Package 'stors'

May 9, 2026

Type Package

Title Step Optimised Rejection Sampling

Version 1.0.1

Maintainer Ahmad ALQabandi <ahmad.alqabandi@durham.ac.uk>

Description Fast and efficient sampling from general univariate probability density functions. Implements a rejection sampling approach designed to take advantage of modern CPU caches and minimise evaluation of the target density for most samples. Many standard densities are internally implemented in 'C' for high performance, with general user defined densities also supported. A paper describing the methodology will be released soon.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

Imports digest, microbenchmark, cli, rlang

Suggests knitr, rmarkdown, testthat (>= 3.0.0), ggplot2

Config/testthat/edition 3

Depends R (>= 4.2.0)

VignetteBuilder knitr

URL https://ahmad-alqabandi.github.io/stors/

NeedsCompilation yes

Author Ahmad ALQabandi [cre, aut, cph] (ORCID: <https://orcid.org/0009-0006-0055-3846>), Louis Aslett [aut, ths, cph] (ORCID: <https://orcid.org/0000-0003-2211-233X>)

Repository CRAN

Date/Publication 2025-03-11 17:00:02 UTC

Contents

`build_proposal` *Build Proposal*

Description

Constructs the step optimized proposal density, squeezing function, and log-linear tail proposal for a user defined probability density function.

Usage

```
build_proposal(
  f,
  modes,
  lower = -Inf,
  upper = Inf,
  h = NULL,
  h_prime = NULL,
  steps = NULL,
  proposal_range = NULL,
  theta = 0.1,
```

```
  target_sample_size = 1000,
  verbose = FALSE,
  ...
)
```

Arguments

f	A function which returns the (unnormalised) probability density function of the target distribution. The first argument must be the value at which the pdf is to be evaluated. Additional arguments may be parameters of the distribution, which should be specified by name in the ... arguments.
modes	Numeric vector of modes of the density function.
lower	Numeric scalar representing the lower bound of the target density. Default is -Inf for unbounded lower support.
upper	Numeric scalar representing the upper bound of the target density. Default is Inf for unbounded upper support.
h	An optional function which returns the (unnormalised) log-probability density function of the target distribution. As for f the first argument must be the value at which the log-pdf is to be evaluated and additional parameters may be named arguments passed to ....
h_prime	An optional function which returns the first derivative of the (unnormalised) log-probability density function of the target distribution. As for f the first argument must be the value at which the log-pdf is to be evaluated and additional parameters may be named arguments passed to ....
steps	Optional integer scalar specifying the number of steps in the step optimised part of the proposal density and squeezing function.
proposal_range	Optional numeric vector of length 2 specifying the lower and upper range of the steps in the step optimised part of the proposal density and squeezing function. This range should be contained within the interval defined by lower and upper.
theta	Optional numeric scalar (between 0.1 and 1) defining the pre-acceptance threshold. This dictates when no further steps should be added in the step optimised part of the proposal density and squeezing function, based on the probability of pre-acceptance.
target_sample_size	Integer scalar indicating the typical sample size that will be requested when sampling from this density using build_sampler. The proposal optimization process bases benchmark timings on this target size in order to select a proposal best suited to the desired sample size. Note this does *not* limit sampling to this number, it is merely a guide should the user be aware that a certain sample size will be most commonly sampled.
verbose	Logical scalar. If TRUE, a table detailing the optimization areas and steps will be displayed during proposal optimization. Defaults to FALSE.
...	Further arguments to be passed to f, h, and h_prime, if they depend on additional parameters.

Details

This function is the starting point when a user wishes to build a custom sampler using StORS. It is responsible for generating the step optimized proposal density, squeezing function, and log-linear tail proposal that can be utilized for this purpose. The minimum information that must be supplied by the user is:

- The (closed) interval of support for the distribution, [`lower`, `upper`] $\in \mathbb{R}$, which may also be half-closed on either side, or all of $\mathbb{R}$.
- The probability density function (pdf), which need not be normalised, `f`.
- Any modes of the pdf, as vector `modes`.

Optionally, the log-pdf and derivative of the log-pdf may be supplied.

Arguments for pdf

The pdf (and log-pdf and first derivative of the log-pdf) may depend on certain parameters. If so, these can be from the second argument onward in `f`. For instance, consider the Kumaraswamy distribution, which has pdf:

$$f(x; a, b) = abx^{a-1}(1 - x^a)^{b-1}, \text{ where } x \in (0, 1)$$

This pdf has known modes.

Then, to implement as a custom StORS sampler, we would first define the pdf in R:

```
dkumaraswamy <- function(x, a, b) a*b*(x^(a-1))*(1-x^a)^(b-1)
```

Then, to construct a StORS proposal for $a = 2$ and $b = 2$, we would call

```
Proposal <- build_Proposal(lower = 0, upper = 1, modes = sqrt(1/3), f = dkumaraswamy, a = 2, b = 2)
```

StORS proposal construction

StORS defines an unnormalised piecewise constant proposal density and squeezing function, with a Proposal defining the change points. To optimise the execution speed on modern CPUs, the unnormalised piecewise constant proposal has fixed area for each segment with one end of the segment coinciding with the user's pdf. That is, each step of the function has width defined by $w_i = (x_i - x_{i-1})$ and a height determined by $h_i = \max(f(x_{i-1}), f(x_i))$, such that $w_i h_i = \alpha \; \forall \, i$ where α is constant.

Once the user has constructed the proposal, the sampling function can be built using `build_sampler()`.

Internal details

The function `build_final_Proposal()` manages the construction of these steps and calculates values critical for the sampling process. When the resultant Proposal is used with the `build_sampler()` function, these values are cached, significantly enhancing the computational efficiency and hence improving sampling speed. During the optimization process, we aim for a certain Proposal size based on L1-3 memory cache size. Therefore, we test the speed of Proposals of sizes 2^m Kb. To achieve this, we estimate the uniform step area based on a certain steps number that leads to the target cache size, $\alpha = \frac{1}{\text{number of steps}}$.

The speed testing for each possible Proposal is initially based on a sample size of 1000. However, if the user wishes to optimize the Proposal for a different sample size, they can do so by specifying the desired sample size using the `target_sample_size` argument.

In case the user wants to select a specific number of steps for the proposal and bypass the optimization process, this can be done by specifying a steps number greater than the number of modes by 2 using the `steps` argument. If the target density is heavy-tailed, and the user wishes to stop the Proposal building process at a certain pre-acceptance threshold, this can be achieved by setting the acceptance probability threshold `theta` θ. Once the steps reach this level of pre-acceptance probability, the step construction will end $\frac{\min(f(x_i),f(x_{i+1}))}{\max(f(x_i),f(x_{i+1}))} < \theta$. Alternatively, if the user wishes to create the steps within certain limits on the x-axis, they can do so by specifying the proposal limits using the `proposal_range` argument.

Value

This returns a list which is used to construct the sampler by passing to `build_sampler` function.

A list containing the optimized proposal and related parameters for the specified built-in distribution:

`data` A data frame with detailed information about the proposal steps, including:

- `x` The start point of each step on the x-axis.
- `s_upper` The height of each step on the y-axis.
- `p_a` Pre-acceptance probability for each step.
- `s_upper_lower` A vector used to scale the uniform random number when the sample is accepted.

`areas` A numeric vector containing the areas under:

- `left_tail` The left tail bound.
- `steps` The middle steps.
- `right_tail` The right tail bound.

`steps_number` An integer specifying the number of steps in the proposal.

`sampling_probabilities` A numeric vector with:

- `left_tail` The probability of sampling from the left tail.
- `left_and_middle` The combined probability of sampling from the left tail and middle steps.

`unif_scaler` A numeric scalar, the inverse probability of sampling from the steps part of the proposal ($\frac{1}{p(lower<x<upper)}$). Used for scaling uniform random values.

`lt_properties` A numeric vector of 5 values required for Adaptive Rejection Sampling (ARS) in the left tail.

`rt_properties` A numeric vector of 6 values required for ARS in the right tail.

`alpha` A numeric scalar representing the uniform step area.

`tails_method` A string, either `"ARS"` (Adaptive Rejection Sampling) or `"IT"` (Inverse Transform), indicating the sampling method for the tails.

