives the density, `plnorm` gives the distribution function, `qlnorm` gives the quantile function, and `rlnorm` generates random deviates.

The length of the result is determined by `n` for `rlnorm`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-plnorm(t, r, lower = FALSE, log = TRUE)`.

Source

`dlnorm` is calculated from the definition (in 'Details'). `[pqr]lnorm` are based on the relationship to the normal.

Consequently, they model a single point mass at `exp(meanlog)` for the boundary case `sdlog = 0`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 14. Wiley, New York.

See Also

Distributions for other standard distributions, including `dnorm` for the normal distribution.

Examples

```
qlnorm2(c(0.25,0.5,0.72), 2, 1)
```

---

qlogitnorm *Logit-normal distribution*

Description

The logit-normal distribution has a support of 0 to 1.

Usage

```
qlogitnorm(p, meanlogit = 0, sdlogit = 1, lower.tail = TRUE, log.p = FALSE)
```

Arguments

`p`	vector of probabilities
`meanlogit`	the mean on the logit scale
`sdlogit`	the sd on the logit scale
`lower.tail`	logical; if TRUE (default), probabilities are `P[X<=x]` otherwise `P[X>x]`.
`log.p`	logical; if TRUE, probabilities p are given as log(p).

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
qlogitnorm(c(0.25,0.5,0.75), 0, 1)
```

qlogitnorm2 *Logit-normal distribution*

Description

The logit-normal distribution has a support of 0 to 1.

Usage

```
qlogitnorm2(
  p,
  prob.0.5 = 0.5,
  kappa = 1 - exp(-1),
  lower.tail = TRUE,
  log.p = FALSE
)
```

Arguments

p	vector of probabilities
prob.0.5	the median on the true scale
kappa	a dispersion parameter from 0 (none) to 1 maximum dispersion
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
log.p	logical; if TRUE, probabilities p are given as log(p).

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
qlnorm2(c(0.25,0.5,0.72), 2, 1)
```

qnbinom2 *The Negative Binomial Distribution*

Description

Density, distribution function, quantile function and random generation for the negative binomial distribution with parameters size and prob.

Usage

```
qnbinom2(p, mean, sd = sqrt(mean), lower.tail = TRUE, log.p = FALSE)
```

Arguments

p	vector of probabilities.
mean	the mean value on the true scale (vectorised)
sd	the standard deviation on the true scale (vectorised)
lower.tail	logical; if TRUE (default), probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.
log.p	logical; if TRUE, probabilities p are given as log(p).

Details

The negative binomial distribution with size $= n$ and prob $= p$ has density

$$p(x) = \frac{\Gamma(x+n)}{\Gamma(n)x!}p^n(1-p)^x$$

for $x = 0, 1, 2, \ldots$, $n > 0$ and $0 < p \leq 1$.

This represents the number of failures which occur in a sequence of Bernoulli trials before a target number of successes is reached. The mean is $\mu = n(1-p)/p$ and variance $n(1-p)/p^2$.

A negative binomial distribution can also arise as a mixture of Poisson distributions with mean distributed as a gamma distribution (see pgamma) with scale parameter (1 - prob)/prob and shape parameter size. (This definition allows non-integer values of size.)

An alternative parametrization (often used in ecology) is by the *mean* mu (see above), and size, the *dispersion parameter*, where prob = size/(size+mu). The variance is mu + mu^2/size in this parametrization.

If an element of x is not integer, the result of dnbinom is zero, with a warning.

The case size == 0 is the distribution concentrated at zero. This is the limiting distribution for size approaching zero, even if mu rather than prob is held constant. Notice though, that the mean of the limit distribution is 0, whatever the value of mu.

The quantile is defined as the smallest value x such that $F(x) \geq p$, where F is the distribution function.

Value

dnbinom gives the density, pnbinom gives the distribution function, qnbinom gives the quantile function, and rnbinom generates random deviates.

Invalid size or prob will result in return value NaN, with a warning.

The length of the result is determined by n for rnbinom, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

rnbinom returns a vector of type integer unless generated values exceed the maximum representable integer when double values are returned.

Source

dnbinom computes via binomial probabilities, using code contributed by Catherine Loader (see dbinom).

pnbinom uses pbeta.

qnbinom uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

rnbinom uses the derivation as a gamma mixture of Poisson distributions, see

Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer-Verlag, New York. Page 480.

See Also

Distributions for standard distributions, including dbinom for the binomial, dpois for the Poisson and dgeom for the geometric distribution, which is a special case of the negative binomial.

