

Package: neuralsbi (via r-universe)

October 10, 2026

Title Neural Simulation-Based Inference

Version 0.6.83

Description A native R implementation of neural simulation-based inference (SBI) methods. Given a prior over parameters and a simulator, 'neuralsbi' trains a conditional neural density estimator for likelihood-free Bayesian inference: Neural Posterior Estimation approximates the posterior directly and samples it in a forward pass, Neural Likelihood Estimation learns a surrogate likelihood that is sampled with MCMC, handles repeated independent observations, and can be exported as 'Stan' code for use inside a larger model, and Neural Ratio Estimation trains a classifier for the likelihood-to-evidence ratio instead of either density. Neural estimators run on the 'torch' back end. This package is developed for applied researchers who want an R-native approachable SBI interface with sensible defaults and built-in posterior diagnostics.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

Depends R (>= 4.1.0)

Imports parallel, stats, utils

Suggests torch (>= 0.11.0), cmdstanr (>= 0.7.0), rstan (>= 2.26.0), posterior (>= 1.4.0), ggplot2 (>= 3.4.0), GGally (>= 2.2.0), ggdensity, future (>= 1.30.0), progressr (>= 0.11.0), cli, rlang, testthat (>= 3.0.0), withr, knitr, rmarkdown, svglite, ragg, covr

Additional_repositories <https://mc-stan.org/r-packages/>

Config/testthat/edition 3

Config/Needs/coverage covr

VignetteBuilder knitr

URL <https://neuralsbi.pedrodelima.com/>,
<https://github.com/pedroliman/neuralsbi>

BugReports <https://github.com/pedroliman/neuralsbi/issues>

Config/roxygen2/version 8.1.0

Repository <https://pedroliman.r-universe.dev>

Date/Publication 2026-10-10 05:58:52 UTC

RemoteUrl <https://github.com/pedroliman/neuralsbi>

RemoteRef HEAD

RemoteSha 48817e3e98c089337dccea58f6612fcf5e64d10b

Contents

c2st	3
density_estimator	5
diagnostics	6
embedding_mlp	6
expected_coverage	7
likelihood_fn	8
log_lik	8
log_prob.nsbi_mcmc_posterior	10
log_ratio	11
map_estimate	12
nle	13
npe	15
npe_sequential	18
nre	20
nsbi_mcmc	23
nsbi_parallel	24
nsbi_progress	25
nsbi_simulator	26
pairplot	27
plot_coverage	28
plot_posterior_predictive	29
plot_sbc	30
plot_tarp	30
posterior	31
posterior.nsbi_nle	32
posterior.nsbi_nre	33
posterior_predictive	35
prior_custom	36
prior_families	37
prior_independent	39
prior_normal	40
prior_truncated	40
prior_uniform	41
priors	42
sample	42
sample.nsbi_mcmc_posterior	43

sample.nsbj_posterior	44
sample_posterior	44
sample_prior	45
save_npe	45
sbc	47
simulate_for_sbj	48
stan_export	49
summaries	51
tarp	52
tasks	53
within_support	54

Index	56
-------	----

c2st	<i>Classifier two-sample test (C2ST)</i>
------	--

Description

Trains a classifier to tell the draws in x apart from the draws in y and reports its cross-validated test accuracy. An accuracy near 0.5 means the two sample sets are indistinguishable (good); near 1.0 means they differ. This is the headline metric of the sbjbm benchmark suite, used there to score an approximate posterior against reference draws.

Usage

```
c2st(  
  x,  
  y,  
  n_folds = 5L,  
  seed = NULL,  
  classifier = c("mlp", "logistic"),  
  z_score = TRUE,  
  noise_scale = NULL,  
  hidden = NULL,  
  max_epochs = 10000L,  
  device = "cpu"  
)
```

Arguments

x, y	Matrices of samples (rows = draws, cols = dimensions). The two must be the same width, since they are draws of the same quantity. Row counts need not match; the larger set is subsampled down to the smaller. Pass reference draws as x, following sbjbm.
n_folds	Number of cross-validation folds. At least 2, and no more than the number of draws in the smaller sample set.

seed	Optional seed. Fixes the fold split, the subsampling, the network initialization and the minibatch order.
classifier	"mlp" for the sbibm network, "logistic" for logistic regression.
z_score	Standardize both sample sets by the mean and standard deviation of x before training. On by default, as in sbibm.
noise_scale	Standard deviation of Gaussian noise added to both sample sets after z-scoring. NULL, the default, adds none. Set it when the draws are discrete or lie on a lower-dimensional set, where a classifier can separate the two sides on an artefact of representation.
hidden	Widths of the hidden layers, one entry per layer. NULL, the default, is sbibm's two layers of $10 * d$ units.
max_epochs	Cap on training epochs per fold. A guard, not a budget: the no-improvement rule normally stops training first.
device	Torch device to train the network on: "cpu" (the default), "cuda", "mps", or "gpu"/"auto" for whichever is available. Ignored by classifier = "logistic".

Details

The defaults reproduce the procedure in `sbibm/metrics/c2st.py`, so a number from here is comparable with a published one: z-score both sample sets using the mean and standard deviation of x, fit a two-hidden-layer ReLU network of $10 * d$ units per layer (d = number of columns) with Adam, L2 penalty $1e-4$, minibatches of 200 and a learning rate of $1e-3$, and average the accuracy over 5 shuffled cross-validation folds. Training stops when the epoch loss fails to improve by $1e-4$ for 10 epochs in a row, which is what ends the fit long before `max_epochs`. `sbibm` passes reference draws first, so pass the reference as x: that is the set whose moments set the scale.

The network trains on torch, like every other neural piece of this package, so `classifier = "mlp"` needs torch installed and errors without it. That is the only part of the diagnostics that does.

Budget for the fit. `sbibm`'s own comparison, 10000 draws against 10000 in two dimensions, takes about 25 seconds on one CPU core; at $d = 5$ it is several minutes, because the hidden layers are $10 * d$ wide and the loss takes longer to flatten. Five folds of an all but unbounded epoch budget is what the `sbibm` recipe asks for, and that is what it costs. `hidden` and `max_epochs` cut it down, and `device` moves the fit to a GPU.

`classifier = "logistic"` swaps the network for cross-validated logistic regression. It needs no torch, it answers in milliseconds, and it is what this function used before it was aligned with `sbibm`. It is also linear: it sees a shift in location and is close to blind to two sample sets that share a mean and differ in spread or in the shape of their dependence. Use it as a cheap screen, or in a session with no torch, and report the MLP number.

Unequal sample sizes are balanced by subsampling the larger set, which `sbibm` does not do because it always compares 10000 draws against 10000. Accuracy against unbalanced classes is not a two-sample test: 8000 draws against 2000 identical ones scores 0.8 for a classifier that has learned nothing except to always answer with the bigger class.

Value

A list with the mean cross-validated accuracy and ROC AUC, the per-fold values of each, the number of draws per class, the classifier used, and a one-line reading of the accuracy.

References

- Lopez-Paz, D. and Oquab, M. (2017). Revisiting classifier two-sample tests. *ICLR*. doi:10.48550/arXiv.1610.06545
- Lueckmann, J.-M., Boelts, J., Greenberg, D. S., Goncalves, P. J. and Macke, J. H. (2021). Benchmarking simulation-based inference. *AISTATS*. doi:10.48550/arXiv.2101.04653

Examples

```
a <- matrix(rnorm(400), ncol = 2)
b <- matrix(rnorm(400), ncol = 2)
c2st(a, b, classifier = "logistic", seed = 1)$accuracy
```

density_estimator	<i>Conditional density estimators</i>
-------------------	---------------------------------------

Description

A conditional density estimator learns $q_\phi(\theta \mid x)$. In `neuralsbi` every estimator is trained in *standardized* space and exposes two generics:

Details

- `de_log_prob(de, theta, x)` – log density of theta given x
- `de_sample(de, x, n)` – draw n parameter vectors given a single x

The contract is really $q(\text{target} \mid \text{condition})$: it makes no assumption about which of the two arguments is the parameter. `npe()` calls it with theta as the target and x as the condition, learning the posterior. `nle()` swaps the two, learning the likelihood instead, with the same estimators and the same two generics.

Four estimators ship today:

- "maf" – a Masked Autoregressive Flow (Papamakarios et al., 2017), a stack of invertible autoregressive transforms with an exact change-of-variables density. This is the default, matching Python `sbi`, and requires the `torch` back end.
- "nsf" – a Neural Spline Flow (Durkan et al., 2019): the same autoregressive structure as the MAF, but with a monotonic rational-quadratic spline transform in place of MAF's affine one, which handles sharply non-Gaussian posteriors better. Requires `torch`.
- "mdn" – a Mixture Density Network (neural network -> Gaussian mixture). Requires `torch`.
- "linear_gaussian" – a closed-form conditional Gaussian baseline (least-squares mean, residual covariance). No neural network, no `torch`. It is exact for linear-Gaussian simulators and doubles as a fast baseline and a regression-test oracle.

References

- Papamakarios, G., Pavlakou, T. and Murray, I. (2017). Masked Autoregressive Flow for Density Estimation. *NeurIPS*. doi:10.48550/arXiv.1705.07057
- Durkan, C., Bekasov, A., Murray, I. and Papamakarios, G. (2019). Neural Spline Flows. *NeurIPS*. doi:10.48550/arXiv.1906.04032

diagnostics	<i>Posterior diagnostics</i>
-------------	------------------------------

Description

Tools to check whether a trained posterior is trustworthy:

Details

- `sbcs()` – Simulation-Based Calibration rank statistics
- `expected_coverage()` – nominal vs. empirical credible-interval coverage
- `c2st()` – classifier two-sample test between two sample sets
- `posterior_predictive()` – draw data from the fitted posterior

embedding_mlp	<i>Embedding (summary) networks for structured observations</i>
---------------	---

Description

Raw observations are often high-dimensional or structured (a time series, a set of summary statistics, an image) where feeding x straight into the density estimator wastes capacity. An embedding network learns a low-dimensional summary $h = f_\psi(x)$ jointly with the density estimator, so the conditioning path becomes $q_\phi(\theta \mid f_\psi(x))$. This mirrors `sbi`'s `embedding_net` argument.

Usage

```
embedding_mlp(output_dim = 16L, hidden = c(64L, 64L))
```

Arguments

<code>output_dim</code>	Number of summary features the network emits. This is the effective data dimension the density estimator conditions on.
<code>hidden</code>	Integer vector of hidden-layer widths (ReLU between layers). An empty vector gives a single linear map to <code>output_dim</code> .

Details

`embedding_mlp()` builds a multilayer-perceptron summary network: a stack of fully connected ReLU layers mapping the (standardized) data to a vector of `output_dim` features. Pass the result to `npe()` via `embedding_net`; it is trained end to end with the estimator and its parameters live inside the fitted network, so sampling and `log_prob` route through it automatically.

The embedding consumes the standardized data (the same z-scoring `npe()` applies to x without an embedding), which keeps the summary network's inputs on a common scale. Standardization of the *features* is intentionally left to the network itself; the estimators operate on the raw embedding output.

Value

An nsbi_embedding specification. It carries no torch objects (the network is built lazily at fit time), so it is safe to construct without torch installed.

See Also

[npe\(\)](#)

Examples

```
emb <- embedding_mlp(output_dim = 8, hidden = c(64, 64))
# fit <- npe(prior, simulator, density_estimator = "maf", embedding_net = emb)
```

expected_coverage	<i>Expected coverage of central credible intervals</i>
-------------------	--

Description

Uses the SBC ranks to compare nominal credible levels with the empirical fraction of trials in which the true parameter falls inside the corresponding central interval. Well-calibrated posteriors lie on the diagonal.

Usage

```
expected_coverage(sbc_result, levels = seq(0.05, 0.95, by = 0.05))
```

Arguments

sbc_result	An nsbi_sbc object from sbc() .
levels	Nominal credibility levels to evaluate, each strictly between 0 and 1.

Value

A data frame with nominal and per-parameter empirical coverage.

likelihood_fn	<i>A surrogate likelihood as a plain R function</i>
---------------	---

Description

Fixes an observation and returns `function(theta)`, so the learned likelihood can be handed to anything in R that wants a log-density: `stats::optim()`, an MCMC package, an importance sampler, a profile likelihood. The returned function is vectorized – pass a matrix of parameter values and get one log-likelihood per row.

Usage

```
likelihood_fn(fit, x_obs, ...)
```

Arguments

<code>fit</code>	An <code>nsbi_nle</code> fit from <code>nle()</code> .
<code>x_obs</code>	The observation to condition on. Rows are independent observations; the log-likelihood sums over them.
<code>...</code>	Passed to <code>log_lik()</code> .

Value

A function of `theta` returning a numeric vector of log-likelihoods, with the observation attached as attribute `x_obs`.

