Package: stanli (via r-universe)

November 20, 2026

Type Package

Title The Stan Language Interpreter

Version 0.30.0

Description Compile and sample Stan models without a C++ toolchain. Stan models are lowered to a graph of operations over precompiled 'stan-math' kernels rather than translated to C++ and compiled, so preparing a model takes milliseconds instead of minutes. Provides adaptive Hamiltonian Monte Carlo with the diagnostics a Bayesian workflow needs (rank-normalized split R-hat, bulk and tail effective sample size, E-BFMI), and L-BFGS optimization. The runtime library is downloaded on first use rather than bundled; see stanli_install().

License BSD_3_clause + file LICENSE

URL https://github.com/seantalts/stanli

BugReports https://github.com/seantalts/stanli/issues

Encoding UTF-8

Depends R (>= 4.0)

Imports utils, tools, methods, stats

Suggests V8, jsonlite, posterior, bayesplot, loo, rstantools, tidybayes, callr, rstan, cmdstanr, shinystan, brms, nutpieR (>= 1.9.0), bridgesampling, coda, Matrix, testthat (>= 3.0.0)

Additional_repositories https://stan-dev.r-universe.dev, https://andytimm.r-universe.dev

NeedsCompilation yes

Config/roxygen2/version 8.1.0

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

Date/Publication 2026-11-20 11:28:35 UTC

RemoteUrl https://github.com/seantalts/stanli

RemoteRef v0.30.0

RemoteSha 9d3ce47d94d24636f3637d6aa26bda9ecf0f2236

RemoteSubdir r

Contents

as_cstanfit . . . 2
as_draws_array . . . 4
as_rfit . . . 4
as_stanfit . . . 6
as_stanmodel . . . 8
bridgestan_data . . . 9
cstan_model . . . 10
laplace_model . . . 11
laplace_whiten . . . 12
log_lik . . . 13
log_prob_grad . . . 14
log_prob_hessian . . . 14
optimize_model . . . 15
parameter_annotations . . . 15
parameter_names . . . 16
pathfinder_model . . . 16
sample_cstan . . . 18
sample_model . . . 19
sample_more . . . 22
sample_nutpie . . . 23
sample_walnuts . . . 25
stanli-ecosystem . . . 26
stanli_available . . . 28
stanli_build_id . . . 28
stanli_check_syntax . . . 29
stanli_diagnose . . . 29
stanli_exact_lp . . . 30
stanli_install . . . 30
stanli_model . . . 31
stanli_runtime_path . . . 32
stanli_thread_safe . . . 33
summary.stanli_fit . . . 33
unconstrain . . . 34
variational_model . . . 35
whiten . . . 36

Index **38**

`as_cstanfit` *Native CmdStanR-style fit interface*

Description

Wrap stored Stanli draws in a native object with CmdStanR-style methods. Neither cmdstanr nor rstan is needed. The original fit is unchanged.

Usage

```
as_cstanfit(x)

## S3 method for class 'stanli_cstanfit'
summary(object, variables = NULL, ...)

## S3 method for class 'stanli_cstanfit'
print(x, variables = NULL, n = 10L, ...)
```

Arguments

x	A ‘stanli_fit‘ or an existing ‘stanli_cstanfit‘.
object	A ‘stanli_cstanfit‘.
variables	Variables to summarize.
...	Arguments passed to ‘posterior::summarise_draws()‘.
n	Maximum number of summary rows to print (default 10).

Details

* ‘$draws(variables = NULL, inc_warmup = FALSE, format = "draws_array")‘ returns posterior draws, including ‘lp__‘ and excluding sampler diagnostics. Base names select all indexed elements. Supported formats are ‘draws_array‘, ‘draws_matrix‘, ‘draws_df‘, ‘draws_list‘, and ‘draws_rvars‘. * ‘$summary(variables = NULL, ...)‘ delegates to ‘posterior::summarise_draws()‘, including custom functions and formulas. It uses rank-normalized R-hat and bulk/tail ESS by default, as CmdStanR does. * ‘$sampler_diagnostics(inc_warmup = FALSE, format = "draws_array")‘ returns the six sampler diagnostics in stable alphabetical order; CmdStanR’s CSV order may differ. * ‘$metadata()‘ includes iteration counts **before** thinning, ‘thin‘, ‘save_warmup‘, variable names, chain IDs, and sampler settings, including ‘metric‘, ‘step_size_adaptation‘ (the step size each chain holds after warmup, ‘NA‘ when unavailable) and ‘adapt_engaged‘ (1 or 0). * ‘$inv_metric(matrix = TRUE)‘ returns the inverse metric each chain holds after warmup, as a list named by chain id. A ‘diag_e‘ metric is a diagonal matrix when ‘matrix = TRUE‘ and a vector otherwise; a ‘dense_e‘ metric is always a matrix; a ‘unit_e‘ metric is the identity. Fits made by an older Stanli version or runtime do not carry it and the method errors. * ‘$num_chains()‘ returns the number of chains. * ‘$time()‘ returns ‘total‘ and a ‘chains‘ data frame with ‘chain_id‘, ‘warmup‘, ‘sampling‘, and ‘total‘ seconds. Unavailable timings are ‘NA‘; overall wall time is unavailable, rather than inferred from chain timings. * ‘$loo(variables = "log_lik", r_eff = TRUE, moment_match = FALSE, ...)‘ computes LOO with optional relative efficiency. Moment matching is unsupported. * ‘$stanli_fit()‘ retrieves the original fit for native Stanli operations.

Draws and summaries require posterior; LOO also requires loo. Stored-draw methods work after serialization without a runtime or live model. The Stanli R package must remain installed. Requests for unsaved warmup fail explicitly. Model recompilation, CSV output, and CmdStan’s model methods are not provided. Consumers that dispatch on CmdStanR class names must explicitly accept this class or call the methods directly.

Value

A ‘stanli_cstanfit‘ with methods described below. This is a native list of functions, not an R6 CmdStanMCMC or RStan S4 object.

as_draws_array *Convert a fit to a draws array*

Description

Returns an object the posterior package understands: a ‘draws_array‘ with dimensions (iteration, chain, variable). Without posterior installed this is still a plain array with the same dimensions and dimnames, which is what most consumers actually read.

Usage

```
as_draws_array(x, include_sampler = FALSE, inc_warmup = FALSE)
```

Arguments

x	A ‘stanli_fit‘.
include_sampler	Append the seven sampler columns (‘lp__‘, ‘accept_stat__‘, ...), which is what a CSV reader would see.
inc_warmup	Include saved warmup draws. Defaults to ‘FALSE‘.

Value

A ‘draws_array‘, or a plain array when posterior is absent.

as_rfit *Native R fit interface*

Description

Give a Stanli fit an RStan-style summary and extraction interface without loading RStan. The arrays, live model, and ecosystem integrations are kept. Existing ‘stanli_fit‘ summary behavior is unchanged until explicitly converted.

Usage

```
as_rfit(x)

extract(
  object,
  pars = NULL,
  permuted = TRUE,
  inc_warmup = FALSE,
  include = TRUE,
  ...
```

```
)

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

get_sampler_params(object, inc_warmup = TRUE)

get_elapsed_time(object)
```

Arguments

x, object	A ‘stanli_fit‘ or ‘stanli_rfit‘.
pars	Parameter names or indexed element names. Defaults to all outputs, including ‘lp__‘. Base names select the complete vector or array.
permuted	Merge post-warmup chains using stored per-chain permutations.
inc_warmup	Include saved warmup. Ignored for permuted extraction.
include	Select ‘pars‘ when TRUE, or exclude them when FALSE.
...	Reserved; unsupported arguments produce an error.
probs	Quantile probabilities.

Details

‘extract()‘ returns a named list of draw-first arrays when permuted, or an iteration-by-chain-by-parameter array otherwise. ‘as_rfit()‘ creates per-chain permutations from the fit’s ‘seed‘, so permuted extraction is the same in every session; neither it nor extraction reads or changes R’s random stream. All parameters share the same permutations, preserving joint draws.

