Package: rstan (via r-universe)

November 1, 2024

Type Package

Title R Interface to Stan

Version 2.32.6

Description User-facing R functions are provided to parse, compile,
test, estimate, and analyze Stan models by accessing the
header-only Stan library provided by the 'StanHeaders' package.
The Stan project develops a probabilistic programming language
that implements full Bayesian statistical inference via Markov
Chain Monte Carlo, rough Bayesian inference via 'variational'
approximation, and (optionally penalized) maximum likelihood
estimation via optimization. In all three cases, automatic
differentiation is used to quickly and accurately evaluate
gradients without burdening the user with the need to derive
the partial derivatives.

License GPL (>= 3)

NeedsCompilation yes

Imports methods, stats4, inline (>= 0.3.19), gridExtra (>= 2.3), Rcpp
(>= 1.0.7), RcppParallel (>= 5.1.4), loo (>= 2.4.1), pkgbuild
(>= 1.2.0), QuickJSR, ggplot2 (>= 3.3.5)

Depends R (>= 3.4.0), StanHeaders (>= 2.32.0)

LinkingTo Rcpp (>= 1.0.7), RcppEigen (>= 0.3.4.0.0), BH (>= 1.75.0-0),
StanHeaders (>= 2.32.0), RcppParallel (>= 5.1.4)

Suggests testthat (>= 3.0.4), parallel, KernSmooth, shinystan,
bayesplot, rmarkdown, rstantools, rstudioapi, Matrix, knitr,
coda, V8

URL https://mc-stan.org/rstan/, https://discourse.mc-stan.org

BugReports https://github.com/stan-dev/rstan/issues/

VignetteBuilder knitr

SystemRequirements GNU make, pandoc

Encoding UTF-8

RoxygenNote 7.1.2

Config/testthat/edition 3

Author Jiqiang Guo [aut], Jonah Gabry [aut], Ben Goodrich [cre, aut],
Andrew Johnson [aut], Sebastian Weber [aut], Hamada S. Badr
[aut] (<https://orcid.org/0000-0002-9808-2344>), Daniel Lee
[ctb], Krzysztof Sakrejda [ctb], Modrak Martin [ctb], Trustees
of Columbia University [cph], Oleg Sklyar [cph]
(R/cxxfunplus.R), The R Core Team [cph] (R/pairs.R,
R/dynGet.R), Jens Oehlschlaegel-Akiyoshi [cph] (R/pairs.R),
John Maddock [cph] (gamma.hpp), Paul Bristow [cph] (gamma.hpp),
Nikhar Agrawal [cph] (gamma.hpp), Christopher Kormanyos [cph]
(gamma.hpp), Bronder Steve [ctb]

Maintainer Ben Goodrich <benjamin.goodrich@columbia.edu>

Contents

---

rstan-package *RStan — the* R *interface to Stan*

Description

RStan is the R interface to the Stan C++ package. The RStan interface (**rstan** R package) provides:

- Full Bayesian inference using the No-U-Turn sampler (NUTS), a variant of Hamiltonian Monte Carlo (HMC)

- Approximate Bayesian inference using automatic differentiation variational inference (ADVI)

- Penalized maximum likelihood estimation using L-BFGS optimization

For documentation on Stan itself, including the manual and user guide for the modeling language, case studies and worked examples, and other tutorial information visit the Users section of the Stan website:

- mc-stan.org/users/documentation

Other R packages from the Stan Development Team

Various related R packages are also available from the Stan Development Team including these and more:

Package	Description	Doc
bayesplot	ggplot-based plotting of parameter estimates, diagnostics, and posterior predictive checks.	bayesplot-package
shinystan	Interactive GUI for exploring MCMC output.	shinystan-package
loo	Out-of-sample predictive performance estimates and model comparison.	loo-package
rstanarm	R formula interface for applied regression modeling.	rstanarm-package

rstantools   Tools for developers of R packages interfacing with Stan.   rstantools-package

Author(s)

Jonah Gabry (author)	<jonah.sol.gabry@columbia.edu>
Ben Goodrich (maintainer, author)	<benjamin.goodrich@columbia.edu>
Jiqiang Guo (author)	<guojq28@gmail.com>

There are also many other important contributors to RStan (github.com/rstan). Please use 'Stan Development Team' whenever citing the R interface to Stan. A BibTex entry is available from `https://mc-stan.org/rstan/authors` or `citation("rstan")`.

See Also

- The RStan vignettes: `https://mc-stan.org/rstan/articles/`.
- `stan` for details on fitting models and `stanfit` for information on the fitted model objects.
- The `lookup` for finding a function in the Stan language that corresponds to a R function or name.
- `https://github.com/stan-dev/rstan/issues/` to submit a bug report or feature request.
- `https://discourse.mc-stan.org` to ask a question on the Stan Forums.

Examples

```
## Not run:

stanmodelcode <- "
data {
  int<lower=0> N;
  array[N] real y;
}

parameters {
  real mu;
}

model {
  target += normal_lpdf(mu | 0, 10);
  target += normal_lpdf(y  | mu, 1);
}
"

y <- rnorm(20)
dat <- list(N = 20, y = y);
fit <- stan(model_code = stanmodelcode, model_name = "example",
            data = dat, iter = 2012, chains = 3, verbose = TRUE,
            sample_file = file.path(tempdir(), 'norm.csv'))
print(fit)
```

```
# extract samples
e <- extract(fit, permuted = FALSE) # return a list of arrays
str(e)

arr <- as.array(fit) # return an array
str(arr)

mat <- as.matrix(fit) # return a matrix
str(mat)

## End(Not run)
```

| as.array | *Create array, matrix, or data.frame objects from samples in a* stanfit *object* |

Description

The samples (without warmup) included in a [stanfit](#) object can be coerced to an array, matrix, or data.frame. Methods are also provided for checking and setting names and dimnames.

Usage

```
  ## S3 method for class 'stanfit'
as.array(x, ...)
  ## S3 method for class 'stanfit'
as.matrix(x, ...)
  ## S3 method for class 'stanfit'
as.data.frame(x, ...)
  ## S3 method for class 'stanfit'
is.array(x)
  ## S3 method for class 'stanfit'
dim(x)
  ## S3 method for class 'stanfit'
dimnames(x)
  ## S3 method for class 'stanfit'
names(x)
  ## S3 replacement method for class 'stanfit'
names(x) <- value
```

Arguments

x	An object of S4 class [stanfit](#).
...	Additional parameters that can be passed to [extract](#) for extracting samples from x. For now, pars is the only additional parameter supported.
value	For the names replacement method, a character vector to set/replace the parameter names in x.

Details

as.array and as.matrix can be applied to a stanfit object to coerce the samples without warmup
to array or matrix. The as.data.frame method first calls as.matrix and then coerces this matrix
to a data.frame.

The array has three named dimensions: iterations, chains, parameters. For as.matrix, all chains
are combined, leaving a matrix of iterations by parameters.

Value

as.array, as.matrix, and as.data.frame return an array, matrix, and data.frame, respectively.

dim and dimnames return the dim and dimnames of the array object that could be created, while
names returns the third element of the dimnames, which are the names of the margins of the posterior
distribution. The names assignment method allows for assigning more interpretable names to them.

is.array returns TRUE for stanfit objects that include samples; otherwise FALSE.

When the stanfit object does not contain samples, empty objects are returned from as.array,
as.matrix, as.data.frame, dim, dimnames, and names.

See Also

S4 class stanfit and its method extract

Examples

```
## Not run:
ex_model_code <- '
  parameters {
    array[2, 3] real alpha;
    array[2] real beta;
  }
  model {
    for (i in 1:2) for (j in 1:3)
      alpha[i, j] ~ normal(0, 1);
    for (i in 1:2)
      beta[i] ~ normal(0, 2);
    # beta ~ normal(0, 2) // vectorized version
  }
'

## fit the model
fit <- stan(model_code = ex_model_code, chains = 4)

dim(fit)
dimnames(fit)
is.array(fit)
a <- as.array(fit)
m <- as.matrix(fit)
d <- as.data.frame(fit)

## End(Not run)
```

As.mcmc.list   *Create an mcmc.list from a stanfit object*

Description

Create an [mcmc.list](#) (**coda**) from a stanfit object.

Usage

```
As.mcmc.list(object, pars, include = TRUE, ...)
```

Arguments

object	object of class "stanfit"
pars	optional character vector of parameters to include
include	logical scalar indicating whether to include (the default) or exclude the parameters named in pars
...	unused

Value

An object of class [mcmc.list](#).

---

check_hmc_diagnostics   *Check HMC diagnostics after sampling*

Description

These functions print summaries of important HMC diagnostics or extract those diagnostics from a stanfit object. See the **Details** section, below.

Usage

```
check_hmc_diagnostics(object)
check_divergences(object)
check_treedepth(object)
check_energy(object)

get_divergent_iterations(object)
get_max_treedepth_iterations(object)
get_num_leapfrog_per_iteration(object)

get_num_divergent(object)
get_num_max_treedepth(object)

get_bfmi(object)
get_low_bfmi_chains(object)
```

Arguments

object   A stanfit object.

Details

The check_hmc_diagnostics function calls the other check_* functions internally and prints an overall summary, but the other functions can also be called directly:

- check_divergences prints the number (and percentage) of iterations that ended with a divergence,
- check_treedepth prints the number (and percentage) of iterations that saturated the max treedepth,
- check_energy prints E-BFMI values for each chain for which E-BFMI is less than 0.2.

The get_* functions are for programmatic access to the diagnostics.

- get_divergent_iterations and get_max_treedepth_iterations return a logical vector indicating problems for individual iterations,
- get_num_divergent and get_num_max_treedepth return the number of offending interations,
- get_num_leapfrog_per_iteration returns an integer vector with the number of leapfrog evalutions for each iteration,
- get_bfmi returns per-chain E-BFMI values and get_low_bfmi_chains returns the indices of chains with low E-BFMI.

The following are several of many resources that provide more information on these diagnostics:

- Brief explanations of some of the problems detected by these diagnostics can be found in the *Brief Guide to Stan's Warnings*.
- Betancourt (2017) provides much more depth on these diagnostics as well as a conceptual introduction to Hamiltonian Monte Carlo in general.
- Gabry et al. (2018) and the **bayesplot** package vignettes demonstrate various visualizations of these diagnostics that can be made in R.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual*. https://mc-stan.org/.

Betancourt, M. (2017). A conceptual introduction to Hamiltonian Monte Carlo. https://arxiv.org/abs/1701.02434.

Gabry, J., Simpson, D., Vehtari, A., Betancourt, M., and Gelman, A. (2018). Visualization in Bayesian workflow. *Journal of the Royal Statistical Society Series A*, accepted for publication. arXiv preprint: https://arxiv.org/abs/1709.01449.

Examples

```
## Not run:
schools <- stan_demo("eight_schools")
check_hmc_diagnostics(schools)
check_divergences(schools)
check_treedepth(schools)
check_energy(schools)

## End(Not run)
```

Diagnostic plots *RStan Diagnostic plots*

Description

Diagnostic plots for HMC and NUTS using ggplot2.

Usage

```
stan_diag(object,
          information = c("sample","stepsize", "treedepth","divergence"),
          chain = 0, ...)
stan_par(object, par, chain = 0, ...)

stan_rhat(object, pars, ...)
stan_ess(object, pars, ...)
stan_mcse(object, pars, ...)
```

Arguments

object	A stanfit or stanreg object.
information	The information to be contained in the diagnostic plot.
par, pars	The name of a single scalar parameter (par) or one or more parameter names (pars).
chain	If chain=0 (the default) all chains are combined. Otherwise the plot for chain is overlaid on the plot for all chains combined.
...	For stan_diag and stan_par, optional arguments to [arrangeGrob](#). For stan_rhat, stan_ess, and stan_mcse, optional arguments to stat_bin in the **ggplot2** package.

Details

stan_rhat**,**stan_ess**,**stan_mcse Respectively, these plots show the distribution of the Rhat statistic, the ratio of effective sample size to total sample size, and the ratio of Monte Carlo standard error to posterior standard deviation for the estimated parameters. These plots are not intended to identify individual parameters, but rather to allow for quickly identifying if the estimated values of these quantities are desireable for all parameters.

stan_par Calling `stan_par` generates three plots: (i) a scatterplot of `par` vs. the accumulated
log-posterior (`lp__`), (ii) a scatterplot of `par` vs. the average Metropolis acceptance rate
(`accept_stat`), and (iii) a violin plot showing the distribution of `par` at each of the sam-
pled step sizes (one per chain). For the scatterplots, red points are superimposed to indicate
which (if any) iterations encountered a divergent transition. Yellow points indicate a transition
that hit the maximum treedepth rather than terminated its evolution normally.

stan_diag The `information` argument is used to specify which plots `stan_diag` should generate:

- `information='sample'` Histograms of `lp__` and `accept_stat`, as well as a scatterplot
  showing their joint distribution.
- `information='stepsize'` Violin plots showing the distributions of `lp__` and `accept_stat`
  at each of the sampled step sizes (one per chain).
- `information='treedepth'` Histogram of `treedepth` and violin plots showing the dis-
  tributions of `lp__` and `accept_stat` for each value of `treedepth`.
- `information='divergence'` Violin plots showing the distributions of `lp__` and `accept_stat`
  for iterations that encountered divergent transitions (`divergent=1`) and those that did not
  (`divergent=0`).

Value

For `stan_diag` and `stan_par`, a list containing the ggplot objects for each of the displayed plots.
For `stan_rhat`, `stan_ess`, and `stan_mcse`, a single ggplot object.

Note

For details about the individual diagnostics and sampler parameters and their interpretations see
the Stan Modeling Language User's Guide and Reference Manual at https://mc-stan.org/documentation/.

See Also

[List of RStan plotting functions](), [Plot options]()

Examples

```
## Not run:
fit <- stan_demo("eight_schools")

stan_diag(fit, info = 'sample') # shows three plots together
samp_info <- stan_diag(fit, info = 'sample') # saves the three plots in a list
samp_info[[3]] # access just the third plot

stan_diag(fit, info = 'sample', chain = 1) # overlay chain 1

stan_par(fit, par = "mu")

## End(Not run)
```

---

expose_stan_functions   *Expose user-defined Stan functions to* R *for testing and simulation*

Description

The Stan modeling language allows users to define their own functions in a `functions` block at the top of a Stan program. The `expose_stan_functions` utility function uses [sourceCpp](#) to export those user-defined functions to the specified environment for testing inside R or for doing posterior predictive simulations in R rather than in the `generated quantities` block of a Stan program.

Usage

```
expose_stan_functions(stanmodel, includes = NULL,
                      show_compiler_warnings = FALSE, ...)
get_rng(seed = 0L)
get_stream()
```

Arguments

stanmodel	A [stanmodel](#) object, a [stanfit](#) object, a list produced by [stanc](#) or the path to a Stan program (`.stan` file). In any of these cases, the underlying Stan program should contain a non-empty `functions` block.
includes	If not `NULL` (the default), then a character vector of length one (possibly containing one or more `"\n"`) of the form `'#include "/full/path/to/my_header.hpp"'`, which will be inserted into the C++ code in the model's namespace and can be used to provide definitions of functions that are declared but not defined in `stanmodel`
show_compiler_warnings	
	Logical scalar defaulting to `FALSE` that controls whether compiler warnings, which can be numerous and have never been relevant, are shown
seed	An integer vector of length one indicating the state of Stan's pseudo-random number generator
...	Further arguments passed to [sourceCpp](#).

Details

The `expose_stan_functions` function requires as much compliance with the C++14 standard as is implemented in the RTools toolchain for Windows. On Windows, you will likely need to specify `CXX14 = g++ -std=c++1y` in the file whose path is [normalizePath](#)(`"~/.R/Makevars"`) in order for expose_stan_functions to work. Outside of Windows, the necessary compiler flags are set programatically, which is likely to suffice.

There are a few special types of user-defined Stan functions for which some additional details are relevant:

(P)RNG functions: If a user-defined Stan function ends in _rng, then it can use the Boost pseudo-random number generator used by Stan. When exposing such functions to R, base_rng__ and pstream__ arguments will be added to the [formals](#). The base_rng__ argument should be passed the result of a call to get_rng (perhaps specifying its seed argument for reproducibility) and the pstream__ should be passed the result of a call to get_stream, which can be used to see the result of print and reject calls in the user-defined Stan functions. These arguments default to get_stream() and get_rng() respectively.

LP functions: If a user-defined Stan function ends in _lp, then it can modify the log-probability used by Stan to evaluate Metropolis proposals or as an objective function for optimization. When exposing such functions to R, a lp__ argument will be added to the [formals](#). This lp__ argument defaults to zero, but a [double](#) precision scalar may be passed to this argument when the function is called from R. Such a user-defined Stan function can terminate with return target(); or can execute print(target()); to verify that the calculation is correct.