Examples

```
prior <- prior_uniform(c(mu = -3), c(mu = 3))
fit <- nle(prior, function(mu) c(y = rnorm(1, mu, 0.5)),
          n_simulations = 1000, density_estimator = "linear_gaussian")

loglik <- likelihood_fn(fit, matrix(rnorm(20, 1, 0.5), ncol = 1))
optimize(function(m) loglik(m), c(-3, 3), maximum = TRUE)$maximum
```

log_lik	<i>Evaluate a surrogate likelihood</i>
---------	--

Description

`log_lik()` evaluates the likelihood learned by `nle()`: $\log q_\phi(x \mid \theta)$, in the original units of both.

Usage

```
log_lik(fit, theta, x, ...)

## S3 method for class 'nsbi_nle'
log_lik(fit, theta, x, sum_iid = TRUE, max_batch = 1e+05, ...)
```

Arguments

fit	An nsbi_nle fit from <code>nle()</code> .
theta	Parameter values: a numeric vector (one parameter set) or an <code>n_theta</code> x <code>dim_theta</code> matrix.
x	Observed data: a numeric vector (one observation) or an <code>n_obs</code> x <code>dim_x</code> matrix whose rows are independent observations.
...	Unused, for S3 consistency.
sum_iid	Sum the log-density over the rows of x (the default). Set FALSE to get the per-observation values instead.
max_batch	Largest number of (theta, x) pairs evaluated in one call to the estimator. Only affects memory and speed.

Details

Rows of x are treated as **independent observations from the same parameter**, so by default the result sums over them, $\sum_i \log q_\phi(x_i | \theta)$. That sum is the whole point of NLE: the estimator is trained on one observation at a time, and the number of observations you condition on afterwards is free.

Value

With `sum_iid = TRUE`, a numeric vector with one entry per row of theta. With `sum_iid = FALSE`, an `n_theta` x `n_obs` matrix.

See Also

`likelihood_fn()` for a closure over a fixed observation, `posterior()` to turn the likelihood into posterior draws.

Examples

```
prior <- prior_uniform(c(mu = -3), c(mu = 3))
fit <- nle(prior, function(mu) c(y = rnorm(1, mu, 0.5)),
          n_simulations = 1000, density_estimator = "linear_gaussian")

x_obs <- matrix(rnorm(20, mean = 1, sd = 0.5), ncol = 1)
grid <- matrix(seq(-2, 2, length.out = 5), ncol = 1)
log_lik(fit, grid, x_obs)
```

```
log_prob.nsbi_mcmc_posterior
```

Posterior log-density

Description

Posterior log-density

Usage

```
## S3 method for class 'nsbi_mcmc_posterior'
log_prob(post, theta, x = NULL, normalize = TRUE, ...)

log_prob(post, theta, x = NULL, ...)

## S3 method for class 'nsbi_posterior'
log_prob(
  post,
  theta,
  x = NULL,
  normalize = TRUE,
  n_normalization = 10000L,
  ...
)
```

Arguments

post	An nsbi_posterior object.
theta	Matrix (or vector) of parameter values to evaluate.
x	Observation to condition on (defaults to x_obs).
normalize	For bounded priors, renormalize by the estimated acceptance probability and return -Inf outside the prior support.
...	Passed to methods.
n_normalization	Number of draws used to estimate the normalizing (acceptance) constant when normalize = TRUE. If none of them land inside the prior support, the estimate is floored at 1 / n_normalization to avoid log(0) and a warning says so – the same warning sample() raises when rejection sampling comes up empty. These draws are internal to the estimate (always taken from a fixed, throwaway stream) and never advance the caller's own RNG, unlike sample() , which is documented to consume randomness.

Value

Numeric vector of log posterior densities. For a posterior built from an [nle\(\)](#) or [nre\(\)](#) fit the value is **unnormalized** – the evidence $p(x)$ is not available – so differences between two theta are meaningful but the absolute level is not.

log_ratio	<i>Evaluate a learned likelihood-to-evidence ratio</i>
-----------	--

Description

log_ratio() evaluates the ratio learned by `nre()`, $\log r(\theta, x) = \log p(x \mid \theta) - \log p(x)$.

Usage

```
log_ratio(fit, theta, x, ...)
```

```
## S3 method for class 'nsbi_nre'
```

```
log_ratio(fit, theta, x, sum_iid = TRUE, max_batch = 1e+05, ...)
```

Arguments

fit	An nsbi_nre fit from <code>nre()</code> .
theta	Parameter values: a numeric vector (one parameter set) or an n_theta x dim_theta matrix.
x	Observed data: a numeric vector (one observation) or an n_obs x dim_x matrix whose rows are independent observations.
...	Unused, for S3 consistency.
sum_iid	Sum the log ratio over the rows of x (the default). Set FALSE to get the per-observation values instead.
max_batch	Largest number of (theta, x) pairs evaluated in one call to the classifier. Only affects memory and speed.

Details

Rows of x are treated as **independent observations from the same parameter**, so by default the result sums over them. That sum is the log-likelihood of the whole data set up to an additive constant that does not depend on theta, which is all a posterior needs and all NRE learns: the evidence term $\log p(x)$ is never estimated separately, so log_ratio() is not a log-likelihood you can compare across observations. Differences between two theta at the same x are exactly the differences in log-likelihood.

Value

With sum_iid = TRUE, a numeric vector with one entry per row of theta. With sum_iid = FALSE, an n_theta x n_obs matrix.

See Also

`posterior()` to turn the ratio into posterior draws, `log_lik()` for the `nle()` equivalent that is a calibrated log-likelihood.

Examples

```
prior <- prior_uniform(c(mu = -3), c(mu = 3))
fit <- nre(prior, function(mu) c(y = rnorm(1, mu, 0.5)),
          n_simulations = 1000, classifier = "logistic")

x_obs <- matrix(rnorm(20, mean = 1, sd = 0.5), ncol = 1)
grid <- matrix(seq(-2, 2, length.out = 5), ncol = 1)
log_ratio(fit, grid, x_obs)
```

map_estimate

Maximum a posteriori (MAP) estimate

Description

Starts from the best of a set of posterior draws and refines with a derivative-free optimizer.

Usage

```
map_estimate(post, x = NULL, n_init = 1000L)
```

Arguments

post	An nsbi_posterior object.
x	Observation to condition on (defaults to x_obs).
n_init	Number of initial draws used to seed the search.

Details

On a posterior from [nle\(\)](#) or [nre\(\)](#) the initial draws come from MCMC, so n_init buys a chain rather than a forward pass. They are cached on the posterior like any other run.

Value

Numeric vector: the MAP parameter estimate. For a bounded prior (from [prior_uniform\(\)](#) or a [prior_custom\(\)](#) with lower/upper), the estimate always falls inside the prior's support – the search never accepts a step that leaves it, the same guarantee [sample\(\)](#) and [log_prob\(\)](#) give. Errors if the seeding draw comes back short of n_init – including empty – which for a bounded prior means the estimator is leaking mass outside the prior support faster than rejection sampling can keep up; there is no starting point to search from in that case, so this stops rather than continuing on a shorter, silently misleading draw.

nle	Neural Likelihood Estimation (NLE)
-----	------------------------------------

Description

`nle()` trains a conditional density estimator on the *other* factorization of the joint: instead of the posterior $p(\theta \mid x)$ that `npe()` learns, it learns a surrogate likelihood $q_\phi(x \mid \theta)$. The posterior then follows from Bayes' rule, $p(\theta \mid x) \propto q_\phi(x \mid \theta) p(\theta)$, and is sampled with MCMC by `posterior()`.

Usage

```
nle(
  prior,
  simulator = NULL,
  n_simulations = 1000,
  sim_args = list(),
  theta = NULL,
  x = NULL,
  density_estimator = c("maf", "mdn", "nsf", "linear_gaussian"),
  n_components = 10L,
  n_transforms = 5L,
  hidden = c(50L, 50L),
  n_bins = 10L,
  tail_bound = 3,
  max_epochs = 2000L,
  batch_size = 200L,
  lr = 5e-04,
  validation_fraction = 0.1,
  patience = 20L,
  n_restarts = 1L,
  clip_grad_norm = 5,
  standardize = TRUE,
  device = "cpu",
  seed = NULL,
  verbose = FALSE
)
```

Arguments

<code>prior</code>	An <code>nsbi_prior</code> (see <code>prior_uniform()</code> , <code>prior_normal()</code>).
<code>simulator</code>	A function called once per parameter set, returning one simulated observation: a numeric vector, a scalar, or a one-row matrix or data frame. Parameters arrive either as named arguments (when the prior's names match the simulator's formal) or as one named vector. Names on the output become the outcome names used in plots. See <code>nsbi_simulator</code> . Ignored if <code>theta</code> and <code>x</code> are given.
<code>n_simulations</code>	Number of prior draws to simulate when <code>simulator</code> is used and <code>theta/x</code> are not supplied.

<code>sim_args</code>	Named list of extra arguments passed to every simulator call: observed data, a time grid, a fixed population size, solver settings. See nsbi_simulator .
<code>theta, x</code>	Optional pre-computed simulations. If supplied, <code>simulator</code> and <code>n_simulations</code> are ignored. Column names on <code>theta</code> (or names on prior's mean/low) and on <code>x</code> are carried through to posterior samples, SBC results, and their plots.
<code>density_estimator</code>	One of "maf" (Masked Autoregressive Flow, needs torch; the default), "mdn" (Mixture Density Network, needs torch), "nsf" (Neural Spline Flow, needs torch), or "linear_gaussian" (closed-form baseline, no torch), or a function <code>function(theta, x)</code> returning a fitted estimator. Note the estimator sees the roles swapped: its target is <code>x</code> and it conditions on <code>theta</code> .
<code>n_components, hidden</code>	MDN settings: number of mixture components and a vector of hidden-layer widths.
<code>n_transforms</code>	MAF/NSF setting: number of stacked autoregressive transforms.
<code>n_bins, tail_bound</code>	NSF settings: number of spline bins per transform (at least 2) and the tail bound. See npe() .
<code>max_epochs, batch_size, lr, validation_fraction, patience</code>	Neural training controls (Adam optimizer, early stopping on validation loss). The defaults (<code>batch_size = 200</code> , <code>lr = 5e-4</code> , <code>validation_fraction = 0.1</code> , <code>patience = 20</code>) match Python <code>sbi</code> ; <code>max_epochs</code> is a high guard cap that early stopping normally reaches first.
<code>n_restarts</code>	Train this many independently initialized networks and keep the one with the best validation loss (guards against bad initializations and MDN mode collapse).
<code>clip_grad_norm</code>	Maximum gradient norm during training (Inf disables clipping). The learning rate also decays 2x after 10 epochs without validation improvement.
<code>standardize</code>	Whether to z-score <code>theta</code> and <code>x</code> before training (strongly recommended; default TRUE).
<code>device</code>	Where to train the neural estimator: "cpu" (the default), "cuda", "mps", or "gpu"/"auto" to resolve CUDA -> MPS -> CPU (mirroring Python <code>sbi</code> 's "gpu"). CPU is the default on purpose – matching <code>sbi</code> , not auto-selecting a GPU – and "cuda"/"mps" error if the requested device is not actually available rather than falling back silently, so a typo or a missing driver is not mistaken for a slow CPU run. Only "gpu"/"auto" falls back to CPU without complaint, since it never named a specific device. Ignored (with no error) by "linear_gaussian", which has no GPU concept. For the small networks typical of SBI models (the SIR example, say), "mps"/"cuda" can be <i>slower</i> than CPU – per-kernel launch overhead dominates until the net and batch are large – so try CPU first and switch only if profiling shows a win.
<code>seed</code>	Optional integer seed for reproducibility.
<code>verbose</code>	Print training progress.

Value

An object of class `nsbi_nle`. Evaluate the surrogate likelihood with [log_lik\(\)](#), turn it into a posterior with [posterior\(\)](#), or export it to Stan with [stan_code\(\)](#).

When NLE beats NPE

An NPE fit learns the posterior for one fixed data dimension, chosen at training time. If the observation is n independent trials from the same parameter, NPE must be retrained for every n , or handed summary statistics that throw information away. NLE learns the density of a *single* trial, so the log-likelihood of n trials is a sum of n evaluations:

$$\log p(x_1, \dots, x_n \mid \theta) = \sum_{i=1}^n \log q_\phi(x_i \mid \theta).$$

Train once, then condition on 50 trials or 5000 without touching the network. The learned likelihood is also a plain differentiable function of θ , so it can be embedded in a larger model written by hand – see `stan_code()`.

The trade-off is real and worth stating: posterior draws now cost an MCMC run rather than a forward pass, and for a single fixed observation with high-dimensional data NPE is usually the better choice.

References

Papamakarios, G., Sterratt, D. and Murray, I. (2019). Sequential Neural Likelihood: Fast Likelihood-free Inference with Autoregressive Flows. *AISTATS*. doi:10.48550/arXiv.1805.07226

See Also

`log_lik()` and `likelihood_fn()` to evaluate the surrogate, `posterior()` to sample it, `stan_code()` to hand it to Stan.