‘summary()‘ returns ‘summary‘ and ‘c_summary‘ matrices/arrays in RStan’s layout. ‘n_eff‘ and ‘Rhat‘ use posterior’s basic ESS and split R-hat; these are distinct from Stanli’s usual bulk ESS and rank-normalized R-hat. posterior’s ESS algorithm also differs from RStan’s legacy ESS estimator, so ‘n_eff‘ and ‘se_mean‘ are not numerically identical to RStan’s summary. Summary requires optional posterior but no native runtime. Sampler and timing extraction require neither posterior nor a native runtime.

This is a native S3 class, not an RStan S4 object. Calls explicitly written as ‘rstan::extract()‘ still require RStan. Use ‘stanli::extract()‘ instead. Packages inspecting RStan slots or requiring its C++ instance need an integration or the optional [as_stanfit()] conversion. The live model pointer does not survive serialization; stored-draw operations do.

Value

‘as_rfit()‘ returns a ‘stanli_rfit‘ inheriting from ‘stanli_fit‘. ‘get_sampler_params()‘ returns one matrix per chain, excluding ‘lp__‘. ‘get_elapsed_time()‘ returns a chain-by-phase matrix in seconds.

`as_stanfit` *Convert draws to an RStan fit without compilation*

Description

Build a ‘stanli_stanfit‘, an S4 subclass of ‘rstan::stanfit‘, directly from stored arrays. Requires optional rstan but never compiles a C++ model.

Usage

```
as_stanfit(x, ..., plain = FALSE)

## S3 method for class 'stanli_fit'
as_stanfit(x, model = x$model, exclude = NULL, ..., plain = FALSE)

## S3 method for class 'stanli_stanfit'
as_stanfit(x, model, ..., plain = FALSE)

## S3 method for class 'stanfit'
as_stanfit(x, model, ..., plain = FALSE)
```

Arguments

`x`	A fitted object. A ‘stanli_fit‘ must contain the sampling metadata recorded by the current [sample_model()]. A ‘stanli_stanfit‘ is returned unchanged unless a replacement ‘model‘ is supplied. A plain ‘stanfit‘, such as ‘rstan::sflist2stanfit()‘ returns from fits of the same model, needs a ‘model‘ and becomes a ‘stanli_stanfit‘ with it.
`...`	Reserved for methods.
`plain`	‘TRUE‘ returns an object whose class is exactly ‘stanfit‘, with the stored draws and no stanli model. Every ‘rstan‘ function that only reads draws works on it, and code that checks ‘class(x) == "stanfit"‘ or ‘identical(class(x), "stanfit")‘ accepts it. ‘log_prob()‘ and the other model methods do not. Attach a model again with ‘as_stanfit(plain_fit, model = model)‘.
`model`	A ‘stanli_model‘ made with the original code and data. Defaults to the model retained by a ‘stanli_fit‘. Use ‘NULL‘ for a draws-only result. A replacement must match the fit’s output columns and parameter count.
`exclude`	Names of variables to leave out of the stored draws, as ‘pars‘ with ‘include = FALSE‘ does in ‘rstan::sampling()‘. Every element of a named container is dropped. The live model is unaffected.

Details

Extraction, summaries, plots, and ordinary LOO inherit RStan’s methods. With a live model, ‘rstan::log_prob‘, ‘grad_log_prob‘, ‘constrain_pars‘, and ‘unconstrain_pars‘ call stanli directly.

‘get_num_upars‘ uses retained metadata and also works without a live model. LOO moment matching works with ‘cores = 1‘ and a live model.

If using rethinking, load rstan before rethinking in a fresh R session. Its placeholder ‘stanfit‘ class can otherwise conflict with RStan's class. Native [as_cstanfit()] and [as_rfit()] do not need this load order.

Density calls support ‘adjust_transform = TRUE‘ (the default). ‘adjust_transform = FALSE‘ is rejected because the graph includes the parameter-transform Jacobian. Native RStan resampling and operations requiring its C++ model instance remain unsupported. ‘constrain_pars‘ includes transformed parameters and generated quantities; RNG-based quantities advance the retained model's RNG stream.

‘saveRDS()‘ preserves draws, but external pointers do not survive loading. Recreate the model with the same code and data and call ‘as_stanfit(saved_fit, model = model)‘ to restore model-dependent methods. Serialized workers likewise cannot use live handles. No model recompiles automatically on loading or conversion.

Per-chain permutations are fixed by the fit's ‘seed‘ and do not read or change R's random stream, so permuted extraction gives the same draws in every session. Stored draw order is unchanged. Empty variables cannot be inferred from column names and are omitted, as in CSV import. Packages that insist on exact class equality instead of checking inheritance may need adaptation.

The converted model reports a model name to ‘brms‘, so ‘brms‘ treats the fit as holding a compiled model and does not recompile it with RStan. RStan itself still cannot sample from it.

Install rstan from a CRAN binary on supported macOS/Windows R versions to avoid a toolchain. On Linux, provision a compatible binary installation.

Value

An S4 object inheriting from ‘stanfit‘, with samples, sampler statistics, timings, and parameter dimensions. Initial values are empty: the first saved draw is not the initial state. Each chain's ‘adaptation_info‘ holds the step size and inverse metric after warmup, as RStan prints them, and is empty without adaptation.

A fit from [variational_model()], [pathfinder_model()] or [laplace_model()] becomes a one-chain fit without warmup or sampler statistics, whose draws end with ‘lp__‘ and ‘lp_approx__‘, and whose ‘stan_args[[1]]$method‘ is ‘"variational"‘, ‘"pathfinder"‘ or ‘"laplace"‘. This is the layout brms builds from CmdStan's output for the same algorithms; unlike [rstan::vb()], no row of means precedes the draws. A fixed-parameter fit keeps its chains and has ‘accept_stat__‘ as its only sampler statistic.

Examples

```
## Not run:
m <- stanli_model(code = "parameters { real mu; } model { mu ~ normal(0, 1); }")
fit <- sample_model(m, refresh = 0)
sf <- as_stanfit(fit)
rstan::extract(sf, pars = "mu")
rstan::log_prob(sf, upars = 0, gradient = TRUE)
rstan::constrain_pars(sf, upars = 0)

## End(Not run)
```

`as_stanmodel` *Run rstan-style code on stanli*

Description

Wraps Stan code as a ‘stanli_stanmodel‘, an S4 subclass of rstan’s ‘stanmodel‘. ‘rstan::sampling()‘ and ‘rstan::vb()‘ on it run stanli’s runtime instead of compiling C++, and return a ‘stanli_stanfit‘. The subclass is registered when rstan loads; stanli does not alter any rstan object.

Usage

```
as_stanmodel(x, model_name = NULL)
```

Arguments

`x`	Stan code as a single string, the path of a ‘.stan‘ file, or an rstan ‘stanmodel‘, of which only the Stan code is used. A ‘stanli_stanmodel‘ is returned as it is.
`model_name`	Name stored in the model and in its fits. Defaults to the file name without extension, the name of ‘x‘ when it is a ‘stanmodel‘, and ‘"anon_model"‘ for code.

Details

The code is checked for syntax and type errors here. The model is built at each ‘sampling()‘ or ‘vb()‘ call, when the data is known.

‘sampling()‘ takes rstan’s arguments. ‘iter‘, ‘warmup‘, ‘thin‘, ‘chains‘, ‘seed‘, ‘init‘, ‘pars‘, ‘include‘, ‘cores‘, ‘control‘ and ‘algorithm = "Fixed_param"‘ work as in rstan, and ‘chain_id‘, ‘init_r‘, ‘refresh‘ and ‘save_warmup‘ are accepted through ‘...‘. ‘control‘ takes ‘adapt_delta‘, ‘max_treedepth‘, ‘stepsize‘, ‘metric‘, ‘adapt_engaged‘ and, as a stanli addition, ‘adapt_metric‘ (see [sample_model()]); any other entry prints an error naming it, then ‘sampling()‘ messages "error in specifying arguments; sampling not done" and returns an empty fit, as rstan does. A model that cannot be built from the data, such as one whose transformed data rejects, is reported the same way with "failed to create the sampler; sampling not done". ‘algorithm = "HMC"‘, ‘sample_file‘ and ‘diagnostic_file‘ are errors. ‘init‘ may be ‘"random"‘, a radius, ‘"0"‘, a list for all chains, a list of lists with one per chain, or a function called as ‘f(chain_id = k)‘ when it has that argument and ‘f()‘ otherwise. Parameters a list leaves out start at random points within ‘init_r‘, and entries that are not parameters are ignored. A start whose log density or gradient is not finite is redrawn up to 100 times.