`proposal_bounds` A numeric vector specifying the left and right bounds of the target density.

`cnum` An integer representing the cache number of the created proposal in memory.

`symmetric` A numeric scalar indicating the symmetry point of the proposal, or `NULL` if not symmetric.

`f_params` A list of parameters for the target density that the proposal is designed for.

`is_symmetric` A logical value indicating whether the proposal is symmetric.

`proposal_type` A string indicating the type of the generated proposal:

- `"scaled"` The proposal is "scalable" and standardized with `rate = 1`. This is used when parameter `rate` is either `NULL` or not provided. Scalable proposals are compatible with `srexp`.
- `"custom"` The proposal is "custom" when `rate` is provided. Custom proposals are compatible with `srexp_custom`.

`target_function_area` A numeric scalar estimating the area of the target distribution.

`dens_func` A string containing the hardcoded density function.

`density_name` A string specifying the name of the target density distribution.

`lock` An identifier used for saving and loading the proposal from disk.

See Also

`build_sampler`: Function to build and return a sampling function based on the provided proposal properties.

Examples

```
# Example 1: Building a proposal for Standard Normal Distribution
# This example demonstrates constructing a proposal for a standard normal distribution
# \( f(x) \sim \mathcal{N}(0,1) \),
# and shows the optimization table by setting \code{verbose} to \code{TRUE}.

# Define the density function, its logarithm,
# and its derivative for the standard normal distribution
modes_norm = 0
f_norm <- function(x) { 1 / sqrt(2 * pi) * exp(-0.5 * x^2) }
h_norm <- function(x) { log(f_norm(x)) }
h_prime_norm <- function(x) { -x }

# Following example takes slightly too long to run on CRAN.

# Build the proposal for the standard normal distribution
norm_proposal = build_proposal(lower = -Inf, upper = Inf, mode = modes_norm,
 f = f_norm, h = h_norm, h_prime = h_prime_norm, verbose = TRUE)

# Plot the generated proposal
plot(norm_proposal)


# Example 2: proposal for a Bimodal Distribution
# This example shows how to build a proposal for sampling from a bimodal distribution,
#combining two normal distributions
# \( f(x) = 0.5 \cdot w_1(x) + 0.5 \cdot w_2(x) \),
# where \( w_1(x) \sim \mathcal{N}(0, 1) \) and \( w_2(x) \sim \mathcal{N}(4, 1) \).

# Define the bimodal density function
f_bimodal <- function(x) {
  0.5 * (1 / sqrt(2 * pi)) * exp(-(x^2) / 2) + 0.5 * (1 / sqrt(2 * pi)) * exp(-((x - 4)^2) / 2)
}
```

```
modes_bimodal = c(0.00134865, 3.99865)

# Build the proposal for the bimodal distribution
bimodal_proposal = build_proposal( f = f_bimodal, lower = -Inf, upper = Inf, mode = modes_bimodal)

# Print and plot the bimodal proposal
print(bimodal_proposal)
plot(bimodal_proposal)

# Example 3: Proposal with 500 Steps for Bimodal Distribution
# This example demonstrates constructing a proposal with 500 steps,
# for the bimodal distribution used in Example 2.

bimodal_proposal_500 = build_proposal(f = f_bimodal, lower = -Inf, upper = Inf,
 mode = modes_bimodal, steps = 500)

# Print and plot the proposal with 500 steps
print(bimodal_proposal_500)
```

`build_sampler` *Sampling Function for User Defined Density*

Description

This function generates a sampling function based on a proposal created by the user using the `build_proposal()` function. The resulting sampling function can then be used to produce samples.

Usage

```
build_sampler(proposal)
```

Arguments

`proposal`	The sampling proposal created using the `build_proposal()` function.

Details

After a user creates a proposal for their desired sampling function using `build_proposal`, this proposal must be passed to `build_sampler()` to create a sampling function for the target distribution. `build_sampler()` first checks whether the proposal was indeed created using `build_proposal()`. If the user has altered or modified the proposal returned from `build_proposal()`, `build_sampler()` will reject the altered proposal; therefore, no changes should be made to the proposal after its creation. Once the proposal is accepted by `build_sampler()`, it is cached in memory, allowing fast access to proposal data for the compiled C code and reducing memory access latency. Subsequently, `build_sampler()` returns a function that can be utilized to generate samples from the target distribution,

Value

Returns a function that can be used to generate samples from the specified `proposal`.

Examples

```
# Example 1
# To sample from a standard normal distribution \( f(x) \sim \mathcal{N}(0,1) \),
# first build the proposal using \code{build_proposal()}

modes_norm = 0
f_norm <- function(x) { 1 / sqrt(2 * pi) * exp(-0.5 * x^2) }
h_norm <- function(x) { log(f_norm(x)) }
h_prime_norm <- function(x) { -x }
normal_proposal = build_proposal(lower = -Inf, upper = Inf, mode = modes_norm,
 f = f_norm, h = h_norm, h_prime = h_prime_norm, steps = 1000)

# Generate samples from the standard normal distribution
sample_normal <- build_sampler(normal_proposal)
hist(sample_normal(100), main = "Normal Distribution Samples")


# Example 2
# Let's consider a bimodal distribution composed of two normal distributions:
# The first normal distribution N(0,1) with weight p = 0.3,
# and the second normal distribution N(4,1) with weight q = 0.7.

f_bimodal <- function(x) {
 0.3 * (1 / sqrt(2 * pi) * exp(-0.5 * (x - 0)^2)) +
 0.7 * (1 / sqrt(2 * pi) * exp(-0.5 * (x - 4)^2))
}

# Define the modes of the bimodal distribution
   modes_bimodal <- c(0.00316841, 3.99942)

# Build the proposal for the bimodal distribution
bimodal_proposal = build_proposal(f = f_bimodal, modes = modes_bimodal,
 lower = -Inf, upper = Inf, steps = 1000)

# Create the sampling function using \code{build_sampler()}
sample_bimodal <- build_sampler(bimodal_proposal)

# Generate and plot samples from the bimodal distribution
bimodal_samples <- sample_bimodal(1000)
hist(bimodal_samples, breaks = 30, main = "Bimodal Distribution Samples")

# Create the truncated sampling function using
# \code{build_sampler()} with truncation bounds [-0.5, 6]
truncated_bimodal_proposal <- build_proposal(f = f_bimodal,
 modes = modes_bimodal, lower = -0.5, upper = 6, steps = 1000)

# Create the sampling function using \code{build_sampler()}
sample_truncated_bimodal <- build_sampler(truncated_bimodal_proposal)
```

```
# Generate and plot samples from the truncated bimodal distribution
truncated_sample <- sample_truncated_bimodal(1000)
hist(truncated_sample, breaks = 30, main = "Truncated Bimodal Distribution Samples")
```

`delete_built_in_proposal` *Delete Built-in Proposal*

Description

This function deletes built-in proposals from disk by specifying the sampling function and proposal type. It is useful for managing cached proposals and freeing up storage space.

Usage

```
delete_built_in_proposal(sampling_function, proposal_type = "custom")
```

Arguments

`sampling_function`	String. The name of the sampling distribution's function in STORS. For example, `"srgamma"` or `"srchisq"`.
`proposal_type`	String. Either `"custom"` to delete the custom proposal or `"scaled"` to delete the scaled proposal. Defaults to `"custom"`.

Details

The function looks for the specified proposal type associated with the sampling function in the built-in proposals directory. If the proposal exists, it deletes the corresponding proposal file from disk and frees its cached resources. If the specified sampling function or proposal type does not exist, an error is thrown.

Value

A message indicating the status of the deletion process, or an error if the operation fails.

Examples

```
# The following examples are not run, since if they are run the srgamma and
# srnorm samplers will no longer work until a new grid is built for them.
# This causes problems if the examples are run by CRAN checks or the website
# build system.
## Not run:
# Delete a custom proposal for the srgamma function (uncomment to run)
delete_built_in_proposal(sampling_function = "srgamma", proposal_type = "custom")
```

```
# Delete a scaled proposal for the srnorm function (uncomment to run)
delete_built_in_proposal(sampling_function = "srnorm", proposal_type = "scaled")

## End(Not run)
```

---

delete_proposal *Delete Proposal*

Description

This function deletes a proposal that was previously stored by the user using the `save_proposal()` function. It is useful for managing stored proposals and freeing up space.