Value

The names of the new functions in env are returned invisibly.

See Also

[sourceCpp](#) and the section in the Stan User Manual on user-defined functions

Examples

```
## Not run:
model_code <-
  '
  functions {
    real standard_normal_rng() {
      return normal_rng(0,1);
   }
  }
'
expose_stan_functions(stanc(model_code = model_code))
standard_normal_rng()
PRNG <- get_rng(seed = 3)
o <- get_stream()
standard_normal_rng(PRNG, o)

## End(Not run)
```

---

extract   *Extract samples from a fitted Stan model*

Description

Extract samples from a fitted model represented by an instance of class [stanfit](#).

Usage

```
   ## S4 method for signature 'stanfit'
extract(object, pars, permuted = TRUE, inc_warmup = FALSE,
  include = TRUE)
```

Arguments

object	An object of class [stanfit]().
pars	An optional character vector providing the parameter names (or other quantity names) of interest. If not specified, all parameters and other quantities are used. The log-posterior with name lp__ is also included by default.
permuted	A logical scalar indicating whether the draws after the *warmup* period in each chain should be *permuted* and *merged*. If FALSE, the original order is kept. For each stanfit object, the permutation is fixed (i.e., extracting samples a second time will give the same sequence of iterations).
inc_warmup	A logical scalar indicating whether to include the warmup draws. This argument is only relevant if permuted is FALSE.
include	A logical scalar indicating whether the parameters named in pars should be included (TRUE) or excluded (FALSE).

Value

When permuted = TRUE, this function returns a named list, every element of which is an array representing samples for a parameter with all chains merged together.

When permuted = FALSE, an array is returned; the first dimension is for the iterations, the second for the number of chains, the third for the parameters. Vectors and arrays are expanded to one parameter (a scalar) per cell, with names indicating the third dimension. See the examples (with comments) below. The [monitor]() function can be applied to the returned array to obtain a summary (similar to the print method for [stanfit]() objects).

Methods

extract signature(object = "stanfit") Extract samples from a fitted model represented by an instance of class [stanfit]().

See Also

S4 class [stanfit](), [as.array.stanfit](), and [monitor]()

Examples

```
## Not run:
ex_model_code <- '
  parameters {
    array[2, 3] real alpha;
    array[2] real beta;
  }
  model {
```

```
    for (i in 1:2) for (j in 1:3)
      alpha[i, j] ~ normal(0, 1);
    for (i in 1:2)
      beta ~ normal(0, 2);
  }
'

## fit the model
fit <- stan(model_code = ex_model_code, chains = 4)

## extract alpha and beta with 'permuted = TRUE'
fit_ss <- extract(fit, permuted = TRUE) # fit_ss is a list
## list fit_ss should have elements with name 'alpha', 'beta', 'lp__'
alpha <- fit_ss$alpha
beta <- fit_ss$beta
## or extract alpha by just specifying pars = 'alpha'
alpha2 <- extract(fit, pars = 'alpha', permuted = TRUE)$alpha
print(identical(alpha, alpha2))

## or extract alpha by excluding beta and lp__
alpha3 <- extract(fit, pars = c('beta', 'lp__'),
                  permuted = TRUE, include = FALSE)$alpha
print(identical(alpha, alpha3))

## get the samples for alpha[1,1] and beta[2]
alpha_11 <- alpha[, 1, 1]
beta_2 <- beta[, 2]

## extract samples with 'permuted = FALSE'
fit_ss2 <- extract(fit, permuted = FALSE) # fit_ss2 is an array

## the dimensions of fit_ss2 should be
## "# of iterations * # of chains * # of parameters"
dim(fit_ss2)

## since the third dimension of `fit_ss2` indicates
## parameters, the names should be
##  alpha[1,1], alpha[2,1], alpha[1,2], alpha[2,2],
##  alpha[1,3], alpha[2,3], beta[1], beta[2], and lp__
## `lp__` (the log-posterior) is always included
## in the samples.
dimnames(fit_ss2)

## End(Not run)

# Create a stanfit object from reading CSV files of samples (saved in rstan
# package) generated by funtion stan for demonstration purpose from model as follows.
#
excode <- '
  transformed data {
    array[20] real y;
    y[1] <- 0.5796;  y[2]  <- 0.2276;   y[3] <- -0.2959;
    y[4] <- -0.3742; y[5]  <- 0.3885;   y[6] <- -2.1585;
```

```
     y[7] <- 0.7111;  y[8]  <- 1.4424;   y[9] <- 2.5430;
     y[10] <- 0.3746; y[11] <- 0.4773;   y[12] <- 0.1803;
     y[13] <- 0.5215; y[14] <- -1.6044;  y[15] <- -0.6703;
     y[16] <- 0.9459; y[17] <- -0.382;   y[18] <- 0.7619;
     y[19] <- 0.1006; y[20] <- -1.7461;
   }
   parameters {
     real mu;
     real<lower=0, upper=10> sigma;
     vector[2] z[3];
     real<lower=0> alpha;
   }
   model {
     y ~ normal(mu, sigma);
     for (i in 1:3)
       z[i] ~ normal(0, 1);
     alpha ~ exponential(2);
   }
'
# exfit <- stan(model_code = excode, save_dso = FALSE, iter = 200,
#               sample_file = "rstan_doc_ex.csv")
#
exfit <- read_stan_csv(dir(system.file('misc', package = 'rstan'),
                       pattern='rstan_doc_ex_[[:digit:]].csv',
                       full.names = TRUE))

ee1 <- extract(exfit, permuted = TRUE)
print(names(ee1))

for (name in names(ee1)) {
  cat(name, "\n")
  print(dim(ee1[[name]]))
}

ee2 <- extract(exfit, permuted = FALSE)
print(dim(ee2))
print(dimnames(ee2))
```

---

extract_sparse_parts   *Extract the compressed representation of a sparse matrix*

Description

Create a list of vectors that represents a sparse matrix.

Usage

```
extract_sparse_parts(A)
```

Arguments

A   A [matrix](#) or [Matrix](#).

Details

The Stan Math Library has a function called `csr_matrix_times_vector`, which inputs a matrix in compressed row storage form and a dense vector and returns their product without fillin. To use the `csr_matrix_times_vector` function with a large sparse matrix, it is optimal in terms of memory to simply pass the three vectors that characterize the compressed row storage form of the matrix to the `data` block of the Stan program. The `extract_sparse_parts` function provides a convenient means of obtaining these vectors.

Value

A named list with components

1. `w` A numeric vector containing the non-zero elements of `A`.

2. `v` An integer vector containing the column indices of the non-zero elements of `A`.

3. `u` An integer vector indicating where in `w` a given row's non-zero values start.

Examples

```
A <- rbind(
  c(19L, 27L,  0L,  0L),
  c( 0L,  0L,  0L,  0L),
  c( 0L,  0L,  0L, 52L),
  c(81L,  0L, 95L, 33L)
)
str(extract_sparse_parts(A))
```

---

gqs   *Draw samples of generated quantities from a Stan model*

Description

Draw samples from the generated quantities block of a [stanmodel](#).

Usage

```
   ## S4 method for signature 'stanmodel'
gqs(object, data = list(), draws,
    seed = sample.int(.Machine$integer.max, size = 1L))
```

Arguments

object	An object of class [stanmodel]().
data	A named `list` or `environment` providing the data for the model or a character vector for all the names of objects used as data. See the **Passing data to Stan** section in [stan]().
draws	A matrix of posterior draws, typically created by calling `as.matrix` on a [stanfit]().
seed	The seed for random number generation. The default is generated from 1 to the maximum integer supported by R on the machine. When a seed is specified by a number, `as.integer` will be applied to it. If `as.integer` produces `NA`, the seed is generated randomly. The seed can also be specified as a character string of digits, such as `"12345"`, which is converted to integer.

Value

An object of S4 class [stanmodel]() representing the fitted results.

Methods

object signature(object = "stanmodel") Evaluate the generated quantities block of a Stan program by supplying `data` and the `draws` output from a previous Stan program.

See Also

[stanmodel](), [stanfit](), [stan]()

Examples

```
## Not run:
m <- stan_model(model_code = 'parameters {real y;} model {y ~ normal(0,1);}')
f <- sampling(m, iter = 300)
mc <-
'
parameters {real y;}
generated quantities {real y_rep = normal_rng(y, 1);}
'
m2 <- stan_model(model_code = mc)
f2 <- gqs(m2, draws = as.matrix(f))
f2

## End(Not run)
```

log_prob-methods   log_prob *and* grad_log_prob *functions*

Description

Using model's `log_prob` and `grad_log_prob` take values from the unconstrained space of model parameters and (by default) return values in the same space. Sometimes we need to convert the values of parameters from their support defined in the parameters block (which might be constrained, and for simplicity, we call it the constrained space) to the unconstrained space and vice versa. The `constrain_pars` and `unconstrain_pars` functions are used for this purpose.

Usage

```
   ## S4 method for signature 'stanfit'
log_prob(object, upars, adjust_transform = TRUE, gradient = FALSE)

   ## S4 method for signature 'stanfit'
grad_log_prob(object, upars, adjust_transform = TRUE)

   ## S4 method for signature 'stanfit'
get_num_upars(object)

   ## S4 method for signature 'stanfit'
constrain_pars(object, upars)

   ## S4 method for signature 'stanfit'
unconstrain_pars(object, pars)
```

Arguments

object   An object of class stanfit.

pars   An list specifying the values for all parameters on the constrained space.

upars   A numeric vector for specifying the values for all parameters on the unconstrained space.

adjust_transform

                  Logical to indicate whether to adjust the log density since Stan transforms parameters to unconstrained space if it is in constrained space. Set to FALSE to make the function return the same values as Stan's lp__ output.

gradient   Logical to indicate whether gradients are also computed as well as the log density.

Details

Stan requires that parameters be defined along with their support. For example, for a variance parameter, we must define it on the positive real line. But inside Stan's samplers all parameters

defined on the constrained space are transformed to an unconstrained space amenable to Hamiltonian Monte Carlo. Because of this, Stan adjusts the log density function by adding the log absolute value of the Jacobian determinant. Once a new iteration is drawn, Stan transforms the parameters back to the original constrained space without requiring interference from the user. However, when using the log density function for a model exposed to R, we need to be careful. For example, if we are interested in finding the mode of parameters on the constrained space, we then do not need the adjustment. For this reason, the `log_prob` and `grad_log_prob` functions accept an `adjust_transform` argument.

Value

`log_prob` returns a value (up to an additive constant) the log posterior. If `gradient` is `TRUE`, the gradients are also returned as an attribute with name `gradient`.

`grad_log_prob` returns a vector of the gradients. Additionally, the vector has an attribute named `log_prob` being the value the same as `log_prob` is called for the input parameters.

`get_num_upars` returns the number of parameters on the unconstrained space.

`constrain_pars` returns a list and `unconstrain_pars` returns a vector.

Methods

log_prob `signature(object = "stanfit")` Compute `lp__`, the log posterior (up to an additive constant) for the model represented by a `stanfit` object. Note that, by default, `log_prob` returns the log posterior in the *unconstrained* space Stan works in internally. set `adjust_transform = FALSE` to make the values match Stan's output.

grad_log_prob `signature(object = "stanfit")` Compute the gradients for `log_prob` as well as the log posterior. The latter is returned as an attribute.

get_num_upars `signature(object = "stanfit")` Get the number of unconstrained parameters.

constrain_pars `signature(object = "stanfit")` Convert values of the parameter from unconstrained space (given as a vector) to their constrained space (returned as a named list).

unconstrain_pars `signature(object = "stanfit")` Contrary to `constrained`, conert values of the parameters from constrained to unconstrained space.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual*. https://mc-stan.org.

See Also

stanfit

Examples

```
## Not run:
# see the examples in the help for stanfit as well
# do a simple optimization problem
opcode <- "
parameters {
```

```
  real y;
}
model {
  target += log(square(y - 5) + 1);
}
"
opfit <- stan(model_code = opcode, chains = 0)
tfun <- function(y) log_prob(opfit, y)
tgrfun <- function(y) grad_log_prob(opfit, y)
or <- optim(1, tfun, tgrfun, method = 'BFGS')
print(or)

# return the gradient as an attribute
tfun2 <- function(y) {
  g <- grad_log_prob(opfit, y)
  lp <- attr(g, "log_prob")
  attr(lp, "gradient") <- g
  lp
}

or2 <- nlm(tfun2, 10)
or2

## End(Not run)
```

---

loo.stanfit   *Approximate leave-one-out cross-validation*

Description

A [loo](loo) method that is customized for stanfit objects. The loo method for stanfit objects —a wrapper
around the array method for [loo](loo) in the **loo** package — computes PSIS-LOO CV, approximate
leave-one-out cross-validation using Pareto smoothed importance sampling (Vehtari, Gelman, and
Gabry, 2017a,2017b).

Usage

```
  ## S3 method for class 'stanfit'
loo(x,
    pars = "log_lik",
    save_psis = FALSE,
    cores = getOption("mc.cores", 1),
    moment_match = FALSE,
    k_threshold = 0.7,
    ...)
```

Arguments

x   An object of S4 class stanfit.

pars	Name of transformed parameter or generated quantity in the Stan program corresponding to the pointwise log-likelihood. If not specified the default behavior is to look for "log_lik".
save_psis	Should the intermediate results from [psis](#) be saved in the returned object? The default is FALSE. This can be useful to avoid repeated computation when using other functions in the **loo** and **bayesplot** packages.
cores	Number of cores to use for parallelization. The default is 1 unless cores is specified or the mc.cores [option](#) has been set.
moment_match	Logical; Whether to use the moment matching algorithm for observations with high Pareto k values to improve accuracy. Note: because the moment matching algorithm relies on the unconstrain_pars method in RStan it is only available if run in the same R session as fitting the model.
k_threshold	Threshold value for Pareto k values above which the moment matching algorithm is used. If moment_match is FALSE, this is ignored.
...	Ignored.

Details

Stan does not automatically compute and store the log-likelihood. It is up to the user to incorporate it into the Stan program if it is to be extracted after fitting the model. In a Stan program, the pointwise log likelihood can be coded as a vector in the transformed parameters block (and then summed up in the model block) or it can be coded entirely in the generated quantities block. We recommend using the generated quantities block so that the computations are carried out only once per iteration rather than once per HMC leapfrog step.

For example, the following is the generated quantities block for computing and saving the log-likelihood for a linear regression model with N data points, outcome y, predictor matrix X (including column of 1s for intercept), coefficients beta, and standard deviation sigma:

```
vector[N] log_lik;
```

```
for (n in 1:N) log_lik[n] = normal_lpdf(y[n] | X[n, ] * beta, sigma);
```

This function automatically uses Pareto k diagnostics for assessing the accuracy of importance sampling for each observation. When the diagnostics indicate that importance sampling for certain observations is inaccurate, a moment matching algorithm can be used, which can improve the accuracy (Paananen et al., 2020).

Value

A list with class c("psis_loo", "loo"), as detailed in the [loo](#) documentation.

References

Vehtari, A., Gelman, A., and Gabry, J. (2017a). Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC. *Statistics and Computing*. 27(5), 1413-1432. doi:10.1007/s11222-016-9696-4. <https://arxiv.org/abs/1507.04544>, <https://link.springer.com/article/10.1007/s11222-016-9696-4>

Vehtari, A., Gelman, A., and Gabry, J. (2017b). Pareto smoothed importance sampling. arXiv preprint: <https://arxiv.org/abs/1507.02646>

Yao, Y., Vehtari, A., Simpson, D., and Gelman, A. (2018). Using stacking to average Bayesian predictive distributions. Bayesian Analysis, advance publication, `doi:10.1214/17-BA1091`.

Paananen, T., Piironen, J., Buerkner, P.-C., Vehtari, A. (2020). Implicitly Adaptive Importance Sampling. arXiv preprint: `https://arxiv.org/abs/1906.08850`.

See Also

- The **loo** package documentation, including the vignettes for many examples (`https://mc-stan.org/loo/`).
- `loo_moment_match` for the moment matching algorithm.
- `loo_model_weights` for model averaging/weighting via stacking or pseudo-BMA weighting.