‘vb()‘ runs ADVI. ‘importance_resampling = TRUE‘, ‘keep_every‘ other than 1 and ‘sample_file‘ are errors. ‘optimizing()‘ is always an error: stanli’s log density includes the change-of-variables Jacobian, so it cannot return the mode rstan reports. Use [optimize_model()].

The caller’s random seed is advanced only by the default ‘seed‘ draw, as in rstan, and is restored afterwards.

Packages that ship precompiled models, such as rstanarm, RBesT, OncoBayes2 and prophet, do not call ‘sampling()‘ on a model you can replace, and are not supported.

Value

A ‘stanli_stanmodel‘.

Examples

```
## Not run:
code <- "
  data { int<lower=0> N; vector[N] y; }
  parameters { real mu; real<lower=0> sigma; }
  model { y ~ normal(mu, sigma); }"
m <- as_stanmodel(code)
fit <- rstan::sampling(m, data = list(N = 3L, y = c(1, 2, 3)))
print(fit)

## End(Not run)
```

`bridgestan_data` *A model as BridgeStan data*

Description

Stanli’s runtime exposes BridgeStan’s C interface, so a BridgeStan client can sample a stanli model without a C++ toolchain. The client loads [stanli_runtime_path()] as the model library and passes the string this function returns as the model’s data. The string is the model’s data object with one added key, ‘"__stanli"‘, holding the runtime build id, the compiled model and its name. The runtime refuses a payload whose build id is not its own.

Usage

```
bridgestan_data(model)
```

Arguments

`model`	A ‘stanli_model‘.

Value

A single JSON string.

See Also

[sample_nutpie()] for a client that is wired up already.

`cstan_model` *Native CmdStanR-style model interface*

Description

Retain Stan source in a native model object. Its `$sample()` method accepts the arguments of `sample_cstan()` (except `model_code`) and returns the same native fit. Model preparation happens when sampling, using the supplied data and seed, including RNG calls in transformed data.

Usage

```
cstan_model(model_code, include_paths = NULL)
```

Arguments

`model_code`	A single string of Stan program source.
`include_paths`	Directories searched for Stan `#include` files; see `stanli_model()`. Resolved when the model object is created; files are read when `$sample()` prepares the model.

Details

Defaults and validation are delegated to `sample_cstan()`. Neither cmdstanr, rstan, CmdStan, nor a C++ toolchain is needed. This object provides the sampling subset of the CmdStanR model interface, with its own class; it does not provide compilation methods, compiler options, or CSV output. Unsupported sampling arguments produce an error before model preparation.

Value

A `stanli_cstanmodel` with `$sample(...)`, `$code()`, and `$model_name()` methods. `$sample()` accepts `threads_per_chain` for native within-chain parallelism and `parallel_chains` for concurrent chains.

Examples

```
model <- cstan_model("parameters { real a; } model { a ~ normal(0, 1); }")
model$code()
if (stanli_available()) model$sample(chains = 1, iter_warmup = 20,
                                     iter_sampling = 20, seed = 1, refresh = 0)
```

`laplace_model` *Draws from a Laplace approximation*

Description

Draws from the normal distribution, on the unconstrained scale, centred at the posterior mode with the curvature of the log density there. This is Stan's own Laplace sampling, the algorithm behind CmdStan's 'laplace' method.

Usage

```
laplace_model(
  model,
  seed = 1,
  draws = 1000,
  mode = NULL,
  jacobian = TRUE,
  iter = 2000,
  init = NULL,
  init_radius = 2,
  calculate_lp = TRUE,
  refresh = 0
)
```

Arguments

`model`	A 'stanli_model'.
`seed`	Seed.
`draws`	Number of draws.
`mode`	The mode to expand around: the result of [optimize_model()], or a point on the unconstrained scale (see [unconstrain()]). 'NULL' runs [optimize_model()] with 'seed', 'iter', 'init' and 'init_radius'.
`jacobian`	Must be 'TRUE'. As for [optimize_model()], the change-of-variables Jacobian is part of the compiled model, so the mode and the curvature are those of the posterior on the unconstrained scale. CmdStan's 'laplace' has the same default.
`iter`, `init`, `init_radius`	Settings for the optimizer when 'mode' is 'NULL'.
`calculate_lp`	Evaluate the log density at each draw. 'FALSE' leaves 'lp__' as 'NaN'.
`refresh`	Any positive value prints Stan's progress every that many draws. 0 prints nothing.

Details

The curvature is Stan's finite-difference Hessian of the gradient. A point that is not a mode, or a mode with a direction of zero or positive curvature, has no normal approximation; Stan does not check for it and the draws are then meaningless or not finite.

Value

A ‘stanli_fit‘ with one chain of ‘draws‘ draws. Its ‘sampler‘ element holds ‘lp__‘, the log density at each draw, and ‘lp_approx__‘, the approximation’s unnormalized log density there. ‘mode‘ holds the unconstrained point that was expanded around.

`laplace_whiten` *Laplace-whiten a model (experimental)*

Description

Finds the Laplace-approximate marginal mode of the hyperparameters and whitens ‘model‘ around it. ‘latent‘ names the parameters that make up the random effects ‘u‘; every other parameter is a hyperparameter ‘theta‘. In the unconstrained vector the latent block is those parameters’ entries, so a simplex of K entries contributes K - 1. With ‘latent = NULL‘ the latent parameters are the ones declared ‘@latent‘ (see [parameter_annotations()]), and a model with none is an error. A ‘latent‘ argument wins when both are present.

Usage

```
laplace_whiten(
  model,
  latent = NULL,
  init = NULL,
  seed = 1,
  max_outer = 60L,
  inner_iter = 2000L,
  pattern_tolerance = 1e-06,
  threshold = 1e-06,
  repair = c("abs", "abs_dense", "clip"),
  clip = NULL,
  hessian = c("coloured", "dense"),
  pattern = c("probe", "dense")
)
```

Arguments

`model`	A ‘stanli_model‘.
`latent`	Names of the latent parameters, or ‘NULL‘.
`init`	An unconstrained starting point that replaces Pathfinder.
`seed`	Seed for Pathfinder and for the model.
`max_outer`, `inner_iter`	L-BFGS iteration limits for ‘theta‘ and ‘u‘.
`pattern_tolerance`, `threshold`	Relative thresholds for the Hessian pattern and for the entries of ‘Q‘.
`repair`, `clip`	How a ‘Q‘ that is not positive definite is repaired.

hessian	‘"coloured"‘ (default) or ‘"dense"‘, which calls [log_prob_hessian()] at the mode.
pattern	‘"probe"‘ (default) finds the pattern from a few dozen gradients and checks it against three random gradient differences, falling back to the dense Hessian with a warning if the check fails; ‘"dense"‘ calls [log_prob_hessian()] at the start point.

Details

The steps, with the defaults the Python builder and its benchmark use: start at the Pathfinder mean (or ‘init‘) and find the sparsity pattern of the Hessian there; maximize over ‘theta‘ the marginal ‘log p(theta, u_hat(theta)) - log det(-H_uu) / 2‘ by L-BFGS, with ‘H_uu‘ from central differences of the gradient along the pattern; at the result take ‘Q = -H‘, drop entries below ‘threshold‘ relative to ‘sqrt(Q_ii Q_jj)‘ and factor it with a sparse LDL’. If ‘Q‘ is not positive definite, ‘repair = "abs"‘ replaces each eigenvalue below ‘clip‘ (default 1e-8) times the largest absolute eigenvalue by its absolute value, floored at that level, as a low-rank correction to the sparse factor, and warns. ‘"abs_dense"‘ does it with a dense eigendecomposition and Cholesky, and ‘"clip"‘ raises eigenvalues below ‘clip‘ (default 1e-3) times the largest to that value; both are O(n^3).

Run the whitened model with ‘metric = "unit_e"‘ and ‘adapt_metric = FALSE‘ (step-size adaptation only); a short warmup is enough. Needs the Matrix package.

Value

A list with ‘model‘, the whitened model, and ‘report‘: counts of gradient evaluations by stage, ‘theta_hat‘, the fill of the factor (‘nnz_L‘), the repaired eigenvalue count (‘n_clipped‘) and ‘rank‘, and timings. ‘perm‘ and ‘latent_index‘ are 1-based.