Usage

```
delete_proposal(proposal_name)
```

Arguments

`proposal_name`	A string specifying the name of the proposal to be deleted.

Value

If `proposal_name` does not exist, the function returns an error message. If the proposal exists and is successfully deleted, a message confirming its successful removal will be displayed.

Examples

```
# First, let's create a proposal to sample from a standard normal distribution
f_normal <- function(x) { 0.3989423 * exp(-0.5 * x^2) }
normal_proposal = build_proposal(f = f_normal, modes = 0, lower = -Inf, upper = Inf, steps = 1000)
print(normal_proposal)

# Then, save this proposal in R's internal data directory using
# `save_proposal()` with the name "normal"
save_proposal(normal_proposal, "normal")

# Now, we can print all proposals stored on this machine using `print_proposals()`
print_proposals()

# The list will include the `normal_proposal` stored under the name "normal"

# To delete the "normal" proposal from the machine, pass its name to `delete_proposal`
delete_proposal("normal")

# Now, when we print all stored proposals, the "normal" proposal will no longer be listed
print_proposals()
```

load_proposal *Load Stored Proposal*

Description

This function loads a proposal into memory that was previously saved using the `save_proposal()` function. It is useful for retrieving saved proposals for further analysis or processing.

Usage

```
load_proposal(proposal_name)
```

Arguments

`proposal_name`	A string specifying the name of the proposal to be loaded.

Value

Returns a list representing the proposal stored under `proposal_name` in R's internal data directory. If the proposal corresponding to the specified name does not exist, an error message is displayed.

Examples

```
# First, let's create a proposal to sample from a standard normal distribution
f_normal <- function(x) { 0.3989423 * exp(-0.5 * x^2) }
normal_proposal = build_proposal(f = f_normal, modes = 0, lower = -Inf, upper = Inf, steps = 1000)
print(normal_proposal)

# Then, save this proposal in R's internal data directory using
# `save_proposal()` with the name "normal"
save_proposal(normal_proposal, "normal")

# Now, in case the R session is restarted and the proposal is no longer in memory,
# it can be loaded from the machine as follows:
loaded_normal_proposal <- load_proposal("normal")
print(loaded_normal_proposal)
```

plot.proposal *Plot Method for Proposal Objects*

Description

This function evaluates the properties of the included target and proposal functions to create a plot for both functions. In cases where the proposal function's steps part is too dense, `x_min` and `x_max` can be set to crop and scale the chart for better visualization.

Usage

```
## S3 method for class 'proposal'
plot(x, x_min = NA, x_max = NA, ...)
```

Arguments

x	A list generated using STORS' `build_proposal()` or `proposal_optimizer()` functions.
x_min	A scalar that represents the left cropping of the chart on the x-axis.
x_max	A scalar that represents the right cropping of the chart on the x-axis.
...	Additional arguments passed to the `plot` function.

Details

This method extends the generic `plot()` function for objects of class `proposal`. It offers custom plotting functionality specifically designed for visualizing proposal objects.

Value

A plot of the target density and proposal. If `ggplot2` is available, it returns a `ggplot` object representing the plot. otherwise, it uses the base `plot()` function.

See Also

`print.proposal`

Examples

```
# Define the density function, its logarithm,
# and its derivative for the standard normal distribution
modes_norm = 0
f_norm <- function(x) { 1 / sqrt(2 * pi) * exp(-0.5 * x^2) }
h_norm <- function(x) { log(f_norm(x)) }
h_prime_norm <- function(x) { -x }

# Build a dense proposal for the standard normal distribution
norm_proposal = build_proposal(lower = -Inf, upper = Inf, mode = modes_norm,
 f = f_norm, h = h_norm, h_prime = h_prime_norm, steps = 4000)

# Plot the generated proposal
plot(norm_proposal)

# To visualize the proposal in a cropped area between -0.1 and 0
plot(norm_proposal, x_min = -0.1, x_max = 0)
```

print.proposal *Print Method for proposal Objects*

Description

The function displays detailed information about the proposal object created by STORS' `build_proposal()` or `proposal_optimizer()` functions. This includes the number of steps within the proposal, the range of values covered by the proposal, and the proposal's sampling efficiency. This information is crucial for understanding the structure and performance of the proposal in sampling processes.

Usage

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

Arguments

`x`	A list generated using STORS' `build_proposal()` or `proposal_optimizer()` functions.
`...`	Additional arguments passed to the `print` function.

Details

This method extends the generic `print` function for objects of class `proposal`. It prints the provided proposal's features such as the number of steps, steps limit, and efficiency.

Value

Prints a summary of the proposal's properties, but does not return any value.

Examples

```
# Define the density function, its logarithm,
#and its derivative for the standard normal distribution
modes_norm = 0
f_norm <- function(x) { 1 / sqrt(2 * pi) * exp(-0.5 * x^2) }
h_norm <- function(x) { log(f_norm(x)) }
h_prime_norm <- function(x) { -x }

# Build a dense proposal for the standard normal distribution
norm_proposal = build_proposal(lower = -Inf, upper = Inf, mode = modes_norm,
 f = f_norm, h = h_norm, h_prime = h_prime_norm, steps = 1000)

# Print the properties of the generated proposal

print(norm_proposal)
```

print_proposals *Print proposals*

Description

This function prints details of all proposals stored by the user. It provides information on each proposal, including the proposal name, size, efficiency, and other relevant details.

Usage

```
print_proposals()
```

Value

Prints a summary of all proposals, but does not return any value.

Examples

```
# First, let's create a proposal to sample from a standard normal distribution
f_normal <- function(x) { 0.3989423 * exp(-0.5 * x^2) }
normal_proposal = build_proposal(f = f_normal, modes = 0, lower = -Inf, upper = Inf, steps = 1000)
print(normal_proposal)

# `print_proposals()` prints all proposals stored in R's internal data directory.
# To see this, we first save `normal_proposal` using `save_proposal()`
save_proposal(normal_proposal, "normal")

# Since `normal_proposal` is now stored on this machine,
# we can confirm this by printing all saved proposals
print_proposals()

# Example 2: Create and Save a proposal for a Bimodal Distribution
f_bimodal <- function(x) {
  0.5 * (1 / sqrt(2 * pi)) * exp(-(x^2) / 2) +
  0.5 * (1 / sqrt(2 * pi)) * exp(-((x - 4)^2) / 2)
}
modes_bimodal = c(0, 4)
bimodal_proposal = build_proposal(f = f_bimodal, modes = modes_bimodal,
lower = -Inf, upper = Inf, steps = 1000)

save_proposal(bimodal_proposal, "bimodal")
print(bimodal_proposal)

# To print all stored proposals after saving bimodal_proposal
print_proposals()
```

save_proposal *Save User Proposal*

Description

This function stores proposals generated by the `build_proposal()` function in R's internal data directory. It is useful when users want to reuse a proposal across multiple R sessions.

Usage

```
save_proposal(proposal, proposal_name)
```

Arguments

`proposal`	list representing an optimized proposal generated using the `build_proposal()` function.
`proposal_name`	string specifying the name under which the proposal will be saved.

Value

This function will produce an error if the proposal is not generated by the `build_proposal()` function. Otherwise, it successfully saves the proposal without returning any value upon completion.

Examples

```
# First, let's create a proposal to sample from a standard normal distribution
f_normal <- function(x) { 0.3989423 * exp( -0.5 * x^2) }
normal_proposal = build_proposal(f = f_normal, modes = 0, lower = -Inf, upper = Inf, steps = 1000)
print(normal_proposal)

# Then, we can save this proposal in R's internal data directory using `save_proposal()`
# with the name "normal"
save_proposal(normal_proposal, "normal")

# To make sure the `normal_proposal` has been stored in R's internal data directory,
# we can print all saved proposals using `print_proposals()`
print_proposals()
```

srbeta_custom *Sampling from Beta Distribution*

Description

The srbeta_custom() function generates random samples from a Beta distribution using the STORS algorithm. It employs an optimized proposal distribution around the mode and Adaptive Rejection Sampling (ARS) for the tails.