Examples

```
## Not run:
# Generate a dataset from N(0,1)
N <- 100
y <- rnorm(N, 0, 1)

# Suppose we have three models for y:
#  1) y ~ N(-1, sigma)
#  2) y ~ N(0.5, sigma)
#  3) y ~ N(0.6,sigma)
#
stan_code <- "
data {
  int N;
  vector[N] y;
  real mu_fixed;
}
  parameters {
  real<lower=0> sigma;
}
model {
  sigma ~ exponential(1);
  y ~ normal(mu_fixed, sigma);
}
generated quantities {
  vector[N] log_lik;
  for (n in 1:N) log_lik[n] = normal_lpdf(y[n]| mu_fixed, sigma);
}"

mod <- stan_model(model_code = stan_code)
fit1 <- sampling(mod, data=list(N=N, y=y, mu_fixed=-1))
fit2 <- sampling(mod, data=list(N=N, y=y, mu_fixed=0.5))
fit3 <- sampling(mod, data=list(N=N, y=y, mu_fixed=0.6))

# use the loo method for stanfit objects
loo1 <- loo(fit1, pars = "log_lik")
print(loo1)
```

```
# which is equivalent to
LL <- as.array(fit1, pars = "log_lik")
r_eff <- loo::relative_eff(exp(LL))
loo1b <- loo::loo.array(LL, r_eff = r_eff)
print(loo1b)

# compute loo for the other models
loo2 <- loo(fit2)
loo3 <- loo(fit3)

# stacking weights
wts <- loo::loo_model_weights(list(loo1, loo2, loo3), method = "stacking")
print(wts)

# use the moment matching for loo with a stanfit object
loo_mm <- loo(fit1, pars = "log_lik", moment_match = TRUE)
print(loo_mm)

## End(Not run)
```

---

lookup   *Look up the Stan function that corresponds to a* R *function or name.*

Description

This function helps to map between R functions and Stan functions.

Usage

```
lookup(FUN, ReturnType = character())
```

Arguments

FUN
: A character string naming a R function or a R function for which the (near) equivalent Stan function is sought. If no matching R function is found, FUN is reinterpreted as a [regexp](#) and matches are sought.

ReturnType
: A character string of positive length naming a valid return type for a Stan function: int, int[], matrix, real, real[,], real[], row_vector, T[], vector, or void. If "ANY" is passed, then the entire [data.frame](#) is returned and can be inspected with the [View](#) function, for example.

Value

Ordinarily, a data.frame with rows equal to the number of partial matches and four columns:

1. StanFunction Character string for the Stan function's name.
2. Arguments Character string indicating the arguments to that Stan function.
3. ReturnType Character string indicating the return type of that Stan function.

4. Page Integer indicating the page of the Stan reference manual where that Stan function is defined.

If there are no matching Stan functions, a character string indicating so is returned.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual.* `https://mc-stan.org/`.

The Stan Development Team *CmdStan Interface User's Guide.* `https://mc-stan.org`.

Examples

```
lookup(dnorm)      # Stan equivalents for the normal PDF (in log form)
lookup("foo")      # fails
lookup("Student")  # succeeds even though there is no such R function
lookup("^poisson") # every Stan function that starts with poisson
```

---

loo_moment_match.stanfit

Moment matching for efficient approximate leave-one-out cross-validation (LOO)

Description

A `loo_moment_match` method that is customized for stanfit objects. The `loo_moment_match` method for stanfit objects —a wrapper around the `loo_moment_match` (**loo** package)— updates a loo object using moment matching (Paananen et al., 2020).

Usage

```
## S3 method for class 'stanfit'
loo_moment_match(x,
    loo = loo,
    ...)
```

Arguments

x	An object of S4 class stanfit.
loo	A loo object that is modified.
...	Further arguments.

Value

The `loo_moment_match()` methods return an updated loo object.

References

Paananen, T., Piironen, J., Buerkner, P.-C., Vehtari, A. (2020). Implicitly Adaptive Importance Sampling. preprint arXiv:1906.08850

See Also

`loo()`, `loo_moment_match()`

---

makeconf_path	*Obtain the full path of file* `Makeconf`

Description

Obtain the full path of file `Makeconf`, in which, for example the flags for compiling C/C++ code are configured.

Usage

```
makeconf_path()
```

Details

The configuration for compiling shared objects using `R CMD SHLIB` are set in file `Makeconf`. To change how the C++ code is compiled, modify this file. For RStan, package **inline** compiles the C++ code using `R CMD SHLIB`. To speed up compiled Stan models, increase the optimization level to `-O3` defined in property `CXXFLAGS` in the file `Makeconf`. This file may also be modified to specify alternative C++ compilers, such as clang++ or later versions of g++.

Value

An character string for the full path of file `Makeconf`.

See Also

`stan`

Examples

```
makeconf_path()
```

---

monitor   *Compute summaries of MCMC draws and monitor convergence*

Description

Similar to the `print` method for `stanfit` objects, but `monitor` takes an array of simulations as its argument rather than a `stanfit` object. For a 3-D array (iterations * chains * parameters) of MCMC draws, `monitor` computes means, standard deviations, quantiles, Monte Carlo standard errors, split Rhats, and effective sample sizes. By default, half of the iterations are considered warmup and are excluded.

Usage

```
monitor(sims, warmup = floor(dim(sims)[1]/2),
        probs = c(0.025, 0.25, 0.5, 0.75, 0.975),
        digits_summary = 1, print = TRUE, ...)
## S3 method for class 'simsummary'
print(x, digits = 3, se = FALSE, ...)
## S3 method for class 'simsummary'
x[i, j, drop = if (missing(i)) TRUE else length(j) == 1]
```

Arguments

sims	A 3-D array (iterations * chains * parameters) of MCMC simulations from any MCMC algorithm.
warmup	The number of warmup iterations to be excluded when computing the summaries. The default is half of the total number of iterations. If `sims` doesn't include the warmup iterations then `warmup` should be set to zero.
probs	A numeric vector specifying quantiles of interest. The defaults is `c(0.025,0.25,0.5,0.75,0.975)`.
digits_summary	The number of significant digits to use when printing the summary, defaulting to 1. Applies to the quantities other than the effective sample size, which is always rounded to the nearest integer.
print	Logical, indicating whether to print the summary after the computations are performed.
...	Additional arguments passed to the underlying `print` method.
x	An object of class `simsummary` created by `monitor`
digits	An integer scalar defaulting to 3 for the number of decimal places to print
se	A logical scalar defaulting to `FALSE` indicating whether to print the estimated standard errors of the estimates
i	A vector indicating which rows of the object created by `monitor` to select
j	A vector indicating which columns of the object crated by `monitor` to select
drop	A logical scalar indicating whether the resulting object should return a vector where possible

Value

A 2-D array with rows corresponding to parameters and columns to the summary statistics that can be printed and subset.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual.* `https://mc-stan.org`.

See Also

S4 class `stanfit` and particularly its `print` method.

Examples

```
csvfiles <- dir(system.file('misc', package = 'rstan'),
                pattern = 'rstan_doc_ex_[0-9].csv', full.names = TRUE)
fit <- read_stan_csv(csvfiles)
# The following is just for the purpose of giving an example
# since print can be used for a stanfit object.
monitor(extract(fit, permuted = FALSE, inc_warmup = TRUE))
```

---

nlist   *Created named lists*

Description

Create a named list using specified names or, if names are omitted, using the names of the objects in the list. The code]list(a = a, b = b) becomes nlist(a,b) and list(a = a, b = 2) becomes nlist(a, b = 2), etc. This is convenient when creating the list of data to pass to Stan.

Usage

```
nlist(...)
```

Arguments

...   The objects to include in the list.

Value

A named list.

Examples

```
# All variables already defined
x <- 1
y <- 2
nlist(x, y)

# Define some variables in the call and take the rest from the environment
nlist(x, y, z = 3)
```

---

optimizing   *Obtain a point estimate by maximizing the joint posterior*

Description

Obtain a point estimate by maximizing the joint posterior from the model defined by class stanmodel.

Usage

```
   ## S4 method for signature 'stanmodel'
optimizing(object, data = list(),
    seed = sample.int(.Machine$integer.max, 1), init = 'random',
    check_data = TRUE, sample_file = NULL,
    algorithm = c("LBFGS", "BFGS", "Newton"),
    verbose = FALSE, hessian = FALSE, as_vector = TRUE,
    draws = 0, constrained = TRUE, importance_resampling = FALSE, ...)
```

Arguments

object   An object of class stanmodel.

data   A named list or environment providing the data for the model or a character
              vector for all the names of objects used as data. See the **Passing data to Stan**
              section in stan.

seed   The seed for random number generation. The default is generated from 1 to the
              maximum integer supported by R on the machine. Even if multiple chains are
              used, only one seed is needed, with other chains having seeds derived from that
              of the first chain to avoid dependent samples. When a seed is specified by a
              number, as.integer will be applied to it. If as.integer produces NA, the seed
              is generated randomly. The seed can also be specified as a character string of
              digits, such as "12345", which is converted to integer.

init   Initial values specification. See the detailed documentation for the init argu-
              ment in stan with one exception. If specifying inits using a list then only a
              single named list of values should be provided. For example, to initialize a pa-
              rameter alpha to value1 and beta to value2 you can specify list(alpha =
              value1, beta = value2).

check_data   Logical, defaulting to TRUE. If TRUE the data will be preprocessed; otherwise
              not. See the **Passing data to Stan** section in stan.

sample_file	A character string of file name for specifying where to write samples for *all* parameters and other saved quantities. If not provided, files are not created. When the folder specified is not writable, tempdir() is used.
algorithm	One of "Newton", "BFGS", and "LBFGS" (the default) indicating which optimization algorithm to use.
verbose	TRUE or FALSE (the default): flag indicating whether to print intermediate output from Stan on the console, which might be helpful for model debugging.
hessian	TRUE or FALSE (the default): flag indicating whether to calculate the Hessian (via numeric differentiation of the gradient function in the unconstrained parameter space).
as_vector	TRUE (the default) or FALSE: flag indicating whether a vector is used to store the point estimate found. A list can be used instead by specifying it to be FALSE.
draws	A non-negative integer (that defaults to zero) indicating how many times to draw from a multivariate normal distribution whose parameters are the mean vector and the inverse negative Hessian in the unconstrained space. If draws > 0 and importance_resampling=TRUE then log_p and log_g will be computed and returned (see description in the **Value** section).
constrained	A logical scalar indicating, if draws > 0, whether the draws should be transformed to the constrained space defined in the parameters block of the Stan program. Defaults to TRUE.
importance_resampling	
	A logical scalar (defaulting to FALSE) indicating whether to do importance resampling to compute diagnostics on the draws from the normal approximation to the posterior distribution. If TRUE and draws > 0 then log_p and log_g will be computed and returned (see description in the **Value** section).
...	Other optional parameters:

- iter (integer), the maximum number of iterations, defaulting to 2000.
- save_iterations (logical), a flag indicating whether to save the iterations, defaulting to FALSE.
- refresh (integer), the number of interations between screen updates, defaulting to 100.
- init_alpha (double), for BFGS and LBFGS, the line search step size for first iteration, defaulting to 0.001.
- tol_obj (double), for BFGS and LBFGS, the convergence tolerance on changes in objective function value, defaulting to 1e-12.
- tol_rel_obj (double), for BFGS and LBFGS, the convergence tolerance on relative changes in objective function value, defaulting to 1e4.
- tol_grad (double), for BFGS and LBFGS, the convergence tolerance on the norm of the gradient, defaulting to 1e-8.
- tol_rel_grad (double), for BFGS and LBFGS, the convergence tolerance on the relative norm of the gradient, defaulting to 1e7.
- tol_param (double), for BFGS and LBFGS, the convergence tolerance on changes in parameter value, defaulting to 1e-8.
- history_size (integer), for LBFGS, the number of update vectors to use in Hessian approximations, defaulting to 5.

Refer to the manuals for both CmdStan and Stan for more details.

Value

A list with components:

par	The point estimate found. Its form (vector or list) is determined by the as_vector argument.
value	The value of the log-posterior (up to an additive constant, the "lp__" in Stan) corresponding to par.
return_code	The value of the return code from the optimizer; anything that is not zero is problematic.
hessian	The Hessian matrix if hessian is TRUE
theta_tilde	If draws > 0, the matrix of parameter draws in the constrained or unconstrained space, depending on the value of the constrained argument.
log_p	If draws > 0 and importance_resampling=TRUE, a vector of length draws that contains the value of the log-posterior evaluated at each row of theta_tilde.
log_g	If draws > 0, a vector of length draws that contains the value of the logarithm of the multivariate normal density evaluated at each row of theta_tilde.

If the optimization is not completed for reasons such as feeding wrong data, it returns NULL.

Methods

optimizing signature(object = "stanmodel") Call Stan's optimization methods to obtain a point estimate for the model defined by S4 class stanmodel given the data, initial values, etc.

See Also

[stanmodel](#)

Examples

```
## Not run:
m <- stan_model(model_code = 'parameters {real y;} model {y ~ normal(0,1);}')
f <- optimizing(m, hessian = TRUE)

## End(Not run)
```

---

pairs.stanfit	*Create a matrix of output plots from a* stanfit *object*

Description

A [pairs](#) method that is customized for MCMC output

Usage

```
   ## S3 method for class 'stanfit'
pairs(x, labels = NULL, panel = NULL, ...,
    lower.panel = NULL,
    upper.panel = NULL, diag.panel = NULL, text.panel = NULL,
    label.pos = 0.5 + 1/3, cex.labels = NULL, font.labels = 1,
    row1attop = TRUE, gap = 1, log = "", pars = NULL, include = TRUE,
    condition = "accept_stat__")
```

Arguments

x	An object of S4 class `stanfit`
`labels`, `panel`, `...`, `lower.panel`, `upper.panel`, `diag.panel`	
	Same as in [pairs](#) syntactically but see the Details section for different default arguments
`text.panel`, `label.pos`, `cex.labels`, `font.labels`, `row1attop`, `gap`	
	Same as in [pairs.default](#)
log	Same as in [pairs.default](#), which makes it possible to utilize logarithmic axes and additionally accepts `log = TRUE`. See the Details section.
pars	If not `NULL`, a character vector indicating which quantities to include in the plots, which is passed to [extract](#). Thus, by default, all unknown quantities are included, which may be far too many to visualize on a small computer screen. If `include = FALSE`, then the named parameters are excluded from the plot.
condition	If `NULL`, it will plot roughly half of the chains in the lower panel and the rest in the upper panel. An integer vector can be passed to select some subset of the chains, of which roughly half will be plotted in the lower panel and the rest in the upper panel. A list of two integer vectors can be passed, each specifying a subset of the chains to be plotted in the lower and upper panels respectively.
	A single number between zero and one exclusive can be passed, which is interpreted as the proportion of realizations (among all chains) to plot in the lower panel starting with the first realization in each chain, with the complement (from the end of each chain) plotted in the upper panel.
	A (possibly abbreviated) character vector of length one can be passed among `"accept_stat__"`, `"stepsize__"`, `"treedepth__"`, `"n_leapfrog__"`, `"divergent__"`,, `"energy__"`, or `"lp__"`, which are the variables produced by [get_sampler_params](#) and [get_logposterior](#). In that case the lower panel will plot realizations that are below the median of the indicated variable (or are zero in the case of `"divergent__"`) and the upper panel will plot realizations that are greater than or equal to the median of the indicated variable (or are one in the case of `"divergent__"`). Finally, any logical vector whose length is equal to the product of the number of iterations and the number of chains can be passed, in which case realizations corresponding to `FALSE` and `TRUE` will be plotted in the lower and upper panel respectively. The default is `"accept_stat__"`.
include	Logical scalar indicating whether to include (the default) or exclude the parameters named in the `pars` argument from the plot.

Details

This method differs from the default [pairs](#) method in the following ways. If unspecified, the [smoothScatter](#) function is used for the off-diagonal plots, rather than [points](#), since the former is more appropriate for visualizing thousands of draws from a posterior distribution. Also, if unspecified, histograms of the marginal distribution of each quantity are placed on the diagonal of the plot, after pooling all of the chains specified by the chain_id argument.

The draws from the warmup phase are always discarded before plotting.

By default, the lower (upper) triangle of the plot contains draws with below (above) median acceptance probability. Also, if condition is not "divergent__", red points will be superimposed onto the smoothed density plots indicating which (if any) iterations encountered a divergent transition. Otherwise, yellow points indicate a transition that hit the maximum treedepth rather than terminated its evolution normally.

You may very well want to specify the log argument for non-negative parameters. However, the pairs function will drop (with a message) parameters that are either constant or duplicative with previous parameters. For example, if a correlation matrix is included among pars, then neither its diagonal elements (which are always 1) nor its upper triangular elements (which are the same as the corresponding lower triangular elements) will be included. Thus, if log is an integer vector, it needs to pertain to the parameters after constant and duplicative ones are dropped. It is perhaps easiest to specify log = TRUE, which will utilize logarithmic axes for all non-negative parameters, except lp__ and any integer valued quantities.

