Package 'rstan'

February 6, 2015

Type Package

Title RStan: R interface to Stan

Version 2.6.0

Date 2015-02-06

Author Jiqiang Guo <`guojq28@gmail.com`>,
Michael Betancourt <`betanalpha@gmail.com`>,
Marcus Brubaker <`mbrubake@cs.toronto.edu`>,
Bob Carpenter <`carp@alias-i.com`>,
Ben Goodrich <`bg2382@columbia.edu`>,
Matt Hoffman <`mdhoffma@cs.princeton.edu`>,
Daniel Lee <`bearlee@alum.mit.edu`>,
Peter Li <`li.peter94@gmail.com`>,
Mitzi Morris <`mitzi@panix.com`>,
Rob Trangucci <`robert.trangucci@gmail.com`>,
Andrew Gelman <`gelman@stat.columbia.edu`>

Maintainer Jiqiang Guo <`guojq28@gmail.com`>

Description R interface to Stan (Stan is an open-source package for obtaining Bayesian inference using the No-U-Turn sampler, a variant of Hamiltonian Monte Carlo.)

License GPL (>=3)

Imports stats4

Depends R (>= 3.0.2), Rcpp (>= 0.9.10), utils, inline, methods

LinkingTo Rcpp, RcppEigen, BH (>= 1.54)

Suggests RUnit, RcppEigen, BH (>= 1.54), parallel, KernSmooth

URL `http://mc-stan.org`

R topics documented:

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

Description

R interface to Stan, which is a C++ package for obtaining Bayesian inference using the No-U-turn sampler, a variant of Hamiltonian Monte Carlo (see `http://mc-stan.org/`).

Details

Package:	rstan
Type:	Package
Version:	2.6.0
Date:	Oct 20, 2014
License:	GPL-3

The RStan package provides an interface to Stan. For more information on Stan and its modeling language, see the *Stan Modeling Language User's Guide and Reference Manual*, which is available `http://mc-stan.org/`.

Author(s)

Author:	Jiqiang Guo <guojq28@gmail.com>
	Michael Betancourt <betanalpha@gmail.com>
	Marcus Brubaker <mbrubake@cs.toronto.edu>
	Bob Carpenter <carp@alias-i.com>
	Ben Goodrich <bg2382@columbia.edu>
	Matt Hoffman <mdhoffma@cs.princeton.edu>
	Daniel Lee <bearlee@alum.mit.edu>
	Peter Li <li.peter94@gmail.com>
	Mitzi Morris <mitzi@panix.com>
	Rob Trangucci <robert.trangucci@gmail.com>
	Andrew Gelman <gelman@stat.columbia.edu>

Maintainer:	Jiqiang Guo <guojq28@gmail.com>

References

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

See Also

`stan`, `stanfit`

Examples

```
## Not run:
library(rstan)
cat("Stan version:", stan_version(), "\n")

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

parameters {
  real mu;
}

model {
  mu ~ normal(0, 10);
  y ~ normal(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, sample_file = 'norm.csv',
            verbose = TRUE)
print(fit)

# extract samples
e <- extract(fit, permuted = TRUE) # return a list of arrays
mu <- e$mu

m <- extract(fit, permuted = FALSE, inc_warmup = FALSE) # return an array
print(dimnames(m))

# using as.array directly on stanfit objects
m2 <- as.array(fit)


## 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 `array`, `matrix`, or `data.frame`

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)
```

Arguments

`x`	An object of S4 class `stanfit`
`...`	Additional parameters that can be passed to `extract` for extracting samples from `stanfit` object. For now, `pars` is the only additional parameter.

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, or data.frame of samples respectively.

`dim` and `dimnames` return the dim and dimnames of the array object that could be created.

`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`, and `dimnames`.

See Also

S4 class `stanfit` and its method `extract`

Examples

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

`cppo` *Set and get the optimization level and debug flag for compiling the C++ code*

Description

Set and get the optimization level, defined in CXXFLAGS, for compiling the C++ code. Also flag `-DDEBUG` or `-DNDEBUG` in `R_XTRA_CPPFLAGS` is set depending on the optimization mode and parameter `NDEBUG`. All flags set by `set_cppo` can be removed by `reset_cppo` to restore the default settings in R.

Usage

```
set_cppo(mode = c("fast", "presentation2", "presentation1", "debug", "small"),
         NDEBUG = FALSE)
get_cppo()
reset_cppo()
```

Arguments

`mode`	A character specifying the optimization level: one of `"fast"`, `"presentation2"`, `"presentation1"`, `"debug"`, `"small"`, corresponding to optimization level `"3"`, `"2"`, `"1"`, `"0"`, `"s"`; defaults to `"fast"`. The order from `"fast"` to `"debug"` generally means that the generated code runs from the fastest to the slowest; `"small"` means optimizing for size. See the notes below.
`NDEBUG`	Logical (defaults to `FALSE`). If `TRUE`, "-DNDEBUG" would be added to compile the generated C++ file (this is the default in R as well). However, if this is set, some index checking in Stan would be turned off, which would crash R if some indices are out of range in the model specification. In `"debug"` mode, the option is neglected (no "-DNDEBUG" is set). When `NDEBUG=FALSE` is specified for calling this function, a warning is issued.

Value

`set_cppo` returns the list with element names being `CXXFLAGS` and `R_XTRA_CPPFLAGS`. Each element is the desired flag that is set if the operation is successful; if any problems is encountered, this function will stop and report an error.

`get_cppo` returns a list. The fist element is string indicating the optimization mode (that is, one of `"fast"`, `"presentation2"`, `"presentation1"`, `"debug"`). The second element indicates if "NDEBUG" is set. The third element indicates if "DEBUG" is set.