Examples

```
qnbinom2(c(0.25,0.5,0.75), 5, sqrt(5))
```

---

qnull *Null distributions always returns NA*

Description

Null distributions always returns NA

Usage

```
qnull(p, ...)
```

Arguments

p	vector of probabilities
...	not used

Examples

```
qnull(c(0.25,0.5,0.75), 0.5, 0.25)
```

qwedge *Wedge distribution*

Description

The wedge distribution has a support of 0 to 1 and has a linear probability density function over that support.

Usage

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

Arguments

p	vector of probabilities
a	a gradient from -2 (left skewed) to 2 (right skewed)
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
log.p	logical; if TRUE, probabilities p are given as log(p).

Details

The rwedge can be combined with quantile functions to skew standard distributions, or introduce correlation or down weight certain parts of the distribution.

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
pwedge(seq(0,1,0.1), a=1)
dwedge(seq(0,1,0.1), a=1)
qwedge(c(0.25,0.5,0.75), a=-1)

stats::cor(
  stats::qnorm(rwedge(1000, a=2)),
  stats::qnorm(rwedge(1000, a=-2))
)
```

rbern *A random Bernoulli sample as a logical value*

Description

A random Bernoulli sample as a logical value

Usage

```
rbern(n, prob)
```

Arguments

n	number of observations
prob	the mean probability (vectorised)

Value

a vector of logical values of size n

Examples

```
table(rbern(100, 0.25))
```

rbeta2 *The Beta Distribution*

Description

Density, distribution function, quantile function and random generation for the Beta distribution with parameters shape1 and shape2 (and optional non-centrality parameter ncp).

Usage

```
rbeta2(n, prob, kappa)
```

Arguments

n	number of observations
prob	the mean probability (vectorised)
kappa	a coefficient of variation. where 0 is no variability and 1 is maximally variability (vectorised)

Value

dbeta gives the density, pbeta the distribution function, qbeta the quantile function, and rbeta generates random deviates.

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

The length of the result is determined by n for rbeta, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

stats::rbeta()

Examples

```
rbeta2(3, c(0.1,0.5,0.9),0.1)
```

---

rcategorical *Sampling from the multinomial equivalent of the Bernoulli distribution*

Description

Sampling from the multinomial equivalent of the Bernoulli distribution

Usage

```
rcategorical(n, prob, factor = FALSE)
```

Arguments

n	a sample size
prob	a (optionally named) vector of probabilities that will be normalised to sum to 1
factor	if not FALSE then factor levels will either be taken from the names of prob first, or if this is a character vector from this.

Value

a vector of random class labels of length n. Labels come from names of prob or from a character vector in factor.

Examples

```
prob = c("one"=0.1,"two"=0.2,"seven"=0.7)
table(rcategorical(1000,prob))
rcategorical(10,prob,factor=TRUE)
rcategorical(10,rep(1,26),factor=letters)
```

`rcgamma` *Sampling: gamma distribution constrained to have mean > sd*

Description

The following conversion describes the parameters mean and kappa

Usage

```
rcgamma(n, mean, kappa = 1/mean)
```

Arguments

`n`	number of observations
`mean`	the mean value on the true scale (vectorised)
`kappa`	a coefficient of variation. where 0 is no variability and 1 is maximally variability (vectorised)

Details

$$\text{shape:}\alpha = \frac{1}{\kappa} \text{rate:}\beta = \frac{1}{\mu \times \kappa} \text{scale:}\sigma = \mu \times \kappa$$

Value

`dgamma` gives the density, `pgamma` gives the distribution function, `qgamma` gives the quantile function, and `rgamma` generates random deviates.

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

The length of the result is determined by `n` for `rgamma`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

`stats::rgamma()`

Examples

```
rcgamma(10, 2, 0.5)
```

rexpgrowth — *Randomly sample incident times in an exponentially growing process*

Description

Randomly sample incident times in an exponentially growing process

Usage

```
rexpgrowth(n, r, t_end, t_start = 0)
```

Arguments

n	the number of items to sample
r	and exponential growth rate (per unit time)
t_end	the end of the observation period
t_start	the start of the observation period

Value

a vector of n samples from an exponential growth process

Examples

```
graphics::hist(rexpgrowth(1000,0.1,40), breaks=40)
graphics::hist(rexpgrowth(1000,-0.1,40), breaks=40)
```

rexpgrowthI0 — *Randomly sample incident times in an exponentially growing process with initial case load*

Description

Randomly sample incident times in an exponentially growing process with initial case load

Usage