Examples

```
# One noisy measurement per simulator call; the observation is 200 of them.
prior <- prior_uniform(c(mu = -3, log_sigma = -1),
                      c(mu = 3, log_sigma = 1))
simulator <- function(mu, log_sigma) c(y = rnorm(1, mu, exp(log_sigma)))

fit <- nle(prior, simulator, n_simulations = 2000,
          density_estimator = "linear_gaussian")

x_obs <- matrix(rnorm(200, mean = 1, sd = 0.5), ncol = 1)
log_lik(fit, theta = c(1, log(0.5)), x = x_obs)
```

Description

`npe()` is the main entry point. Given a prior and either a simulator (which it will call) or a set of pre-computed simulations (`theta`, `x`), it trains a conditional density estimator whose output directly approximates the posterior $p(\theta \mid x)$. This is single-round, *amortized* NPE: after training once, you can condition on any observation without re-simulating.

Usage

```
npe(
  prior,
  simulator = NULL,
  n_simulations = 1000,
  sim_args = list(),
  theta = NULL,
  x = NULL,
  density_estimator = c("maf", "mdn", "nsf", "linear_gaussian"),
  n_components = 10L,
  n_transforms = 5L,
  hidden = c(50L, 50L),
  n_bins = 10L,
  tail_bound = 3,
  embedding_net = NULL,
  max_epochs = 2000L,
  batch_size = 200L,
  lr = 5e-04,
  validation_fraction = 0.1,
  patience = 20L,
  n_restarts = 1L,
  clip_grad_norm = 5,
  standardize = TRUE,
  device = "cpu",
  seed = NULL,
  verbose = FALSE
)
```

Arguments

<code>prior</code>	An <code>nsbi_prior</code> (see prior_uniform() , prior_normal()).
<code>simulator</code>	A function called once per parameter set, returning one simulated observation: a numeric vector, a scalar, or a one-row matrix or data frame. Parameters arrive either as named arguments (when the prior's names match the simulator's formal) or as one named vector. Names on the output become the outcome names used in plots. See nsbi_simulator . Ignored if <code>theta</code> and <code>x</code> are given.
<code>n_simulations</code>	Number of prior draws to simulate when <code>simulator</code> is used and <code>theta/x</code> are not supplied.
<code>sim_args</code>	Named list of extra arguments passed to every simulator call: observed data, a time grid, a fixed population size, solver settings. See nsbi_simulator .
<code>theta, x</code>	Optional pre-computed simulations. If supplied, <code>simulator</code> and <code>n_simulations</code> are ignored. Column names on <code>theta</code> (or names on prior's mean/low) and on <code>x</code> are carried through to posterior samples, SBC results, and their plots.
<code>density_estimator</code>	One of "maf" (Masked Autoregressive Flow, needs torch; the default, matching Python sbi), "mdn" (neural Mixture Density Network, needs torch), "nsf"

	(Neural Spline Flow, needs torch), or "linear_gaussian" (closed-form baseline, no torch), or a function <code>function(theta, x)</code> returning a fitted estimator.
<code>n_components, hidden</code>	MDN settings: number of mixture components (default 10, as in <code>sbi</code>) and a vector of hidden-layer widths.
<code>n_transforms</code>	MAF/NSF setting: number of stacked autoregressive transforms (default 5, as in <code>sbi</code>).
<code>n_bins, tail_bound</code>	NSF settings: number of spline bins per transform (at least 2, since the spline needs an interior derivative to fit) and the half-width of the interval the spline acts on (outside it the transform is the identity).
<code>embedding_net</code>	Optional summary network built with <code>embedding_mlp()</code> . When supplied, the neural estimators condition on the learned features $f_\psi(x)$ instead of the raw data, training the embedding jointly. Ignored (with a warning) by "linear_gaussian" and by a function-valued <code>density_estimator</code> , since a custom fitter only ever receives <code>theta</code> and <code>x</code> .
<code>max_epochs, batch_size, lr, validation_fraction, patience</code>	Neural training controls (Adam optimizer, early stopping on validation loss). The defaults (<code>batch_size = 200</code> , <code>lr = 5e-4</code> , <code>validation_fraction = 0.1</code> , <code>patience = 20</code>) match Python <code>sbi</code> ; <code>max_epochs</code> is a high guard cap that early stopping normally reaches first.
<code>n_restarts</code>	Train this many independently initialized networks and keep the one with the best validation loss (guards against bad initializations and MDN mode collapse).
<code>clip_grad_norm</code>	Maximum gradient norm during training (Inf disables clipping). The learning rate also decays 2x after 10 epochs without validation improvement.
<code>standardize</code>	Whether to z-score <code>theta</code> and <code>x</code> before training (strongly recommended; default <code>TRUE</code>).
<code>device</code>	Where to train the neural estimator: "cpu" (the default), "cuda", "mps", or "gpu"/"auto" to resolve CUDA -> MPS -> CPU (mirroring Python <code>sbi</code> 's "gpu"). CPU is the default on purpose – matching <code>sbi</code> , not auto-selecting a GPU – and "cuda"/"mps" error if the requested device is not actually available rather than falling back silently, so a typo or a missing driver is not mistaken for a slow CPU run. Only "gpu"/"auto" falls back to CPU without complaint, since it never named a specific device. Ignored (with no error) by "linear_gaussian", which has no GPU concept. For the small networks typical of SBI models (the SIR example, say), "mps"/"cuda" can be <i>slower</i> than CPU – per-kernel launch overhead dominates until the net and batch are large – so try CPU first and switch only if profiling shows a win.
<code>seed</code>	Optional integer seed for reproducibility.
<code>verbose</code>	Print training progress.

Value

An object of class `nsbi_npe`. Turn it into a usable posterior with `posterior()`, or sample directly with `sample()`. Save it to disk with `save_npe()`: a torch-backed fit does not survive `saveRDS()`.

Parallel simulation and progress

The simulator runs sequentially unless you declare a **future** plan – `library(future); plan(multisession)` – in which case the simulations are spread across workers. Simulation and training both report progress with an ETA. See [nsbi_parallel](#) and [nsbi_progress](#).

Examples

```
prior <- prior_uniform(c(mu = -2, nu = -2), c(mu = 2, nu = 2))
simulator <- function(mu, nu) c(a = mu + rnorm(1, sd = 0.1),
                                b = nu + rnorm(1, sd = 0.1))
fit <- npe(prior, simulator, n_simulations = 2000,
           density_estimator = "linear_gaussian")
post <- posterior(fit, x_obs = c(0.8, 0.6))
draws <- sample(post, 1000)
```

npe_sequential

Sequential NPE with truncated-prior proposals (TSNPE)

Description

Multi-round NPE targeting a single observation `x_obs`. Single-round `npe()` spends its simulation budget across the whole prior; when only one observation matters, most of those simulations land in regions the posterior never visits. `npe_sequential()` implements truncated sequential NPE (TSNPE, Deistler et al. 2022): after each round the prior is truncated to the highest-probability region of the current posterior estimate, and the next round’s parameters are drawn from that truncated prior. Because every proposal is proportional to the prior on its support, the standard NPE loss stays valid – no importance-weight or atomic correction is needed, which is what makes TSNPE the simplest correct sequential scheme.

Usage

```
npe_sequential(
  prior,
  simulator,
  x_obs,
  n_rounds = 2L,
  n_simulations = 1000L,
  sim_args = list(),
  density_estimator = c("maf", "mdn", "nsf", "linear_gaussian"),
  epsilon = 1e-04,
  n_truncation_samples = 5000L,
  max_proposal_batches = 200L,
  seed = NULL,
  verbose = FALSE,
  ...
)
```

Arguments

prior	An nsbi_prior (see prior_uniform() , prior_normal()).
simulator	A function called once per parameter set, returning one simulated observation; see nsbi_simulator .
x_obs	The observation to target, as one numeric vector, one-row matrix or one-row data frame whose width matches the simulator's output. Sequential inference concentrates simulations around the posterior for this observation.
n_rounds	Number of rounds, at least 1. Round 1 is ordinary single-round NPE.
n_simulations	Simulation budget per round; either a scalar or a vector of length n_rounds.
sim_args	Named list of extra arguments passed to every simulator call; see nsbi_simulator .
density_estimator	Passed to npe() each round.
epsilon	Mass cut for the truncation: the proposal region is the 1 - epsilon highest-probability region of the current posterior. Must be a single number strictly between 0 and 1.
n_truncation_samples	Posterior draws used to locate the truncation threshold each round.
max_proposal_batches	Cap on rejection-sampling batches per round.
seed	Optional integer seed for reproducibility.
verbose	Print per-round progress.
...	Passed to npe() (estimator and training settings). Checked before round 1 simulates, the same as the arguments above.

Details

The rounds accumulate: each round's estimator is trained on all simulations so far. The final fit is returned as an nsbi_npe (subclass nsbi_snpe) and works with [posterior\(\)](#), [sample\(\)](#) and the diagnostics, but unlike single-round NPE it is *not* amortized: it is only trustworthy at (or very near) x_obs.

Proposal draws are obtained by rejection: prior candidates are kept when their posterior log-density clears the epsilon-quantile threshold of the current posterior's own draws. If the posterior is much narrower than the prior the acceptance rate falls; the round then stops after max_proposal_batches batches and continues with the draws it has, with a warning.

Value

An object of class c("nsbi_snpe", "nsbi_npe") with a rounds field recording per-round budgets, acceptance rates, and thresholds.

References

Deistler, Goncalves & Macke (2022), "Truncated proposals for scalable and hassle-free simulation-based inference", NeurIPS. [doi:10.48550/arXiv.2210.04815](https://arxiv.org/abs/2210.04815)

Examples

```
prior <- prior_normal(mean = c(mu = 0, nu = 0), sd = 1)
simulator <- function(mu, nu) c(mu, nu) + rnorm(2, sd = 0.3)
fit <- npe_sequential(prior, simulator, x_obs = c(0.5, -0.5),
                     n_rounds = 2, n_simulations = 1000,
                     density_estimator = "linear_gaussian")
post <- posterior(fit, x_obs = c(0.5, -0.5))
draws <- sample(post, 1000)
```

nre

Neural Ratio Estimation (NRE)

Description

nre() learns the third factorization of the joint. npe() learns the posterior and nle() learns the likelihood; nre() learns neither density but their ratio, $r(\theta, x) = p(x | \theta)/p(x)$, by training a binary classifier to tell (θ, x) pairs drawn from the joint apart from pairs whose parameter came from a different simulation. The posterior follows from Bayes' rule, $p(\theta | x) \propto r(\theta, x) p(\theta)$, and is sampled with MCMC by posterior().

Usage

```
nre(
  prior,
  simulator = NULL,
  n_simulations = 1000,
  sim_args = list(),
  theta = NULL,
  x = NULL,
  classifier = c("resnet", "mlp", "linear", "logistic"),
  num_atoms = 10L,
  hidden = 50L,
  n_blocks = 2L,
  embedding_net = NULL,
  max_epochs = 2000L,
  batch_size = 200L,
  lr = 5e-04,
  validation_fraction = 0.1,
  patience = 20L,
  n_restarts = 1L,
  clip_grad_norm = 5,
  standardize = TRUE,
  device = "cpu",
  seed = NULL,
  verbose = FALSE
)
```

Arguments

prior	An nsbi_prior (see prior_uniform() , prior_normal()).
simulator	A function called once per parameter set, returning one simulated observation: a numeric vector, a scalar, or a one-row matrix or data frame. Parameters arrive either as named arguments (when the prior's names match the simulator's formal) or as one named vector. Names on the output become the outcome names used in plots. See nsbi_simulator . Ignored if theta and x are given.
n_simulations	Number of prior draws to simulate when simulator is used and theta/x are not supplied.
sim_args	Named list of extra arguments passed to every simulator call: observed data, a time grid, a fixed population size, solver settings. See nsbi_simulator .
theta, x	Optional pre-computed simulations. If supplied, simulator and n_simulations are ignored. Column names on theta (or names on prior's mean/low) and on x are carried through to posterior samples, SBC results, and their plots.
classifier	One of "resnet" (a residual MLP, needs torch; the default, matching Python sbi), "mlp" (a plain MLP, needs torch), "linear" (a single linear layer on the raw inputs, needs torch), or "logistic" (a closed-form logistic regression on quadratic features, no torch), or a function function(theta, x) returning a fitted ratio estimator – one whose class has a de_log_ratio() method, which is the only thing the rest of the pipeline asks of it.
num_atoms	Number of parameter values the classifier compares per simulation: one true and num_atoms - 1 contrasts. Clamped to the minibatch size, as in sbi.
hidden	Width of the classifier's hidden layers. One number, not a per-layer vector as in npe() : sbi's classifiers use a single width throughout, and n_blocks is what sets the depth.
n_blocks	Depth of the classifier: residual blocks for "resnet", hidden layers for "mlp". Ignored by "linear" and "logistic". sbi fixes its MLP at two hidden layers and only lets num_blocks reach the residual net; here the one argument sets both.
embedding_net	Optional summary network built with embedding_mlp() . The classifier then sees $(\theta, f_\psi(x))$, with the embedding trained jointly. Ignored (with a warning) by "logistic" and by a function-valued classifier, since a custom fitter only ever receives theta and x.
max_epochs, batch_size, lr, validation_fraction, patience	Neural training controls (Adam optimizer, early stopping on validation loss). The defaults (batch_size = 200, lr = 5e-4, validation_fraction = 0.1, patience = 20) match Python sbi; max_epochs is a high guard cap that early stopping normally reaches first.
n_restarts	Train this many independently initialized networks and keep the one with the best validation loss (guards against bad initializations and MDN mode collapse).
clip_grad_norm	Maximum gradient norm during training (Inf disables clipping). The learning rate also decays 2x after 10 epochs without validation improvement.
standardize	Whether to z-score theta and x before training (strongly recommended; default TRUE).

device	Where to train the neural estimator: "cpu" (the default), "cuda", "mps", or "gpu"/"auto" to resolve CUDA -> MPS -> CPU (mirroring Python sbi's "gpu"). CPU is the default on purpose – matching sbi, not auto-selecting a GPU – and "cuda"/"mps" error if the requested device is not actually available rather than falling back silently, so a typo or a missing driver is not mistaken for a slow CPU run. Only "gpu"/"auto" falls back to CPU without complaint, since it never named a specific device. Ignored (with no error) by "linear_gaussian", which has no GPU concept. For the small networks typical of SBI models (the SIR example, say), "mps"/"cuda" can be <i>slower</i> than CPU – per-kernel launch overhead dominates until the net and batch are large – so try CPU first and switch only if profiling shows a win.
seed	Optional integer seed for reproducibility.
verbose	Print training progress.