---

log_lik *Extract pointwise log likelihoods*

Description

Declare the pointwise log likelihood in Stan’s generated quantities block, for example ‘vector[N] log_lik‘ with one log density per observation.

Usage

```
log_lik(object, ...)

## S3 method for class 'stanli_fit'
log_lik(object, variable = "log_lik", ...)
```

Arguments

object	A fitted model.
...	Arguments passed to methods.
variable	Name of the pointwise log-likelihood variable.

Value

A numeric array (iteration, chain, observation), in Stan column order, excluding saved warmup. A scalar variable retains a singleton third axis.

`log_prob_grad` *Log density and gradient*

Description

Log density and gradient

Usage

```
log_prob_grad(model, q)
```

Arguments

`model`	A ‘stanli_model‘.
`q`	A point on the unconstrained scale.

Value

A list with ‘lp‘ and ‘grad‘.

`log_prob_hessian` *Log density, gradient and Hessian*

Description

The density and gradient of [log_prob_grad()] plus the Hessian at ‘q‘. The Hessian is Stan’s finite-difference Hessian of the gradient, the routine behind its Laplace approximation, so it costs ‘1 + 2n‘ gradient evaluations for ‘n‘ unconstrained parameters and carries finite-difference error.

Usage

```
log_prob_hessian(model, q)
```

Arguments

`model`	A ‘stanli_model‘.
`q`	A point on the unconstrained scale.

Value

A list with ‘lp‘, ‘grad‘ and ‘hessian‘, an ‘n‘ x ‘n‘ symmetric matrix.

optimize_model *Find the posterior mode by L-BFGS*

Description

Returns the posterior MODE. CmdStan's ‘optimize‘ defaults to ‘jacobian=0‘, the penalized maximum likelihood, which stanli cannot offer: the change-of-variables Jacobian is folded into the graph when the model is lowered.

Usage

```
optimize_model(model, seed = 1, iter = 2000, init = NULL, init_radius = 2)
```

Arguments

model	A ‘stanli_model‘.
seed, iter	Optimizer configuration.
init	Optional start on the unconstrained scale; see [unconstrain()] to build one from constrained values.
init_radius	Random start radius.

Value

A list with ‘values‘ (named), ‘unconstrained‘, and ‘lp‘.

parameter_annotations *Annotations on a model's parameters*

Description

The declaration annotations on each parameter, without the ‘@‘: ‘@latent vector[J] a;‘ gives ‘list(a = "latent")‘. Only parameters with at least one annotation are listed, in source order. Transformed parameters and generated quantities are not listed; an annotation on one has no effect.

Usage

```
parameter_annotations(model)
```

Arguments

model	A ‘stanli_model‘.

Value

A named list of character vectors.

parameter_names *Names of a model's parameters*

Description

The variables declared in the ‘parameters‘ block, in declaration order. Transformed parameters and generated quantities are not included, so this is the set of names [unconstrain()] accepts.

Usage

```
parameter_names(model, scalars = FALSE)
```

Arguments

model	A ‘stanli_model‘.
scalars	‘FALSE‘ returns one name per declared variable (‘"sigma"‘). ‘TRUE‘ returns one name per scalar element in Stan's column order (‘"sigma[1]"‘, ‘"sigma[2]"‘, ‘"B[2,3]"‘), as ‘model$columns‘ and draw columns spell them.

Value

A character vector.

pathfinder_model *Pathfinder variational inference*

Description

Follows an L-BFGS path towards the posterior mode, builds a normal approximation at each point along it, keeps the one with the highest evidence lower bound, and draws from that. This is Stan's own single-path Pathfinder, what CmdStan's ‘pathfinder‘ method runs with ‘num_paths = 1‘.

Usage

```
pathfinder_model(
  model,
  seed = 1,
  num_paths = 1,
  draws = 1000,
  max_lbfgs_iters = 1000,
  num_elbo_draws = 25,
  history_size = 5,
  init_alpha = 0.001,
  tol_obj = 1e-12,
  tol_rel_obj = 10000,
```

```
  tol_grad = 1e-08,
  tol_rel_grad = 1e+07,
  tol_param = 1e-08,
  calculate_lp = TRUE,
  init = NULL,
  init_radius = 2,
  refresh = 0
)
```

Arguments

`model`	A ‘stanli_model‘.
`seed`	Seed.
`num_paths`	Must be 1. Multi-path Pathfinder, which runs several paths and resamples their draws by importance, is not available in this version; see Details.
`draws`	Number of draws to take from the approximation.
`max_lbfgs_iters`	Maximum L-BFGS iterations.
`num_elbo_draws`	Draws used to compare the approximations along the path.
`history_size`, `init_alpha`, `tol_obj`, `tol_rel_obj`, `tol_grad`, `tol_rel_grad`, `tol_param`	L-BFGS settings, as in CmdStan.
`calculate_lp`	Evaluate the log density at each draw. With ‘FALSE‘, ‘lp__‘ is ‘NaN‘ except for the first ‘num_elbo_draws‘ draws, which Stan has already evaluated.
`init`	Optional starting point on the unconstrained scale; see [unconstrain()].
`init_radius`	A random start is drawn uniform(-r, r).
`refresh`	Any positive value prints Stan’s progress every that many L-BFGS iterations. 0 prints nothing.

Details

Multi-path Pathfinder is held back because the multi-path service of Stan 2.40, the version stanli is built on, reads past the end of an array while it resamples. ‘num_paths‘ is kept as an argument so that code written for it says what it means; any value but 1 is an error. cmdstanr’s default is four paths, so its default results are not what this function returns.

A single path is one normal approximation, with no importance resampling and no Pareto k diagnostic. It can be far from the posterior and nothing in the fit says so: compare with [sample_model()] before relying on it. ‘sample_model(pathfinder_init =)‘ uses the same algorithm to start NUTS.

Value

A ‘stanli_fit‘ with one chain of ‘draws‘ draws. Its ‘sampler‘ element holds ‘lp__‘, ‘lp_approx__‘, and ‘path__‘, which is 1 on every draw.

`sample_cstan` *Sample with CmdStanR-style arguments*

Description

Prepare and sample a Stan model with Stanli, returning [as_cstanfit()]. This is a convenience interface; it does not install or call CmdStanR.

Usage

```
sample_cstan(
  model_code,
  data = list(),
  chains = 4,
  parallel_chains = chains,
  iter_warmup = 1000,
  iter_sampling = 1000,
  seed = NULL,
  init = NULL,
  adapt_delta = 0.8,
  max_treedepth = 10,
  thin = 1,
  save_warmup = FALSE,
  refresh = 100,
  init_radius = 2,
  pathfinder_init = NULL,
  threads_per_chain = 1,
  include_paths = NULL,
  metric = c("diag_e", "dense_e", "unit_e"),
  inv_metric = NULL,
  step_size = NULL,
  adapt_engaged = TRUE,
  adapt_metric = TRUE,
  ...
)
```

Arguments

`model_code`	Stan program source.
`data`	Named list of model data.
`chains`	Number of chains.
`parallel_chains`	Maximum concurrent chains.
`iter_warmup, iter_sampling`	Iterations per chain before thinning.
`seed`	Integer seed. If NULL, draw one from R's RNG.

`init` Constrained parameter list, one list per chain, a function returning a list (optionally accepting ‘chain_id‘), or NULL for random initialization. A list may leave parameters out; those start at random within ‘init_radius‘, as in CmdStan. Numeric radii and files are unsupported; use ‘init_radius‘ for random initialization. Extra non-parameter names (such as transformed parameters) are ignored, as in CmdStanR.

`adapt_delta` Target acceptance probability.

`max_treedepth` Maximum NUTS tree depth.

`thin` Retain every thin-th iteration, separately in warmup and sampling.

`save_warmup` Retain warmup draws.

`refresh` Progress interval; zero suppresses progress.

`init_radius` Radius for random initialization on the unconstrained scale.

`pathfinder_init`
Optional Pathfinder settings passed to [sample_model()].

`threads_per_chain`
Threads per chain for eligible native ‘reduce_sum‘ calls. The model is built for this count; see [stanli_model()]. Defaults to 1.

`include_paths` Directories searched for Stan ‘#include‘ files; see [stanli_model()].