```
rexpgrowthI0(I0, r, t_end, t_start = 0)
```

Arguments

I0	the expected number of cases observed in the first day
r	and exponential growth rate (per unit time)
t_end	the end of the observation period
t_start	the start of the observation period

Value

a vector of n samples from an exponential growth process

Examples

```
graphics::hist(rexpgrowthI0(10,0.1,20), breaks=40)
graphics::hist(rexpgrowthI0(1000,-0.1,40), breaks=40)
```

---

rgamma2 *The Gamma Distribution*

Description

Density, distribution function, quantile function and random generation for the Gamma distribution with parameters shape and scale.

Usage

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

Arguments

n	number of observations
mean	the mean value on the true scale (vectorised)
sd	the standard deviation on the true scale (vectorised)

Value

dgamma gives the density, pgamma gives the distribution function, qgamma gives the quantile function, and rgamma generates random deviates.

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

The length of the result is determined by n for rgamma, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

See Also

stats::rgamma()

Examples

```
rgamma2(10, 2, 1)
```

`rlnorm2` *The Log Normal Distribution*

Description

Density, distribution function, quantile function and random generation for the log normal distribution whose logarithm has mean equal to `meanlog` and standard deviation equal to `sdlog`.

Usage

```
rlnorm2(n, mean = 1, sd = sqrt(exp(1) - 1))
```

Arguments

`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`mean`	the mean value on the true scale (vectorised)
`sd`	the standard deviation on the true scale (vectorised)

Details

The log normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-(\log(x)-\mu)^2/2\sigma^2}$$

where μ and σ are the mean and standard deviation of the logarithm. The mean is $E(X) = exp(\mu + 1/2\sigma^2)$, the median is $med(X) = exp(\mu)$, and the variance $Var(X) = exp(2\mu+\sigma^2)(exp(\sigma^2)-1)$ and hence the coefficient of variation is $\sqrt{exp(\sigma^2) - 1}$ which is approximately σ when that is small (e.g., $\sigma < 1/2$).

Value

`dlnorm` gives the density, `plnorm` gives the distribution function, `qlnorm` gives the quantile function, and `rlnorm` generates random deviates.

The length of the result is determined by `n` for `rlnorm`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

The cumulative hazard $H(t) = -\log(1 - F(t))$ is `-plnorm(t, r, lower = FALSE, log = TRUE)`.

Source

`dlnorm` is calculated from the definition (in 'Details'). `[pqr]lnorm` are based on the relationship to the normal.

Consequently, they model a single point mass at `exp(meanlog)` for the boundary case `sdlog = 0`.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, N. L., Kotz, S. and Balakrishnan, N. (1995) *Continuous Univariate Distributions*, volume 1, chapter 14. Wiley, New York.

See Also

Distributions for other standard distributions, including `dnorm` for the normal distribution.

Examples

```
rlnorm2(10, 2, 1)
```

`rlogitnorm` *Logit-normal distribution*

Description

The logit-normal distribution has a support of 0 to 1.

Usage

```
rlogitnorm(n, meanlogit = 0, sdlogit = 1)
```

Arguments

`n`	number of observations
`meanlogit`	the mean on the logit scale
`sdlogit`	the sd on the logit scale

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
rlogitnorm(10, 0, 1)
```

rlogitnorm2 *Logit-normal distribution*

Description

The logit-normal distribution has a support of 0 to 1.

Usage

```
rlogitnorm2(n, prob.0.5 = 0.5, kappa = 1 - exp(-1))
```

Arguments

n	number of observations
prob.0.5	the median on the true scale
kappa	a dispersion parameter from 0 (none) to 1 maximum dispersion

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
mean(rlogitnorm2(10000,0.75,0.2))
```

rnbinom2 *The Negative Binomial Distribution*

Description

Density, distribution function, quantile function and random generation for the negative binomial distribution with parameters size and prob.

Usage