Value

An object of class `nsbi_nre`. Evaluate the learned ratio with `log_ratio()` or turn it into a posterior with `posterior()`.

Why a classifier instead of a density

A density estimator has to spend capacity describing the shape of a distribution – normalization, tails, the lot – even where that shape does not affect the posterior. A classifier only has to say which of two parameter values explains the data better, and the optimal classifier's logit *is* the log ratio. That makes NRE the natural choice when the data are high-dimensional or awkward to model directly (discrete counts, mixed types, anything a flow handles badly) but easy to discriminate.

Like `nle()` and unlike `npe()`, the ratio is learned for a single observation, so the log-likelihood of n independent trials from the same parameter is a sum:

$$\log \frac{p(x_1, \dots, x_n | \theta)}{\prod_i p(x_i)} = \sum_{i=1}^n \log r(\theta, x_i).$$

Train once, condition on as many trials as you like. The price is the same one `nle()` pays: posterior draws cost an MCMC run rather than a forward pass.

The training objective

The default is the *atomic* loss of Durkan et al. (2020), which is what sbi's NRE (an alias for NRE_B) trains. For each simulation (θ_i, x_i) in a minibatch, `num_atoms` - 1 contrasting parameters are taken from the other simulations in that batch, and the classifier is scored on a `num_atoms`-way softmax over which of them produced x_i :

$$\mathcal{L} = -\frac{1}{b} \sum_i \left[f(\theta_i, x_i) - \log \sum_k \exp f(\theta_{ik}, x_i) \right].$$

More atoms mean a harder discrimination problem and a sharper ratio, at a linear cost in forward passes per epoch. `num_atoms = 10` is sbi's default and this one. The softmax is invariant to adding any function of x to f , so the learned ratio is calibrated up to a constant at each fixed observation – exactly what a posterior needs, and the reason `log_prob()` on an NRE posterior is unnormalized.

Standardization has no Jacobian here

The estimators in `npe()` and `nle()` train on z-scored data and need a change-of-variables term to report densities in the original units. A ratio needs none: both $p(x \mid \theta)$ and $p(x)$ pick up the same Jacobian factor and it cancels, so `log_ratio()` in standardized space is already the ratio in the units the simulator returned.

References

Hermans, J., Begy, V. and Louppe, G. (2020). Likelihood-free MCMC with Amortized Approximate Ratio Estimators. *ICML*. doi:10.48550/arXiv.1903.04057

Durkan, C., Murray, I. and Papamakarios, G. (2020). On Contrastive Learning for Likelihood-free Inference. *ICML*. doi:10.48550/arXiv.2002.03712

See Also

`log_ratio()` to evaluate the ratio, `posterior()` to sample it, `nle()` for the likelihood factorization and `npe()` for the posterior one.

Examples

```
# One noisy measurement per simulator call; the observation is 50 of them.
prior <- prior_uniform(c(mu = -3), c(mu = 3))
simulator <- function(mu) c(y = rnorm(1, mu, 0.5))

fit <- nre(prior, simulator, n_simulations = 2000, classifier = "logistic")

x_obs <- matrix(rnorm(50, mean = 1, sd = 0.5), ncol = 1)
log_ratio(fit, theta = c(1), x = x_obs)
```

nsbi_mcmc

MCMC over a learned likelihood

Description

An `nle()` fit gives an unnormalized posterior, $q_\phi(x \mid \theta) p(\theta)$, but no way to draw from it directly. `neuralnsbi` samples it with a univariate slice sampler (Neal, 2003) run over many chains at once.

Details

Slice sampling is the default in Python `sbi` for the same reason it is the default here: it has no step size to tune, adapts its scale to the target as it goes, and handles bounded supports without any special casing, because a prior that returns $-\text{Inf}$ outside its support simply shrinks the slice interval.

The vectorization runs across chains rather than across dimensions. Every chain proposes its next value for the same coordinate at the same time, so one step costs one batched call to the density estimator instead of `n_chains` separate forward passes. With a neural likelihood that difference is the whole running time.

References

Neal, R. M. (2003). Slice sampling. *The Annals of Statistics* 31(3), 705-767. doi:[10.1214/aos/1056562461](https://doi.org/10.1214/aos/1056562461)

See Also

`posterior.nsbi_nle()` for the arguments that control it, and `stan_code()` for handing the same likelihood to NUTS instead.

nsbi_parallel

Running the simulator in parallel

Description

Every `neuralsbi` function that calls a simulator – `npe()`, `simulate_for_sbi()`, `npe_sequential()`, `sbc()`, `tarp()`, `posterior_predictive()` – runs it through one execution path. That path is sequential by default and parallel as soon as you declare a **future** plan:

Details

```
library(future)
plan(multisession)      # or multicore, cluster, batchtools, ...
fit <- npe(prior, simulator, n_simulations = 10000)
```

Nothing else changes: no argument to set, no variant of the function to call. With no plan (or `plan(sequential)`) the simulations run in the current process, and `neuralsbi` mentions the two lines above once per session. Silence that hint with `options(neuralsbi.parallel_hint = FALSE)`.

Random numbers

Each *simulation* gets its own L’Ecuyer-CMRG random-number stream, derived from the session’s RNG state at the moment simulation starts. So `set.seed(42)` before a `fit` (or `npe(..., seed = 42)`) gives the same simulations whatever the plan and whatever the worker count: results depend on the seed alone.

Because the simulations are drawn from separate streams, the simulator no longer consumes the caller’s RNG state directly; the state advances by one draw per simulation phase.

Scheduling

Under a multi-process plan the draws are dealt out to workers in batches, because one future per simulation would spend longer shipping the simulator to a worker than running it. Batch sizes follow the worker count and are not a setting: they cannot change a result, since every simulation has its own RNG stream. Running sequentially there are no batches at all, just a loop.

Each batch ships the simulator, and everything its environment captures, to a worker. A simulator closing over a large object therefore pays for that object once per batch, which is a reason to pass it through `sim_args` instead of capturing it.

What is not parallelized

Training runs in the calling process. Torch tensors and modules are external pointers that cannot be shipped to a worker, and libtorch already uses multiple threads internally (`torch::torch_set_num_threads()`). The same goes for the posterior-sampling loops in `sbc()` and `tarp()` – those call the trained network, not the simulator, and are reported with their own progress bar.

See Also

[nsbi_progress](#) for controlling progress bars.

nsbi_progress

Progress reporting

Description

Long-running steps – running the simulator, training a neural density estimator – report progress through one mechanism, so a fit that simulates and then trains shows the same style of bar for both phases. Each bar reports the elapsed time and an ETA extrapolated from the work done so far.

Details

If the **progressr** package is installed, `neuralnsbi` emits standard `progressr` updates, so `progressr::handlers()` and `progressr::with_progress()` control reporting exactly as they do for any other `progressr`-aware package:

```
library(progressr)
handlers(global = TRUE)
handlers("cli")          # or "progress", "txtprogressbar", "rstudio"
fit <- npe(prior, simulator, n_simulations = 10000)
```

Without `progressr` installed (or when it is installed but no handler is registered) `neuralnsbi` draws its own bar, which needs no extra packages.

Reporting is controlled by `options(neuralnsbi.progress = )`:

- "auto" (default) – report in interactive sessions, stay quiet in scripts, R CMD check, and knitr.
- TRUE – always report, interactive or not.
- FALSE – never report.
- "builtin" – always report with the built-in bar, ignoring `progressr`.

Training progress

While a neural estimator trains, one progress step is one epoch. The total is not known in advance because training stops early, so the bar targets the epoch at which training would stop if the validation loss never improved again (best epoch + patience). The target moves out whenever the validation loss improves, so the ETA is a running estimate, not a promise.

See Also

[nsbi_parallel](#) for running the simulator across cores.

nsbi_simulator

The simulator contract

Description

A simulator in `neuralnsbi` is called **once per parameter set** and returns **one simulated observation**. Everything that calls a simulator – `npe()`, `simulate_for_sbi()`, `npe_sequential()`, `sbc()`, `tarp()`, `posterior_predictive()` – uses this contract.

How parameters arrive

Two signatures are accepted, and which one applies is decided once per run from `formals(simulator)`. If every parameter name appears among the simulator’s formals, the parameters are passed **by name**, one scalar each:

```
prior <- prior_uniform(low = c(mu = -5, sigma = 0.1),
                      high = c(mu = 5, sigma = 3))
```

```
simulator <- function(mu, sigma) {
  y <- rnorm(100, mean = mu, sd = sigma)
  c(mean = mean(y), sd = sd(y))
}
```

Otherwise the whole **named parameter vector** goes to the first argument:

```
simulator <- function(theta) {
  y <- rnorm(100, mean = theta["mu"], sd = theta["sigma"])
  c(mean = mean(y), sd = sd(y))
}
```

An unnamed prior always takes the vector form, and so does a prior whose names are not syntactic R names (`"beta[1]"`), since those can never match a formal. A partial match warns and names the parameters that found no formal, because the vector form then hands every parameter to the first argument and the remaining formals fall back to their defaults.

Everything else the simulator needs

Observed data, a time grid, a population size, a design matrix, a solver tolerance: anything that is not a calibrated parameter goes in `sim_args`, a named list forwarded to every simulator call.

```
simulator <- function(alpha, beta, sigma, x_grid) {
  rnorm(length(x_grid), mean = alpha + beta * x_grid, sd = sigma)
}
```

```
fit <- npe(prior, simulator, n_simulations = 10000,
         sim_args = list(x_grid = seq(-1, 1, length.out = 50)))
```

A list is used rather than `...` because every one of `npe()`'s own arguments would otherwise be exposed to R's partial matching: `x`, `theta`, `n` and `seed` are all natural names for a simulator argument and all of them would be captured silently.

What the simulator returns

One observation per call: a numeric vector of length `d`, a scalar, or a one-row matrix or data frame. Names on the vector, or `colnames` on the matrix or data frame, become the outcome names used in summaries and plots. Every draw must agree on `d`. Lists, character or factor columns, and anything with more than one row are rejected by name.

One sharp edge: arithmetic on a single named parameter keeps the name, so `theta["beta"] / theta["gamma"]` returns a scalar named `beta` and would quietly become an outcome called `beta`. Vectors are unaffected – `theta["alpha"] + theta["beta"] * x` comes back unnamed. Use `unname()`, or name the outcome deliberately with `c(r0 = ...)`.

Failed simulations

A draw whose output contains NA, NaN or an infinite value is dropped, together with its parameters, and one warning per run reports the count and the rate. A single non-finite value would otherwise poison the training loss for the whole fit and surface much later as a NaN validation loss. The parameters are checked the same way, so a pre-computed pair with a missing `theta` is dropped instead of reaching the density estimator.

Dropping conditions on the simulator having succeeded. When failure depends on the parameters – an ODE that diverges for large `beta`, a model that returns NA outside a stability region – the surviving draws are no longer a sample from the prior, and the fit targets the posterior *given success* rather than the posterior. A high drop rate is a modelling signal, and the honest fix is usually to handle the failure inside the simulator.

See Also

[nsbi_parallel](#) for how the calls are spread across workers.

pairplot

Visualize posterior samples

Description

A pair plot built on `GGally::ggpairs()`: 1-D marginal densities on the diagonal and 2-D highest-density regions (via `ggdensity::geom_hdr()`) in the lower triangle, with optional markers for a reference (e.g. true) parameter value. Analogous to Python `sbi`'s `pairplot`.