`metric`, `inv_metric`, `step_size`, `adapt_engaged`, `adapt_metric`
Metric kind (‘"diag_e"‘, ‘"dense_e"‘ or ‘"unit_e"‘), starting inverse metric, initial step size, whether adaptation is engaged and whether it adapts the metric, passed to [sample_model()], which describes them. ‘inv_metric‘ may be one vector or matrix, or a list with one per chain.

`...` Reserved. Unsupported options produce an error.

Details

Model preparation uses the sampling seed, including RNG calls in transformed data. C++ compiler options are not supported. Existing Stanli models and unconstrained initial values can be used through [sample_model()] followed by [as_cstanfit()].

Value

A native ‘stanli_cstanfit‘.

`sample_model` *Sample a model with NUTS*

Description

Four chains by default, run in parallel: R-hat needs more than one chain, and a single-chain run cannot be checked for convergence at all. Chain ‘c‘ of the run uses CmdStan’s stream for ‘(seed, chain_id + c - 1)‘.

Usage

```
sample_model(
  model,
  chains = 4,
  seed = 1,
  warmup = 1000,
  samples = 1000,
  thin = 1,
  delta = 0.8,
  max_depth = 10,
  save_warmup = FALSE,
  init = NULL,
  init_radius = 2,
  pathfinder_init = NULL,
  parallel_chains = NULL,
  refresh = 100,
  threads_per_chain = NULL,
  fixed_param = FALSE,
  metric = c("diag_e", "dense_e", "unit_e"),
  inv_metric = NULL,
  step_size = NULL,
  adapt_engaged = TRUE,
  adapt_metric = TRUE,
  chain_id = 1L
)
```

Arguments

model	A ‘stanli_model‘.
chains, seed, warmup, samples, thin	Sampler configuration. ‘seed‘ is also the model-construction seed for this run when transformed data draws from it; see [stanli_model()].
delta	Target acceptance statistic.
max_depth	Maximum treedepth.
save_warmup	Keep the warmup draws.
init	Optional starting point on the UNCONSTRAINED scale: one vector shared by every chain, or a matrix with one row per chain. Start from constrained values by passing them through [unconstrain()] first – one scale here means one contract for what a start is.
init_radius	Random inits are drawn uniform(-r, r); 0 starts at the origin.
pathfinder_init	Optional named list enabling Pathfinder-generated starts. An empty list uses defaults; supported entries are ‘num_iterations‘ (1000), ‘num_elbo_draws‘ (25), ‘history_size‘ (5), and Pathfinder’s own ‘init_radius‘ (2). Uses ‘seed‘, returns one start per chain, and cannot be combined with ‘init‘. Single-path Pathfinder does not perform PSIS resampling.

`parallel_chains`
: Chains to run at once. Defaults to all of them.

`refresh`
: Print a progress update every ‘refresh‘ transitions within each phase, plus the first and last transition of the phase. Set to 0 to suppress all automatic sampling output.

`threads_per_chain`
: Deprecated. Set the thread count when building the model, with ‘stanli_model(threads_per_chain =)‘; sampling uses the count the model was built with. ‘NULL‘ (the default) or the model’s own count changes nothing. Any other count still works but gives a warning of class ‘deprecatedWarning‘, and the model is rebuilt with that count for this run. Reduction rounding and NUTS draws can differ across thread counts. Active chains may each use the model’s count of threads. ‘fit$model$reduce_sum_count‘ and ‘reduce_sum_fallbacks‘ report retained reductions and graph-lowering refusals (not opaque regions).

`fixed_param`
: ‘TRUE‘ runs Stan’s fixed-parameter sampler instead of NUTS: every chain keeps its parameters at its starting point and only generated quantities are drawn, which is how a model with no parameters, or one that simulates data at given values, is run. There is no warmup: ‘warmup‘, ‘delta‘, ‘max_depth‘, ‘save_warmup‘ and ‘parallel_chains‘ are not used, and the chains run one after another. The fit’s ‘sampler‘ element then holds only ‘lp__‘ and ‘accept_stat__‘, both zero, and its ‘report‘ only the time each chain took.

`metric`
: The metric: ‘"diag_e"‘ (the default), ‘"dense_e"‘ or ‘"unit_e"‘, as in CmdStan.

`inv_metric`
: Optional starting inverse metric. For ‘"diag_e"‘, a numeric vector with one value per unconstrained parameter. For ‘"dense_e"‘, a symmetric positive definite matrix with one row and column per unconstrained parameter. A list with one such element per chain gives each chain its own; a single vector or matrix is used for every chain. ‘NULL‘ starts from the identity. A ‘"unit_e"‘ metric takes none. Stan checks the values and its message is reported when they are not valid. With adaptation on, the metric is adapted during warmup from this start.

`step_size`
: Optional initial step size, a single positive number used by every chain. ‘NULL‘ uses 1, as CmdStan does. With adaptation on it is the starting point of the step size search and of dual averaging.

`adapt_engaged`
: ‘TRUE‘ (the default) adapts the step size and, unless ‘metric = "unit_e"‘, the metric during warmup. ‘FALSE‘ runs no adaptation: ‘warmup‘ iterations are ordinary transitions and ‘inv_metric‘ and ‘step_size‘ are used as given. With ‘warmup = 0‘ and ‘adapt_engaged = TRUE‘, stanli still runs Stan’s initial step size search, as Stan’s adaptive sampler does, so the step size in use differs from ‘step_size‘. To use a supplied step size exactly, set ‘adapt_engaged = FALSE‘, as with CmdStanR.

`adapt_metric`
: ‘TRUE‘ (the default) adapts the metric during warmup when ‘adapt_engaged‘ is ‘TRUE‘. ‘FALSE‘ adapts only the step size and keeps ‘inv_metric‘ (the identity when ‘NULL‘) as given. It has no effect with ‘metric = "unit_e"‘ or ‘adapt_engaged = FALSE‘.

`chain_id`
: Id of the first chain, a positive integer, default 1. Chain ‘c‘ of the run is labelled ‘chain_id + c - 1‘ and uses that id’s random stream, as in CmdStan. A run of chains 1 to 4 gives the same draws, chain for chain, as four one-chain runs

with ‘chain_id = 1:4‘ and the same ‘seed‘. With ‘pathfinder_init‘ the starts of all chains come from one Pathfinder run seeded by the first chain’s id, so that equality does not hold.

Details

Interrupting R (Ctrl-C, or the stop button in RStudio) stops every chain after its current transition and raises the usual interrupt; no fit is returned.

Value

An object of class ‘stanli_fit‘. Its ‘report‘ element contains per-chain warmup and sampling times plus exact divergence and maximum-treedepth counts. Sampling metadata (‘warmup‘, ‘samples‘, ‘thin‘, ‘save_warmup‘, ‘chains‘, and ‘delta‘) records the configuration for downstream conversion; iteration counts are before thinning.

‘metric‘ is the metric used. ‘step_size‘ is a numeric vector with the step size each chain holds after warmup (the value CmdStan writes as "Step size"). ‘inv_metric‘ is a list with one element per chain holding the inverse metric after warmup: a numeric vector for ‘"diag_e"‘, an n x n matrix for ‘"dense_e"‘ and ‘NULL‘ for ‘"unit_e"‘. Passing ‘inv_metric = fit$inv_metric‘, ‘step_size = mean(fit$step_size)‘ and ‘adapt_engaged = FALSE‘ to a later call reuses an adapted metric and step size. ‘initial_step_size‘ is the step size the chains started from: ‘step_size‘ as passed, or 1.

---

sample_more *Continue sampling from a fit*

Description

Runs ‘samples‘ more transitions of each chain of a [sample_model()] fit, starting where the chains stopped with their own step size, inverse metric and random stream, and no warmup or adaptation. The result is the fit one call with the summed ‘samples‘ would have returned: the same draws, sampler columns and generated quantities, bit for bit.

Usage

```
sample_more(
  fit,
  samples = 1000,
  refresh = 100,
  parallel_chains = NULL,
  threads_per_chain = NULL,
  ...
)
```

Arguments

`fit`	A ‘stanli_fit‘ from [sample_model()].
`samples`	Further transitions per chain, before thinning.
`refresh`	Print a progress update every ‘refresh‘ transitions; the numbering continues from the fit’s transitions. 0 is silent.
`parallel_chains`	Chains to run at once. Defaults to all of them.
`threads_per_chain`	Deprecated, as in [sample_model()]: the run uses the thread count of the fit’s model. Another count gives a warning of class ‘deprecatedWarning‘ and rebuilds the model with that count.
`...`	Not used; any other sampler setting is an error.