See Also

S4 class [stanfit](#) and its method [extract](#) as well as the [pairs](#) generic function. Also, see [get_sampler_params](#) and [get_logposterior](#).

Examples

```
example(read_stan_csv)
pairs(fit, pars = c("mu", "sigma", "alpha", "lp__"), log = TRUE, las = 1)
# sigma and alpha will have logarithmic axes
```

---

plot-methods   *Plots for stanfit objects*

Description

The default plot shows posterior uncertainty intervals and point estimates for parameters and generated quantities. The plot method can also be used to call the other **rstan** plotting functions via the plotfun argument (see Examples).

Usage

```
   ## S4 method for signature 'stanfit,missing'
plot(x, ..., plotfun)
```

Arguments

x	An instance of class [stanfit](#).
plotfun	A character string naming the plotting function to apply to the stanfit object. If plotfun is missing, the default is to call [stan_plot](#), which generates a plot of credible intervals and point estimates. See [rstan-plotting-functions](#) for the names and descriptions of the other plotting functions. plotfun can be either the full name of the plotting function (e.g. "stan_hist") or can be abbreviated to the part of the name following the underscore (e.g. "hist").
...	Optional arguments to plotfun.

Value

A [ggplot](#) object that can be further customized using the **ggplot2** package.

Note

Because the **rstan** plotting functions use **ggplot2** (and thus the resulting plots behave like ggplot objects), when calling a plotting function within a loop or when assigning a plot to a name (e.g., graph <- plot(fit, plotfun = "rhat")), if you also want the side effect of the plot being displayed you must explicity print it (e.g., (graph <- plot(fit, plotfun = "rhat")), print(graph <- plot(fit, plotfun = "rhat"))).

See Also

[List of RStan plotting functions](#), [Plot options](#)

Examples

```
## Not run:
library(rstan)
fit <- stan_demo("eight_schools")
plot(fit)
plot(fit, show_density = TRUE, ci_level = 0.5, fill_color = "purple")
plot(fit, plotfun = "hist", pars = "theta", include = FALSE)
plot(fit, plotfun = "trace", pars = c("mu", "tau"), inc_warmup = TRUE)
plot(fit, plotfun = "rhat") + ggtitle("Example of adding title to plot")

## End(Not run)
```

Plots   *ggplot2 for RStan*

Description

Visual posterior analysis using ggplot2.

Usage

```
stan_plot(object, pars, include = TRUE, unconstrain = FALSE, ...)
stan_trace(object, pars, include = TRUE, unconstrain = FALSE,
          inc_warmup = FALSE, nrow = NULL, ncol = NULL, ...,
          window = NULL)
stan_scat(object, pars, unconstrain = FALSE,
          inc_warmup = FALSE, nrow = NULL, ncol = NULL, ...)
stan_hist(object, pars, include = TRUE, unconstrain = FALSE,
          inc_warmup = FALSE, nrow = NULL, ncol = NULL, ...)
stan_dens(object, pars, include = TRUE, unconstrain = FALSE,
          inc_warmup = FALSE, nrow = NULL, ncol = NULL, ...,
          separate_chains = FALSE)
stan_ac(object, pars, include = TRUE, unconstrain = FALSE,
          inc_warmup = FALSE, nrow = NULL, ncol = NULL, ...,
          separate_chains = FALSE, lags = 25, partial = FALSE)
quietgg(gg)
```

Arguments

`object`	A stanfit or stanreg object.
`pars`	Optional character vector of parameter names. If `object` is a stanfit object, the default is to show all user-defined parameters or the first 10 (if there are more than 10). If `object` is a stanreg object, the default is to show all (or the first 10) regression coefficients (including the intercept). For `stan_scat` only, `pars` should not be missing and should contain exactly two parameter names.
`include`	Should the parameters given by the `pars` argument be included (the default) or excluded from the plot?
`unconstrain`	Should parameters be plotted on the unconstrained space? Defaults to `FALSE`. Only available if `object` is a stanfit object.
`inc_warmup`	Should warmup iterations be included? Defaults to `FALSE`.
`nrow, ncol`	Passed to [facet_wrap]().
`...`	Optional additional named arguments passed to geoms (e.g. for `stan_trace` the geom is [geom_path]() and we could specify `linetype`, `size`, `alpha`, etc.). For `stan_plot` there are also additional arguments that can be specified in `...` (see Details).
`window`	For `stan_trace` window is used to control which iterations are shown in the plot. See [traceplot]().
`separate_chains`	For `stan_dens`, should the density for each chain be plotted? The default is `FALSE`, which means that for each parameter the draws from all chains are combined. For `stan_ac`, if `separate_chains=FALSE` (the default), the autocorrelation is averaged over the chains. If `TRUE` each chain is plotted separately.
`lags`	For `stan_ac`, the maximum number of lags to show.
`partial`	For `stan_ac`, should partial autocorrelations be plotted instead? Defaults to `FALSE`.
`gg`	A ggplot object or an expression that creates one.

Details

For `stan_plot`, there are additional arguments that can be specified in `...`. The optional arguments and their default values are:

`point_est = "median"` The point estimate to show. Either "median" or "mean".

`show_density = FALSE` Should kernel density estimates be plotted above the intervals?

`ci_level = 0.8` The posterior uncertainty interval to highlight. Central `100*ci_level`% intervals are computed from the quantiles of the posterior draws.

`outer_level = 0.95` An outer interval to also draw as a line (if `show_outer_line` is TRUE) but not highlight.

`show_outer_line = TRUE` Should the `outer_level` interval be shown or hidden? Defaults to = TRUE (to plot it).

`fill_color`, `outline_color`, `est_color` Colors to override the defaults for the highlighted interval, the outer interval (and density outline), and the point estimate.

Value

A [ggplot](#) object that can be further customized using the **ggplot2** package.

Note

Because the **rstan** plotting functions use **ggplot2** (and thus the resulting plots behave like ggplot objects), when calling a plotting function within a loop or when assigning a plot to a name (e.g., graph `<- plot(fit, plotfun = "rhat")`), if you also want the side effect of the plot being displayed you must explicity print it (e.g., (graph `<- plot(fit, plotfun = "rhat")`), `print(graph <- plot(fit, plotfun = "rhat"))`).

See Also

[List of RStan plotting functions](#), [Plot options](#)

Examples

```
## Not run:
example("read_stan_csv")
stan_plot(fit)
stan_trace(fit)

library(gridExtra)
fit <- stan_demo("eight_schools")

stan_plot(fit)
stan_plot(fit, point_est = "mean", show_density = TRUE, fill_color = "maroon")


# histograms
stan_hist(fit)
# suppress ggplot2 messages about default bindwidth
quietgg(stan_hist(fit))
```

```
quietgg(h <- stan_hist(fit, pars = "theta", binwidth = 5))

# juxtapose histograms of tau and unconstrained tau
tau <- stan_hist(fit, pars = "tau")
tau_unc <- stan_hist(fit, pars = "tau", unconstrain = TRUE) +
             xlab("tau unconstrained")
grid.arrange(tau, tau_unc)

# kernel density estimates
stan_dens(fit)
(dens <- stan_dens(fit, fill = "skyblue", ))
dens <- dens + ggtitle("Kernel Density Estimates\n") + xlab("")
dens

(dens_sep <- stan_dens(fit, separate_chains = TRUE, alpha = 0.3))
dens_sep + scale_fill_manual(values = c("red", "blue", "green", "black"))
(dens_sep_stack <- stan_dens(fit, pars = "theta", alpha = 0.5,
                              separate_chains = TRUE, position = "stack"))

# traceplot
trace <- stan_trace(fit)
trace +
  scale_color_manual(values = c("red", "blue", "green", "black"))
trace +
  scale_color_brewer(type = "div") +
  theme(legend.position = "none")

facet_style <- theme(strip.background = ggplot2::element_rect(fill = "white"),
                     strip.text = ggplot2::element_text(size = 13, color = "black"))
(trace <- trace + facet_style)

# scatterplot
(mu_vs_tau <- stan_scat(fit, pars = c("mu", "tau"), color = "blue", size = 4))
mu_vs_tau +
  ggplot2::coord_flip() +
  theme(panel.background = ggplot2::element_rect(fill = "black"))


## End(Not run)
```

---

print   *Print a summary for a fitted model represented by a* stanfit *object*

Description

Print basic information regarding the fitted model and a summary for the parameters of interest estimated by the samples included in a stanfit object.

Usage

```
   ## S3 method for class 'stanfit'
print(x, pars = x@sim$pars_oi,
      probs = c(0.025, 0.25, 0.5, 0.75, 0.975),
      digits_summary = 2, include = TRUE, ...)
```

Arguments

x	An object of S4 class stanfit.
pars	A character vector of parameter names. The default is all parameters for which samples are saved. If include = FALSE, then the specified parameters are excluded from the printed summary.
probs	A numeric vector of quantiles of interest. The default is c(0.025,0.25,0.5,0.75,0.975).
digits_summary	The number of significant digits to use when printing the summary, defaulting to 2. Applies to the quantities other than the effective sample size, which is always rounded to the nearest integer.
include	Logical scalar (defaulting to TRUE) indicating whether to include or exclude the parameters named by the pars argument.
...	Additional arguments passed to the summary method for stanfit objects.

Details

The information regarding the fitted model includes the number of iterations, the number of chains, the total number of saved iterations, the estimation algorithm used, and the timestamp indicating when sampling finished.

The parameter summaries computed include means, standard deviations (sd), quantiles, Monte Carlo standard errors (se_mean), split Rhats, and effective sample sizes (n_eff). The summaries are computed after dropping the warmup iterations and merging together the draws from all chains.

In addition to the model parameters, summaries for the log-posterior (lp__) are also reported.

See Also

S4 class stanfit and particularly its method summary, which is used to obtain the values that are printed.

---

read_rdump	*Read data in an* R *dump file to a list*

Description

Create an R list from an R dump file

Usage

```
read_rdump(f, keep.source = FALSE, ...)
```

Arguments

`f`	A character string providing the dump file name.
`keep.source`	logical: should the source formatting be retained when echoing expressions, if possible?
`...`	passed to `source`

Details

The R dump file can be read directly by R function `source`, which by default would read the data into the user's workspace (the global environment). This function instead read the data to a list, making it convenient to prepare data for the `stan` model-fitting function.

Value

A list containing all the data defined in the dump file with keys corresponding to variable names.

See Also

[stan_rdump](); [dump]()

Examples

```
x <- 1; y <- 1:10; z <- array(1:10, dim = c(2,5))
stan_rdump(ls(pattern = '^[xyz]'), file.path(tempdir(), "xyz.Rdump"))
l <- read_rdump(file.path(tempdir(), 'xyz.Rdump'))
print(l)
unlink(file.path(tempdir(), "xyz.Rdump"))
```

---

read_stan_csv   *Read CSV files of samples generated by (R)Stan into a* stanfit *object*

Description

Create a `stanfit` object from the saved CSV files that are created by Stan or RStan and that include the samples drawn from the distribution of interest to facilitate analysis of samples using RStan.

Usage

```
read_stan_csv(csvfiles, col_major = TRUE)
```

Arguments

`csvfiles`	A character vector providing CSV file names
`col_major`	The order for array parameters; default to TRUE

Details

Stan and RStan could save the samples to CSV files. This function reads the samples and using the comments (beginning with "#") to create a stanfit object. The model name is derived from the first CSV file.

col_major specifies how array parameters are ordered in each row of the CSV files. For example, parameter "a[2,2]" would be ordered as "a[1,1], a[2,1], a[1,2], a[2,2]" if col_major is TRUE.

Value

A stanfit object (with invalid stanmodel slot). This stanfit object cannot be used to re-run the sampler.

See Also

stanfit

Examples

```
csvfiles <- dir(system.file('misc', package = 'rstan'),
                pattern = 'rstan_doc_ex_[0-9].csv', full.names = TRUE)
fit <- read_stan_csv(csvfiles)
```

Rhat   *Convergence and efficiency diagnostics for Markov Chains*

Description

These functions are improved versions of the traditional Rhat (for convergence) and Effective Sample Size (for efficiency).

Usage

```
Rhat(sims)
ess_bulk(sims)
ess_tail(sims)
```

Arguments

sims   A two-dimensional array whose rows are equal to the number of iterations of the
              Markov Chain(s) and whose columns are equal to the number of Markov Chains
              (preferably more than one). The cells are the realized draws for a particular
              parameter or function of parameters.

Value

The Rhat function produces R-hat convergence diagnostic, which compares the between- and within-chain estimates for model parameters and other univariate quantities of interest. If chains have not mixed well (ie, the between- and within-chain estimates don't agree), R-hat is larger than 1. We recommend running at least four chains by default and only using the sample if R-hat is less than 1.05. Stan reports R-hat which is the maximum of rank normalized split-R-hat and rank normalized folded-split-R-hat, which works for thick tailed distributions and is sensitive also to differences in scale.

The ess_bulk function produces an estimated Bulk Effective Sample Size (bulk-ESS) using rank normalized draws. Bulk-ESS is useful measure for sampling efficiency in the bulk of the distribution (related e.g. to efficiency of mean and median estimates), and is well defined even if the chains do not have finite mean or variance.

The ess_tail function produces an estimated Tail Effective Sample Size (tail-ESS) by computing the minimum of effective sample sizes for 5% and 95% quantiles. Tail-ESS is useful measure for sampling efficiency in the tails of the distribution (related e.g. to efficiency of variance and tail quantile estimates).

Both bulk-ESS and tail-ESS should be at least 100 (approximately) per Markov Chain in order to be reliable and indicate that estimates of respective posterior quantiles are reliable.

Author(s)

Paul-Christian Burkner and Aki Vehtari

References

Aki Vehtari, Andrew Gelman, Daniel Simpson, Bob Carpenter, and Paul-Christian Burkner (2019). Rank-normalization, folding, and localization: An improved R-hat for assessing convergence of MCMC. *arXiv preprint* arXiv:1903.08008.

See Also

monitor

Examples

```
# pretend these draws came from five actual Markov Chins
sims <- matrix(rnorm(500), nrow = 100, ncol = 5)
Rhat(sims)
ess_bulk(sims)
ess_tail(sims)
```

```
rstan-plotting-functions
```
RStan Plotting Functions

Description

List of RStan plotting functions that return ggplot objects

RStan plotting functions

Posterior intervals and point estimates `stan_plot`

Traceplots `stan_trace`

Histograms `stan_hist`

Kernel density estimates `stan_dens`

Scatterplots `stan_scat`

Diagnostics for Hamiltonian Monte Carlo and the No-U-Turn Sampler `stan_diag`

Rhat `stan_rhat`

Ratio of effective sample size to total posterior sample size `stan_ess`

Ratio of Monte Carlo standard error to posterior standard deviation `stan_mcse`

Autocorrelation `stan_plot`

See Also

`Plot options`

```
rstan.package.skeleton
```
Create a Skeleton for a New Source Package with Stan Programs

Description

This function has been removed from **rstan**. Please use the new `rstan_package_skeleton` function in the **rstantools** package.

---

rstan_gg_options   *Set default appearance options*

Description

Set default appearance options

Usage

```
rstan_gg_options(...)

rstan_ggtheme_options(...)
```

Arguments

`...`   For rstan_ggtheme_options, see [theme](#) for the theme elements that can be
                    specified in `...`. For rstan_gg_options, `...` can be `fill`, `color`, `chain_colors`,
                    `size`, `pt_color`, or `pt_size`. See Examples.

See Also

[List of RStan plotting functions](#)

Examples

```
rstan_ggtheme_options(panel.background = ggplot2::element_rect(fill = "gray"),
                      legend.position = "top")
rstan_gg_options(fill = "skyblue", color = "skyblue4", pt_color = "red")
```

---

rstan_options   *Set and read options used in RStan*

Description

Set and read options used in RStan. Some settings as options can be controlled by the user.

Usage

```
rstan_options(...)
```

Arguments

`...`   Arguments of the form `opt = val` set option `opt` to value `val`. Arguments of the
                    form `opt` set the function to return option `opt`'s value. Each argument must be
                    a character string.

Details

The available options are:

1. plot_rhat_breaks: The cut off points for Rhat for which we would indicate using a different color. This is a numeric vector, defaulting to c(1.1, 1.2, 1.5, 2). The value for this option will be sorted in ascending order, so for example plot_rhat_breaks = c(1.2, 1.5) is equivalent to plot_rhat_breaks = c(1.5, 1.2).

2. plot_rhat_cols: A vector of the same length as plot_rhat_breaks that indicates the colors for the breaks.