Usage

```
srbeta_custom(n = 1, x = NULL)
```

Arguments

n	Integer, length 1. Number of samples to draw.
x	(optional) Numeric vector of length n. If provided, this vector is overwritten in place to avoid any memory allocation.

Details

The Beta Distribution

The Beta distribution has the probability density function (PDF):

$$f(x|\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}, \quad 0 \le x \le 1,$$

where:

α is the first shape parameter ($\alpha > 0$).

β is the second shape parameter ($\beta > 0$).

The Beta distribution is widely used in Bayesian statistics and in modelling probabilities and proportions.

Value

A numeric vector of length n containing random samples from the Beta distribution. The shape1 and shape2 parameters are specified during the optimization process using srbeta_optimize().

NOTE: When the x parameter is specified, it is updated in-place with the simulation for performance reasons.

TODO : This density instead of this function.

This function samples from a proposal constructed using srbeta_optimize, employing the STORS algorithm.

By default, srbeta_custom() samples from the standard Beta distribution with shape1 = 1 and shape2 = 1. The proposal distribution is pre-optimized at package load time using srbeta_optimize() with steps = 4091, creating a scalable proposal centred around the mode.

Note

This function is not scalable. Therefore, only the `srbeta_custom()` version is available, which requires the proposal to be pre-optimized using `srbeta_optimize()` before calling this function.

See Also

`srbeta_optimize` to optimize the custom proposal.

Examples

```
# Generate 10 samples from Beta Distribution
samples <- srbeta_custom(10)
print(samples)

# Generate 10 samples using a pre-allocated vector
x <- numeric(10)
srbeta_custom(10, x = x)
print(x)
```

---

srbeta_optimize *Optimizing Beta Distribution proposal*

Description

The `srbeta_optimize()` function generates an optimized proposal for a targeted Beta distribution. The proposal can be customized and adjusted based on various options provided by the user.

Usage

```
srbeta_optimize(
  shape1 = 2,
  shape2 = 2,
  xl = 0,
  xr = 1,
  steps = NULL,
  proposal_range = NULL,
  theta = 0.1,
  target_sample_size = 1000,
  verbose = FALSE
)
```

Arguments

`shape1`	(optional) Numeric. The first shape parameter (α) of the Beta distribution. Defaults to 1.
`shape2`	(optional) Numeric. The second shape parameter (β) of the Beta distribution. Defaults to 1.

xl	Numeric. Left truncation bound for the target distribution. Defaults to 0, as the Beta distribution is defined only on the interval [0, 1].
xr	Numeric. Right truncation bound for the target distribution. Defaults to 1, as the Beta distribution is defined only on the interval [0, 1].
steps	(optional) Integer. Desired number of steps in the proposal. Defaults to 4091.
proposal_range	(optional) Numeric vector. Specifies the range for optimizing the steps part of the proposal. Defaults to NULL, indicating automatic range selection.
theta	Numeric. A parameter for proposal optimization. Defaults to 0.1.
target_sample_size	(optional) Integer. Target sample size for proposal optimization. Defaults to 1000.
verbose	Boolean. If TRUE, detailed optimization information, including areas and steps, will be displayed. Defaults to FALSE.

Details

When srbeta_optimize() is explicitly called:

- A proposal is created and cached. If no parameters are provided, a standard proposal is created with shape1 = 1 and shape2 = 1.
- Providing shape1 and shape2 creates a custom proposal, which is cached for use with srbeta_custom().
- The optimization process can be controlled via parameters such as steps, proposal_range, or theta. If no parameters are provided, the proposal is optimized via brute force based on the target_sample_size.

Value

A list containing the optimized proposal and related parameters for the specified Beta distribution. The proposal is also cached for internal use.

data Detailed information about the proposal steps, including x, s_upper, p_a, and s_upper_lower.

areas The areas under the left tail, steps, and right tail of the proposal distribution.

steps_number The number of steps in the proposal.

f_params The parameters (shape1 and shape2) of the Beta distribution.

See Also

srbeta_custom: Function to sample from a custom proposal generated by srbeta_optimize().

Examples

```
# Generate a standard proposal with shape1 = 1 and shape2 = 1
standard_proposal <- srbeta_optimize()

# Generate a custom proposal with shape1 = 2 and shape2 = 3
custom_proposal <- srbeta_optimize(shape1 = 2, shape2 = 3)
```

`srchisq_custom` *Sampling from Chi-squared Distribution*

Description

The `srchisq_custom()` function generates random samples from a Chi-squared Distribution using the STORS algorithm. It employs an optimized proposal distribution around the mode and Adaptive Rejection Sampling (ARS) for the tails.

Usage

```
srchisq_custom(n = 1, x = NULL)
```

Arguments

`n`	Integer, length 1. Number of samples to draw.
`x`	(optional) Numeric vector of length n. If provided, this vector is over written in place to avoid any memory allocation.

Details

The Chi-squared Distribution

The Chi-squared distribution has the probability density function (PDF):

$$f(x|k) = \frac{1}{2^{k/2}\Gamma(k/2)} x^{(k/2)-1} \exp(-x/2), \quad x \geq 0,$$

where:

k is the degrees of freedom ($k > 0$), which determines the shape of the distribution.

The Chi-squared distribution is widely used in hypothesis testing and constructing confidence intervals, particularly in the context of variance estimation.

this function is sampling from proposal that has been constructed using `srchisq_optimize`, using the STORS algorithm.

By default, `srchisq_custom()` samples from Chi-squared Distribution `df = 2`. The proposal distribution is pre-optimized at package load time using `srchisq_optimize()` with `steps = 4091`, creating a scalable proposal centred around the mode.

Value

A numeric vector of length `n` containing random samples from the Chi-squared distribution. The degrees of freedom (`df`) for the distribution are specified during the optimization process using `srchisq_optimize()`. **NOTE:** When the `x` parameter is specified, it is updated in-place with the simulation for performance reasons.

Note

This function is not scalable. Therefore, only the `srchisq_custom()` version is available, which requires the proposal to be pre-optimized using `srchisq_optimize()` before calling this function.

See Also

`srchisq_optimize` to optimize the custom proposal.

Examples

```
# Genedf 10 samples from Chi-squared Distribution
samples <- srchisq_custom(10)
print(samples)

# Genedf 10 samples using a pre-allocated vector
x <- numeric(10)
srchisq_custom(10, x = x)
print(x)
```

---

srchisq_optimize *Optimizing Chi-squared Distribution proposal*

Description

The `srchisq_optimize()` function generates an optimized proposal for a targeted Chi-squared Distribution. The proposal can be customized and adjusted based on various options provided by the user.

Usage

```
srchisq_optimize(
  df = 2,
  xl = NULL,
  xr = NULL,
  steps = 4091,
  proposal_range = NULL,
  theta = 0.1,
  target_sample_size = 1000,
  verbose = FALSE
)
```

Arguments

df	(optional) Numeric. degrees of freedom parameter of the Chi-squared Distribution. Defaults to NULL, which implies proposal with df = 2.
xl	Numeric. Left truncation bound for the target distribution. Defaults to -Inf, representing no left truncation.
xr	Numeric. Right truncation bound for the target distribution. Defaults to Inf, representing no right truncation.
steps	(optional) Integer. Desired number of steps in the proposal. Defaults to NULL, which means the number of steps is determined automatically during optimization.
proposal_range	(optional) Numeric vector. Specifies the range for optimizing the steps part of the proposal. Defaults to NULL, indicating automatic range selection.
theta	Numeric. A parameter for proposal optimization. Defaults to 0.1.
target_sample_size	(optional) Integer. Target sample size for proposal optimization. Defaults to 1000.
verbose	Boolean. If TRUE, detailed optimization information, including areas and steps, will be displayed. Defaults to FALSE.

Details

When srchisq_optimize() is explicitly called:

- A proposal is created and cached. If no parameters are provided, a standard proposal is created with df = 2.
- Providing df creates a custom proposal, which is cached for use with srchisq_custom().
- The optimization process can be controlled via parameters such as steps, proposal_range, or theta. If no parameters are provided, the proposal is optimized via brute force based on the. target_sample_size.

Value

The user does not need to store the returned value, because the package internally cashes the proposal. However, we explain here the full returned proposal for advanced users.