Details

The other settings (seed, thinning, ‘delta‘, ‘max_depth‘, the metric) come from the fit, and the model is the one the fit carries, ‘fit$model‘. A model with different code or data is refused.

Interrupting R stops every chain after its current transition and raises the usual interrupt; no fit is returned.

Value

A new ‘stanli_fit‘ with the new draws, sampler columns and unconstrained draws appended to the old ones in each chain, after any saved warmup. ‘samples‘ is the total, the ‘report‘ sums sampling times and problem counts, and ‘seed‘, ‘warmup‘, ‘step_size‘ and ‘inv_metric‘ are those of ‘fit‘, which is left unchanged.

`sample_nutpie` *Sample with nutpie*

Description

Runs the nutpie sampler (through the nutpieR package) on a stanli model. nutpie drives the stanli runtime over its BridgeStan interface, so the log density and gradient are stanli’s. [sample_model()] stays the default sampler; this is the alternative to reach for when nutpie’s adaptation suits a model better, for instance the low-rank mass matrix on a posterior with strong correlations.

Usage

```
sample_nutpie(
  model,
  chains = 4,
  seed = 1,
  warmup = NULL,
  samples = 1000,
```

```
  adaptation = c("diag", "low_rank"),
  save_warmup = FALSE,
  ...
)
```

Arguments

`model`	A ‘stanli_model‘.
`chains`, `seed`, `samples`	Number of chains, seed, and post-warmup draws per chain.
`warmup`	Warmup draws per chain. ‘NULL‘ uses nutpie’s default.
`adaptation`	‘"diag"‘ for a diagonal mass matrix or ‘"low_rank"‘ for nutpie’s low-rank-plus-diagonal one.
`save_warmup`	Keep the warmup draws, placed before the post-warmup draws as in [sample_model()].
`...`	Further arguments for ‘nutpieR::nutpie_sample()‘, such as ‘max_treedepth‘, ‘target_accept‘ or ‘progress‘. Progress output is off unless ‘progress‘ is given.

Details

nutpieR 1.9.0 or later is needed and is not installed with stanli. Install it with ‘install.packages("nutpieR", repos = c("https://andytimm.r-universe.dev", "https://cloud.r-project.org"))‘.

The result is a ‘stanli_fit‘, so [summary.stanli_fit()], [stanli_diagnose()], [as_draws_array()], [log_lik()] and the conversions to other fit classes work on it. Three things differ from a fit made by [sample_model()]. ‘metric‘ and ‘inv_metric‘ are ‘NULL‘, because nutpie’s mass matrix is not a Stan metric that another run can start from, and ‘as_cstanfit(fit)$inv_metric()‘ says so. ‘report$warmup_seconds‘ is ‘NA‘, and ‘report$sampling_seconds‘ is the wall time of the whole run, warmup included, since nutpie times the two together. ‘step_size‘ holds the adapted step size of each chain, before nutpie’s per-draw jitter.

Transformed data that draws random numbers uses ‘seed‘, as it does in [sample_model()], and the fit’s ‘model‘ is the model built under it.

Value

An object of class ‘stanli_fit‘ with ‘sampler_backend‘ set to ‘"nutpie"‘ and ‘adaptation‘ recording the choice.

See Also

[bridgestan_data()]

`sample_walnuts` *Sample a model with WALNUTS*

Description

Runs WALNUTS, the within-orbit adaptive step-length No-U-Turn sampler (arXiv:2506.18746), through the walnutpie implementation built into the runtime. NUTS uses one step size for a whole trajectory. WALNUTS halves the step inside a trajectory where the energy error grows too large, which helps on funnels and other posteriors whose curvature changes from place to place. [sample_model()] stays the default sampler.

Usage

```
sample_walnuts(
  model,
  chains = 4,
  seed = 1,
  warmup = 1000,
  samples = 1000,
  max_error = 0.5,
  init = NULL,
  init_radius = 2,
  chain_id = 1L,
  refresh = 100,
  ...
)
```

Arguments

`model`	A ‘stanli_model‘.
`chains`, `seed`, `warmup`, `samples`	Number of chains, seed, warmup iterations and post-warmup draws per chain.
`max_error`	The largest change in the Hamiltonian allowed across one macro step before WALNUTS halves the step. Smaller values take smaller steps where the posterior is hard.
`init`	Optional starting point on the unconstrained scale: one vector for every chain, or a matrix with one row per chain, as in [sample_model()]. See [unconstrain()].
`init_radius`	Random starts are drawn uniform(-r, r); 0 starts at the origin.
`chain_id`	Id of the first chain, a positive integer, as in [sample_model()].
`refresh`	Print a progress update every ‘refresh‘ iterations within each phase, plus the first and last iteration of the phase. Set to 0 for no output.
`...`	Not used. Options that WALNUTS lacks, such as ‘thin‘ or ‘delta‘, are an error.

Details

The chains run one after another on the calling thread, because the runtime samples one WALNUTS chain at a time on one model. Each chain uses the model's ‘threads_per_chain‘.

Chain ‘c‘ uses the random stream for ‘(seed, chain_id + c - 1)‘, as in [sample_model()]. A run of chains 1 to 4 gives the same draws, chain for chain, as four one-chain runs with ‘chain_id = 1:4‘ and the same ‘seed‘. Without ‘init‘, a chain starts where the NUTS chain with the same ‘seed‘ and chain id starts. Generated quantities draw from the stream that [sample_model()] would give that seed and chain id, but they do not match its draws, because NUTS shares that stream with the sampler.

The result is a ‘stanli_fit‘, so [summary.stanli_fit()], [as_draws_array()], [log_lik()], [as_stanfit()], [as_rfit()] and [as_cstanfit()] work on it. The runtime reports less about a WALNUTS run than about a NUTS run: * ‘fit$sampler‘ holds ‘lp__‘, the log density at each draw, the same quantity as in [sample_model()], and ‘stepsize__‘, the macro step size that warmup adapted. WALNUTS halves that step within a trajectory, so a transition can use smaller steps. There are no divergences, tree depths, leapfrog counts, energies or acceptance statistics. * The adapted metric is not reported, so ‘fit$inv_metric‘ is ‘NULL‘ and ‘as_cstanfit(fit)$inv_metric()‘ is an error. * The maximum tree depth is 10 and a macro step can be halved at most 5 times. Neither can be changed. * There is no thinning and no saved warmup. * ‘report$warmup_seconds‘ is ‘NA‘, and ‘report$sampling_seconds‘ is the wall time of each chain, warmup included. * [sample_more()], [stanli_diagnose()], ‘nuts_params()‘ and [get_sampler_params()] need a NUTS fit and refuse this one.

Transformed data that draws random numbers uses ‘seed‘, as it does in [sample_model()], and the fit's ‘model‘ is the model built under it.

Interrupting R (Ctrl-C, or the stop button in RStudio) stops the run when the current chain finishes, because the runtime cannot stop a WALNUTS chain partway. It then raises the usual interrupt and returns no fit.

Value

An object of class ‘stanli_fit‘ with ‘algorithm‘ and ‘sampler_backend‘ set to ‘"walnuts"‘, ‘max_error‘ recording the setting and ‘step_size‘ the adapted macro step size of each chain.

`stanli-ecosystem` *R workflow integrations*

Description

Optional methods for bayesplot, loo, tidybayes, bridgesampling, and shinystan. Install the relevant package to use its generic; no plotting package is needed to load stanli. Saved warmup is excluded by default.