3. plot_rhat_nan_col: The color for Rhat when it is Inf or NaN.

4. plot_rhat_large_col: The color for Rhat when it is larger than the largest value of plot_rhat_breaks.

5. rstan_alert_col: The color used in method plot of S4 class stanfit to show that the vector/array parameters are truncated.

6. rstan_chain_cols: The colors used in methods plot and traceplot of S4 class stanfit for coloring different chains.

7. rstan_warmup_bg_col: The background color for the warmup area in the traceplots.

8. boost_lib: The path for the Boost C++ library used to compile Stan models. This option is valid for the whole R session if not changed again.

9. eigen_lib: The path for the Eigen C++ library used to compile Stan models. This option is valid for the whole R session if not changed again.

10. auto_write: A logical scalar (defaulting to FALSE) that controls whether a compiled instance of a stanmodel-class is written to the hard disk in the same directory as the .stan program.

11. threads_per_chain: A positive integer (defaulting to 1). If the model was compiled with threading support, the number of threads to use in parallelized sections _within_ an MCMC chain (e.g., when using the Stan functions 'reduce_sum()' or 'map_rect()'). The actual number of CPU cores used is 'chains * threads_per_chain' where 'chains' is the number of parallel chains. For an example of using threading, see the Stan case study [Reduce Sum: A Minimal Example](https://mc-stan.org/users/documentation/case-studies/reduce_sum_tutorial.html).

Value

The values as a list for existing options and NA for non-existent options. When only one option is specified, its old value is returned.

| sampling | *Draw samples from a Stan model* |

Description

Draw samples from the model defined by class stanmodel.

Usage

```
   ## S4 method for signature 'stanmodel'
sampling(object, data = list(), pars = NA,
    chains = 4, iter = 2000, warmup = floor(iter/2), thin = 1,
    seed = sample.int(.Machine$integer.max, 1),
    init = 'random', check_data = TRUE,
    sample_file = NULL, diagnostic_file = NULL, verbose = FALSE,
    algorithm = c("NUTS", "HMC", "Fixed_param"),
    control = NULL, include = TRUE,
    cores = getOption("mc.cores", 1L),
    open_progress = interactive() && !isatty(stdout()) &&
                    !identical(Sys.getenv("RSTUDIO"), "1"),
    show_messages = TRUE, ...)
```

Arguments

object	An object of class [stanmodel]().
data	A named list or environment providing the data for the model or a character vector for all the names of objects used as data. See the **Passing data to Stan** section in [stan]().
pars	A vector of character strings specifying parameters of interest. The default is NA indicating all parameters in the model. If include = TRUE, only samples for parameters named in pars are stored in the fitted results. Conversely, if include = FALSE, samples for all parameters *except* those named in pars are stored in the fitted results.
chains	A positive integer specifying the number of Markov chains. The default is 4.
iter	A positive integer specifying the number of iterations for each chain (including warmup). The default is 2000.
warmup	A positive integer specifying the number of warmup (aka burnin) iterations per chain. If step-size adaptation is on (which it is by default), this also controls the number of iterations for which adaptation is run (and hence these warmup samples should not be used for inference). The number of warmup iterations should be smaller than iter and the default is iter/2.
thin	A positive integer specifying the period for saving samples. The default is 1, which is usually the recommended value.
seed	The seed for random number generation. The default is generated from 1 to the maximum integer supported by R on the machine. Even if multiple chains are used, only one seed is needed, with other chains having seeds derived from that of the first chain to avoid dependent samples. When a seed is specified by a number, as.integer will be applied to it. If as.integer produces NA, the seed is generated randomly. The seed can also be specified as a character string of digits, such as "12345", which is converted to integer.
init	Initial values specification. See the detailed documentation for the init argument in [stan]().
check_data	Logical, defaulting to TRUE. If TRUE the data will be preprocessed; otherwise not. See the **Passing data to Stan** section in [stan]().

sample_file	An optional character string providing the name of a file. If specified the draws for *all* parameters and other saved quantities will be written to the file. If not provided, files are not created. When the folder specified is not writable, tempdir() is used. When there are multiple chains, an underscore and chain number are appended to the file name prior to the .csv extension.
diagnostic_file	
	An optional character string providing the name of a file. If specified the diagnostics data for *all* parameters will be written to the file. If not provided, files are not created. When the folder specified is not writable, tempdir() is used. When there are multiple chains, an underscore and chain number are appended to the file name prior to the .csv extension.
verbose	TRUE or FALSE: flag indicating whether to print intermediate output from Stan on the console, which might be helpful for model debugging.
algorithm	One of sampling algorithms that are implemented in Stan. Current options are "NUTS" (No-U-Turn sampler, Hoffman and Gelman 2011, Betancourt 2017), "HMC" (static HMC), or "Fixed_param". The default and preferred algorithm is "NUTS".
control	A named list of parameters to control the sampler's behavior. See the details in the documentation for the control argument in [stan]().
include	Logical scalar defaulting to TRUE indicating whether to include or exclude the parameters given by the pars argument. If FALSE, only entire multidimensional parameters can be excluded, rather than particular elements of them.
cores	Number of cores to use when executing the chains in parallel, which defaults to 1 but we recommend setting the mc.cores option to be as many processors as the hardware and RAM allow (up to the number of chains).
open_progress	Logical scalar that only takes effect if cores > 1 but is recommended to be TRUE in interactive use so that the progress of the chains will be redirected to a file that is automatically opened for inspection. For very short runs, the user might prefer FALSE.
show_messages	Either a logical scalar (defaulting to TRUE) indicating whether to print the summary of Informational Messages to the screen after a chain is finished or a character string naming a path where the summary is stored. Setting to FALSE is not recommended unless you are very sure that the model is correct up to numerical error.
...	Additional arguments can be chain_id, init_r, test_grad, append_samples, refresh, enable_random_init. See the documentation in [stan]().

Value

An object of S4 class stanfit representing the fitted results. Slot mode for this object indicates if the sampling is done or not.

Methods

sampling signature(object = "stanmodel") Call a sampler (NUTS, HMC, or Fixed_param depending on parameters) to draw samples from the model defined by S4 class stanmodel given the data, initial values, etc.

See Also

stanmodel, stanfit, stan

Examples

```
## Not run:
m <- stan_model(model_code = 'parameters {real y;} model {y ~ normal(0,1);}')
f <- sampling(m, iter = 100)

## End(Not run)
```

---

sbc   *Simulation Based Calibration (sbc)*

Description

Check whether a model is well-calibrated with respect to the prior distribution and hence possibly amenable to obtaining a posterior distribution conditional on observed data.

Usage

```
   sbc(stanmodel, data, M, ..., save_progress, load_incomplete=FALSE)
   ## S3 method for class 'sbc'
plot(x, thin = 3, ...)
   ## S3 method for class 'sbc'
print(x, ...)
```

Arguments

stanmodel	An object of stanmodel-class that is first created by calling the stan_model function
data	A named `list` or `environment` providing the data for the model, or a character vector for all the names of objects to use as data. This is the same format as in stan or sampling.
M	The number of times to condition on draws from the prior predictive distribution
...	Additional arguments that are passed to sampling, such as `refresh = 0` when calling sbc. For the plot and print methods, the additional arguments are not used.
x	An object produced by sbc
thin	An integer vector of length one indicating the thinning interval when plotting, which defaults to 3
save_progress	If a directory is provided, stanfit objects are saved to disk making it easy to resume a partial sbc run after interruption.
load_incomplete	
	When `save_progress` is used, load whatever runs have been saved to disk and ignore argument M.

Details

This function assumes adherence to the following conventions in the underlying Stan program:

1. Realizations of the unknown parameters are drawn in the `transformed data` block of the Stan program and are postfixed with an underscore, such as `theta_`. These are considered the "true" parameters being estimated by the corresponding symbol declared in the `parameters` block, which should have the same name except for the trailing underscore, such as `theta`.

2. The realizations of the unknown parameters are then conditioned on when drawing from the prior predictive distribution, also in the `transformed data` block. There is no restriction on the symbol name that holds the realizations from the prior predictive distribution but for clarity, it should not end with a trailing underscore.

3. The realizations of the unknown parameters should be copied into a `vector` in the `generated quantities` block named `pars_`.

4. The realizations from the prior predictive distribution should be copied into an object (of the same type) in the `generated quantities` block named `y_`. Technically, this step is optional and could be omitted to conserve RAM, but inspecting the realizations from the prior predictive distribution is a good way to judge whether the priors are reasonable.

5. The `generated quantities` block must contain an integer array named `ranks_` whose only values are zero or one, depending on whether the realization of a parameter from the posterior distribution exceeds the corresponding "true" realization, such as `theta > theta_;`. These are not actually "ranks" but can be used afterwards to reconstruct (thinned) ranks.

6. The `generated quantities` block may contain a vector named `log_lik` whose values are the contribution to the log-likelihood by each observation. This is optional but facilitates calculating Pareto k shape parameters to judge whether the posterior distribution is sensitive to particular observations.

Although the user can pass additional arguments to sampling through the . . . , the following arguments are hard-coded and should not be passed through the . . . :

1. `pars = "ranks_"` because nothing else needs to be stored for each posterior draw

2. `include = TRUE` to ensure that `"ranks_"` is included rather than excluded

3. `chains = 1` because only one chain is run for each integer less than `M`

4. `seed` because a sequence of seeds is used across the `M` runs to preserve independence across runs

5. `save_warmup = FALSE` because the warmup realizations are not relevant

6. `thin = 1` because thinning can and should be done after the Markov Chain is finished, as is done by the `thin` argument to the `plot` method in order to make the histograms consist of approximately independent realizations

Other arguments will take the default values used by sampling unless passed through the . . . . Specifying `refresh = 0` is recommended to avoid printing a lot of intermediate progress reports to the screen. It may be necessary to pass a list to the `control` argument of sampling with elements `adapt_delta` and / or `max_treedepth` in order to obtain adequate results.

Ideally, users would want to see the absence of divergent transitions (which is shown by the `print` method) and other warnings, plus an approximately uniform histogram of the ranks for each parameter (which are shown by the `plot` method). See the vignette for more details.

Value

The sbc function outputs a list of S3 class "sbc", which contains the following elements:

1. ranks A list of M matrices, each with number of rows equal to the number of saved iterations and number of columns equal to the number of unknown parameters. These matrices contain the realizations of the ranks_ object from the generated quantities block of the Stan program.

2. Y If present, a matrix of realizations from the prior predictive distribution whose rows are equal to the number of observations and whose columns are equal to M, which are taken from the y_ object in the generated quantities block of the Stan program.

3. pars A matrix of realizations from the prior distribution whose rows are equal to the number of parameters and whose columns are equal to M, which are taken from the pars_ object in the generated quantities block of the Stan program.

4. pareto_k A matrix of Pareto k shape parameter estimates or NULL if there is no log_lik symbol in the generated quantities block of the Stan program

5. sampler_params A three-dimensional array that results from combining calls to get_sampler_params for each of the M runs. The resulting matrix has rows equal to the number of post-warmup iterations, columns equal to six, and M floors. The columns are named "accept_stat__", "stepsize__", "treedepth__", "n_leapfrog__", "divergent__", and "energy__". The most important of which is "divergent__", which should be all zeros and perhaps "treedepth__", which should only rarely get up to the value of max_treedepth passed as an element of the control list to sampling or otherwise defaults to 10.

The print method outputs the number of divergent transitions and returns NULL invisibly. The plot method returns a ggplot object with histograms whose appearance can be further customized.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual.* https://mc-stan.org.

Talts, S., Betancourt, M., Simpson, D., Vehtari, A., and Gelman, A. (2018). Validating Bayesian Inference Algorithms with Simulation-Based Calibration. arXiv preprint arXiv:1804.06788. https://arxiv.org/abs/1804.06788

See Also

stan_model and sampling

Examples

```
scode <- "
data {
  int<lower = 1> N;
  real<lower = 0> a;
  real<lower = 0> b;
}
transformed data { // these adhere to the conventions above
  real pi_ = beta_rng(a, b);
  int y = binomial_rng(N, pi_);
```

```
}
parameters {
  real<lower = 0, upper = 1> pi;
}
model {
  target += beta_lpdf(pi | a, b);
  target += binomial_lpmf(y | N, pi);
}
generated quantities { // these adhere to the conventions above
  int y_ = y;
  vector[1] pars_;
  int ranks_[1] = {pi > pi_};
  vector[N] log_lik;
  pars_[1] = pi_;
  for (n in 1:y) log_lik[n] = bernoulli_lpmf(1 | pi);
  for (n in (y + 1):N) log_lik[n] = bernoulli_lpmf(0 | pi);
}
"
```

---

set_cppo   *Defunct function to set the compiler optimization level*

Description

This function returns nothing and does nothing except throw a warning. See https://cran.r-project.org/doc/manuals/r-release/R-admin.html#Customizing-package-compilation for information on customizing the compiler options, but doing so should be unnecessary for normal useage.

Usage

```
set_cppo(...)
```

Arguments

...   Any input is ignored

Value

An invisible NULL

---

sflist2stanfit *Merge a list of stanfit objects into one*

Description

This function takes a list of stanfit objects and returns a consolidated stanfit object. The stanfit objects to be merged need to have the same configuration of iteration, warmup, and thin, besides being from the same model. This could facilitate some parallel usage of RStan. For example, if we call stan by parallel and it returns a list of stanfit objects, this function can be used to create one stanfit object from the list.

Usage

```
sflist2stanfit(sflist)
```

Arguments

sflist   A list of stanfit objects.

Value

An S4 object of stanfit consolidated from all the input stanfit objects.

Note

This function should be called in rare circumstances because sampling has a cores argument that allows multiple chains to be executed in parallel. However, if you need to depart from that, the best practice is to use sflist2stanfit on a list of stanfit objects created with the same seed but different chain_id (see example below). Using the same seed but different chain_id can make sure the random number generations for all chains are not correlated.

This function would do some check to see if the stanfit objects in the input list can be merged. But the check is not sufficient. So generally, it is the user's responsibility to make sure the input is correct so that the merging makes sense.

The date in the new stanfit object is when it is merged.

get_seed function for the new consolidated stanfit object only returns the seed used in the first chain of the new object.

The sampler such as NUTS2 that is displayed in the printout by print is the sampler used for the first chain. The print method assumes the samplers are the same for all chains.

The included stanmodel object, which includes the compiled model, in the new stanfit object is from the first element of the input list.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual*. https://mc-stan.org/.

See Also

stan

Examples

```
## Not run:
library(rstan)
scode <- "
data {
  int<lower=1> N;
}
parameters {
  array[N] real y1;
  array[N] real y2;
}
model {
  y1 ~ normal(0, 1);
  y2 ~ double_exponential(0, 2);
}
"
seed <- 123 # or any other integer
foo_data <- list(N = 2)
foo <- stan(model_code = scode, data = foo_data, chains = 1, iter = 1)
f1 <- stan(fit = foo, data = foo_data, chains = 1, seed = seed, chain_id = 1)
f2 <- stan(fit = foo, data = foo_data, chains = 2, seed = seed, chain_id = 2:3)
f12 <- sflist2stanfit(list(f1, f2))

## parallel stan call for unix-like OS
library(parallel)

if (.Platform$OS.type == "unix") {
sflist1 <-
  mclapply(1:4, mc.cores = 2,
           function(i) stan(fit = foo, data = foo_data, seed = seed,
                      chains = 1, chain_id = i, refresh = -1))
f3 <- sflist2stanfit(sflist1)
}
if (.Platform$OS.type == "windows") { # also works on non-Windows
CL <- makeCluster(2)
clusterExport(cl = CL, c("foo_data", "foo", "seed"))
sflist1 <- parLapply(CL, 1:4, fun = function(cid) {
  require(rstan)
  stan(fit = foo, data = foo_data, chains = 1,
       iter = 2000, seed = seed, chain_id = cid)
})

fit <- sflist2stanfit(sflist1)
print(fit)
stopCluster(CL)
} # end example for Windows

## End(Not run)
```

---

stan   *Fit a model with Stan*

Description

Fit a model defined in the Stan modeling language and return the fitted result as an instance of
`stanfit`.