A list containing the optimized proposal and related parameters for the specified built-in distribution:

data A data frame with detailed information about the proposal steps, including:

- x The start point of each step on the x-axis.
- s_upper The height of each step on the y-axis.
- p_a Pre-acceptance probability for each step.
- s_upper_lower A vector used to scale the uniform random number when the sample is accepted.

areas A numeric vector containing the areas under:

- left_tail The left tail bound.

`steps` The middle steps.

`right_tail` The right tail bound.

`steps_number` An integer specifying the number of steps in the proposal.

`sampling_probabilities` A numeric vector with:

`left_tail` The probability of sampling from the left tail.

`left_and_middle` The combined probability of sampling from the left tail and middle steps.

`unif_scaler` A numeric scalar, the inverse probability of sampling from the steps part of the proposal ($\frac{1}{p(lower<x<upper)}$). Used for scaling uniform random values.

`lt_properties` A numeric vector of 5 values required for Adaptive Rejection Sampling (ARS) in the left tail.

`rt_properties` A numeric vector of 6 values required for ARS in the right tail.

`alpha` A numeric scalar representing the uniform step area.

`tails_method` A string, either `"ARS"` (Adaptive Rejection Sampling) or `"IT"` (Inverse Transform), indicating the sampling method for the tails.

`proposal_bounds` A numeric vector specifying the left and right bounds of the target density.

`cnum` An integer representing the cache number of the created proposal in memory.

`symmetric` A numeric scalar indicating the symmetry point of the proposal, or `NULL` if not symmetric.

`f_params` A list of parameters for the target density that the proposal is designed for.

`df` the df of the target distribution.

`is_symmetric` A logical value indicating whether the proposal is symmetric.

`proposal_type` A string indicating the type of the genedfd proposal:

`"custom"` The proposal is "custom" when `df` is provided. Custom proposals are compatible with `srchisq_custom`.

`target_function_area` A numeric scalar estimating the area of the target distribution.

`dens_func` A string containing the hardcoded density function.

`density_name` A string specifying the name of the target density distribution.

`lock` An identifier used for saving and loading the proposal from disk.

See Also

`srchisq_custom`: Function to sample from a custom proposal genedfd by `srchisq_optimize()`.

Examples

```
# Genedf custom proposal that with df = 2, that has 4096 steps
scalable_proposal <- srchisq_optimize(steps = 4096)

# Genedf custom proposal that with df = 4
scalable_proposal <- srchisq_optimize(df = 4)
```

srexp *Sampling from Exponential Distribution*

Description

The `srexp()` function generates random samples from a Exponential Distribution using the STORS algorithm. It employs an optimized proposal distribution around the mode and Inverse Transform (IT) method for the tails.

Usage

```
srexp(n = 1, rate = 1, x = NULL)

srexp_custom(n = 1, x = NULL)
```

Arguments

`n`	Integer, length 1. Number of samples to draw.
`rate`	Numeric. is the rate parameter of the Exponential Distribution.
`x`	(optional) Numeric vector of length n. If provided, this vector is over written in place to avoid any memory allocation.

Details

The Exponential distribution has the probability density function (PDF): $f(x|\lambda) = \lambda \exp(-\lambda x), \quad x \geq 0$, where:

λ is the rate parameter ($\lambda > 0$), which determines the rate of decay of the distribution.

The Exponential distribution is commonly used to model the time between independent events that occur at a constant average rate.

These two functions are for sampling using the STORS algorithm based on the proposal that has been constructed using `srexp_optimize`.

By default, `srexp()` samples from a standard Exponential Distribution `rate = 1`. The proposal distribution is pre-optimized at package load time using `srexp_optimize()` with `steps = 4091`, creating a scalable proposal centred around the mode.

If `srexp()` is called with custom `rate` parameter, the samples are generated from the standard Exponential Distribution, then scaled accordingly.

Value

A numeric vector of length `n` containing samples from the Exponential Distribution with the specified `rate`.

NOTE: When the `x` parameter is specified, it is updated in-place with the simulation for performance reasons.

See Also

srexp_optimize to optimize the custom or the scaled proposal.

Examples

```
# Generate 10 samples from the standard Exponential Distribution
samples <- srexp(10)
print(samples)

# Generate 10 samples using a pre-allocated vector
x <- numeric(10)
srexp(10, x = x)
print(x)

# Generate 10 samples from a Exponential Distribution with rate = 4
samples <- srexp(10, rate = 4)
print(samples)
```

---

srexp_optimize *Optimizing Exponential Distribution proposal*

Description

The srexp_optimize() function generates an optimized proposal for a targeted Exponential Distribution. The proposal can be customized and adjusted based on various options provided by the user.

Usage

```
srexp_optimize(
  rate = NULL,
  xl = NULL,
  xr = NULL,
  steps = 4091,
  proposal_range = NULL,
  theta = 0.1,
  target_sample_size = 1000,
  verbose = FALSE
)
```

Arguments

rate	(optional) Numeric. rate parameter of the Exponential Distribution. Defaults to NULL, which implies a scalable proposal with rate = 1.
xl	Numeric. Left truncation bound for the target distribution. Defaults to -Inf, representing no left truncation.

xr	Numeric. Right truncation bound for the target distribution. Defaults to Inf, representing no right truncation.
steps	(optional) Integer. Desired number of steps in the proposal. Defaults to NULL, which means the number of steps is determined automatically during optimization.
proposal_range	(optional) Numeric vector. Specifies the range for optimizing the steps part of the proposal. Defaults to NULL, indicating automatic range selection.
theta	Numeric. A parameter for proposal optimization. Defaults to 0.1.
target_sample_size	(optional) Integer. Target sample size for proposal optimization. Defaults to 1000.
verbose	Boolean. If TRUE, detailed optimization information, including areas and steps, will be displayed. Defaults to FALSE.

Details

When srexp_optimize() is explicitly called:

- A proposal is created and cached. If no parameters are provided, a standard proposal is created with rate = 1.
- Providing rate creates a custom proposal, which is cached for use with srexp_custom().
- The optimization process can be controlled via parameters such as steps, proposal_range, or theta. If no parameters are provided, the proposal is optimized via brute force based on the. target_sample_size.

Value

The user does not need to store the returned value, because the package internally cashes the proposal. However, we explain here the full returned proposal for advanced users.

A list containing the optimized proposal and related parameters for the specified built-in distribution:

data A data frame with detailed information about the proposal steps, including:

- x The start point of each step on the x-axis.
- s_upper The height of each step on the y-axis.
- p_a Pre-acceptance probability for each step.
- s_upper_lower A vector used to scale the uniform random number when the sample is accepted.

areas A numeric vector containing the areas under:

- left_tail The left tail bound.
- steps The middle steps.
- right_tail The right tail bound.

steps_number An integer specifying the number of steps in the proposal.

sampling_probabilities A numeric vector with:

- left_tail The probability of sampling from the left tail.

`left_and_middle` The combined probability of sampling from the left tail and middle steps.

`unif_scaler` A numeric scalar, the inverse probability of sampling from the steps part of the proposal ($\frac{1}{p(lower<x<upper)}$). Used for scaling uniform random values.

`lt_properties` A numeric vector of 5 values required for Adaptive Rejection Sampling (ARS) in the left tail.

`rt_properties` A numeric vector of 6 values required for ARS in the right tail.

`alpha` A numeric scalar representing the uniform step area.

`tails_method` A string, either `"ARS"` (Adaptive Rejection Sampling) or `"IT"` (Inverse Transform), indicating the sampling method for the tails.

`proposal_bounds` A numeric vector specifying the left and right bounds of the target density.

`cnum` An integer representing the cache number of the created proposal in memory.

`symmetric` A numeric scalar indicating the symmetry point of the proposal, or `NULL` if not symmetric.

`f_params` A list of parameters for the target density that the proposal is designed for.

`rate` the rate of the target distribution.

`is_symmetric` A logical value indicating whether the proposal is symmetric.

`proposal_type` A string indicating the type of the generated proposal:

`"scaled"` The proposal is "scalable" and standardized with `rate = 1`. This is used when parameter `rate` is either `NULL` or not provided. Scalable proposals are compatible with `srexp`.

`"custom"` The proposal is "custom" when `rate` is provided. Custom proposals are compatible with `srexp_custom`.