```
rnbinom2(n, mean, sd = sqrt(mean))
```

Arguments

n	number of observations. If length(n) > 1, the length is taken to be the number required.
mean	the mean value on the true scale (vectorised)
sd	the standard deviation on the true scale (vectorised)

Details

The negative binomial distribution with `size` $= n$ and `prob` $= p$ has density

$$p(x) = \frac{\Gamma(x+n)}{\Gamma(n)x!}p^n(1-p)^x$$

for $x = 0, 1, 2, \ldots$, $n > 0$ and $0 < p \leq 1$.

This represents the number of failures which occur in a sequence of Bernoulli trials before a target number of successes is reached. The mean is $\mu = n(1-p)/p$ and variance $n(1-p)/p^2$.

A negative binomial distribution can also arise as a mixture of Poisson distributions with mean distributed as a gamma distribution (see `pgamma`) with scale parameter `(1 - prob)/prob` and shape parameter `size`. (This definition allows non-integer values of `size`.)

An alternative parametrization (often used in ecology) is by the *mean* `mu` (see above), and `size`, the *dispersion parameter*, where `prob = size/(size+mu)`. The variance is `mu + mu^2/size` in this parametrization.

If an element of `x` is not integer, the result of `dnbinom` is zero, with a warning.

The case `size == 0` is the distribution concentrated at zero. This is the limiting distribution for `size` approaching zero, even if `mu` rather than `prob` is held constant. Notice though, that the mean of the limit distribution is 0, whatever the value of `mu`.

The quantile is defined as the smallest value x such that $F(x) \geq p$, where F is the distribution function.

Value

`dnbinom` gives the density, `pnbinom` gives the distribution function, `qnbinom` gives the quantile function, and `rnbinom` generates random deviates.

Invalid `size` or `prob` will result in return value `NaN`, with a warning.

The length of the result is determined by `n` for `rnbinom`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

`rnbinom` returns a vector of type integer unless generated values exceed the maximum representable integer when `double` values are returned.

Source

`dnbinom` computes via binomial probabilities, using code contributed by Catherine Loader (see `dbinom`).

`pnbinom` uses `pbeta`.

`qnbinom` uses the Cornish–Fisher Expansion to include a skewness correction to a normal approximation, followed by a search.

`rnbinom` uses the derivation as a gamma mixture of Poisson distributions, see

Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer-Verlag, New York. Page 480.

See Also

Distributions for standard distributions, including dbinom for the binomial, dpois for the Poisson and dgeom for the geometric distribution, which is a special case of the negative binomial.

Examples

```
rnbinom2(10, 5, sqrt(5))
```

---

rnull *Null distributions always returns NA*

Description

Null distributions always returns NA

Usage

```
rnull(n, ...)
```

Arguments

n	number of observations
...	not used

Examples

```
rnull(3, c(0.1,0.5,0.9),0.1)
```

---

rwedge *Wedge distribution*

Description

The wedge distribution has a support of 0 to 1 and has a linear probability density function over that support.

Usage

```
rwedge(n, a)
```

Arguments

n	number of observations
a	a gradient from -2 (left skewed) to 2 (right skewed)

Details

The rwedge can be combined with quantile functions to skew standard distributions, or introduce correlation or down weight certain parts of the distribution.

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
pwedge(seq(0,1,0.1), a=1)
dwedge(seq(0,1,0.1), a=1)
qwedge(c(0.25,0.5,0.75), a=-1)

stats::cor(
  stats::qnorm(rwedge(1000, a=2)),
  stats::qnorm(rwedge(1000, a=-2))
)
```

---

sim_outbreak *The* sim_outbreak *dataset*

Description

This is a generated data set of an outbreak based on a branching process model with fixed parameters. There is a both the full infection line list as it happened in the simulation and the subset of this data that might have been observed in a contact tracing exercise where identification is done through symptoms.

Usage

```
sim_outbreak
```

Format

An object of class list of length 3.

Details

sim_outbreak **is a named list with 3 items:**

Item	Type	Description
parameters	list[dbl]	the ground truth of the simulation parameters
outbreak_truth	df[outbreak_truth]*	the full infection line list
contact_tracing	df[contact_tracing]*	the "observed" contact tracing subset

`df[outbreak_truth]` **dataframe with 663 rows and 11 columns:**

The simulation details

Column	Type	Description
time	dbl	The true time of infection
id	int	Person unique id
generation_interval	dbl	The time since infector's infection
infector	int	The unique id of the infector
generation	dbl	Which generation is this infection since the simulation start
symptom	lgl	Did this person experience symptoms
symptom_delay	dbl	How long after infection were their symptoms?
symptom_time	dbl	When? (from the simulation start)
observation	lgl	Was this person detected (only if symptoms)
observation_delay	dbl	How long after symptoms were they observed?
observation_time	dbl	When? (from the simulation start)

`df[contact_tracing]` **dataframe with 663 rows and 4 columns:**

A minimal set of data that might be collected in a contact tracing exercise.

Column	Type	Description
id	int	Unique person id
contact_id	int	Unique id of infectious contact
onset_time	int	Time of symptom onset (from the simulation start)
obs_time	int	Time of observation (from the simulation start)

Source

https://ai4ci.github.io/ggoutbreak/articles/simulation-test-models.html#line-list-simulations

---

skew *Calculate the skew of a set of data*

Description

Calculate the skew of a set of data

Usage

```
skew(x, na.rm = FALSE)
```

Arguments

x	a vector of observations
na.rm	remove NAs?