Usage

```
stan(file, model_name = "anon_model", model_code = "", fit = NA,
  data = list(), pars = NA,
  chains = 4, iter = 2000, warmup = floor(iter/2), thin = 1,
  init = "random", seed = sample.int(.Machine$integer.max, 1),
  algorithm = c("NUTS", "HMC", "Fixed_param"),
  control = NULL, sample_file = NULL, diagnostic_file = NULL,
  save_dso = TRUE, verbose = FALSE, include = TRUE,
  cores = getOption("mc.cores", 1L),
  open_progress = interactive() && !isatty(stdout()) &&
                  !identical(Sys.getenv("RSTUDIO"), "1"),
  ...,
  boost_lib = NULL, eigen_lib = NULL
  )
```

Arguments

`file`	The path to the Stan program to use. `file` should be a character string file name or a connection that R supports containing the text of a model specification in the Stan modeling language.
	A model may also be specified directly as a character string using the `model_code` argument, but we recommend always putting Stan programs in separate files with a `.stan` extension.
	The `stan` function can also use the Stan program from an existing `stanfit` object via the `fit` argument. When `fit` is specified, the `file` argument is ignored.
`model_code`	A character string either containing the model definition or the name of a character string object in the workspace. This argument is used only if arguments `file` and `fit` are not specified.
`fit`	An instance of S4 class `stanfit` derived from a previous fit; defaults to NA. If `fit` is not NA, the compiled model associated with the fitted result is re-used; thus the time that would otherwise be spent recompiling the C++ code for the model can be saved.
`model_name`	A string to use as the name of the model; defaults to `"anon_model"`. However, the model name will be derived from `file` or `model_code` (if `model_code` is the name of a character string object) if `model_name` is not specified. This is not a particularly important argument, although since it affects the name used in printed messages, developers of other packages that use **rstan** to fit models may want to use informative names.

data	A named `list` or `environment` providing the data for the model, or a character vector for all the names of objects to use as data. See the **Passing data to Stan** section below.
pars	A character vector specifying parameters of interest to be saved. The default is to save all parameters from the model. If `include = TRUE`, only samples for parameters named in `pars` are stored in the fitted results. Conversely, if `include = FALSE`, samples for all parameters *except* those named in `pars` are stored in the fitted results.
include	Logical scalar defaulting to `TRUE` indicating whether to include or exclude the parameters given by the `pars` argument. If `FALSE`, only entire multidimensional parameters can be excluded, rather than particular elements of them.
iter	A positive integer specifying the number of iterations for each chain (including warmup). The default is 2000.
warmup	A positive integer specifying the number of warmup (aka burnin) iterations per chain. If step-size adaptation is on (which it is by default), this also controls the number of iterations for which adaptation is run (and hence these warmup samples should not be used for inference). The number of warmup iterations should be smaller than `iter` and the default is `iter/2`.
chains	A positive integer specifying the number of Markov chains. The default is 4.
cores	The number of cores to use when executing the Markov chains in parallel. The default is to use the value of the `"mc.cores"` option if it has been set and otherwise to default to 1 core. However, we recommend setting it to be as many processors as the hardware and RAM allow (up to the number of chains). See detectCores if you don't know this number for your system.
thin	A positive integer specifying the period for saving samples. The default is 1, which is usually the recommended value. Unless your posterior distribution takes up too much memory we do *not* recommend thinning as it throws away information. The tradition of thinning when running MCMC stems primarily from the use of samplers that require a large number of iterations to achieve the desired effective sample size. Because of the efficiency (effective samples per second) of Hamiltonian Monte Carlo, rarely should this be necessary when using Stan.
init	Specification of initial values for all or some parameters. Can be the digit 0, the strings `"0"` or `"random"`, a function that returns a named list, or a list of named lists:

init="random" **(default):** Let Stan generate random initial values for all parameters. The seed of the random number generator used by Stan can be specified via the `seed` argument. If the seed for Stan is fixed, the same initial values are used. The default is to randomly generate initial values between -2 and 2 *on the unconstrained support*. The optional additional parameter `init_r` can be set to some value other than 2 to change the range of the randomly generated inits.

init="0", init=0: Initialize all parameters to zero on the unconstrained support.

inits via list: Set inital values by providing a list equal in length to the number of chains. The elements of this list should themselves be named lists, where

each of these named lists has the name of a parameter and is used to specify the initial values for that parameter for the corresponding chain.

inits via function: Set initial values by providing a function that returns a list for specifying the initial values of parameters for a chain. The function can take an optional parameter chain_id through which the chain_id (if specified) or the integers from 1 to chains will be supplied to the function for generating initial values. See the **Examples** section below for examples of defining such functions and using a list of lists for specifying initial values.

When specifying initial values via a list or function, any parameters for which values are not specified will receive initial values generated as described in the init="random" description above.

seed   The seed for random number generation. The default is generated from 1 to the maximum integer supported by R on the machine. Even if multiple chains are used, only one seed is needed, with other chains having seeds derived from that of the first chain to avoid dependent samples. When a seed is specified by a number, as.integer will be applied to it. If as.integer produces NA, the seed is generated randomly. The seed can also be specified as a character string of digits, such as "12345", which is converted to integer.

Using R's set.seed function to set the seed for Stan will not work.

algorithm   One of the sampling algorithms that are implemented in Stan. The default and preferred algorithm is "NUTS", which is the No-U-Turn sampler variant of Hamiltonian Monte Carlo (Hoffman and Gelman 2011, Betancourt 2017). Currently the other options are "HMC" (Hamiltonian Monte Carlo), and "Fixed_param". When "Fixed_param" is used no MCMC sampling is performed (e.g., for simulating with in the generated quantities block).

sample_file   An optional character string providing the name of a file. If specified the draws for *all* parameters and other saved quantities will be written to the file. If not provided, files are not created. When the folder specified is not writable, tempdir() is used. When there are multiple chains, an underscore and chain number are appended to the file name.

diagnostic_file

An optional character string providing the name of a file. If specified the diagnostics data for *all* parameters will be written to the file. If not provided, files are not created. When the folder specified is not writable, tempdir() is used. When there are multiple chains, an underscore and chain number are appended to the file name.

save_dso   Logical, with default TRUE, indicating whether the dynamic shared object (DSO) compiled from the C++ code for the model will be saved or not. If TRUE, we can draw samples from the same model in another R session using the saved DSO (i.e., without compiling the C++ code again). This parameter only takes effect if fit is not used; with fit defined, the DSO from the previous run is used. When save_dso=TRUE, the fitted object can be loaded from what is saved previously and used for sampling, if the compiling is done on the same platform, that is, same operating system and same architecture (32bits or 64bits).

verbose   TRUE or FALSE: flag indicating whether to print intermediate output from Stan on the console, which might be helpful for model debugging.

control
A named `list` of parameters to control the sampler's behavior. It defaults to `NULL` so all the default values are used. First, the following are adaptation parameters for sampling algorithms. These are parameters used in Stan with similar names here.

- `adapt_engaged` (logical)
- `adapt_gamma` (double, positive, defaults to 0.05)
- `adapt_delta` (double, between 0 and 1, defaults to 0.8)
- `adapt_kappa` (double, positive, defaults to 0.75)
- `adapt_t0` (double, positive, defaults to 10)
- `adapt_init_buffer` (integer, positive, defaults to 75)
- `adapt_term_buffer` (integer, positive, defaults to 50)
- `adapt_window` (integer, positive, defaults to 25)

In addition, algorithm HMC (called 'static HMC' in Stan) and NUTS share the following parameters:

- `stepsize` (double, positive, defaults to 1) Note: this controls the *initial* stepsize only, unless adapt_engaged=FALSE.
- `stepsize_jitter` (double, [0,1], defaults to 0)
- `metric` (string, one of "unit_e", "diag_e", "dense_e", defaults to "diag_e")

For algorithm NUTS, we can also set:

- `max_treedepth` (integer, positive, defaults to 10)

For algorithm HMC, we can also set:

- `int_time` (double, positive)

For `test_grad` mode, the following parameters can be set:

- `epsilon` (double, defaults to 1e-6)
- `error` (double, defaults to 1e-6)

open_progress
Logical scalar that only takes effect if `cores` > 1 but is recommended to be `TRUE` in interactive use so that the progress of the chains will be redirected to a file that is automatically opened for inspection. For very short runs, the user might prefer `FALSE`.

...
Other optional parameters:

- `chain_id` (integer)
- `init_r` (double, positive)
- `test_grad` (logical)
- `append_samples` (logical)
- `refresh`(integer)
- `save_warmup`(logical)
- deprecated: `enable_random_init`(logical)

`chain_id` can be a vector to specify the chain_id for all chains or an integer. For the former case, they should be unique. For the latter, the sequence of integers starting from the given `chain_id` are used for all chains.

`init_r` is used only for generating random initial values, specifically when init="random" or not all parameters are initialized in the user-supplied list

or function. If specified, the initial values are simulated uniformly from inter-
val [-init_r, init_r] rather than using the default interval (see the manual of
(cmd)Stan).

test_grad (logical). If test_grad=TRUE, Stan will not do any sampling. In-
stead, the gradient calculation is tested and printed out and the fitted stanfit
object is in test gradient mode. By default, it is FALSE.

append_samples (logical). Only relevant if sample_file is specified *and* is
an existing file. In that case, setting append_samples=TRUE will append the
samples to the existing file rather than overwriting the contents of the file.

refresh (integer) can be used to control how often the progress of the sam-
pling is reported (i.e. show the progress every refresh iterations). By default,
refresh = max(iter/10, 1). The progress indicator is turned off if refresh
<= 0.

Deprecated: enable_random_init (logical) being TRUE enables specifying
initial values randomly when the initial values are not fully specified from the
user.

save_warmup (logical) indicates whether to save draws during the warmup
phase and defaults to TRUE. Some memory related problems can be avoided by
setting it to FALSE, but some diagnostics are more limited if the warmup draws
are not stored.

boost_lib   The path for an alternative version of the Boost C++ to use instead of the one in
                   the **BH** package.

eigen_lib   The path for an alternative version of the Eigen C++ library to the one in **RcppEigen**.

Details

The stan function does all of the work of fitting a Stan model and returning the results as an instance
of stanfit. The steps are roughly as follows:

1. Translate the Stan model to C++ code. ([stanc](#))

2. Compile the C++ code into a binary shared object, which is loaded into the current R session
   (an object of S4 class stanmodel is created). ([stan_model](#))

3. Draw samples and wrap them in an object of S4 class stanfit. ([sampling](#))

The returned object can be used with methods such as print, summary, and plot to inspect and
retrieve the results of the fitted model.

stan can also be used to sample again from a fitted model under different settings (e.g., different
iter, data, etc.) by using the fit argument to specify an existing stanfit object. In this case, the
compiled C++ code for the model is reused.

Value

An object of S4 class [stanfit](#). However, if cores > 1 and there is an error for any of the chains,
then the error(s) are printed. If all chains have errors and an error occurs before or during sampling,
the returned object does not contain samples. But the compiled binary object for the model is still
included, so we can reuse the returned object for another sampling.

Passing data to Stan

The data passed to stan are preprocessed before being passed to Stan. If data is not a character vector, the data block of the Stan program is parsed and R objects of the same name are searched starting from the calling environment. Then, if data is list-like but not a data.frame the elements of data take precedence. This behavior is similar to how a formula is evaluated by the lm function when data is supplied. In general, each R object being passed to Stan should be either a numeric vector (including the special case of a 'scalar') or a numeric array (matrix). The first exception is that an element can be a logical vector: TRUE's are converted to 1 and FALSE's to 0. An element can also be a data frame or a specially structured list (see details below), both of which will be converted into arrays in the preprocessing. Using a specially structured list is not encouraged though it might be convenient sometimes; and when in doubt, just use arrays.

This preprocessing for each element mainly includes the following:

1. Change the data of type from double to integer if no accuracy is lost. The main reason is that by default, R uses double as data type such as in a <- 3. But Stan will not read data of type int from real and it reads data from int if the data type is declared as real.

2. Check if there is NA in the data. Unlike BUGS, Stan does not allow missing data. Any NA values in supplied data will cause the function to stop and report an error.

3. Check data types. Stan allows only numeric data, that is, doubles, integers, and arrays of these. Data of other types (for example, characters and factors) are not passed to Stan.

4. Check whether there are objects in the data list with duplicated names. Duplicated names, if found, will cause the function to stop and report an error.

5. Check whether the names of objects in the data list are legal Stan names. If illegal names are found, it will stop and report an error. See (Cmd)Stan's manual for the rules of variable names.

6. When an element is of type data.frame, it will be converted to matrix by function data.matrix.

7. When an element is of type list, it is supposed to make it easier to pass data for those declared in Stan code such as "vector[J] y1[I]" and "matrix[J,K] y2[I]". Using the latter as an example, we can use a list for y2 if the list has "I" elements, each of which is an array (matrix) of dimension "J*K". However, it is not possible to pass a list for data declared such as "vector[K] y3[I,J]"; the only way for it is to use an array with dimension "I*J*K". In addition, technically a data.frame in R is also a list, but it should not be used for the purpose here since a data.frame will be converted to a matrix as described above.

Stan treats a vector of length 1 in R as a scalar. So technically if, for example, "array[1] real y;" is defined in the data block, an array such as "y = array(1.0, dim = 1)" in R should be used. This is also the case for specifying initial values since the same underlying approach for reading data from R in Stan is used, in which vector of length 1 is treated as a scalar.

In general, the higher the optimization level is set, the faster the generated binary code for the model runs, which can be set in a Makevars file. However, the binary code generated for the model runs fast by using a higher optimization level at the cost of longer times to compile the C++ code.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual*. https://mc-stan.org.

The Stan Development Team *CmdStan Interface User's Guide*. https://mc-stan.org.

See Also

- The package vignettes for an example of fitting a model and accessing the contents of stanfit objects (https://mc-stan.org/rstan/articles/).

- stanc for translating model code in Stan modeling language to C++, sampling for sampling, and stanfit for the fitted results.

- as.array.stanfit and extract for extracting samples from stanfit objects.

Examples

```
## Not run:
#### example 1
library(rstan)
scode <- "
parameters {
  array[2] real y;
}
model {
  y[1] ~ normal(0, 1);
  y[2] ~ double_exponential(0, 2);
}
"
fit1 <- stan(model_code = scode, iter = 10, verbose = FALSE)
print(fit1)
fit2 <- stan(fit = fit1, iter = 10000, verbose = FALSE)

## using as.array on the stanfit object to get samples
a2 <- as.array(fit2)

## extract samples as a list of arrays
e2 <- extract(fit2, permuted = FALSE)

#### example 2
#### the result of this package is included in the package

excode <- '
  transformed data {
    array[20] real y;
    y[1] = 0.5796;  y[2] = 0.2276;   y[3]  = -0.2959;
    y[4] = -0.3742; y[5] = 0.3885;   y[6]  = -2.1585;
    y[7] = 0.7111;  y[8] = 1.4424;   y[9]  = 2.5430;
    y[10] = 0.3746; y[11] = 0.4773;  y[12] = 0.1803;
    y[13] = 0.5215; y[14] = -1.6044; y[15] = -0.6703;
    y[16] = 0.9459; y[17] = -0.382;  y[18] = 0.7619;
    y[19] = 0.1006; y[20] = -1.7461;
  }
  parameters {
    real mu;
    real<lower=0, upper=10> sigma;
    vector[2] z[3];
    real<lower=0> alpha;
  }
```

```
  model {
    y ~ normal(mu, sigma);
    for (i in 1:3)
      z[i] ~ normal(0, 1);
    alpha ~ exponential(2);
  }
'

exfit <- stan(model_code = excode, save_dso = FALSE, iter = 500)
print(exfit)
plot(exfit)

## End(Not run)
## Not run:
## examples of specify argument `init` for function stan

## define a function to generate initial values that can
## be fed to function stan's argument `init`
# function form 1 without arguments
initf1 <- function() {
  list(mu = 1, sigma = 4, z = array(rnorm(6), dim = c(3,2)), alpha = 1)
}
# function form 2 with an argument named `chain_id`
initf2 <- function(chain_id = 1) {
  # cat("chain_id =", chain_id, "\n")
  list(mu = 1, sigma = 4, z = array(rnorm(6), dim = c(3,2)), alpha = chain_id)
}

# generate a list of lists to specify initial values
n_chains <- 4
init_ll <- lapply(1:n_chains, function(id) initf2(chain_id = id))

exfit0 <- stan(model_code = excode, init = initf1)
stan(fit = exfit0, init = initf2)
stan(fit = exfit0, init = init_ll, chains = n_chains)

## End(Not run)
```

| stanc | *Translate Stan model specification to C++ code* |

Description

Translate a model specification in Stan code to C++ code, which can then be compiled and loaded for sampling.