`target_function_area` A numeric scalar estimating the area of the target distribution.

`dens_func` A string containing the hardcoded density function.

`density_name` A string specifying the name of the target density distribution.

`lock` An identifier used for saving and loading the proposal from disk.

See Also

`srexp`: Function to sample from a scalable proposal generated by `srexp_optimize()`. `srexp_custom`: Function to sample from a custom proposal tailored to user specifications.

Examples

```
# Generate scalable proposal that with rate = 1, that has 4096 steps
scalable_proposal <- srexp_optimize(steps = 4096)

# Generate custom proposal that with rate = 4
scalable_proposal <- srexp_optimize(rate = 4)
```

srgamma_custom *Sampling from Gamma Distribution*

Description

The srgamma_custom() function generates random samples from a Gamma distribution using the STORS algorithm. It employs an optimized proposal distribution around the mode and Adaptive Rejection Sampling (ARS) for the tails.

Usage

```
srgamma_custom(n = 1, x = NULL)
```

Arguments

n	Integer, length 1. Number of samples to draw.
x	(optional) Numeric vector of length n. If provided, this vector is overwritten in place to avoid any memory allocation.

Details

The Gamma Distribution

The Gamma distribution has the probability density function (PDF):

$$f(x|\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} \exp(-\beta x), \quad x \geq 0,$$

where:

α is the shape parameter ($\alpha > 0$), which determines the shape of the distribution.

β is the rate parameter ($\beta > 0$), which determines the rate of decay.

The Gamma distribution is widely used in statistics, particularly in Bayesian inference and modelling waiting times.

This function samples from a proposal constructed using srgamma_optimize, employing the STORS algorithm.

By default, srgamma_custom() samples from the standard Gamma distribution with shape = 1 and rate = 1. The proposal distribution is pre-optimized at package load time using srgamma_optimize() with steps = 4091, creating a scalable proposal centred around the mode.

Value

A numeric vector of length n containing random samples from the Gamma distribution. The shape and rate parameters are specified during the optimization process using srgamma_optimize().

NOTE: When the x parameter is specified, it is updated in-place with the simulation for performance reasons.

Note

This function is not scalable. Therefore, only the `srgamma_custom()` version is available, which requires the proposal to be pre-optimized using `srgamma_optimize()` before calling this function.

See Also

`srgamma_optimize` to optimize the custom proposal.

Examples

```
# Generate 10 samples from Gamma Distribution
samples <- srgamma_custom(10)
print(samples)

# Generate 10 samples using a pre-allocated vector
x <- numeric(10)
srgamma_custom(10, x = x)
print(x)
```

`srgamma_optimize` *Optimizing Gamma Distribution proposal*

Description

The `srgamma_optimize()` function generates an optimized proposal for a targeted Gamma distribution. The proposal can be customized and adjusted based on various options provided by the user.

Usage

```
srgamma_optimize(
  shape = NULL,
  rate = NULL,
  scale = NULL,
  xl = NULL,
  xr = NULL,
  steps = 4091,
  proposal_range = NULL,
  theta = 0.1,
  target_sample_size = 1000,
  verbose = FALSE
)
```

Arguments

shape	(optional) Numeric. The shape parameter (α) of the Gamma distribution. Defaults to 1.
rate	(optional) Numeric. The rate parameter (β) of the Gamma distribution. Defaults to 1.
scale	(optional) Numeric. The scale parameter of the Gamma distribution. Defaults to 1.
xl	Numeric. Left truncation bound for the target distribution. Defaults to 0, representing no left truncation.
xr	Numeric. Right truncation bound for the target distribution. Defaults to Inf, representing no right truncation.
steps	(optional) Integer. Desired number of steps in the proposal. Defaults to 4091.
proposal_range	(optional) Numeric vector. Specifies the range for optimizing the steps part of the proposal. Defaults to NULL, indicating automatic range selection.
theta	Numeric. A parameter for proposal optimization. Defaults to 0.1.
target_sample_size	(optional) Integer. Target sample size for proposal optimization. Defaults to 1000.
verbose	Boolean. If TRUE, detailed optimization information, including areas and steps, will be displayed. Defaults to FALSE.

Details

When srgamma_optimize() is explicitly called:

- A proposal is created and cached. If no parameters are provided, a standard proposal is created with shape = 1 and rate = 1.
- Providing shape and rate creates a custom proposal, which is cached for use with srgamma_custom().
- The optimization process can be controlled via parameters such as steps, proposal_range, or theta. If no parameters are provided, the proposal is optimized via brute force based on the target_sample_size.

Value

A list containing the optimized proposal and related parameters for the specified Gamma distribution. The proposal is also cached for internal use.

data Detailed information about the proposal steps, including x, s_upper, p_a, and s_upper_lower.

areas The areas under the left tail, steps, and right tail of the proposal distribution.

steps_number The number of steps in the proposal.

f_params The parameters (shape and rate) of the Gamma distribution.

See Also

srgamma_custom: Function to sample from a custom proposal generated by srgamma_optimize().

Examples

```
# Generate a standard proposal with shape = 1 and rate = 1
standard_proposal <- srgamma_optimize()

# Generate a custom proposal with shape = 2 and rate = 3
custom_proposal <- srgamma_optimize(shape = 2, rate = 3)
```

`srlaplace` *Sampling from Laplace Distribution*

Description

The `srlaplace()` function generates random samples from a Laplace Distribution using the STORS algorithm. It employs an optimized proposal distribution around the mode and Inverse Transform (IT) method for the tails.

Usage

```
srlaplace(n = 1, mu = 0, b = 1, x = NULL)

srlaplace_custom(n = 1, x = NULL)
```

Arguments

`n`	Integer, length 1. Number of samples to draw.
`mu`	Numeric, location parameter.
`b`	Numeric, scale parameter.
`x`	(optional) Numeric vector of length n. If provided, this vector is over written in place to avoid any memory allocation.

Details

The Laplace distribution has the probability density function (PDF): $f(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|x-\mu|}{b}\right)$, where:

μ is the location parameter (mean of the distribution).

`b` is the scale parameter, which controls the spread of the distribution (`b > 0`).

These two functions are for sampling using the STORS algorithm based on the proposal that has been constructed using `srlaplace_optimize`.

By default, `srlaplace()` samples from a standard Laplace Distribution (`mu = 0`, `b = 1`). The proposal distribution is pre-optimized at package load time using `srlaplace_optimize()` with `steps = 4091`, creating a scalable proposal centred around the mode.

If `srlaplace()` is called with custom `mu` or `b` parameters, the samples are generated from the standard Laplace Distribution, then scaled and location shifted accordingly.

Value

A numeric vector of length n containing samples from the Laplace Distribution with the specified mu and b.

NOTE: When the x parameter is specified, it is updated in-place with the simulation for performance reasons.

See Also

srlaplace_optimize to optimize the custom or the scaled proposal.

Examples

```
# Generate 10 samples from the standard Laplace Distribution
samples <- srlaplace(10)
print(samples)

# Generate 10 samples using a pre-allocated vector
x <- numeric(10)
srlaplace(10, x = x)
print(x)

# Generate 10 samples from a Laplace Distribution with mu = 2 and b = 3
samples <- srlaplace(10, mu = 2, b = 3)
print(samples)
```

---

srlaplace_optimize *Optimizing Laplace Distribution proposal*

Description

The srlaplace_optimize() function generates an optimized proposal for a targeted Laplace Distribution. The proposal can be customized and adjusted based on various options provided by the user.