Usage

```
pairplot(
  samples,
  truth = NULL,
  labels = NULL,
```

```

    limits = NULL,
    col = "steelblue",
    alpha = 0.4,
    ...
  )

```

Arguments

<code>samples</code>	A matrix of posterior draws (rows = draws), or an <code>nsbi_samples</code> object.
<code>truth</code>	Optional reference parameter vector to overlay.
<code>labels</code>	Optional parameter labels. Defaults to <code>colnames(samples)</code> (set automatically for <code>sample()</code> draws from a fit with named parameters – see <code>prior_uniform()/prior_normal()</code> or <code>theta[1]</code> , <code>theta[2]</code> , Labels that parse as R syntax (" <code>beta[1]</code> ", " <code>rho</code> ") render as their plotmath symbol.
<code>limits</code>	Optional list (one <code>c(lo, hi)</code> per parameter, in column order) or matrix of per-parameter axis limits. Defaults to each parameter's own data range (padded 5%), applied consistently across every panel that plots it – without this, the lower-triangle and diagonal panels for the same parameter can each draw a different range, since <code>ggdensity::geom_hdr()</code> and <code>ggplot2::geom_density()</code> estimate their density grid independently per panel.
<code>col</code>	Density-region and marginal-density fill colour.
<code>alpha</code>	Marginal-density fill transparency. The lower-triangle highest-density regions shade themselves by probability level (99/95/80/50%) instead, via <code>ggdensity</code> 's own alpha mapping.
<code>...</code>	Passed to the lower-triangle <code>ggdensity::geom_hdr()</code> layer.

Value

A `ggmatrix` object (also drawn as a side effect), invisibly.

<code>plot_coverage</code>	<i>Plot nominal vs. empirical credible-interval coverage</i>
----------------------------	--

Description

Well-calibrated posteriors lie on the diagonal. Curves above the diagonal mean the posterior is too wide (conservative); below means overconfident. A shaded band shows the Monte-Carlo uncertainty from the finite number of SBC trials.

Usage

```
plot_coverage(sbc_result, levels = seq(0.05, 0.95, by = 0.05))
```

Arguments

sbc_result	An nsbi_sbc object from sbc() .
levels	Nominal credibility levels to evaluate, each strictly between 0 and 1. Passed to expected_coverage() , which checks them.

Value

A ggplot object (also drawn as a side effect), invisibly.

plot_posterior_predictive

Plot posterior predictive checks

Description

Compares data simulated from posterior parameter draws (see [posterior_predictive\(\)](#)) with the observed data, one marginal histogram per data dimension with the observation marked. If the observation falls in the tails of the predictive distribution, the model (or the fit) does not reproduce the data it is conditioned on.

Usage

```
plot_posterior_predictive(pred, x_obs, labels = NULL, bins = 30L)
```

Arguments

pred	A matrix of predictive draws from posterior_predictive() .
x_obs	The observed data vector the posterior was conditioned on.
labels	Optional labels for the data dimensions. Defaults to <code>colnames(pred)</code> (set automatically when the simulator names its output columns) or <code>x[1]</code> , <code>x[2]</code> , Labels that parse as R syntax render as their plotmath symbol.
bins	Number of histogram bins.

Value

A ggplot object (also drawn as a side effect), invisibly.

plot_sbc	<i>Plot an SBC rank histogram</i>
----------	-----------------------------------

Description

Uniform bars indicate calibration; a U shape means the posterior is too narrow (overconfident); an inverted-U means it is too wide.

Usage

```
plot_sbc(sbc_result, param = 1L, bins = 20L)
```

Arguments

sbc_result	An nsbi_sbc object from sbc() .
param	Which parameter to plot: an index (default 1), or a parameter name when sbc() was run against a fit with named parameters (see prior_uniform()/prior_normal()). The title uses that parameter's name when there is one, rendered as a plotmath symbol if the name parses as R syntax.
bins	Number of histogram bins.

Value

A ggplot object (also drawn as a side effect), invisibly.

plot_tarp	<i>Plot TARP expected coverage</i>
-----------	------------------------------------

Description

Draws the expected coverage probability (ECP) curve from [tarp\(\)](#) against the nominal credibility level. A calibrated posterior lies on the diagonal; a curve above the diagonal means the posterior is too wide (conservative), below means overconfident. The shaded band shows the Monte-Carlo uncertainty from the finite number of TARP trials.

Usage

```
plot_tarp(tarp_result)
```

Arguments

tarp_result	An nsbi_tarp object from tarp() .
-------------	---

Value

A ggplot object (also drawn as a side effect), invisibly.

posterior

Posterior objects

Description

A posterior wraps a trained `npe()` fit together with (optionally) a default observation `x_obs`. It knows how to draw posterior samples, evaluate the posterior log-density, and find the maximum-a-posteriori (MAP) estimate. All transforms between standardized training space and the original parameter space are handled internally.

The three inference methods reach a posterior by different routes, and `posterior()` hides the difference. An `npe()` fit already *is* a posterior estimator, so the returned object samples with a forward pass. An `nle()` fit only knows the likelihood and an `nre()` fit only the likelihood ratio, so both return an object that samples with MCMC and takes the extra arguments that implies.

Usage

```
posterior(fit, x_obs = NULL, ...)

## Default S3 method:
posterior(fit, x_obs = NULL, ...)

## S3 method for class 'nsbi_npe'
posterior(fit, x_obs = NULL, ...)
```

Arguments

<code>fit</code>	An <code>nsbi_npe</code> object from <code>npe()</code> or <code>npe_sequential()</code> , an <code>nsbi_nle</code> object from <code>nle()</code> , or an <code>nsbi_nre</code> object from <code>nre()</code> .
<code>x_obs</code>	Optional default observation to condition on. If supplied it becomes the default <code>x</code> for <code>sample()</code> , <code>log_prob()</code> and <code>map_estimate()</code> . For an NLE or NRE fit, rows of <code>x_obs</code> are independent observations.
<code>...</code>	Passed to methods. See <code>posterior.nsbi_nle()</code> and <code>posterior.nsbi_nre()</code> for the MCMC controls those fits accept.

Details

For bounded priors, samples that fall outside the prior support are rejected ("leakage" correction), and `log_prob()` is renormalized by the estimated acceptance probability so it integrates to one over the support.

Value

An `nsbi_posterior` object.

See Also

`save_npe()`, which is how a torch-backed fit gets to disk and back; `readRDS()` returns one whose network is dead, and `posterior()` says so rather than failing later.

posterior.nsbi_nle *Posterior from a neural likelihood*

Description

Bayes' rule turns an `nle()` fit into a posterior, $p(\theta \mid x) \propto q_\phi(x \mid \theta) p(\theta)$, but the result has no closed form and no direct sampler. `posterior()` on an `nsbi_nle` therefore returns an object that samples with MCMC, and `sample()` on it runs a chain rather than a forward pass.

Usage

```
## S3 method for class 'nsbi_nle'
posterior(
  fit,
  x_obs = NULL,
  sampler = c("slice", "stan"),
  n_chains = NULL,
  warmup = 200L,
  thin = 2L,
  init_strategy = c("resample", "proposal"),
  seed = NULL,
  max_batch = 1e+05,
  ...
)
```

Arguments

<code>fit</code>	An <code>nsbi_nle</code> object from <code>nle()</code> .
<code>x_obs</code>	Observation to condition on. Rows are treated as independent observations from the same parameter, and the log-likelihood sums over them.
<code>sampler</code>	"slice" (the default) or "stan".
<code>n_chains</code>	Number of chains. The default depends on the sampler: 20 for "slice", which evaluates every chain in one batched call per step and so pays almost nothing for more of them, and 4 for "stan", where each chain is a separate process with its own warmup to pay for.
<code>warmup</code>	Steps discarded at the start of each chain.
<code>thin</code>	Keep one draw in thin. The default is 2: the slice width is adapted during warmup, and with that the retained draws are already close to independent – on a Gaussian target with 20 chains, <code>thin = 2</code> gives a bulk ESS of about 96% of them. Since each evaluation is a forward pass over every observation, thinning harder buys very little for what it costs. Raise it if the reported ESS says you need to. (Python <code>sbi</code> thins by 1 by default; it thinned by 10 up to v0.21.)

`init_strategy` "resample" (default, and `sbi`'s) weights a pool of prior draws by the posterior density and resamples the starting points from it, drawing more pools as needed if the first one lands fewer than `n_chains` draws in the posterior's support; "proposal" skips the weighting and keeps whichever prior draws land inside the posterior's support, also drawing more pools as needed. "proposal" is cheaper per accepted draw, but every draw the posterior excludes is wasted: if only a fraction α of the prior's mass survives, roughly $1/\alpha$ draws are needed per starting point, so a posterior that rules out most of the prior can make "proposal" far slower than "resample" to find enough starting points, and it errors out once its draw budget is spent. The pool is 1000 draws, set with `n_pool`. `sbi` uses 10,000 *per chain*, which on a surrogate summed over thousands of observations is a large bill before the first MCMC step.

`seed` Optional integer seed.

`max_batch` Largest number of (θ, x) pairs evaluated in one call to the estimator, both at every MCMC step and in `log_prob()`. Only affects memory and speed; see `max_batch` on `log_lik()` for the same knob on a direct likelihood evaluation.

... Further arguments to the sampler: `width`, `max_steps` and `n_pool` for "slice", or `iter_warmup`, `iter_sampling` and `refresh` for "stan".

Details

Two samplers are available. "slice" is the default: a vectorized univariate slice sampler (see `nsbj_mcmc`) with nothing to tune and no dependencies. "stan" writes the fitted likelihood out as a Stan program (see `stan_code()`) and runs NUTS on it through `cmdstanr` or `rstan`, which mixes better on correlated posteriors at the cost of a one-time model compile.

Value

An object of class `c("nsbj_nle_posterior", "nsbj_mcmc_posterior", "nsbj_posterior")`.

Examples

```
prior <- prior_uniform(c(mu = -3), c(mu = 3))
fit <- nle(prior, function(mu) c(y = rnorm(1, mu, 0.5)),
          n_simulations = 1000, density_estimator = "linear_gaussian")

x_obs <- matrix(rnorm(50, mean = 1, sd = 0.5), ncol = 1)
post <- posterior(fit, x_obs, n_chains = 4, warmup = 50, thin = 2)
draws <- sample(post, 400)
colMeans(draws)
```

posterior.nsbj_nre *Posterior from a neural ratio*

Description

Bayes' rule turns an `nre()` fit into a posterior, $p(\theta \mid x) \propto r_\phi(\theta, x)p(\theta)$, and as with `nle()` the result has no closed form and no direct sampler. `posterior()` on an `nsbj_nre` therefore returns an object that samples with MCMC, and `sample()` on it runs a chain rather than a forward pass.

Usage

```
## S3 method for class 'nsbi_nre'
posterior(
  fit,
  x_obs = NULL,
  n_chains = 20L,
  warmup = 200L,
  thin = 2L,
  init_strategy = c("resample", "proposal"),
  seed = NULL,
  max_batch = 1e+05,
  ...
)
```

Arguments

<code>fit</code>	An <code>nsbi_nre</code> object from <code>nre()</code> .
<code>x_obs</code>	Observation to condition on. Rows are treated as independent observations from the same parameter, and the log ratio sums over them.
<code>n_chains</code>	Number of chains. The slice sampler evaluates every chain in one batched call per step, so more chains cost almost nothing.
<code>warmup</code>	Steps discarded at the start of each chain.
<code>thin</code>	Keep one draw in thin. See <code>posterior.nsbi_nle()</code> for why the default is 2.
<code>init_strategy</code>	"resample" (default, and <code>sbi</code> 's) weights a pool of prior draws by the posterior density and resamples the starting points from it, drawing more pools as needed if the first one lands fewer than <code>n_chains</code> draws in the posterior's support; "proposal" skips the weighting and keeps whichever prior draws land inside the posterior's support, also drawing more pools as needed. "proposal" is cheaper per accepted draw, but every draw the posterior excludes is wasted: if only a fraction α of the prior's mass survives, roughly $1/\alpha$ draws are needed per starting point, so a posterior that rules out most of the prior can make "proposal" far slower than "resample" to find enough starting points, and it errors out once its draw budget is spent.
<code>seed</code>	Optional integer seed.
<code>max_batch</code>	Largest number of (θ, x) pairs evaluated in one call to the classifier, both at every MCMC step and in <code>log_prob()</code> . Only affects memory and speed; see <code>max_batch</code> on <code>log_ratio()</code> for the same knob on a direct ratio evaluation, and matters most here since the ratio classifier has no i.i.d. fast path – every pair costs a forward pass.
<code>...</code>	Further arguments to the sampler: <code>width</code> , <code>max_steps</code> and <code>n_pool</code> .

Details

There is one sampler here, the vectorized slice sampler of `nsbi_mcmc`, and no `sampler` argument to choose it with. `nle()`'s "stan" option works by transpiling the fitted density into a Stan functions block; the residual classifier behind a ratio estimator has no such export, so NUTS is not on offer.

Value

An object of class `c("nsbi_nre_posterior", "nsbi_mcmc_posterior", "nsbi_posterior")`.