Value

the skew

Examples

```
skew(stats::rnorm(1000))
skew(stats::rbeta(1000, 1, 8)) # positively (left) skewed
skew(stats::rbeta(1000, 8, 1)) # negatively (right) skewed

skew(stats::rlnorm(1000))
```

---

test_simulation *Run the simulation for one set of parameters*

Description

Run the simulation for one set of parameters

Usage

```
test_simulation(
  sim_fn,
  scorer_fn,
  ...,
  params = NULL,
  obsdata = NULL,
  seed = NULL,
  debug = FALSE,
  debug_errors = !debug
)
```

Arguments

sim_fn	a user defined function that takes a set of parameters named the same as priors_list. It must return a simulated data set in the same format as obsdata, or that can be compared to simdata by scorer_fn. This function must not refer to global parameters, and will be automatically crated with carrier.
scorer_fn	a user supplied function that matches the following signature scorer_fn(simdata, obsdata, ...), i.e. it takes data in the format of simdata paired with the original obsdata and returns a named list of component scores per simulation. This function can make use of the calculate_*() set of functions to compare components of the simulation to the original data. This function must not refer to global parameters, and will be automatically crated with carrier. If this is a purrr style function then .x will refer to simulation output and .y to original observation data.
...	simulation parameters, must be named

params	a named list of simulation parameters to test (as an alternative to including them in ...)
obsdata	The observational data. The data in this will typically be a named list, but could be anything, e.g. dataframe. It is the reference data that the simulation model is aiming to replicate.
seed	an optional random seed
debug	start the simulation function in debug mode. This will step through both the sim_fn and the scorer_fn line by line to check that the behaviour is as intended.
debug_errors	Errors that crop up in sim_fn during a simulation due to anomolous value combinations are hard to debug. If this flag is set, whenever a sim_fn or scorer_fn throws an error an interactive debugging session is started with the failing parameter combinations. This is not compatible with running in parallel.

Value

a list containing the parameters as truth and an instance of the simulation as obsdata, obsscores is the result of comparing the obsdata with itself and is usually going to result in zeros. Both sim_fn and scorer_fn are rewritten to make sure that all package names are fully qualified. The rewritten versions are refurned in the result list (as sim_fn and scorer_fn respectively)

Examples

```
test = test_simulation(
  example_sim_fn,
  example_scorer_fn,
  # Model parameters to test with:
  mean = 4, sd1 = 3, sd2 = 2,
  obsdata = example_obsdata()
)

# the rewritten function:
cat(test$sim_fn, sep="\n")

# The scores resulting from this one simulation, when compared to the
# reference `obsdata`.
test$obsscores
```

---

transform *Generate a distribution from a link transform of another*

Description

Generates a new distribution by applying a link transformation to an existing distribution dist. If $X \sim$ dist and h is the link function, this function returns the distribution of $Y = h^{-1}(X)$. The CDF F_Y, quantile function Q_Y, PDF f_Y, and RNG R_Y of the transformed distribution are derived

from the original distribution's functions F_X, Q_X, f_X, R_X and the link function h and its inverse h^{-1} as follows:

$$F_Y(y) = F_X(h(y))$$
$$Q_Y(p) = h^{-1}(Q_X(p))$$
$$f_Y(y) = f_X(h(y)) \cdot |h'(y)|$$
$$R_Y(n) = h^{-1}(R_X(n))$$

where $h'(y)$ is the derivative of the link function. This function implements these transformations for the p, q, d, and r functions of the resulting dist_fns object.

Usage

```
transform(link, dist, ..., name = NULL)
```

Arguments

link	a link function (or name of a link function)
dist	distribution(s) as a name, function or a dist_fn S3 object
...	parameters for the underlying distribution if dist is a name or function.
name	a name for the link function

Value

a dist_fn or dist_fn_list holding the transformed distribution(s)

Unit tests

```
t = transform("log","norm")
ps = seq(0,1,0.1)
qs = 0:6
testthat::expect_equal(t$q(ps),qlnorm(ps))
testthat::expect_equal(t$p(qs),plnorm(qs))
testthat::expect_equal(t$d(qs),dlnorm(qs))

t2 = transform("log","norm", 0.4, 0.1)
testthat::expect_equal(t2$q(ps),qlnorm(ps,0.4,0.1))
```

Examples

```
n = as.dist_fns("norm",mean=0.5, sd=0.1)
t = transform("log",n)

plot(t)+ggplot2::geom_function(fun=~ dlnorm(.x, 0.5, 0.1), linetype="dashed")
```