Usage

```
nuts_params.stanli_fit(object, pars = NULL, inc_warmup = FALSE, ...)

log_posterior.stanli_fit(object, inc_warmup = FALSE, ...)

rhat.stanli_fit(object, pars = NULL, ...)

neff_ratio.stanli_fit(object, pars = NULL, ...)

loo.stanli_fit(x, ..., variable = "log_lik", r_eff = NULL)

loo_compare.stanli_fit(x, ...)

tidy_draws.stanli_fit(model, ...)

bridge_sampler.stanli_fit(
  samples,
  repetitions = 1,
  method = "normal",
  cores = 1,
  use_neff = TRUE,
  maxiter = 1000,
  silent = FALSE,
  verbose = FALSE,
  ...
)

launch_shinystan.stanli_fit(object, ...)
```

Arguments

`object`, `x`, `model`	A 'stanli_fit'.
`pars`	Optional variable names (base names select all indexed elements). For 'nuts_params', exact sampler names such as 'divergent__'.
`inc_warmup`	Include saved warmup in sampler tables.
`...`	Further arguments passed to the underlying package where applicable.
`variable`	Pointwise log-likelihood variable passed to [log_lik()].
`r_eff`	Optional relative effective sample sizes. By default computed from the likelihood with 'loo::relative_eff()' using the chain axis.
`samples`	A 'stanli_fit' for 'bridge_sampler'.
`repetitions`, `method`, `cores`, `use_neff`, `maxiter`, `silent`, `verbose`	Passed to 'bridgesampling::bridge_sampler()'.

Details

R-hat and effective sample size use [summary.stanli_fit()]. The ESS ratio is bulk ESS divided by the number of retained post-warmup draws. Bayesplot's CmdStanMCMC method instead uses basic

ESS.

‘launch_shinystan‘ and ‘shinystan::as.shinystan()‘ convert the fit with [as_stanfit()] first, so they need rstan as well as shinystan.

‘bridge_sampler‘ estimates the log marginal likelihood from the unconstrained post-warmup draws. Write the model with ‘target +=‘ statements: ‘~‘ drops normalizing constants, as in rstan, and the estimate is then wrong. It needs the fit’s live model, so a fit restored with ‘readRDS()‘ must be sampled again from a new [stanli_model()]. On Windows, ‘cores‘ is set to 1.

Value

‘nuts_params‘ returns a data frame with ‘Chain‘, ‘Iteration‘, ‘Parameter‘ (a factor), and ‘Value‘; ‘log_posterior‘ omits ‘Parameter‘. ‘rhat‘ and ‘neff_ratio‘ return named numeric vectors. ‘tidy_draws‘ returns a tibble with draw indices. ‘loo‘ returns a ‘psis_loo‘ object and ‘loo_compare‘ compares fits via their LOO results. ‘bridge_sampler‘ returns a ‘bridge‘ object.

---

stanli_available *Is the stanli runtime available?*

Description

Is the stanli runtime available?

Usage

```
stanli_available()
```

Value

‘TRUE‘ when the runtime library is present and loadable.

---

stanli_build_id *Build id of the stanli runtime*

Description

The string a BridgeStan client’s model payload must carry for the runtime to accept it; see [bridgestan_data()].

Usage

```
stanli_build_id()
```

Value

A single string.

stanli_check_syntax *Check Stan code for syntax and type errors*

Description

Runs the Stan compiler front end on the code and nothing else: no data is needed and no model is built.

Usage

```
stanli_check_syntax(code, include_paths = NULL)
```

Arguments

code	Stan model source.
include_paths	Directories searched for '#include' files, as in [stanli_model()].

Value

'TRUE', invisibly. An invalid program is an error carrying the compiler's message.

stanli_diagnose *Convergence diagnostics*

Description

Divergent transitions, treedepth saturation, E-BFMI, R-hat and bulk/tail effective sample size – each either confirmed or reported with the number that failed and what to do about it.

Usage

```
stanli_diagnose(fit)
```

Arguments

fit	A 'stanli_fit' from NUTS. The other algorithms have no chains to diagnose.

Value

The report, invisibly, after printing it.

stanli_exact_lp *Does this build report CmdStan's 'lp__' exactly?*

Description

Does this build report CmdStan's 'lp__' exactly?

Usage

```
stanli_exact_lp()
```

Value

Always 'TRUE': every build reproduces CmdStan's 'lp__' exactly.

stanli_install *Download the stanli runtime*

Description

Fetches the prebuilt runtime for this platform into the user cache directory. This is a one-time step; the library is about 16 MB.

Usage

```
stanli_install(
  version = stanli_runtime_release,
  quiet = FALSE,
  overwrite = FALSE
)
```

Arguments

version	Release tag to fetch. Defaults to the release this version of the package was built against, which is the pairing its ABI check will accept. Only that release's runtime is found by [stanli_runtime_path()]; to run a different build, set 'STANLI_RUNTIME'.
quiet	Passed to [utils::download.file()].
overwrite	Re-download even if the runtime is already present.

Value

The path it was installed to, invisibly.

`stanli_model` *Compile a Stan model*

Description

Compile a Stan model

Usage

```
stanli_model(
  file = NULL,
  code = NULL,
  data = NULL,
  mir = NULL,
  seed = 1,
  threads_per_chain = 1,
  include_paths = NULL,
  fast_math = FALSE,
  allow_interpreter = TRUE
)
```

Arguments

`file`	Path to a '.stan' file.
`code`	Model source, as an alternative to 'file'.
`data`	A named list of data, or a path to a JSON data file. An R array of any rank is written with its first index outermost, so a matrix is an array of rows; an unnamed list is an array of its elements, which is how to pass 'array[K] matrix[N, M]', 'array[K] vector[N]' and arrays of them. Zero-length extents are kept.
`mir`	Transformed MIR text, for a build without the embedded compiler. Rarely needed.
`seed`	The model-construction seed: RNG calls in transformed data draw from it once, here, the way CmdStan's generated constructor does. [sample_model()] and [optimize_model()] forward their own 'seed' here when transformed data drew from it, rebuilding the model for that run, so one seed governs a whole run as it does in CmdStan. The rebuilt model is the one the fit carries as 'fit$model'; the object passed in is unchanged. A rebuilt model keeps the 'threads_per_chain' it was built with.
`threads_per_chain`	Positive thread count for eligible native 'reduce_sum' calls. Defaults to 1. The model is compiled for this count, and [sample_model()], optimization, the approximate methods and direct gradients all use it. To run with another count, build a new model with 'stanli_model()'. Requires a thread-safe runtime for values above 1.

include_paths Character vector of directories searched in order for Stan ‘#include‘ files, followed by the directory containing ‘file‘ (or the current directory for ‘code‘). Nested includes use the same search path. Included source is compiled once and retained when sampling rebuilds the model, even if the files later change or move. Ignored when ‘mir‘ is supplied.

fast_math ‘TRUE‘ opts this model into fast mode: the Stan source is compiled with fused multiply-add rewriting and the runtime may use faster, less exactly rounded evaluation, so results can differ from the default in the last bits. Defaults to ‘FALSE‘. With ‘mir‘, use MIR compiled in fast mode.

allow_interpreter ‘FALSE‘ makes it an error, naming the parts, when part of the model has no compiled path and would run through the much slower MIR interpreter. Defaults to ‘TRUE‘.

Value

An object of class ‘stanli_model‘ whose ‘columns‘ name every output the way the posterior package reads them, ‘theta[1,2]‘ for an indexed value. Warns, naming the part and the reason, when part of the model has no compiled path and runs through the much slower MIR interpreter; with ‘allow_interpreter = FALSE‘ (or the environment variable ‘STANLI_NO_INTERPRETER‘ set) that is an error instead. Either message is what to include in a bug report. When the compiler skips O1 for a very large model, the warning has class ‘stanli_o1_fallback‘.

---

stanli_runtime_path *Where the stanli runtime lives*

Description

The path is taken from ‘STANLI_RUNTIME‘ when that is set (which is how a development build is used), then from a runtime bundled into the package (wasm builds bundle one at build time, since a browser cannot download it), and from the user cache directory otherwise. The cache path includes the release the package pins, so a runtime downloaded by an older version of the package is never picked up by a newer one.

Usage

```
stanli_runtime_path()
```

Value

A file path, which may not exist yet.

stanli_thread_safe *Can this build run chains in parallel threads?*

Description

Can this build run chains in parallel threads?

Usage

```
stanli_thread_safe()
```

Value

'TRUE' when the runtime was built with thread support.

summary.stanli_fit *Posterior summary*

Description

Mean, MCSE, sd, quantiles, bulk and tail effective sample size, and rank-normalized split R-hat – stan's own estimators, so the numbers agree with 'stansummary' rather than approximating it.

Usage

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

Arguments

object	A 'stanli_fit'.
...	Unused.