Usage

```
stanc(
  file, model_code = '', model_name = "anon_model", verbose = FALSE,
```

```
    obfuscate_model_name = TRUE,
    allow_undefined = isTRUE(getOption("stanc.allow_undefined", FALSE)),
  allow_optimizations = isTRUE(getOption("stanc.allow_optimizations", FALSE)),
  standalone_functions = isTRUE(getOption("stanc.standalone_functions", FALSE)),
    use_opencl = isTRUE(getOption("stanc.use_opencl", FALSE)),
    warn_pedantic = isTRUE(getOption("stanc.warn_pedantic", FALSE)),
    warn_uninitialized = isTRUE(getOption("stanc.warn_uninitialized", FALSE)),
    isystem = c(if (!missing(file)) dirname(file), getwd()))
  stanc_builder(
    file, isystem = c(dirname(file), getwd()),
    verbose = FALSE, obfuscate_model_name = FALSE,
    allow_undefined = isTRUE(getOption("stanc.allow_undefined", FALSE)),
  allow_optimizations = isTRUE(getOption("stanc.allow_optimizations", FALSE)),
  standalone_functions = isTRUE(getOption("stanc.standalone_functions", FALSE)),
    use_opencl = isTRUE(getOption("stanc.use_opencl", FALSE)),
    warn_pedantic = isTRUE(getOption("stanc.warn_pedantic", FALSE)),
    warn_uninitialized = isTRUE(getOption("stanc.warn_uninitialized", FALSE)))
```

Arguments

`file`	A character string or a connection that R supports specifying the Stan model specification in Stan's modeling language.
`model_code`	Either a character string containing a Stan model specification or the name of a character string object in the workspace. This parameter is used only if parameter `file` is not specified, so it defaults to the empty string.
`model_name`	A character string naming the model. The default is `"anon_model"`. However, the model name will be derived from `file` or `model_code` (if `model_code` is the name of a character string object) if `model_name` is not specified.
`verbose`	Logical, defaulting to `FALSE`. If `TRUE` more intermediate information is printed during the translation procedure.
`obfuscate_model_name`	
	Logical, defaulting to `TRUE`, indicating whether to use a randomly-generated character string for the name of the C++ class. This prevents name clashes when compiling multiple models in the same R session.
`isystem`	A character vector naming a path to look for file paths in `file` that are to be included within the Stan program named by `file`. See the Details section below.
`allow_undefined`	
	A logical scalar defaulting to `FALSE` indicating whether to allow Stan functions to be declared but not defined in `file` or `model_code`. If `TRUE`, then it is the caller's responsibility to provide a function definition in another header file or linked shared object.
`allow_optimizations`	
	A logical scalar defaulting to `FALSE` indicating whether to allow level-1 compiler optimization to the **Stan** code.
`standalone_functions`	
	A logical scalar defaulting to `FALSE` indicating whether to generate the standalone functions **C++** code.

use_opencl	A logical scalar defaulting to FALSE indicating whether to try to use matrix_cl signatures.
warn_pedantic	A logical scalar defaulting to FALSE indicating whether to emit warnings about common mistakes in **Stan** programs.
warn_uninitialized	
	A logical scalar defaulting to FALSE indicating whether emit warnings about uninitialized variables.

Details

The stanc_builder function supports the standard C++ convention of specifying something like #include "my_includes.txt" on an entire line within the file named by the file argument. In other words, stanc_builder would look for "my_includes.txt" in (or under) the directories named by the isystem argument and — if found — insert its contents verbatim at that position before calling stanc on the resulting model_code. This mechanism reduces the need to copy common chunks of code across Stan programs. It is possible to include such files recursively.

Note that line numbers referred to in parser warnings or errors refer to the postprocessed Stan program rather than file. In the case of a parser error, the postprocessed Stan program will be printed after the error message. Line numbers referred to in messages while Stan is executing also refer to the postprocessed Stan program which can be obtained by calling get_stancode.

Value

A list with named entries:

1. model_name Character string for the model name.

2. model_code Character string for the model's Stan specification.

3. cppcode Character string for the model's C++ code.

4. status Logical indicating success/failure (always TRUE) of translating the Stan code.

Note

Unlike R, in which variable identifiers may contain dots (e.g. a.1), Stan prohibits dots from occurring in variable identifiers. Furthermore, C++ reserved words and Stan reserved words may not be used for variable names; see the Stan User's Guide for a complete list.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual*. https://mc-stan.org/.

The Stan Development Team *CmdStan Interface User's Guide*. https://mc-stan.org.

See Also

stan_model and stan

Examples

```
stanmodelcode <- "
data {
  int<lower=0> N;
  array[N] real y;
}

parameters {
  real mu;
}

model {
  mu ~ normal(0, 10);
  y ~ normal(mu, 1);
}
"

r <- stanc(model_code = stanmodelcode, model_name = "normal1")
str(r)
```

---

stanfit-class   *Class* stanfit*: fitted Stan model*

Description

The components (slots) of a stanfit object and the various available methods are described below. When methods have their own more detailed documentation pages links are provided.

Objects from the Class

An object of class stanfit contains the output derived from fitting a Stan model as returned by the top-level function [stan](#) or the lower-level methods [sampling](#) and [vb](#) (which are defined on class [stanmodel](#)). Many methods (e.g., print, plot, summary) are provided for summarizing results and various access methods also allow the underlying data (e.g., simulations, diagnostics) contained in the object to be retrieved.

Slots

model_name: The model name as a string.

model_pars: A character vector of names of parameters (including transformed parameters and derived quantities).

par_dims: A named list giving the dimensions for all parameters. The dimension for a scalar parameter is given as numeric(0).

mode: An integer indicating the mode of the fitted model. 0 indicates sampling mode, 1 indicates test gradient mode (no sampling is done), and 2 indicates error mode (an error occurred before sampling). Most methods for stanfit objects are useful only if mode=0.

sim: A list containing simulation results including the posterior draws as well as various pieces of metadata used by many of the methods for stanfit objects.

inits: The initial values (either user-specified or generated randomly) for all chains. This is a list with one component per chain. Each component is a named list containing the initial values for each parameter for the corresponding chain.

stan_args: A list with one component per chain containing the arguments used for sampling (e.g. iter, seed, etc.).

stanmodel: The instance of S4 class stanmodel.

date: A string containing the date and time the object was created.

.MISC: Miscellaneous helper information used for the fitted model. This is an object of type environment. Users rarely (if ever) need to access the contents of .MISC.

Methods

Printing, plotting, and summarizing:

show Print the default summary for the model.

print Print a customizable summary for the model. See print.stanfit.

plot Create various plots summarizing the fitted model. See plot,stanfit-method.

summary Summarize the distributions of estimated parameters and derived quantities using the posterior draws. See summary,stanfit-method.

get_posterior_mean Get the posterior mean for parameters of interest (using pars to specify a subset of parameters). Returned is a matrix with one column per chain and an additional column for all chains combined.

Extracting posterior draws:

extract Extract the draws for all chains for all (or specified) parameters. See extract.

as.array, as.matrix, as.data.frame Coerce the draws (without warmup) to an array, matrix or data frame. See as.array.stanfit.

As.mcmc.list Convert a stanfit object to an mcmc.list as in package **coda**. See As.mcmc.list.

get_logposterior Get the log-posterior at each iteration. Each element of the returned list is the vector of log-posterior values (up to an additive constant, i.e. up to a multiplicative constant on the linear scale) for a single chain. The optional argument inc_warmup (defaulting to TRUE) indicates whether to include the warmup period.

Diagnostics, log probability, and gradients:

get_sampler_params Obtain the parameters used for the sampler such as stepsize and treedepth. The results are returned as a list with one component (an array) per chain. The array has number of columns corresponding to the number of parameters used in the sampler and its column names provide the parameter names. Optional argument inc_warmup (defaulting to TRUE) indicates whether to include the warmup period.

get_adaptation_info Obtain the adaptation information for the sampler if NUTS was used. The results are returned as a list, each element of which is a character string with the info for a single chain.

log_prob Compute the log probability density (lp__) for a set of parameter values (on the *uncon-strained* space) up to an additive constant. The unconstrained parameters are specified using a numeric vector. The number of parameters on the unconstrained space can be obtained using method get_num_upars. A numeric value is returned. See also the documentation in [log_prob](#).

grad_log_prob Compute the gradient of log probability density function for a set of parameter values (on the *unconstrained* space) up to an additive constant. The unconstrained parameters are specified using a numeric vector with the length being the number of unconstrained parameters. A numeric vector is returned with the length of the number of unconstrained parameters and an attribute named log_prob being the lp__. See also the documentation in [grad_log_prob](#).

get_num_upars Get the number of unconstrained parameters of the model. The number of parameters for a model is not necessarily equal to this number of unconstrained parameters. For example, when a parameter is specified as a simplex of length K, the number of unconstrained parameters is K-1.

unconstrain_pars Transform the parameters to unconstrained space. The input is a named list as for specifying initial values for each parameter. A numeric vector is returned. See also the documentation in [unconstrain_pars](#).

constrain_pars Get the parameter values from their unconstrained space. The input is a numeric vector. A list is returned. This function is contrary to unconstrain_pars. See also the documentation in [constrain_pars](#).

Metadata and miscellaneous:

get_stancode Get the Stan code for the fitted model as a string. The result can be printed in a readable format using [cat](#).

get_stanmodel Get the object of S4 class [stanmodel](#) of the fitted model.

get_elapsed_time Get the warmup time and sample time in seconds. A matrix of two columns is returned with each row containing the warmup and sample times for one chain.

get_inits, iter = NULL Get the initial values for parameters used in sampling all chains. The returned object is a list with the same structure as the inits slot described above. If object@mode=2 (error mode) an empty list is returned. If iter is not NULL, then the draw from that iteration is returned for each chain rather than the initial state.

get_cppo_mode Get the optimization mode used for compilation. The returned string is one of "fast", "presentation2", "presentation1", and "debug".

get_seed Get the (P)RNG seed used. When the fitted object is empty (mode=2), NULL might be returned. In the case that the seeds for all chains are different, use get_seeds.

get_seeds Get the seeds used for all chains. When the fitted object is empty (mode=2), NULL might be returned.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual*. https://mc-stan.org.

See Also

[stan](#) and [stanmodel](#)

Examples

```
## Not run:
showClass("stanfit")
ecode <- '
  parameters {
    array[2] real<lower=0> y;
  }
  model {
    y ~ exponential(1);
  }
'
fit <- stan(model_code = ecode, iter = 10, chains = 1)
fit2 <- stan(fit = fit)
print(fit2)
plot(fit2)
traceplot(fit2)
ainfo <- get_adaptation_info(fit2)
cat(ainfo[[1]])
seed <- get_seed(fit2)
sp <- get_sampler_params(fit2)
sp2 <- get_sampler_params(fit2, inc_warmup = FALSE)
head(sp[[1]])

lp <- log_prob(fit, c(1, 2))
grad <- grad_log_prob(fit, c(1, 2))
lp2 <- attr(grad, "log_prob") # should be the same as "lp"

# get the number of parameters on the unconstrained space
n <- get_num_upars(fit)

# parameters on the positive real line (constrained space)
y1 <- list(y = rep(1, 2))

uy <- unconstrain_pars(fit, y1)
## uy should be c(0, 0) since here the log transformation is used
y1star <- constrain_pars(fit, uy)

print(y1)
print(y1star) # y1start should equal to y1

## End(Not run)

# Create a stanfit object from reading CSV files of samples (saved in rstan
# package) generated by funtion stan for demonstration purpose from model as follows.
#
excode <- '
  transformed data {
    array[20] real y;
    y[1] <- 0.5796;  y[2]  <- 0.2276;   y[3] <- -0.2959;
    y[4] <- -0.3742; y[5]  <- 0.3885;   y[6] <- -2.1585;
    y[7] <- 0.7111;  y[8]  <- 1.4424;   y[9] <- 2.5430;
    y[10] <- 0.3746; y[11] <- 0.4773;   y[12] <- 0.1803;
```

```
    y[13] <- 0.5215; y[14] <- -1.6044;  y[15] <- -0.6703;
    y[16] <- 0.9459; y[17] <- -0.382;   y[18] <- 0.7619;
    y[19] <- 0.1006; y[20] <- -1.7461;
  }
  parameters {
    real mu;
    real<lower=0, upper=10> sigma;
    vector[2] z[3];
    real<lower=0> alpha;
  }
  model {
    y ~ normal(mu, sigma);
    for (i in 1:3)
      z[i] ~ normal(0, 1);
    alpha ~ exponential(2);
  }
'

# exfit <- stan(model_code = excode, save_dso = FALSE, iter = 200,
#                sample_file = "rstan_doc_ex.csv")
#

exfit <- read_stan_csv(dir(system.file('misc', package = 'rstan'),
                       pattern='rstan_doc_ex_[[:digit:]].csv',
                       full.names = TRUE))

print(exfit)
## Not run:
plot(exfit)

## End(Not run)

adaptinfo <- get_adaptation_info(exfit)
inits <- get_inits(exfit) # empty
inits <- get_inits(exfit, iter = 101)
seed <- get_seed(exfit)
sp <- get_sampler_params(exfit)
ml <- As.mcmc.list(exfit)
cat(get_stancode(exfit))
```

---

stanmodel-class   *Class representing model compiled from C++*

Description

A stanmodel object represents the model compiled from C++ code. The sampling method defined in this class may be used to draw samples from the model and optimizing method is for obtaining a point estimate by maximizing the log-posterior.

Objects from the Class

Instances of stanmodel are usually created by calling function stan_model or function stan.

Slots

model_name: The model name, an object of type character.

model_code: The Stan model specification, an object of type character.

model_cpp: Object of type list that includes the C++ code for the model.

mk_cppmodule: A function to return a RCpp module. This function will be called in function sampling and optimzing with one argument (the instance of stanmodel itself).

dso: Object of S4 class cxxdso. The container for the dynamic shared objects compiled from the C++ code of the model, returned from function cxxfunction in package **inline**.

Methods

show signature(object = "stanmodel"): print the Stan model specification.

vb signature(object = "stanmodel"): use the variational Bayes algorithms.

sampling signature(object = "stanmodel"): draw samples for the model (see [sampling](#)).

optimizing signature(object = "stanmodel"): obtain a point estimate by maximizing the posterior (see [optimizing](#)).

get_cppcode signature(object = "stanmodel"): returns the C++ code for the model as a character string. This is part of the C++ code that is compiled to the dynamic shared object for the model.

get_stancode signature(object = "stanmodel"): returns the Stan code for the model as a character string

get_cxxflags signature(object = "stanmodel"): return the CXXFLAGS used for compiling the model. The returned string is like CXXFLAGS = -O3.

Note

Objects of class stanmodel can be saved for use across R sessions only if save_dso = TRUE is set during calling functions that create stanmodel objects (e.g., stan and stan_model).

Even if save_dso = TRUE, the model cannot be loaded on a platform (operating system, 32 bits or 64 bits, etc.) that differs from the one on which it was compiled.

See Also

[stan_model](#), [stanc sampling](#), [optimizing](#), [vb](#)

Examples

```
showClass("stanmodel")
```

---

stan_demo   *Demonstrate examples included in Stan*

Description

Stan includes a variety of examples and most of the BUGS example models that are translated into Stan modeling language. One example is chosen from a list created from matching user input and gets fitted in the demonstration.

Usage

```
stan_demo(model = character(0),
          method = c("sampling", "optimizing", "meanfield", "fullrank"), ...)
```

Arguments

model
: A character string for model name to specify which model will be used for demonstration. The default is an empty string, which prompts the user to select one the available models. If model = 0, a character vector of all models is returned without any user intervention. If model = i where i > 0, then the ith available model is chosen without user intervention, which is useful for testing.

method
: Whether to call [sampling](#) (the default), [optimizing](#), or one of the variants of [vb](#) for the demonstration

...
: Further arguments passed to method.

Value

An S4 object of stanfit, unless model = 0, in which case a character vector of paths to available models is returned.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual.* https://mc-stan.org/.

See Also

[sampling](#), [optimizing](#)

Examples

```
## Not run:
   dogsfit <- stan_demo("dogs") # run the dogs model
   fit1 <- stan_demo(1) # run model_names[1]

## End(Not run)
```

---

stan_model | *Construct a Stan model*

Description

Construct an instance of S4 class stanmodel from a model specified in Stan's modeling language. A stanmodel object can then be used to draw samples from the model. The Stan program (the model expressed in the Stan modeling language) is first translated to C++ code and then the C++ code for the model plus other auxiliary code is compiled into a dynamic shared object (DSO) and then loaded. The loaded DSO for the model can be executed to draw samples, allowing inference to be performed for the model and data.