`truncate` *Generate a distribution from a truncation of another*

Description

Generates a truncated distribution from an existing distribution `dist`. The new distribution is restricted to the interval $[x_{left}, x_{right}]$ (or (x_{left}, x_{right}) if open bounds are intended, though the quantile function will map 0 to x_{left} and 1 to x_{right}). The probability density function (PDF) of the truncated distribution $f_T(x)$ is related to the original PDF $f(x)$ by a normalization constant:

$$f_T(x) = \frac{f(x)}{F(x_{right}) - F(x_{left})} \mathbb{I}_{[x_{left}, x_{right}]}(x)$$

where F is the original CDF and $\mathbb{I}$ is the indicator function. Consequently, the CDF F_T and quantile function Q_T are:

$$F_T(x) = \frac{F(x) - F(x_{left})}{F(x_{right}) - F(x_{left})}$$

$$Q_T(p) = Q(F(x_{left}) + p \cdot (F(x_{right}) - F(x_{left})))$$

where Q is the quantile function of the original distribution. This function implements these transformations for the `p`, `q`, and `r` functions of the resulting `dist_fns` object.

Usage

```
truncate(dist, x_left, x_right, ..., name = NULL)
```

Arguments

`dist`	a distribution as a name, function or a `dist_fn` S3 object
`x_left`	The lower end of the interval or NA for open
`x_right`	The upper end of the interval or NA for open
`...`	parameters for the underlying distribution if `dist` is a name or function.
`name`	a name for the truncation

Value

a `dist_fn` or `dist_fn_list` holding the truncated distribution(s)

Unit tests

```
shape2_gamma = as.dist_fns(pgamma, shape=2)
g2 = truncate(shape2_gamma, 0.5, 4)

testthat::expect_equal(
  format(g2),
```

```
  "trunc(gamma(shape = 2, rate = 1), 0.50 - 4.00); Median (IQR) 1.68 [1.08 — 2.46]"
)

testthat::expect_equal(
  g2$p(0:5),
  c(0, 0.212702667019627, 0.615716430100344, 0.868531239886668, 1, 1)
)

testthat::expect_equal(g2$q(seq(0, 1, 0.2)), c(
  0.5,
  0.971743593113941,
  1.42711359317907,
  1.95304152540128,
  2.6642447272259,
  4
))


g3 = truncate(shape2_gamma, NA, 4)
testthat::expect_equal(
  format(g3),
  "trunc(gamma(shape = 2, rate = 1), 0.00 - 4.00); Median (IQR) 1.54 [0.899 — 2.35]"
)

g4 = truncate(shape2_gamma, 2, NA)
testthat::expect_equal(
  format(g4),
  "trunc(gamma(shape = 2, rate = 1), 2.00 - Inf); Median (IQR) 2.97 [2.42 — 3.87]"
)
```

Examples

```
shape2_gamma = as.dist_fns(pgamma, shape=2)
g2 = truncate(shape2_gamma, 0.5, 4)
plot(g2)+ggplot2::geom_function(fun = ~ dgamma(.x,shape=2), xlim = c(0,5))
```

`wasserstein_calculator` *Generate a function to calculate a wasserstein distance*

Description

[Experimental]

Usage

```
wasserstein_calculator(obs, debias = FALSE, bootstraps = 1)
```

Arguments

obs	A vector of observations
debias	Should the simulations be shifted to match the mean of the observed data
bootstraps	Randomly resample from the simulated data points to match the observed size this many times and combine the output by averaging. The alternative, when this is 1 (the default) matches the sizes by selecting and/or repeating the simulated data points in order (deterministically)

Details

This function takes reference data in the forms of individual observations in the form of for example event times and returns a crated function to calculate the wasserstein distance from simulated data to the observed data. For large simulations this will be quicker than `calculate_wasserstein` but must be used with a crated `scorer_fn`

In the comparison unequal lengths of the data can be accommodated. The simulated data is recycled, and sampled, until the same length as the observed data before the comparison.

Value

a function that takes parameter `sim` and returns a length normalised wasserstein distance. This is the average distance an individual data point must be shifted to match the reference data normalised by the average distance of the reference data from the mean. The function also takes a `p` parameter which can be a `progressr` progress bar which must be named.