Value

A data frame, one row per column.

`unconstrain` *Starting values on the constrained scale, as the free vector*

Description

Unless ‘partial = TRUE‘, every declared parameter must appear, at its declared size. A missing, unknown, wrong-length, or out-of-support value is an error naming the parameter. Containers are listed in Stan’s own serialization order (the first index fastest, the order a CSV column carries) and may be nested or flat – the declaration owns the shape either way.

Usage

```
unconstrain(
  model,
  values,
  partial = FALSE,
  init_radius = 2,
  seed = NULL,
  ignore_unknown = FALSE
)
```

Arguments

`model`	A ‘stanli_model‘.
`values`	A named list of constrained starting values, or a JSON string in CmdStan’s data format.
`partial`	‘TRUE‘ allows ‘values‘ to leave parameters out. Each one left out starts where Stan would start it: at the constrained image of a point drawn uniformly in ‘(-init_radius, init_radius)‘ on the unconstrained scale. Only for a list, not a JSON string.
`init_radius`	Radius of that draw.
`seed`	Seed for that draw. ‘NULL‘ uses, and advances, R’s random stream; a number leaves the stream untouched.
`ignore_unknown`	‘TRUE‘ drops the entries of a ‘values‘ list whose names are not declared parameters, as RStan does with transformed parameters, generated quantities and other extra entries, instead of refusing them. See [parameter_names()]. Only for a list, not a JSON string.

Details

The result is what ‘sample_model(init =)‘ and ‘optimize_model(init =)‘ take, so unconstraining is a step per starting point rather than a second kind of argument.

Value

A numeric vector of length ‘model$n_unconstrained‘.

`variational_model` *Variational inference with ADVI*

Description

Fits a normal approximation on the unconstrained scale by stochastic gradient ascent on the evidence lower bound, then draws from it. This is Stan's own ADVI, the algorithm behind CmdStan's 'variational' method and 'rstan::vb()', with their defaults.

Usage

```
variational_model(
  model,
  algorithm = c("meanfield", "fullrank"),
  seed = 1,
  iter = 10000,
  grad_samples = 1,
  elbo_samples = 100,
  eta = 1,
  adapt_engaged = TRUE,
  adapt_iter = 50,
  tol_rel_obj = 0.01,
  eval_elbo = 100,
  draws = 1000,
  init = NULL,
  init_radius = 2,
  refresh = 100
)
```

Arguments

`model`	A 'stanli_model'.
`algorithm`	'"meanfield"' for independent normals, '"fullrank"' for a normal with a full covariance matrix.
`seed`	Seed. Also the model-construction seed for this run when transformed data draws from it; see [stanli_model()].
`iter`	Maximum number of iterations.
`grad_samples, elbo_samples`	Monte Carlo draws for each gradient and for each evaluation of the lower bound.
`eta`	Step size. Used as given when 'adapt_engaged' is 'FALSE'.
`adapt_engaged`	Choose the step size by a short search first.
`adapt_iter`	Iterations for each step size the search tries.
`tol_rel_obj`	Relative change in the lower bound below which the fit is taken as converged.
`eval_elbo`	Evaluate the lower bound every this many iterations.

draws	Number of draws to take from the approximation.
init	Optional starting point on the unconstrained scale; see [unconstrain()].
init_radius	Random starts are drawn uniform(-r, r); 0 starts at the origin.
refresh	Any positive value prints Stan's progress, including the table of lower-bound values. 0 prints nothing.

Details

The approximation is not checked against the posterior. ADVI can stop early at a poor optimum, and a mean-field fit ignores correlation by construction, so compare with [sample_model()] before relying on the spread of the draws. Reaching ‘iter‘ without converging is reported as a warning and the fit reached so far is returned.

Stan measures the lower bound with every constant of the log density included. stanli evaluates the density ‘lp__‘ reports, which leaves out the constants of ‘~‘ statements, so for a model written with them the lower bound printed here differs from CmdStan's by a constant. The gradients and the random stream are the same, but the stopping rule is a relative change of the bound, so the fit can stop on a different iteration than CmdStan's with the same seed. A model that uses ‘target +=‘ throughout, as brms models do, is unaffected.

Value

A ‘stanli_fit‘ with one chain of ‘draws‘ draws. Its ‘sampler‘ element holds ‘lp__‘, the log density at each draw, and ‘lp_approx__‘, the approximation's unnormalized log density there. ‘mean‘ holds every column at the mean of the approximation, and ‘report$sampling_seconds‘ the wall time of the run.

---

whiten *Whiten a model around a point (experimental)*

Description

A new model over coordinates ‘z‘ with ‘q = q_hat + P' L^-T z‘, where ‘q‘ is the unconstrained vector of ‘model‘, ‘(P v)[i] = v[perm[i]]‘, and ‘L‘ is a lower triangular factor with ‘P Q P' = L L'‘. ‘Q‘ is usually the negative Hessian of the log density at ‘q_hat‘, so ‘z‘ is close to standard normal. The log density is that of ‘model‘ at ‘q(z)‘ with the constant ‘log|det L|‘ dropped. Every algorithm runs on ‘z‘ unchanged; sample with ‘metric = "unit_e"‘. Columns, constrained names and output rows are those of ‘model‘. [unconstrain()] returns ‘z‘, and unconstrained inits, modes and Hessians are in ‘z‘. The result does not depend on ‘model‘ afterwards, and it survives the rebuild a run with a new seed does.

Usage

```
whiten(model, q_hat, perm, L, lowrank = NULL)
```

Arguments

`model`	A ‘stanli_model‘.
`q_hat`	The point to whiten around, on the unconstrained scale.
`perm`	A permutation of ‘1:n‘.
`L`	A lower triangular factor with a nonzero diagonal: a ‘Matrix‘ (the Matrix package must be installed), or a list with the 0-based CSC arrays ‘p‘, ‘i‘ and ‘x‘.
`lowrank`	‘NULL‘, or ‘list(c, b, w)‘ with ‘c‘ of length ‘n‘ and ‘b‘ and ‘w‘ ‘n‘ by ‘k‘ matrices. It inserts the factor ‘diag(c) + tril(w b’, -1)‘, so that with ‘L‘ unit lower triangular and ‘C C’ = D + W W’‘ the precision is ‘P’ L (D + W W’) L’ P‘. See [laplace_whiten()].

Value

A ‘stanli_model‘.

Index

as_cstanfit, 2
as_draws_array, 4
as_rfit, 4
as_stanfit, 6
as_stanmodel, 8

bridge_sampler.stanli_fit
    (stanli-ecosystem), 26
bridgestan_data, 9

cstan_model, 10

extract *(as_rfit)*, 4

get_elapsed_time *(as_rfit)*, 4
get_sampler_params *(as_rfit)*, 4

laplace_model, 11
laplace_whiten, 12
launch_shinystan.stanli_fit
    (stanli-ecosystem), 26
log_lik, 13
log_posterior.stanli_fit
    (stanli-ecosystem), 26
log_prob_grad, 14
log_prob_hessian, 14
loo.stanli_fit *(stanli-ecosystem)*, 26
loo_compare.stanli_fit
    (stanli-ecosystem), 26

neff_ratio.stanli_fit
    (stanli-ecosystem), 26
nuts_params.stanli_fit
    (stanli-ecosystem), 26

optimize_model, 15

parameter_annotations, 15
parameter_names, 16
pathfinder_model, 16
print.stanli_cstanfit *(as_cstanfit)*, 2

rhat.stanli_fit *(stanli-ecosystem)*, 26

sample_cstan, 18
sample_cstan(), *10*
sample_model, 19
sample_more, 22
sample_nutpie, 23
sample_walnuts, 25
stanli-ecosystem, 26
stanli_available, 28
stanli_build_id, 28
stanli_check_syntax, 29
stanli_diagnose, 29
stanli_exact_lp, 30
stanli_install, 30
stanli_model, 31
stanli_model(), *10*
stanli_runtime_path, 32
stanli_thread_safe, 33
summary.stanli_cstanfit *(as_cstanfit)*, 2
summary.stanli_fit, 33
summary.stanli_rfit *(as_rfit)*, 4

tidy_draws.stanli_fit
    (stanli-ecosystem), 26

unconstrain, 34

variational_model, 35

whiten, 36