Examples

```
prior <- prior_uniform(c(mu = -3), c(mu = 3))
fit <- nre(prior, function(mu) c(y = rnorm(1, mu, 0.5)),
          n_simulations = 1000, classifier = "logistic")

x_obs <- matrix(rnorm(50, mean = 1, sd = 0.5), ncol = 1)
post <- posterior(fit, x_obs, n_chains = 4, warmup = 50, thin = 2)
draws <- sample(post, 400)
colMeans(draws)
```

posterior_predictive *Posterior predictive draws*

Description

Samples parameters from the posterior and pushes them back through the simulator, giving predictive data to compare against the observation. A draw whose simulation returns non-finite output is dropped, which lowers the number of predictive draws returned.

Usage

```
posterior_predictive(post, simulator, n = 1000L, x = NULL, sim_args = list())
```

Arguments

<code>post</code>	An <code>nsbi_posterior</code> object.
<code>simulator</code>	The simulator; called once per posterior draw (see nsbi_simulator).
<code>n</code>	Number of predictive draws.
<code>x</code>	Observation to condition on (defaults to <code>x_obs</code>).
<code>sim_args</code>	Named list of extra arguments passed to every simulator call; see nsbi_simulator .

Value

An $n \times d$ matrix of simulated data from posterior parameter draws.

prior_custom

*Build a prior from arbitrary sampling / density functions***Description**

This is the one prior a user writes by hand, so it is the one where a mistake is most likely, and the mistakes are quiet. A `log_prob_fn` that returns a single number instead of one per row is legal R that surfaces much later as an MCMC initialization failure; a lower of the wrong length is recycled by `sweep()` into a support test that rejects the wrong draws. Everything is therefore checked at construction, including one probe call of `sample_fn(2)` and, when it is given, `log_prob_fn()` on those two rows.

Usage

```
prior_custom(
  sample_fn,
  log_prob_fn = NULL,
  dim,
  lower = NULL,
  upper = NULL,
  param_names = NULL
)
```

Arguments

<code>sample_fn</code>	Function <code>function(n)</code> returning an $n \times \text{dim}$ matrix, one row per draw.
<code>log_prob_fn</code>	Function <code>function(theta)</code> returning a length- n vector of log densities, one per row of <code>theta</code> . Optional; required only for methods/diagnostics that need it.
<code>dim</code>	Number of parameters.
<code>lower, upper</code>	Optional support bounds enabling out-of-support rejection. Numeric of length <code>dim</code> , or length 1 to apply the same bound to every parameter.
<code>param_names</code>	Optional character vector of parameter names, one per parameter. These name the columns of every downstream parameter matrix, posterior sample and diagnostic plot, and they decide how the simulator is called: a simulator whose formals are exactly these names receives one scalar per formal, and any other simulator receives the parameter vector as its first argument. See nsbi_simulator .

Value

An `nsbi_prior` object.

Stan export

A custom prior is arbitrary R code rather than a named distribution with parameters, so `stan_code()` cannot restate it as a Stan sampling statement. Take `stan_code(fit, model = FALSE)` and write the model block yourself, or build the prior out of `prior_families` and `prior_independent()` instead, which `stan_code()` does write out.

Examples

```
prior <- prior_custom(
  sample_fn = function(n) cbind(rexp(n, 1), rexp(n, 2)),
  log_prob_fn = function(theta) {
    dexp(theta[, 1], 1, log = TRUE) + dexp(theta[, 2], 2, log = TRUE)
  },
  dim = 2, lower = 0, param_names = c("beta", "gamma")
)
```

prior_families	<i>Named prior families</i>
----------------	-----------------------------

Description

Priors built from a named distribution family, one marginal per parameter, following Stan's argument names and argument order. Each constructor is vectorized: pass a vector where Stan would write one sampling statement per component, and pass a scalar to use the same value for every parameter. Naming any of the vectors (e.g. `c(beta = 2, gamma = 3)`) names the parameters, exactly as naming `low` does in `prior_uniform()`.

Usage

```
prior_lognormal(meanlog = 0, sdlog = 1)

prior_exponential(rate = 1)

prior_gamma(shape, rate = 1)

prior_beta(shape1, shape2)

prior_student_t(df, location = 0, scale = 1)

prior_cauchy(location = 0, scale = 1)

prior_half_normal(sd = 1)

prior_half_cauchy(scale = 1)
```

Arguments

meanlog, sdlog	Log-scale mean and standard deviation of the log-normal, as in <code>stats::dlnorm()</code> and Stan's <code>lognormal</code> .
rate	Rate of the exponential or gamma, as in <code>stats::dexp()</code> and Stan's <code>exponential/gamma</code> . Must be positive.
shape	Shape of the gamma. Must be positive.
shape1, shape2	Beta shape parameters, as in <code>stats::dbeta()</code> . Both must be positive.

df	Degrees of freedom of the Student-t. Must be positive.
location, scale	Location and scale of the Student-t, the Cauchy or the half-Cauchy. scale must be positive.
sd	Standard deviation of the normal underlying the half-normal. Must be positive.

Details

Families with constrained support set lower/upper on the prior, so the posterior's leakage correction (see `posterior()`) rejects and renormalizes against the right region without any further declaration. A gamma prior bounds at zero, a beta prior at zero and one, and the half families at zero.

A prior built here can be truncated further with `prior_truncated()`, combined with others through `prior_independent()`, and written out by `stan_code()` as the sampling statement it came from. That is the reason these exist as families rather than as recipes for `prior_custom()`: the family is what carries over to Stan and to the truncation constant.

`prior_half_normal()` and `prior_half_cauchy()` are the zero-centred families truncated to $[0, \infty)$, matching Stan's idiom of declaring `real<lower=0>` and writing `sigma ~ normal(0, 1)`. Their densities carry the $\log(2)$ renormalization, which Stan drops as a constant and this package cannot. For a half-normal centred somewhere other than zero, wrap `prior_normal()` in `prior_truncated()`.

Value

An `nsbi_prior` object.

See Also

`prior_independent()` to combine several of these into one joint prior, `prior_truncated()` to bound one, and `priors` for the rest.

Examples

```
# One log-normal per parameter, named through the vector.
prior <- prior_lognormal(meanlog = c(beta = log(0.4), gamma = log(0.125)),
                        sdlog = c(0.5, 0.2))
theta <- sample_prior(prior, 5)

# A scalar argument is shared: three gammas with the same rate.
prior_gamma(shape = c(2, 5, 9), rate = 3)

# Support bounds come from the family, and drive the leakage correction.
# A bare vector is read as a single row, so several 1-d draws need an
# explicit column matrix.
within_support(prior_beta(2, 15), matrix(c(0.1, 1.5), ncol = 1))
```

prior_independent *Combine independent priors into one joint prior*

Description

Stan writes a joint prior as one sampling statement per parameter and lets the product take care of itself. This is that, as an object: give it the per-parameter priors in the order the simulator expects them and it returns the product prior, with `dim` the total, lower/upper stacked, and a `log_prob` that sums the parts. It is the thing most `prior_custom()` calls were written to do by hand.

Usage

```
prior_independent(...)
```

Arguments

... Priors, one per block of parameters. Name them (`prior_independent(beta = ..., gamma = ...)`) to name the parameters: a name given here beats the component's own `param_names` for a one-parameter component, and a multi-parameter component keeps its own names, or takes `name1`, `name2`, ... when it has none.

Details

Components may themselves cover several parameters, so `prior_independent(prior_normal(mean = c(0, 0)), prior_beta(2, 15))` is a three-parameter prior. Any `nsbi_prior` works as a component, including a `prior_custom()`; the result can only be written out by `stan_code()` when every component comes from a named family (see `prior_families`).

Value

An `nsbi_prior` object.

See Also

`prior_families` for the components, `prior_truncated()` to bound one, `priors` for the rest.

Examples

```
prior <- prior_independent(
  p_S1S2 = prior_beta(2, 15),
  hr_S1  = prior_lognormal(log(3), 0.3),
  hr_S2  = prior_lognormal(log(10), 0.25)
)
prior
sample_prior(prior, 3)
```

prior_normal	<i>Independent normal prior</i>
--------------	---------------------------------

Description

Independent normal prior

Usage

```
prior_normal(mean, sd = 1)
```

Arguments

mean	Numeric vector of means (one per parameter). Naming the vector (e.g. <code>c(beta = 0, gamma = 0)</code>) attaches those names to every downstream parameter matrix, posterior sample, and diagnostic plot. Must be finite; a normal prior has no legitimate use for an infinite mean.
sd	Numeric scalar or vector of standard deviations. Must be finite and positive.

Value

An `nsbi_prior` object.

Examples

```
prior <- prior_normal(mean = c(0, 0), sd = 1)
```

prior_truncated	<i>Truncate a prior to a box</i>
-----------------	----------------------------------

Description

Stan's `T[lower, upper]`, as an object. The returned prior is the original restricted to the box and renormalized by the mass it keeps, so its `log_prob` is a proper log density rather than the original shifted by an unknown constant. That matters here in a way it does not in Stan: the density is compared against a learned posterior in `nle()`'s MCMC target and summed with it in `c2st()` and the diagnostics, so a missing constant is a wrong answer rather than a constant offset.

Usage

```
prior_truncated(prior, lower = NULL, upper = NULL)
```

Arguments

prior	An <code>nsbi_prior</code> from a named family.
lower, upper	Truncation bounds. Numeric of length <code>prior\$dim</code> , or length 1 to apply the same bound to every parameter. Give one or both; <code>-Inf/Inf</code> leaves that side alone.

Details

Renormalization needs the family's CDF, so prior has to come from a named family: `prior_uniform()`, `prior_normal()`, anything in `prior_families`, or a `prior_independent()` built from those. A `prior_custom()` is arbitrary R code with no CDF behind it; give it lower/upper there instead, which rejects out-of-support draws without claiming to renormalize.

Value

An `nsbi_prior` object.

See Also

`prior_families`, `prior_independent()`, `priors`.

Examples

```
# A half-normal, the long way round (prior_half_normal() is the short one).
prior_truncated(prior_normal(mean = 0, sd = 1), lower = 0)

# A contact rate known to be between 0.1 and 2, log-normal in between.
prior <- prior_truncated(prior_lognormal(log(0.4), 0.5),
                        lower = 0.1, upper = 2)
range(sample_prior(prior, 100))
```

prior_uniform	<i>Box-uniform (independent uniform) prior</i>
---------------	--

Description

Box-uniform (independent uniform) prior

Usage

```
prior_uniform(low, high)
```

Arguments

low	Numeric vector of lower bounds (one per parameter). Naming the vector (e.g. <code>c(beta = 0, gamma = 0)</code>) attaches those names to every downstream parameter matrix, posterior sample, and diagnostic plot. NA/NaN are rejected; <code>-Inf</code> is accepted (matching high's <code>Inf</code>) but makes the prior improper, so <code>sample_prior()</code> on it errors unless it is first bounded with <code>prior_truncated()</code> .
high	Numeric vector of upper bounds (one per parameter). Same finiteness rule as low, with <code>Inf</code> in place of <code>-Inf</code> .

Value

An `nsbi_prior` object.

Examples

```
prior <- prior_uniform(low = c(-2, -2, -2), high = c(2, 2, 2))
theta <- sample_prior(prior, 5)
```

priors

Priors for neural simulation-based inference

Description

A prior in `neuralsbi` is a lightweight object (class `nsbi_prior`) that knows how to (a) draw samples and (b) evaluate its log-density. Bounded priors also carry lower/upper support limits, which are used to reject out-of-support posterior samples ("leakage" correction).

Details

There are four ways to build one. `prior_uniform()` is the box prior most benchmark tasks use. `prior_normal()` and the named families in `prior_families` (log-normal, exponential, gamma, beta, Student-t, Cauchy and the half versions of the last two) give one marginal per parameter, under Stan's argument names. `prior_independent()` multiplies those together into a joint prior, and `prior_truncated()` bounds one, renormalizing the density by the mass it keeps. `prior_custom()` takes a sampler and a density you write yourself, for anything the families do not cover.

Prefer a named family over `prior_custom()` where one fits. A family carries its support bounds into the leakage correction, survives `prior_truncated()` with an exact normalizing constant, and is what `stan_code()` writes out as a sampling statement; a custom prior does none of those.

See Also

`prior_families`, `prior_independent()`, `prior_truncated()`, `sample_prior()`, `within_support()`.

sample

Draw samples (S3 generic)

Description

`neuralsbi` turns `base::sample()` into an S3 generic so that `sample(posterior, n)` reads the way statisticians expect. For any object without a dedicated method (vectors, etc.) this falls back to `base::sample()` unchanged.

Usage

```
sample(x, ...)
```

```
## Default S3 method:
```

```
sample(x, ...)
```

Arguments

x Object to sample from.
 ... Passed on to methods / `base::sample()`.