Unit tests

```
tmp = wasserstein_calculator(0:10)

# zero if no distance
testthat::expect_equal(tmp(0:10), 0)
testthat::expect_equal(tmp(10:0), 0)
```

Examples

```
# example case counts from an exponential growth process
sim = rexpgrowth(1000, 0.05, 40, 0)
obs = rexpgrowth(1000, 0.075, 40, 0)
obs2 = rexpgrowth(1000, 0.05, 40, 0)

calc = wasserstein_calculator(sim)

# obs is a different distribution to sim (larger growth)
calc(obs)
```

```
# obs2 is from the same distribution as sim so the RMSE should be lower:
calc(obs2)
```

---

wbw.nrd *Weighted bandwidth selector*

Description

Weighted bandwidth selector

Usage

```
wbw.nrd(x, w = NULL)
```

Arguments

x	data
w	weights

Value

a bandwidth based on weighted data

Examples

```
x = runif(1000)
w = rgamma(1000,2)

wbw.nrd(x)
wbw.nrd(x,w)
```

---

wedge *Wedge distribution*

Description

The wedge distribution has a support of 0 to 1 and has a linear probability density function over that support.

Arguments

n	number of observations
x	vector of quantiles
q	vector of quantiles
p	vector of probabilities
log	logical; if TRUE, probabilities p are given as log(p).
log.p	logical; if TRUE, probabilities p are given as log(p).
lower.tail	logical; if TRUE (default), probabilities are P[X<=x] otherwise P[X>x].
a	a gradient from -2 (left skewed) to 2 (right skewed)

Details

The rwedge can be combined with quantile functions to skew standard distributions, or introduce correlation or down weight certain parts of the distribution.

Value

a vector of probabilities, quantiles, densities or samples.

Examples

```
pwedge(seq(0,1,0.1), a=1)
dwedge(seq(0,1,0.1), a=1)
qwedge(c(0.25,0.5,0.75), a=-1)

stats::cor(
  stats::qnorm(rwedge(1000, a=2)),
  stats::qnorm(rwedge(1000, a=-2))
)
```

---

widen *Increase the dispersion of a distribution*

Description

Increases the dispersion (spread) of a distribution by transforming its quantile function in a standardized Q-Q space.

Usage

```
widen(x, scale, knots = NULL, name = NULL)
```

Arguments

x	a distribution as a dist_fns S3 object
scale	acts as a multiplier for the log-odds difference from the median, effectively acting like an odds-ratio parameter. A scale > 1 increases dispersion (stretches quantiles away from the median in logit space), while 0 < scale < 1 decreases dispersion (compresses quantiles towards the median in logit space). The median value is preserved in the original parameter space.
knots	the number of knots in the transformed distribution, if it is not already an empirical CDF distribution.
name	a name for the widened distribution

Details

The transformation aims to increase the standard deviation by a factor scale while preserving the median. It operates on the internal Q-Q space representation (qx, qy) of an empirical distribution generated by empirical_cdf.

Applies a logit-space scaling transformation centred on the median quantile. This transformation modifies the quantile axis (qx) in the Q-Q space of an empirical distribution to change its dispersion while preserving the median value.

Let qx be the original quantile coordinate in [0, 1], and qmedian be the quantile corresponding to the median (e.g., qx_from_qy(0.5)). The transformation is defined as:

$$qx_2 = \text{expit}\left((\text{logit}(qx) - \text{logit}(qmedian)) \times \text{scale} + \text{logit}(qmedian)\right)$$

where $\text{logit}(x) = \log(x/(1-x))$ and $\text{expit}(x) = 1/(1+\exp(-x))$ are the standard logit and inverse-logit functions, respectively.

Value

an empirical dist_fn with the same median and increased SD. This transformation will change the mean of skewed distributions.

Examples

```
d1 = as.dist_fns("norm",4,2)
w1 = widen(d1, scale=1.5)
```

wmean *Weighted mean*

Description

a simple alias for base `weighted,mean`

Usage

```
wmean(x, w = NULL, na.rm = TRUE)
```

Arguments

x	either a vector of samples from a distribution X or cut-offs for cumulative probabilities when combined with p
w	for data fits, a vector the same length as x giving the importance weight of each observation. This does not need to be normalised. There must be some non zero weights, and all must be finite.
na.rm	remove NAs (default TRUE)

Value

a standard deviation

Examples

```
#' # unweighted:
wmean(x = stats::rnorm(1000))

# weighted:
wmean(x = seq(-2,2,0.1), w = stats::dnorm(seq(-2,2,0.1)))
```

wquantile *Quantile from weighted data with link function support*

Description

This quantile function has different order of parameters from base quantile. It takes a weight and a link function specification which allows us to define the support of the quantile function. It is optimised for imputing the tail of distributions and not speed.