Usage

```
stan_model(
  file, model_name = "anon_model",
  model_code = "", stanc_ret = NULL,
  boost_lib = NULL, eigen_lib = NULL,
  save_dso = TRUE, verbose = FALSE,
  auto_write = rstan_options("auto_write"),
  obfuscate_model_name = TRUE,
  allow_undefined = isTRUE(getOption("stanc.allow_undefined", FALSE)),
  allow_optimizations = isTRUE(getOption("stanc.allow_optimizations", FALSE)),
  standalone_functions = isTRUE(getOption("stanc.standalone_functions", FALSE)),
  use_opencl = isTRUE(getOption("stanc.use_opencl", FALSE)),
  warn_pedantic = isTRUE(getOption("stanc.warn_pedantic", FALSE)),
  warn_uninitialized = isTRUE(getOption("stanc.warn_uninitialized", FALSE)),
  includes = NULL,
  isystem = c(if (!missing(file)) dirname(file), getwd()))
```

Arguments

file	A character string or a connection that R supports specifying the Stan model specification in Stan's modeling language.
model_name	A character string naming the model; defaults to "anon_model". However, the model name will be derived from file or model_code (if model_code is the name of a character string object) if model_name is not specified.
model_code	Either a character string containing the model specification or the name of a character string object in the workspace. This is an alternative to specifying the model via the file or stanc_ret arguments.
stanc_ret	A named list returned from a previous call to the [stanc](#) function. The list can be used to specify the model instead of using the file or model_code arguments.
boost_lib	The path to a version of the Boost C++ library to use instead of the one in the **BH** package.
eigen_lib	The path to a version of the Eigen C++ library to use instead of the one in the **RcppEigen** package.

save_dso	Logical, defaulting to TRUE, indicating whether the dynamic shared object (DSO) compiled from the C++ code for the model will be saved or not. If TRUE, we can draw samples from the same model in another R session using the saved DSO (i.e., without compiling the C++ code again).
verbose	Logical, defaulting to FALSE, indicating whether to report additional intermediate output to the console, which might be helpful for debugging.
auto_write	Logical, defaulting to the value of rstan_options("auto_write"), indicating whether to write the object to the hard disk using saveRDS. Although this argument is FALSE by default, we recommend calling rstan_options("auto_write" = TRUE) in order to avoid unnecessary recompilations. If file is supplied and its dirname is writable, then the object will be written to that same directory, substituting a .rds extension for the .stan extension. Otherwise, the object will be written to the tempdir.

obfuscate_model_name

A logical scalar that is TRUE by default and passed to stanc.

allow_undefined

A logical scalar that is FALSE by default and passed to stanc.

allow_optimizations

A logical scalar that is FALSE by default and passed to stanc.

standalone_functions

A logical scalar that is FALSE by default and passed to stanc.

use_opencl	A logical scalar that is FALSE by default and passed to stanc.
warn_pedantic	A logical scalar that is FALSE by default and passed to stanc.

warn_uninitialized

A logical scalar that is FALSE by default and passed to stanc.

includes	If not NULL (the default), then a character vector of length one (possibly containing one or more "\n") of the form '#include "/full/path/to/my_header.hpp"', which will be inserted into the C++ code in the model's namespace and can be used to provide definitions of functions that are declared but not defined in file or model_code when allow_undefined = TRUE
isystem	A character vector naming a path to look for file paths in file that are to be included within the Stan program named by file. See the Details section below.

Details

If a previously compiled stanmodel exists on the hard drive, its validity is checked and then returned without recompiling. The most common form of invalidity seems to be Stan code that ends with a } rather than a blank line, which causes the hash checker to think that the current model is different than the one saved on the hard drive. To avoid reading previously compiled stanmodels from the hard drive, supply the stanc_ret argument rather than the file or model_code arguments.

There are three ways to specify the model's code for stan_model:

1. parameter model_code: a character string containing the Stan model specification,
2. parameter file: a file name (or a connection) from which to read the Stan model specification, or
3. parameter stanc_ret: a list returned by stanc to be reused.

Value

An instance of S4 class stanmodel that can be passed to the sampling, optimizing, and vb functions.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual.* https://mc-stan.org/.

See Also

stanmodel for details on the class.

sampling, optimizing, and vb, which take a stanmodel object as input, for estimating the model parameters.

More details on Stan, including the full user's guide and reference manual, can be found at https://mc-stan.org/.

Examples

```
## Not run:
stancode <- 'data {real y_mean;} parameters {real y;} model {y ~ normal(y_mean,1);}'
mod <- stan_model(model_code = stancode, verbose = TRUE)
fit <- sampling(mod, data = list(y_mean = 0))
fit2 <- sampling(mod, data = list(y_mean = 5))

## End(Not run)
```

---

stan_rdump   *Dump the data for a Stan model to* R *dump file in the limited format that Stan can read.*

Description

This function takes a vector of names of R objects and outputs text representations of the objects to a file or connection. The file created by stan_rdump is typically used as data input of the Stan package (https://mc-stan.org/) or sourced into another R session. The usage of this function is very similar to dump in R.

Usage

```
stan_rdump(list, file = "", append = FALSE,
          envir = parent.frame(),
          width = options("width")$width,
          quiet = FALSE)
```

Arguments

list	A vector of character string: the names of one or more R objects to be dumped. See the note below.
file	Either a character string naming a file or a [connection](). `""` indicates output to the console.
append	Logical: if TRUE and `file` is a character string, output will be appended to `file`; otherwise, it will overwrite the contents of `file`.
envir	The environment to search for objects.
width	The width for maximum characters on a line. The output is broken into lines with `width`.
quiet	Whether to suppress warning messages that would appear when a variable is not found or not supported for dumping (not being numeric or it would not be converted to numeric) or a variable name is not allowed in Stan.

Value

An invisible character vector containing the names of the objects that were dumped.

Note

stan_rdump only dumps numeric data, which first can be a scalar, vector, matrix, or (multidimensional) array. Additional types supported are logical (TRUE and FALSE), factor, data.frame and a specially structured list.

The conversion for logical variables is to map TRUE to 1 and FALSE to 0. For factor variable, function as.integer is used to do the conversion (If we want to transform a factor f to approximately its original numeric values, see the help of function factor and do the transformation before calling stan_rdump). In the case of data.frame, function data.matrix is applied to the data frame before dumping. See the notes in [stan]() for the specially structured list, which will be converted to array before dumping.

stan_rdump will check whether the names of objects are legal variable names in Stan. If an illegal name is found, data will be dumped with a warning. However, passing the name checking does not necessarily mean that the name is legal. More details regarding rules of variable names in Stan can be found in Stan's manual.

If objects with specified names are not found, a warning will be issued.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual.* [https://mc-stan.org]().

See Also

[dump]()

Examples

```
# set variables in global environment
a <- 17.5
b <- c(1,2,3)
# write variables a and b to file ab.data.R in temporary directory
stan_rdump(c('a','b'), file.path(tempdir(), "ab.data.R"))
unlink(file.path(tempdir(), "ab.data.R"))

x <- 1; y <- 1:10; z <- array(1:10, dim = c(2,5))
stan_rdump(ls(pattern = '^[xyz]'), file.path(tempdir(), "xyz.Rdump"))
cat(paste(readLines(file.path(tempdir(), "xyz.Rdump")), collapse = '\n'), '\n')
unlink(file.path(tempdir(), "xyz.Rdump"))
```

---

stan_version   *Obtain the version of Stan*

Description

The stan version is in form of `major.minor.patch`; the first version is 1.0.0, indicating major version 1, minor version 0 and patch level 0. Functionality only changes with minor versions and backward compatibility will only be affected by major versions.

Usage

```
stan_version()
```

Value

A character string giving the version of Stan used in this version of RStan.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual*. `https://mc-stan.org/`.

See Also

`stan` and `stan_model`

Examples

```
stan_version()
```

---

summary-methods   *Summary method for stanfit objects*

Description

Summarize the distributions of estimated parameters and derived quantities using the posterior draws.

Usage

```
  ## S4 method for signature 'stanfit'
summary(object, pars, probs = c(0.025, 0.25, 0.50, 0.75, 0.975),
  use_cache = TRUE, ...)
```

Arguments

object	An instance of class [stanfit]().
pars	A character vector of parameter names. Defaults to all parameters as well as the log-posterior (`lp__`).
probs	A numeric vector of [quantile]()s of interest. The default is `c(0.025,0.25,0.5,0.75,0.975)`.
use_cache	Logical, defaulting to `TRUE`. When `use_cache=TRUE` the summary quantities for all parameters are computed and cached for future use. Setting `use_cache=FALSE` can be used to avoid performing the summary computations for all parameters if `pars` is given as some specific parameters.
...	Currently unused.

Value

The `summary` method returns a named list with elements `summary` and `c_summary`, which contain summaries for for all chains merged and individual chains, respectively. Included in the summaries are quantiles, means, standard deviations (sd), effective sample sizes (n_eff), and split Rhats (the potential scale reduction derived from all chains after splitting each chain in half and treating the halves as chains). For the summary of all chains merged, Monte Carlo standard errors (se_mean) are also reported.

See Also

- [monitor](), which computes similar summaries but accepts an array of MCMC draws as its input rather than a `stanfit` object.

- The RStan vignettes for more example usage.

Examples

```
## Not run:
ecode <- '
  parameters {
    array[2] real<lower=0> y;
  }
  model {
    y ~ exponential(1);
  }
'
fit <- stan(model_code = ecode)
s <- summary(fit, probs = c(0.1, 0.9))
s$summary  # all chaines merged
s$c_summary  # individual chains

## End(Not run)
```

---

traceplot	*Markov chain traceplots*

Description

Draw the traceplot corresponding to one or more Markov chains, providing a visual way to inspect sampling behavior and assess mixing across chains and convergence.

Usage

```
   ## S4 method for signature 'stanfit'
traceplot(object, pars, include = TRUE, unconstrain = FALSE,
          inc_warmup = FALSE, window = NULL, nrow = NULL, ncol = NULL, ...)
```

Arguments

object	An instance of class `stanfit`.
pars	A character vector of parameter names. Defaults to all parameters or the first 10 parameters (if there are more than 10).
include	Should the parameters given by the pars argument be included (the default) or excluded from the plot? Only relevant if pars is not missing.
inc_warmup	TRUE or FALSE, indicating whether the warmup sample are included in the trace plot; defaults to FALSE.
window	A vector of length 2. Iterations between window[1] and window[2] will be shown in the plot. The default is to show all iterations if inc_warmup is TRUE and all iterations from the sampling period only if inc_warmup is FALSE. If inc_warmup is FALSE the iterations specified in window should not include iterations from the warmup period.
unconstrain	Should parameters be plotted on the unconstrained space? Defaults to FALSE.
nrow, ncol	Passed to `facet_wrap`.
...	Optional arguments to pass to `geom_path` (e.g. size, linetype, alpha, etc.).

Value

A [ggplot](#) object that can be further customized using the **ggplot2** package.

Methods

traceplot signature(object = ″stanfit″) Plot the sampling paths for all chains.

See Also

[List of RStan plotting functions](#), [Plot options](#)

Examples

```
## Not run:
# Create a stanfit object from reading CSV files of samples (saved in rstan
# package) generated by funtion stan for demonstration purpose from model as follows.
#
excode <- '
  transformed data {
    array[20] real y;
    y[1] <- 0.5796;  y[2]  <- 0.2276;   y[3] <- -0.2959;
    y[4] <- -0.3742; y[5]  <- 0.3885;   y[6] <- -2.1585;
    y[7] <- 0.7111;  y[8]  <- 1.4424;   y[9] <- 2.5430;
    y[10] <- 0.3746; y[11] <- 0.4773;   y[12] <- 0.1803;
    y[13] <- 0.5215; y[14] <- -1.6044;  y[15] <- -0.6703;
    y[16] <- 0.9459; y[17] <- -0.382;   y[18] <- 0.7619;
    y[19] <- 0.1006; y[20] <- -1.7461;
  }
  parameters {
    real mu;
    real<lower=0, upper=10> sigma;
    vector[2] z[3];
    real<lower=0> alpha;
  }
  model {
    y ~ normal(mu, sigma);
    for (i in 1:3)
      z[i] ~ normal(0, 1);
    alpha ~ exponential(2);
  }
'
# exfit <- stan(model_code = excode, save_dso = FALSE, iter = 200,
#               sample_file = ″rstan_doc_ex.csv″)
#
exfit <- read_stan_csv(dir(system.file('misc', package = 'rstan'),
                       pattern='rstan_doc_ex_[[:digit:]].csv',
                       full.names = TRUE))

print(exfit)
traceplot(exfit)
traceplot(exfit, size = 0.25)
traceplot(exfit, pars = ″sigma″, inc_warmup = TRUE)
```

```
trace <- traceplot(exfit, pars = c("z[1,1]", "z[3,1]"))
trace + scale_color_discrete() + theme(legend.position = "top")

## End(Not run)
```

---

vb   *Run Stan's variational algorithm for approximate posterior sampling*

Description

Approximately draw from a posterior distribution using variational inference.

This is still considered an experimental feature. We recommend calling stan or sampling for final inferences and only using vb to get a rough idea of the parameter distributions.

Usage

```
  ## S4 method for signature 'stanmodel'
vb(object, data = list(), pars = NA, include = TRUE,
    seed = sample.int(.Machine$integer.max, 1),
    init = 'random', check_data = TRUE,
    sample_file = tempfile(fileext = '.csv'),
    algorithm = c("meanfield", "fullrank"),
    importance_resampling = FALSE, keep_every = 1,
    ...)
```

Arguments

object	An object of class stanmodel.
data	A named list or environment providing the data for the model or a character vector for all the names of objects used as data. See the **Passing data to Stan** section in stan.
pars	If not NA, then a character vector naming parameters, which are included in the output if include = TRUE and excluded if include = FALSE. By default, all parameters are included.
include	Logical scalar defaulting to TRUE indicating whether to include or exclude the parameters given by the pars argument. If FALSE, only entire multidimensional parameters can be excluded, rather than particular elements of them.
seed	The seed for random number generation. The default is generated from 1 to the maximum integer supported by R on the machine. Even if multiple chains are used, only one seed is needed, with other chains having seeds derived from that of the first chain to avoid dependent samples. When a seed is specified by a number, as.integer will be applied to it. If as.integer produces NA, the seed is generated randomly. The seed can also be specified as a character string of digits, such as "12345", which is converted to integer.

init Initial values specification. See the detailed documentation for the init argument in [stan](#).

check_data Logical, defaulting to TRUE. If TRUE the data will be preprocessed; otherwise not. See the **Passing data to Stan** section in [stan](#).

sample_file A character string of file name for specifying where to write samples for *all* parameters and other saved quantities. This defaults to a temporary file.

algorithm Either "meanfield" (the default) or "fullrank", indicating which variational inference algorithm is used. The "meanfield" option uses a fully factorized Gaussian for the approximation whereas the fullrank option uses a Gaussian with a full-rank covariance matrix for the approximation. Details and additional references are available in the Stan manual.

importance_resampling

Logical scalar (defaulting to FALSE) indicating whether to do importance resampling to adjust the draws at the optimum to be more like draws from the posterior distribution

keep_every Integer scalar (defaulting to 1) indicating the interval by which to thin the draws when imporance_resampling = TRUE

... Other optional parameters:

- iter (positive integer), the maximum number of iterations, defaulting to 10000.
- grad_samples (positive integer), the number of samples for Monte Carlo estimate of gradients, defaulting to 1.
- elbo_samples (positive integer), the number of samples for Monte Carlo estimate of ELBO (objective function), defaulting to 100. (ELBO stands for "the evidence lower bound".)
- eta (double), positive stepsize weighting parameter for variational inference but is ignored if adaptation is engaged, which is the case by default.
- adapt_engaged (logical), a flag indicating whether to automatically adapt the stepsize, defaulting to TRUE.
- tol_rel_obj (positive double), the convergence tolerance on the relative norm of the objective, defaulting to 0.01.
- eval_elbo (positive integer), evaluate ELBO every Nth iteration, defaulting to 100.
- output_samples (positive integer), number of posterior samples to draw and save, defaults to 1000.
- adapt_iter (positive integer), the maximum number of iterations to adapt the stepsize, defaulting to 50. Ignored if adapt_engaged = FALSE.

Refer to the manuals for both CmdStan and Stan for more details.

Value

An object of [stanfit-class](#).

Methods

vb signature(object = "stanmodel") Call Stan's variational Bayes methods for the model defined by S4 class stanmodel given the data, initial values, etc.

References

The Stan Development Team *Stan Modeling Language User's Guide and Reference Manual.* `https://mc-stan.org`.

The Stan Development Team *CmdStan Interface User's Guide.* `https://mc-stan.org`.

See Also

`stanmodel`

The manuals of CmdStan and Stan.

Examples

```
## Not run:
m <- stan_model(model_code = 'parameters {real y;} model {y ~ normal(0,1);}')
f <- vb(m)

## End(Not run)
```

Index