Usage

```
srlaplace_optimize(
  mu = NULL,
  b = NULL,
  xl = NULL,
  xr = NULL,
  steps = 4091,
  proposal_range = NULL,
  theta = 0.1,
  target_sample_size = 1000,
  verbose = FALSE,
  symmetric = FALSE
)
```

Arguments

mu	(optional) Numeric, location parameter.
b	(optional) Numeric, scale parameter.
xl	Numeric. Left truncation bound for the target distribution. Defaults to -Inf, representing no left truncation.
xr	Numeric. Right truncation bound for the target distribution. Defaults to Inf, representing no right truncation.
steps	(optional) Integer. Desired number of steps in the proposal. Defaults to NULL, which means the number of steps is determined automatically during optimization.
proposal_range	(optional) Numeric vector. Specifies the range for optimizing the steps part of the proposal. Defaults to NULL, indicating automatic range selection.
theta	Numeric. A parameter for proposal optimization. Defaults to 0.1.
target_sample_size	(optional) Integer. Target sample size for proposal optimization. Defaults to 1000.
verbose	Boolean. If TRUE, detailed optimization information, including areas and steps, will be displayed. Defaults to FALSE.
symmetric	Boolean. If TRUE, the proposal will target only the right tail of the distribution, reducing the size of the cached proposal and making sampling more memory-efficient. An additional uniform random number will be sampled to determine the sample's position relative to the mode of the distribution. While this improves memory efficiency, the extra sampling may slightly impact performance, especially when the proposal efficiency is close to 1. Defaults to FALSE.

Details

When srlaplace_optimize() is explicitly called:

- A proposal is created and cached. If no parameters are provided, a standard proposal is created (mu = 0, b = 1).
- Providing mu or b creates a custom proposal, which is cached for use with srlaplace_custom().
- The optimization process can be controlled via parameters such as steps, proposal_range, or theta. If no parameters are provided, the proposal is optimized via brute force based on the. target_sample_size.

Value

The user does not need to store the returned value, because the package internally cashes the proposal. However, we explain here the full returned proposal for advanced users.

A list containing the optimized proposal and related parameters for the specified built-in distribution:

data A data frame with detailed information about the proposal steps, including:

x The start point of each step on the x-axis.

`s_upper` The height of each step on the y-axis.

`p_a` Pre-acceptance probability for each step.

`s_upper_lower` A vector used to scale the uniform random number when the sample is accepted.

`areas` A numeric vector containing the areas under:

`left_tail` The left tail bound.

`steps` The middle steps.

`right_tail` The right tail bound.

`steps_number` An integer specifying the number of steps in the proposal.

`sampling_probabilities` A numeric vector with:

`left_tail` The probability of sampling from the left tail.

`left_and_middle` The combined probability of sampling from the left tail and middle steps.

`unif_scaler` A numeric scalar, the inverse probability of sampling from the steps part of the proposal ($\frac{1}{p(lower<x<upper)}$). Used for scaling uniform random values.

`lt_properties` A numeric vector of 5 values required for Adaptive Rejection Sampling (ARS) in the left tail.

`rt_properties` A numeric vector of 6 values required for ARS in the right tail.

`alpha` A numeric scalar representing the uniform step area.

`tails_method` A string, either `"ARS"` (Adaptive Rejection Sampling) or `"IT"` (Inverse Transform), indicating the sampling method for the tails.

`proposal_bounds` A numeric vector specifying the left and right bounds of the target density.

`cnum` An integer representing the cache number of the created proposal in memory.

`symmetric` A numeric scalar indicating the symmetry point of the proposal, or `NULL` if not symmetric.

`f_params` A list of parameters for the target density that the proposal is designed for.

`mu` is the location parameter (location of the distribution).

`b` is the scale parameter, which controls the spread of the distribution (`b > 0`).

`is_symmetric` A logical value indicating whether the proposal is symmetric.

`proposal_type` A string indicating the type of the generated proposal:

`"scaled"` The proposal is "scalable" and standardized with `mu = 0` and `b = 1`. This is used when parameters `mu` and `b` are either `NULL` or not provided. Scalable proposals are compatible with `srlaplace`.

`"custom"` The proposal is "custom" when either `mu` or `b` is provided. Custom proposals are compatible with `srlaplace_custom`.

`target_function_area` A numeric scalar estimating the area of the target distribution.

`dens_func` A string containing the hardcoded density function.

`density_name` A string specifying the name of the target density distribution.

`lock` An identifier used for saving and loading the proposal from disk.

See Also

srlaplace: Function to sample from a scalable proposal generated by srlaplace_optimize().
srlaplace_custom: Function to sample from a custom proposal tailored to user specifications.

Examples

```
# Generate scalable proposal that with mu = 0 and b = 1, that has 4096 steps
scalable_proposal <- srlaplace_optimize(steps = 4096)

# Generate custom proposal that with mu = 2 and b = 1
scalable_proposal <- srlaplace_optimize(mu = 2, b = 1)
```

---

srnorm *Sampling from Normal Distribution*

Description

The srnorm() function generates random samples from a Normal distribution using the STORS algorithm. It employs an optimized proposal distribution around the mode and Adaptive Rejection Sampling (ARS) for the tails.

Usage

```
srnorm(n = 1, mean = 0, sd = 1, x = NULL)

srnorm_custom(n = 1, x = NULL)
```

Arguments

n	Integer, length 1. Number of samples to draw.
mean	Numeric. Mean parameter of the Normal distribution.
sd	Numeric. Standard deviation of the target Normal distribution.
x	(optional) Numeric vector of length n. If provided, this vector is over written in place to avoid any memory allocation.

Details

The Normal distribution has the probability density function (PDF): $f(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$, where:

- μ is the mean of the distribution, which determines the centre of the bell curve.
- σ is the standard deviation, which controls the spread of the distribution ($\sigma > 0$).

These two functions are for sampling using the STORS algorithm based on the proposal that has been constructed using srnorm_optimize.

By default, srnorm() samples from a standard Normal distribution (mean = 0, sd = 1). The proposal distribution is pre-optimized at package load time using srnorm_optimize() with steps = 4091, creating a scalable proposal centred around the mode.

If srnorm() is called with custom mean or sd parameters, the samples are generated from the standard Normal distribution, then scaled and location shifted accordingly.

Value

A numeric vector of length n containing samples from the Normal distribution with the specified mean and sd.

NOTE: When the x parameter is specified, it is updated in-place with the simulation for performance reasons.

See Also

srnorm_optimize to optimize the custom or the scaled proposal.

Examples

```
# Generate 10 samples from the standard Normal distribution
samples <- srnorm(10)
print(samples)

# Generate 10 samples using a pre-allocated vector
x <- numeric(10)
srnorm(10, x = x)
print(x)

# Generate 10 samples from a Normal distribution with mean = 2 and sd = 3
samples <- srnorm(10, mean = 2, sd = 3)
print(samples)
```

---

srnorm_optimize *Optimizing Normal Distribution proposal*

Description

The srnorm_optimize() function generates an optimized proposal for a targeted Normal distribution. The proposal can be customized and adjusted based on various options provided by the user.

Usage

```
srnorm_optimize(
  mean = NULL,
  sd = NULL,
  xl = -Inf,
  xr = Inf,
  steps = NULL,
  proposal_range = NULL,
  theta = 0.1,
  target_sample_size = 1000,
  verbose = FALSE,
  symmetric = FALSE
)
```

Arguments

mean	(optional) Numeric. Mean parameter of the Normal distribution. Defaults to NULL, which implies a scalable proposal with mean = 0.
sd	(optional) Numeric. Standard deviation of the target Normal distribution. Defaults to NULL, which implies a scalable proposal with sd = 1.
xl	Numeric. Left truncation bound for the target distribution. Defaults to -Inf, representing no left truncation.
xr	Numeric. Right truncation bound for the target distribution. Defaults to Inf, representing no right truncation.
steps	(optional) Integer. Desired number of steps in the proposal. Defaults to NULL, which means the number of steps is determined automatically during optimization.
proposal_range	(optional) Numeric vector. Specifies the range for optimizing the steps part of the proposal. Defaults to NULL, indicating automatic range selection.
theta	Numeric. A parameter for proposal optimization. Defaults to 0.1.
target_sample_size	(optional) Integer. Target sample size for proposal optimization. Defaults to 1000.
verbose	Boolean. If TRUE, detailed optimization information, including areas and steps, will be displayed. Defaults to FALSE.
symmetric	Boolean. If TRUE, the proposal will target only the right tail of the distribution, reducing the size of the cached proposal and making sampling more memory-efficient. An additional uniform random number will be sampled to determine the sample's position relative to the mode of the distribution. While this improves memory efficiency, the extra sampling may slightly impact performance, especially when the proposal efficiency is close to 1. Defaults to FALSE.