Usage

```
wquantile(p, x, w = NULL, link = "identity", names = TRUE, window = 7)
```

Arguments

p	the probabilities for which to estimate quantiles from the data
x	a set of observations
w	for data fits, a vector the same length as x giving the importance weight of each observation. This does not need to be normalised. There must be some non zero weights, and all must be finite.
link	a link function. Either as name, or a link_fns S3 object. In the latter case this could be derived from a statistical distribution by as.link_fns(<dist_fns>). This supports the use of a prior to define the support of the empirical function, and is designed to prevent tail truncation. Support for the updated quantile function will be the same as the provided prior.
names	name the resulting quantile vector
window	the number of data points to include when estimating the quantile. The closest window points are picked and used as a distance weighted linear interpolation of the weighted CDF in logit-link space. This tends to give good results for extrapolating tails.

Details

The process involves:

1. Link transformation: x values are transformed using the link function: $x_1 = T(x)$.
2. Standardization: Transformed values are standardized: $x_2 = \frac{x_1 - \mu_{w,1}}{\sigma_{w,1}}$, where $\mu_{w,1}$ and $\sigma_{w,1}$ are the weighted mean and standard deviation of x_1.
3. Weighted CDF calculation: The empirical CDF y is calculated from weights w.
4. Logit transformation: y is transformed: $y_2 = \text{logit}(y)$.
5. Local interpolation: For a target probability p, $p_2 = \text{logit}(p)$ is calculated. A window of $window$ points is selected from the (y_2, x_2) pairs around p_2. A weighted linear model is fitted using Gaussian kernel weights based on distance in y_2 space: $K = \exp(-\frac{1}{2}u^2)$, where u is the normalized distance.
6. Quantile estimation: The local model predicts q_2 for p_2.
7. Back-transformation: The quantile is transformed back: $q = T^{-1}(q_2 \cdot \sigma_{w,1} + \mu_{w,1})$.

This is a moderately expensive function to call (in memory terms), as it needs to construct the whole quantile function. if there are multiple calls consider using empirical() to build a quantile function and using that.

Value

a vector of quantiles

Unit tests

```
test = function(rfn,qfn, link,..., n = 100000, tol=1000/(n+10000)) {
  testthat::expect_equal(abs(
   unname(wquantile(c(0.025, 0.5, 0.975),rfn(100000,...),link=link,names=FALSE)-
     qfn(c(0.025, 0.5, 0.975), ...))
  ), c(0,0,0), tolerance=tol)
}

withr::with_seed(123, {

  test(stats::rnorm,stats::qnorm,"identity",n = 10000)
  test(stats::rnorm,stats::qnorm,"identity",mean=4,n = 10000)
  test(stats::rnorm,stats::qnorm,"identity",sd=3, n = 100000, tol=0.05)

  test(stats::rnorm,stats::qnorm,"identity",n = 5000)
  test(stats::rnorm,stats::qnorm,"identity",n = 1000)
  test(stats::rnorm,stats::qnorm,"identity",n = 100)
  test(stats::rnorm,stats::qnorm,"identity",n = 30)

  test(stats::rgamma,stats::qgamma,"log", 4,n = 10000)
  test(stats::rgamma,stats::qgamma,"log", 4,n = 5000)
  test(stats::rgamma,stats::qgamma,"log", 4,n = 1000)
  test(stats::rgamma,stats::qgamma,"log", 4, 3,n = 100)
  test(stats::rgamma,stats::qgamma,"log", 4,n = 30)

  test(stats::runif,stats::qunif,as.link_fns(c(0,10)),0,10)

})
```

Examples

```
# fit weighted data
samples = seq(0,10,0.01)
weights = dgamma2(samples, mean=5, sd=2)

wquantile(c(0.25,0.5,0.75), x = samples, w = weights, link="log")

# compared to the sampled distribution
qgamma2(c(0.25,0.5,0.75), mean=5, sd=2)

# unweighted:
wquantile(p = c(0.25,0.5,0.75), x = stats::rnorm(1000))
qnorm(p = c(0.25,0.5,0.75))
```

wsd *Weighted standard deviation*

Description

Weighted standard deviation

Usage

```
wsd(x, w = NULL, na.rm = TRUE)
```

Arguments

x	either a vector of samples from a distribution X or cut-offs for cumulative probabilities when combined with p
w	for data fits, a vector the same length as x giving the importance weight of each observation. This does not need to be normalised. There must be some non zero weights, and all must be finite.
na.rm	remove NAs (default TRUE)

Value

a standard deviation

Examples

```
# unweighted:
wsd(x = stats::rnorm(1000))

# weighted:
wsd(x = seq(-2,2,0.1), w = stats::dnorm(seq(-2,2,0.1)))
```

Index