Value

Whatever the dispatched method returns. The default method returns the result of `base::sample()`; `sample.nsbi_posterior()` returns an $n \times \text{dim}$ matrix of posterior draws.

```
sample.nsbi_mcmc_posterior
```

Sample an MCMC posterior

Description

Runs the sampler chosen by `posterior.nsbi_nle()` or `posterior.nsbi_nre()`. Draws are cached on the posterior object, so asking for the same or fewer draws from the same observation returns immediately instead of re-running a chain – which is what makes `summary()` and repeated calls tolerable.

Usage

```
## S3 method for class 'nsbi_mcmc_posterior'
sample(
  x,
  size = 1000,
  n = size,
  obs = NULL,
  refresh = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

x An nsbi_mcmc_posterior object (named x to satisfy the `sample()` generic).
 size, n Number of posterior draws (n is an alias for size).
 obs Observation to condition on (defaults to the posterior's x_obs).
 refresh Force a new run even when a cached one would do.
 verbose Report sampling progress.
 ... Unused.

Value

An $n \times \text{dim}$ matrix of posterior draws (class nsbi_samples), with convergence diagnostics attached as attribute diagnostics.

sample.nsbi_posterior *Sample from a posterior*

Description

Sample from a posterior

Usage

```
## S3 method for class 'nsbi_posterior'
sample(x, size = 1000, n = size, obs = NULL, max_sampling_batches = 100L, ...)
```

Arguments

x	An nsbi_posterior object (named x to satisfy the sample() generic).
size, n	Number of posterior draws (n is an alias for size).
obs	Observation to condition on (defaults to the posterior's x_obs).
max_sampling_batches	Safety cap on rejection-sampling rounds for bounded priors. The first round draws n rows; later rounds size the batch from the running acceptance rate (at most 100 * n rows), so low acceptance still fills n draws within a few rounds.
...	Unused.

Value

An n x dim matrix of posterior draws (class nsbi_samples).

sample_posterior *Sample from a posterior (non-generic alias)*

Description

Identical to sample(post, n); provided for users who prefer not to rely on the generic.

Usage

```
sample_posterior(post, n = 1000, obs = NULL, ...)
```

Arguments

post	An nsbi_posterior object.
n	Number of posterior draws.
obs	Observation to condition on (defaults to the posterior's x_obs).
...	Passed to the dispatched sample() method.

Value

An $n \times \text{dim}$ matrix of posterior draws.

sample_prior	<i>Draw samples from a prior</i>
--------------	----------------------------------

Description

Draw samples from a prior

Usage

```
sample_prior(prior, n)
```

Arguments

prior	An nsbi_prior object.
n	Number of samples.

Value

An $n \times \text{dim}$ matrix of parameter draws.

save_npe	<i>Save and reload a fitted model</i>
----------	---------------------------------------

Description

A fit whose estimator is "maf", "mdn" or "nsf" holds a torch module, and a torch module is an external pointer. saveRDS() writes the pointer, not the network: the file reloads without complaint and the object prints normally, but the first call that touches the network fails with external pointer is not valid. save_npe() and load_npe() are the round trip that works.

Usage

```
save_npe(fit, path)
```

```
load_npe(path)
```

```
save_nle(fit, path)
```

```
load_nle(path)
```

```
save_nre(fit, path)
```

```
load_nre(path)
```

Arguments

fit	An nsbi_npe object from <code>npe()</code> or <code>npe_sequential()</code> , an nsbi_nle object from <code>nle()</code> , or an nsbi_nre object from <code>nre()</code> .
path	File to write to (or read from). The convention is <code>.rds</code> . <code>load_npe()</code> says so when there is no such file.

Details

`save_npe()` writes one `.rds` file holding the network's weights (via `torch::torch_save()` on its `state_dict`) alongside everything else the fit carries as ordinary R objects: the prior, the standardization centers and scales, parameter and outcome names, the simulation count, and the training history. `load_npe()` rebuilds the network from the recorded architecture and restores the weights, returning an nsbi_npe that behaves exactly like the one you trained.

A "linear_gaussian" fit holds no torch objects and round-trips through `saveRDS()` unharmed; `save_npe()` accepts it anyway, so saving code does not have to know which estimator was used.

`npe()/nle()/nre()` accept a custom `density_estimator/classifier` function that returns its own S3-classed object around a torch module. `save_npe()` only knows how to rebuild the estimators shipped with this package ("mdn", "maf", "nsf", and the ratio-estimation network), so it errors immediately, naming the offending class, on a torch-backed fit whose estimator is none of those, rather than writing a file that only fails later, in `load_npe()`, possibly in a different session.

Weights are saved, not code. A fit saved by one version of `neuralnsbi` loads into a later one as long as the estimator's architecture has not changed; `load_npe()` reports the version that wrote the file when the rebuild fails.

`save_nle()/load_nle()` and `save_nre()/load_nre()` are aliases; every pair handles every kind of fit, and the extra names exist only so calling code reads the way the fit was made.

A fit trained with `device = "cuda"/"mps"` (see `npe()/nle()`) always reloads onto CPU, never onto the original device: `torch::torch_load()` defaults to `device = "cpu"` regardless of where the tensors were saved from, and `de_rebuild_net()` builds the network fresh with no `torch::with_device()` in effect. Move it back with `fit2$de$net$to(device = "cuda")` if you want the reloaded fit on a GPU.

Value

`save_npe()` returns `path` invisibly. `load_npe()` returns the fit.

Examples

```
prior <- prior_uniform(c(mu = -2, nu = -2), c(mu = 2, nu = 2))
simulator <- function(mu, nu) c(a = mu + rnorm(1, sd = 0.1),
                                b = nu + rnorm(1, sd = 0.1))
fit <- npe(prior, simulator, n_simulations = 500,
           density_estimator = "linear_gaussian")

path <- tempfile(fileext = ".rds")
save_npe(fit, path)
fit2 <- load_npe(path)
sample(posterior(fit2, x_obs = c(0.8, 0.6)), 100)
unlink(path)
```

Description

Repeatedly draws a "true" parameter from the prior, simulates data, and ranks the true parameter within posterior samples conditioned on that data. If the posterior is well calibrated, the ranks are uniformly distributed.

Usage

```
sbc(
  fit,
  simulator,
  prior = fit$prior,
  n_sbc = 200L,
  n_posterior_samples = 1000L,
  sim_args = list(),
  seed = NULL,
  ...
)
```

Arguments

<code>fit</code>	An <code>nsbi_npe</code> fit from <code>npe()</code> , an <code>nsbi_nle</code> fit from <code>nle()</code> , or an <code>nsbi_nre</code> fit from <code>nre()</code> . With an NLE or NRE fit every trial is a separate MCMC run, so start with a small <code>n_sbc</code> and raise it once the cost is known.
<code>simulator</code>	The simulator used for inference; called once per trial (see nsbi_simulator).
<code>prior</code>	The prior to draw the true parameters from (defaults to <code>fit\$prior</code>). SBC is a test of the posterior against the prior it was trained on, so the default is the only choice that answers "is this fit calibrated". Overriding it changes the question to how the fit behaves on parameters drawn from somewhere else, which is a reasonable local check but is no longer SBC. It must cover the same parameters as the fit.
<code>n_sbc</code>	Number of SBC trials (fresh (theta, x) pairs).
<code>n_posterior_samples</code>	Posterior draws per trial (rank resolution).
<code>sim_args</code>	Named list of extra arguments passed to every simulator call; see nsbi_simulator .
<code>seed</code>	Optional seed.
<code>...</code>	Passed to <code>posterior()</code> , which is how the MCMC controls (<code>n_chains</code> , <code>warmup</code> , <code>thin</code> , <code>sampler</code>) reach an NLE or NRE fit.

Details

A trial whose simulation returns non-finite output is dropped, which lowers the effective `n_sbc`. A trial whose posterior comes back with fewer than `n_posterior_samples` draws, which happens when a bounded prior and a leaky estimator defeat rejection sampling, is an error: ranks are binned against `n_posterior_samples`, so a short draw would be scored on a scale it was never drawn on and would read as miscalibration.

The `n_sbc` simulations run across future workers when a plan is set (see [nsbi_parallel](#)); the ranking loop that follows calls the trained network and always runs locally.

Value

An object of class `nsbi_sbc` with the rank matrix and a per-parameter uniformity test.

<code>simulate_for_sbi</code>	<i>Run a simulator over prior draws</i>
-------------------------------	---

Description

Draws `n` parameter vectors from the prior and calls the simulator once per draw. Under a **future** plan the draws are spread across workers. See [nsbi_simulator](#), [nsbi_parallel](#) and [nsbi_progress](#).

Usage

```
simulate_for_sbi(
  simulator,
  prior,
  n,
  sim_args = list(),
  seed = NULL,
  verbose = FALSE
)
```

Arguments

<code>simulator</code>	A function called once per parameter set, returning one simulated observation: a numeric vector, a scalar, or a one-row matrix or data frame. See nsbi_simulator . Note the order: the simulator is the first argument here and the second in <code>npe(prior, simulator, ...)</code> .
<code>prior</code>	An <code>nsbi_prior</code> (see prior_uniform() , prior_normal()).
<code>n</code>	Number of simulations.
<code>sim_args</code>	Named list of extra arguments passed to every simulator call: observed data, a time grid, a fixed population size, solver settings. See nsbi_simulator .
<code>seed</code>	Optional integer seed for reproducibility.
<code>verbose</code>	Print training progress.

Details

Simulations whose output is not finite are dropped together with their parameters, with a warning.

The simulator comes first here and second in `npe()`, `nle()` and `npe_sequential()`. That is the reverse of the fitting functions and it is easy to get backwards, so a call with the two swapped is detected and named rather than left to fail inside `sample_prior()`.

Value

A list with `theta` ($n \times \text{dim}$) and `x` ($n \times d$) matrices and `n_dropped`, the number of simulations discarded for non-finite output.

Examples

```
prior <- prior_uniform(c(a = -1, b = -1), c(a = 1, b = 1))
sims <- simulate_for_sbi(function(a, b) c(a^2, b^2), prior, n = 100)
str(sims)
```

stan_export	<i>Export a learned likelihood to Stan</i>
-------------	--

Description

`stan_code()` writes the likelihood learned by `nle()` out as Stan source: a functions block that recomputes $\log q_\phi(x \mid \theta)$ in Stan's own language, with the trained weights travelling alongside as data. Stan differentiates the generated code itself, so NUTS gets exact gradients and nothing has to be linked against torch at run time.

Usage

```
stan_code(fit, name = "nsbi_log_lik", model = TRUE)

write_stan_model(fit, file, name = "nsbi_log_lik", model = TRUE)

stan_data(fit, x_obs = NULL, model = TRUE)
```

Arguments

<code>fit</code>	An <code>nsbi_nle</code> object from <code>nle()</code> .
<code>name</code>	Prefix for the generated functions.
<code>model</code>	Generate a complete, runnable model (the default) or only the functions block, for <code>#include</code> -ing into a model of your own. In <code>stan_data()</code> , this must agree with the model a paired <code>stan_code()</code> call used: <code>model = TRUE</code> requires <code>x_obs</code> , because the generated model's data block declares <code>N</code> and <code>x</code> and there is nothing to fill them with otherwise; <code>model = FALSE</code> matches a functions-only export, which has no <code>N/x</code> to fill, so <code>x_obs</code> is optional there.
<code>file</code>	Path to write to.
<code>x_obs</code>	Observation to put in the data list. Rows are independent observations. Required when <code>model = TRUE</code> .

Value

`stan_code()` returns the Stan program as a single string; `write_stan_model()` returns file invisibly; `stan_data()` returns a named list ready for `cmdstanr` or `rstan`.

Why this is useful

The exported function is an ordinary Stan function of `theta`. That means the surrogate likelihood stops being the whole model and becomes one term in a model you write: put a hierarchical prior over the parameters, add covariates, mix in a second data source with a likelihood you *do* know, and let NUTS sample the lot. None of that is reachable from a posterior estimator, which only ever knows the one conditional it was trained on.

The generated interface

Two entry points, where `w` is the packed weight vector from `stan_data()`:

```
real nsbi_log_lik_lpdf(vector x, vector theta, vector w);    // one observation
real nsbi_log_lik_sum_lpdf(matrix x, vector theta, vector w); // rows are i.i.d.
```

Both take `x` and `theta` in the original units the simulator and prior use; the standardization the estimator trained under is folded into the generated code. Prefer the `_sum` form for repeated observations: for the MDN and the linear-Gaussian estimator the conditional distribution depends on `theta` alone, so it is computed once and reused across every row.

Why the code is generated rather than called

Python `sbi` hands its learned likelihood to Pyro or PyMC as a callable, because everything lives in one process and the sampler can differentiate the same graph the estimator was trained in. Stan cannot work that way: it compiles to C++ and needs the gradient inside its own autodiff. The alternative would be an external C++ header (`--allow-undefined` plus a `USER_HEADER`) linking `torch` into every Stan compile, with the gradients plumbed by hand. Generating source keeps the run-time dependency at zero and makes the result something you can read, edit, and check – which is what the package’s own tests do, by evaluating the emitted functions and comparing them against `log_lik()`.

Supported estimators

`"mdn"`, `"maf"` and `"linear_gaussian"`. `"nsf"` is not exported – its rational-quadratic spline transform would be a large and fragile block of generated Stan. Refit with `"maf"` if you need a flow.

`nre()` fits are not exported at all. What gets transpiled here is a density; a ratio estimator holds a classifier instead, and there is no $p(x \mid \theta)$ in it to write out.

Supported priors

Only `model = TRUE` needs a prior, since only then is there a model block to put one in. `prior_uniform()` and `prior_normal()` travel as data, so their bounds and their mean and standard deviation come from `stan_data()` and the compiled model does not care what they are. Every other named family (see `prior_families`), including `prior_independent()` products and `prior_truncated()`

bounds, is written out as literal sampling statements with `T[,]` where the support was cut down. A `prior_custom()` cannot be written out at all: take `stan_code(fit, model = FALSE)` and write the model block yourself.

Examples

```
prior <- prior_uniform(c(mu = -3), c(mu = 3))
fit <- nle(prior, function(mu) c(y = rnorm(1, mu, 0.5)),
  n_simulations = 500, density_estimator = "linear_gaussian")

cat(substr(stan_code(fit), 1, 400))
str(stan_data(fit, matrix(rnorm(10), ncol = 1)), max.level = 1)
str(stan_data(fit, model = FALSE), max.level = 1)
```

summaries

Summaries and tidy accessors

Description

`summary()` methods for fits, posteriors, and samples, plus `as.data.frame()` for posterior draws so results drop straight into data-frame workflows (`dplyr`, `ggplot2`, ...).

Usage

```
## S3 method for class 'nsbi_samples'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'nsbi_samples'
summary(object, probs = c(0.025, 0.25, 0.5, 0.75, 0.975), ...)