Details

When srnorm_optimize() is explicitly called:

- A proposal is created and cached. If no parameters are provided, a standard proposal is created (`mean = 0`, `sd = 1`).
- Providing `mean` or `sd` creates a custom proposal, which is cached for use with `srnorm_custom()`.
- The optimization process can be controlled via parameters such as `steps`, `proposal_range`, or `theta`. If no parameters are provided, the proposal is optimized via brute force based on the. `target_sample_size`.

Value

The user does not need to store the returned value, because the package internally cashes the proposal. However, we explain here the full returned proposal for advanced users.

A list containing the optimized proposal and related parameters for the specified built-in distribution:

`data` A data frame with detailed information about the proposal steps, including:

- `x` The start point of each step on the x-axis.
- `s_upper` The height of each step on the y-axis.
- `p_a` Pre-acceptance probability for each step.
- `s_upper_lower` A vector used to scale the uniform random number when the sample is accepted.

`areas` A numeric vector containing the areas under:

- `left_tail` The left tail bound.
- `steps` The middle steps.
- `right_tail` The right tail bound.

`steps_number` An integer specifying the number of steps in the proposal.

`sampling_probabilities` A numeric vector with:

- `left_tail` The probability of sampling from the left tail.
- `left_and_middle` The combined probability of sampling from the left tail and middle steps.

`unif_scaler` A numeric scalar, the inverse probability of sampling from the steps part of the proposal ($\frac{1}{p(lower<x<upper)}$). Used for scaling uniform random values.

`lt_properties` A numeric vector of 5 values required for Adaptive Rejection Sampling (ARS) in the left tail.

`rt_properties` A numeric vector of 6 values required for ARS in the right tail.

`alpha` A numeric scalar representing the uniform step area.

`tails_method` A string, either `"ARS"` (Adaptive Rejection Sampling) or `"IT"` (Inverse Transform), indicating the sampling method for the tails.

`proposal_bounds` A numeric vector specifying the left and right bounds of the target density.

`cnum` An integer representing the cache number of the created proposal in memory.

`symmetric` A numeric scalar indicating the symmetry point of the proposal, or `NULL` if not symmetric.

`f_params` A list of parameters for the target density that the proposal is designed for.

- `mean` The mean of the target distribution.

`sd` The standard deviation of the target distribution.

`is_symmetric` A logical value indicating whether the proposal is symmetric.

`proposal_type` A string indicating the type of the generated proposal:

- `"scaled"` The proposal is "scalable" and standardized with `mean = 0` and `sd = 1`. This is used when parameters `mean` and `sd` are either `NULL` or not provided. Scalable proposals are compatible with `srnorm`.
- `"custom"` The proposal is "custom" when either `mean` or `sd` is provided. Custom proposals are compatible with `srnorm_custom`.

`target_function_area` A numeric scalar estimating the area of the target distribution.

`dens_func` A string containing the hardcoded density function.

`density_name` A string specifying the name of the target density distribution.

`lock` An identifier used for saving and loading the proposal from disk.

See Also

`srnorm`: Function to sample from a scalable proposal generated by `srnorm_optimize()`. `srnorm_custom`: Function to sample from a custom proposal tailored to user specifications.

Examples

```
# Generate scalable proposal that with mean = 0 and sd = 1, that has 4096 steps
scalable_proposal <- srnorm_optimize(steps = 4096)


# Generate custom proposal that with mean = 2 and sd = 1
scalable_proposal <- srnorm_optimize(mean = 2, sd = 1)
```

---

srpareto_custom *Sampling from Pareto Distribution*

Description

The `srpareto_custom()` function generates random samples from a Pareto distribution using the STORS algorithm. It employs an optimized proposal distribution around the mode and Inverse Transform (IT) method for the tails.

Usage

```
srpareto_custom(n = 1, x = NULL)
```

Arguments

`n`	Integer, length 1. Number of samples to draw.
`x`	(optional) Numeric vector of length n. If provided, this vector is overwritten in place to avoid any memory allocation.

Details

The Pareto Distribution

The Pareto distribution has the probability density function (PDF):

$$f(x|\alpha,\sigma) = \frac{\alpha\sigma^{\alpha}}{x^{\alpha+1}}, \quad x \geq \sigma,$$

where:

α is the shape parameter ($\alpha > 0$), which determines the tail heaviness of the distribution.

σ is the scale parameter ($\sigma > 0$), which determines the minimum possible value of x.

The Pareto distribution is widely used in modelling phenomena with heavy tails, such as wealth distribution, insurance losses, and natural events.

This function samples from a proposal constructed using `srpareto_optimize`, employing the STORS algorithm.

By default, `srpareto_custom()` samples from the standard Pareto distribution with `shape = 1` and `rate = 1`. The proposal distribution is pre-optimized at package load time using `srpareto_optimize()` with `steps = 4091`, creating a scalable proposal centred around the mode.

Value

A numeric vector of length `n` containing random samples from the Pareto distribution. The `shape` and `scale` parameters are specified during the optimization process using `srpareto_optimize()`.

NOTE: When the `x` parameter is specified, it is updated in-place with the simulation for performance reasons.

Note

This function is not scalable. Therefore, only the `srpareto_custom()` version is available, which requires the proposal to be pre-optimized using `srpareto_optimize()` before calling this function.

See Also

`srpareto_optimize` to optimize the custom proposal.

Examples

```
# Generate 10 samples from Pareto Distribution
samples <- srpareto_custom(10)
print(samples)

# Generate 10 samples using a pre-allocated vector
x <- numeric(10)
srpareto_custom(10, x = x)
print(x)
```

srpareto_optimize *Optimizing Pareto Distribution proposal*

Description

The srpareto_optimize() function generates an optimized proposal for a targeted Pareto Distribution. The proposal can be customized and adjusted based on various options provided by the user.

Usage

```
srpareto_optimize(
  scale = NULL,
  shape = NULL,
  xl = NULL,
  xr = NULL,
  steps = 4091,
  proposal_range = NULL,
  theta = 0.1,
  target_sample_size = 1000,
  verbose = FALSE
)
```

Arguments

scale	(optional) Numeric. scale parameter of the Pareto Distribution. Defaults to NULL, which implies a scalable proposal with scale = 1.
shape	(optional) Numeric. shape parameter of the Pareto Distribution. Defaults to NULL, which implies a scalable proposal with shape = 1.
xl	Numeric. Left truncation bound for the target distribution. Defaults to -Inf, representing no left truncation.
xr	Numeric. Right truncation bound for the target distribution. Defaults to Inf, representing no right truncation.
steps	(optional) Integer. Desired number of steps in the proposal. Defaults to NULL, which means the number of steps is determined automatically during optimization.
proposal_range	(optional) Numeric vector. Specifies the range for optimizing the steps part of the proposal. Defaults to NULL, indicating automatic range selection.
theta	Numeric. A parameter for proposal optimization. Defaults to 0.1.
target_sample_size	(optional) Integer. Target sample size for proposal optimization. Defaults to 1000.
verbose	Boolean. If TRUE, detailed optimization information, including areas and steps, will be displayed. Defaults to FALSE.

Details

When `srpareto_optimize()` is explicitly called:

- A proposal is created and cached. If no parameters are provided, a standard proposal is created with `rate = 1`.
- Providing `rate` creates a custom proposal, which is cached for use with `srpareto_custom()`.
- The optimization process can be controlled via parameters such as `steps`, `proposal_range`, or `theta`. If no parameters are provided, the proposal is optimized via brute force based on the. `target_sample_size`.

Value

The user does not need to store the returned value, because the package internally cashes the proposal. However, we explain here the full returned proposal for advanced users.

A list containing the optimized proposal and related parameters for the specified built-in distribution: #'

`data` Detailed information about the proposal steps, including `x`, `s_upper`, `p_a`, and `s_upper_lower`.

`areas` The areas under the left tail, steps, and right tail of the proposal distribution.

`steps_number` The number of steps in the proposal.

`f_params` The parameters (`scale` and `shape`) of the Beta distribution.

See Also

`srpareto_custom`: Function to sample from a custom proposal tailored to user specifications.

Examples

```
# Generate scalable proposal that with rate = 1, that has 4096 steps
scalable_proposal <- srpareto_optimize(steps = 4096)

# Generate custom proposal that with scale = 4
scalable_proposal <- srpareto_optimize(scale = 4)
```

Index