## S3 method for class 'nsbi_posterior'
summary(object, n = 1000L, x = NULL, ...)

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

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

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

Arguments

`x` Observation to condition on (defaults to the posterior's `x_obs`); for `as.data.frame()`, the samples object.

`row.names, optional` Standard `as.data.frame()` arguments.

...	Additional arguments passed to methods.
object	An <code>nsbi_samples</code> matrix from <code>sample()</code> , an <code>nsbi_posterior</code> , or an <code>nsbi_npe</code> , <code>nsbi_nle</code> or <code>nsbi_nre</code> fit.
probs	Quantiles to report.
n	Number of draws used to summarize a posterior.

Value

For samples and posteriors, a data frame with one row per parameter (mean, sd, and quantiles). For fits, an invisible list of training metadata. In that list `dim_x` counts the data dimension the estimator was trained on, which for an `nsbi_nle` or `nsbi_nre` fit is one observation rather than the whole data set.

tarp	<i>TARP expected coverage</i>
------	-------------------------------

Description

Tests of Accuracy with Random Points (Lemos et al. 2023). For each trial a true parameter is drawn from the prior, data are simulated, and posterior samples are drawn conditioned on those data. Given a random reference point, the fraction of posterior samples closer to the reference than the truth is the credibility level of the smallest distance-based credible region that contains the truth. For a calibrated posterior these fractions are uniform, so the expected coverage probability (ECP) at credibility level α equals α .

Usage

```
tarp(
  fit,
  simulator,
  prior = fit$prior,
  n_tarp = 200L,
  n_posterior_samples = 1000L,
  references = c("uniform", "prior"),
  sim_args = list(),
  seed = NULL,
  ...
)
```

Arguments

fit	An <code>nsbi_npe</code> fit from <code>npe()</code> , an <code>nsbi_nle</code> fit from <code>nle()</code> , or an <code>nsbi_nre</code> fit from <code>nre()</code> . With an NLE or NRE fit every trial is a separate MCMC run, so start with a small <code>n_tarp</code> and raise it once the cost is known.
simulator	The simulator used for inference; called once per trial (see <code>nsbi_simulator</code>).

prior	The prior to draw the true parameters from, and the reference points when <code>references = "prior"</code> (defaults to <code>fit\$prior</code>). As in <code>sbc()</code> , the coverage claim is about the prior the fit was trained on, so the default is what tests this fit; another prior tests how the fit does on parameters it was not calibrated against. It must cover the same parameters as the fit.
n_tarp	Number of TARP trials (fresh (theta, x) pairs). At least 2: the true draws are standardized by their own spread before distances are computed, and a single draw leaves that spread undefined.
n_posterior_samples	Posterior draws per trial.
references	How to draw reference points: "uniform" (default, uniform over the hyper-rectangle spanned by the true parameter draws, as in the paper) or "prior" (draws from the prior).
sim_args	Named list of extra arguments passed to every simulator call; see <code>nsbi_simulator</code> .
seed	Optional seed.
...	Passed to <code>posterior()</code> , which is how the MCMC controls (<code>n_chains</code> , <code>warmup</code> , <code>thin</code> , <code>sampler</code>) reach an NLE or NRE fit.

Details

Unlike `sbc()`, which ranks each parameter marginally, TARP is a *joint* test: it can detect posteriors whose marginals are calibrated but whose correlation structure is wrong. Distances are computed after z-scoring each parameter (using the spread of the true draws), so parameters on different scales contribute comparably.

A trial whose simulation returns non-finite output is dropped, which lowers the effective `n_tarp`. A trial whose posterior comes back with fewer than `n_posterior_samples` draws is an error, for the same reason as in `sbc()`: a trial scored on a different number of draws is not comparable to the rest.

Value

An object of class `nsbi_tarp` with the per-trial coverage values and the ECP curve. Plot it with `plot_tarp()`.

References

Lemos, Coogan, Hezaveh & Perreault-Levasseur (2023), "Sampling-based accuracy testing of posterior estimators for general inference", ICML. [doi:10.48550/arXiv.2302.03026](https://arxiv.org/abs/2302.03026)

Description

Standard simulation-based-inference benchmark tasks, following the definitions in the `sbibm` benchmark suite. Each task bundles a prior, a simulator, and (where one exists) an analytic reference posterior, so the same object drives unit tests, calibration studies, and the benchmark harness in `inst/benchmarks/`.

Each task's simulator follows the package contract: one parameter set in, one simulated observation out (see [nsbi_simulator](#)).

- `task_gaussian_linear()` – conjugate Gaussian; analytic posterior.
- `task_two_moons()` – crescent-shaped, bimodal posterior.
- `task_slcp()` – 5 parameters, 8-dimensional data, strongly non-Gaussian posterior.
- `task_sir()` – SIR epidemic dynamics with log-normal priors; no closed-form posterior (verify with SBC / coverage).

Usage

```
task_gaussian_linear(dim = 10L, prior_var = 0.1, noise_var = 0.1)
```

```
task_two_moons()
```

```
task_slcp()
```

```
task_sir(N = 1e6, days = 160, n_points = 10L, n_obs_draws = 1000L)
```

Arguments

<code>dim</code>	Parameter/data dimension for the Gaussian linear task (default 10).
<code>prior_var, noise_var</code>	Prior and likelihood variances for the Gaussian linear task.
<code>N, days, n_points, n_obs_draws</code>	SIR task: population size, horizon in days, number of observation times, and binomial trials per observation.

Value

A list of class `nsbi_task` with elements `name`, `prior`, `simulator`, `dim_theta`, `dim_x`, and optionally `reference_posterior(x_obs, n)` returning exact posterior draws.

<code>within_support</code>	<i>Test whether parameters lie within the prior support</i>
-----------------------------	---

Description

Test whether parameters lie within the prior support

Usage

```
within_support(prior, theta)
```

Arguments

<code>prior</code>	An <code>nsbi_prior</code> object.
<code>theta</code>	A matrix (or vector) of parameters.

Value

Logical vector, one entry per row of `theta`.

Index

`as.data.frame()`, [51](#)
`as.data.frame.nsbi_samples(summaries)`,
[51](#)
`base::sample()`, [42, 43](#)

`c2st`, [3](#)
`c2st()`, [6, 40](#)

`de_rebuild_net()`, [46](#)
`density_estimator`, [5](#)
`diagnostics`, [6](#)

`embedding_mlp`, [6](#)
`embedding_mlp()`, [17, 21](#)
`expected_coverage`, [7, 29](#)
`expected_coverage()`, [6](#)

`GGally::ggpairs()`, [27](#)
`ggdensity::geom_hdr()`, [27, 28](#)
`ggplot2::geom_density()`, [28](#)

`likelihood_fn`, [8](#)
`likelihood_fn()`, [9, 15](#)
`load_nle(save_npe)`, [45](#)
`load_npe(save_npe)`, [45](#)
`load_nre(save_npe)`, [45](#)
`log_lik`, [8](#)
`log_lik()`, [8, 11, 14, 15, 33, 50](#)
`log_prob`
 (`log_prob.nsbi_mcmc_posterior`),
 [10](#)
`log_prob()`, [12, 22, 31, 33, 34](#)
`log_prob.nsbi_mcmc_posterior`, [10](#)
`log_ratio`, [11](#)
`log_ratio()`, [22, 23, 34](#)

`map_estimate`, [12](#)
`map_estimate()`, [31](#)

`nle`, [13](#)

`nle()`, [5, 8–12, 20, 22, 23, 31–34, 40, 46, 47, 49, 52](#)
`npe`, [15](#)
`npe()`, [5–7, 13, 14, 18–24, 26, 31, 46, 47, 49, 52](#)
`npe_sequential`, [18](#)
`npe_sequential()`, [24, 26, 31, 46, 49](#)
`nre`, [20](#)
`nre()`, [10–12, 31, 33, 34, 46, 47, 50, 52](#)
`nsbi_mcmc`, [23, 33, 34](#)
`nsbi_parallel`, [18, 24, 26, 27, 48](#)
`nsbi_progress`, [18, 25, 25, 48](#)
`nsbi_simulator`, [13, 14, 16, 19, 21, 26, 35, 36, 47, 48, 52–54](#)

`pairplot`, [27](#)
`plot_coverage`, [28](#)
`plot_posterior_predictive`, [29](#)
`plot_sbc`, [30](#)
`plot_tarp`, [30](#)
`plot_tarp()`, [53](#)
`posterior`, [31](#)
`posterior()`, [9, 11, 13–15, 17, 19, 20, 22, 23, 38, 47, 53](#)
`posterior.nsbi_nle`, [32](#)
`posterior.nsbi_nle()`, [24, 31, 34, 43](#)
`posterior.nsbi_nre`, [33](#)
`posterior.nsbi_nre()`, [31, 43](#)
`posterior_predictive`, [35](#)
`posterior_predictive()`, [6, 24, 26, 29](#)
`prior_beta(prior_families)`, [37](#)
`prior_cauchy(prior_families)`, [37](#)
`prior_custom`, [36](#)
`prior_custom()`, [12, 38, 39, 41, 42, 51](#)
`prior_exponential(prior_families)`, [37](#)
`prior_families`, [36, 37, 39, 41, 42, 50](#)
`prior_gamma(prior_families)`, [37](#)
`prior_half_cauchy(prior_families)`, [37](#)
`prior_half_normal(prior_families)`, [37](#)
`prior_independent`, [39](#)

prior_independent(), 36, 38, 41, 42, 50
 prior_lognormal (prior_families), 37
 prior_normal, 40
 prior_normal(), 13, 16, 19, 21, 28, 30, 38, 41, 42, 48, 50
 prior_student_t (prior_families), 37
 prior_truncated, 40
 prior_truncated(), 38, 39, 41, 42, 50
 prior_uniform, 41
 prior_uniform(), 12, 13, 16, 19, 21, 28, 30, 37, 41, 42, 48, 50
 priors, 38, 39, 41, 42

 sample, 42
 sample(), 10, 12, 17, 19, 28, 31–33, 43, 44, 52
 sample.nsbi_mcmc_posterior, 43
 sample.nsbi_posterior, 44
 sample.nsbi_posterior(), 43
 sample_posterior, 44
 sample_prior, 45
 sample_prior(), 41, 42, 49
 save_nle (save_npe), 45
 save_npe, 45
 save_npe(), 17, 32
 save_nre (save_npe), 45
 sbc, 29, 47
 sbc(), 6, 7, 24–26, 30, 53
 simulate_for_sbi, 48
 simulate_for_sbi(), 24, 26
 stan_code (stan_export), 49
 stan_code(), 14, 15, 24, 33, 36, 38, 39, 42, 49
 stan_data (stan_export), 49
 stan_data(), 50
 stan_export, 49
 stats::dbeta(), 37
 stats::dexp(), 37
 stats::dlnorm(), 37
 stats::optim(), 8
 summaries, 51
 summary(), 43
 summary.nsbi_nle (summaries), 51
 summary.nsbi_npe (summaries), 51
 summary.nsbi_nre (summaries), 51
 summary.nsbi_posterior (summaries), 51
 summary.nsbi_samples (summaries), 51

 tarp, 52
 tarp(), 24–26, 30
 task_gaussian_linear (tasks), 53
 task_sir (tasks), 53
 task_slcp (tasks), 53
 task_two_moons (tasks), 53
 tasks, 53
 torch::torch_save(), 46

 within_support, 54
 within_support(), 42
 write_stan_model (stan_export), 49