`reset_cppo` removes the compiler flags set by `set_cppo` and restores the default settings in R.

Note

Since the optimization level is set by using a file with name `Makevars` (or a similar file with names such as `Makevars.win64`, `Makevars.win`, `Makevars-$R_PLATFROM` depending on platforms) in folder `.R` under the user's home directory, the *side effect* is that the optimization level set here will be used by `R CMD SHLIB` and possibly installing other R package from source (including installing **rstan** again). If this is not desired, the created file can be removed. For the same reason, the optimization level that is set stays once `set_cppo` is called. If a different optimization level is needed, `set_cppo` needs to be called again.

Generally for compiling models, what is strongly recommended is to use `mode="fast"`. When debugging C++ code using for example gdb, `mode="debug"` can be used, in which case `-g` flag is also set in `CXXFLAGS`. And we recommend calling `set_cppo` again to set the optimization mode back to `"fast"` after finishing debugging.

In modes other than `"debug"`, `"-DNDEBUG"` is added to the flags for compiling model's C++ code if `NDEBUG=TRUE`. By default (`NDEBUG=FALSE`), there is more index checking in Stan, which might slow down the execution. So if it is certain there is no problem with the indices in the model specification, we can set `NDEBUG=TRUE` to recompile the model.

References

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

help of `R CMD SHLIB`.

manual of *gcc* and *clang*.

See Also

`stan` and `stan_model`

Examples

```
## Not run:
set_cppo("fast")
get_cppo()
set_cppo("fast", NDEBUG = FALSE) # warning: this might disable some index range-checking

## End(Not run)
```

`extract-methods`   `extract`*: extract samples from 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)
```

Arguments

`object`	An object of class `stanfit`.
`pars`	An object of class `"character"` 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 in the samples.
`permuted`	`TRUE` of `FALSE`. If `TRUE`, draws after the *warmup* period in each chain are *permuted* and *merged*. If `FALSE`, the original order are kept. For each `stanfit` object, the permutation is fixed (that is, extracting samples a second time will give the same sequence of iterations as the previous).
`inc_warmup`	`TRUE` of `FALSE`. The argument matters only if `permuted` is `FALSE`. If `TRUE`, warmup draws are included; otherwise excluded.

Value

When `permuted = TRUE`, this function returns a named list, every element of which is an array representing samples for a parameter merged from all chains. 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 examples below and comments in the code. Function `monitor` can be applied to the returned array to obtain a summary of model inference (similar to method `print` for `stanfit`).

Methods

extract `signature(object = "stanfit")` Extract samples for a fitted model defined by class `stanfit`.

See Also

S4 class `stanfit`, `as.array.stanfit`, and `monitor`

Examples

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

  ## 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 {
      real y[20];
      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))
```

---

log_prob-methods — *model's* log_prob *and* grad_log_prob *functions*

Description

Using model's log_prob and grad_log_prob functions on the unconstrained space of model parameters. So sometimes we need convert the values of parameters from their support defined in parameter block (which might be constrained, and for simplicity, we call it constrained space) to unconstrained space and vice versa. constrained_pars and unconstrain_pars can be used then.

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

Details

In Stan, the parameters need be defined with their supports. For example, for a variance parameter, we must define it on the positive real line. But inside Stan's sampler, all parameters defined on the constrained space are transformed to unconstrained space, so the log density function need be adjusted (i.e., adding the log of the absolute value of the Jacobian determinant). With the transformation, Stan's samplers work on the unconstrained space and once a new iteration is drawn, Stan transforms the parameters back to their supports. All the transformation are done inside Stan without interference from the users. 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, there is an argument named `adjust_transform` for functions `log_prob` and `grad_log_prob`.

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 the log posterior (`lp__`) for the model represented by a `stanfit` object.

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*. http://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 {
  lp__ <- log(square(y - 5) + 1);
}
"
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)
```

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++.

As of rstan 1.0.2, `set_cppo` can be used to create/change a file that can be used to define compiling flags in the user's home directoty, in which a flag can be set to overwrite what is set in the system's Makeconf file.

Value

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

See Also

`stan`, `set_cppo`

Examples

```
makeconf_path()
```

monitor — *Compute the summary for MCMC simulation samples and monitor the convergence*

Description

For a 3-d array (the number of iterations * the number of chains * the number of parameters) of MCMC simulation samples, this function computes the summaries such as mean, standard deviation, standard error of the mean, and quantiles. And for monitoring the convergence, split Rhat and the effective sample size are also computed. By default, half of the iterations are considered to be warmup samples and thus 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, ...)
```

Arguments

`sims`	A 3-dimension array of samples simulated from any MCMC algorithm. The first dimension is for the number of iterations; the second for the number of chains; the third for the parameters.
`warmup`	The number of warmup iterations that would be excluded for computing the summaries; default to half of the total number of iterations.
`probs`	Quantiles of interest; defaults to "`c(0.025,0.25,0.5,0.75,0.975)`"
`digits_summary`	The number of significant digits for printing out the summary; defaults to 1. The effective sample size is always rounded to integers.
`...`	Additional arguments for the underlying `print` method.
`print`	Logical; indicating whether to print the summary.

Details

Similar to the `print` function for `stanfit` object, the function prints out a summary for the simulated samples. In particular, for monitoring the convergence, the printout includes the split Rhat and the effective sample size.

Value

A summary given as a 2 dimension array for the input samples: each row is for one parameter; the columns are the mean, standard deviation, quantiles, split Rhat, the effective sample size, etc.

References

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

See Also

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

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))
```

optimizing-methods — `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`. This method is a generic function of the S4 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,
    algorithm = c("LBFGS", "BFGS", "Newton"),
    verbose = FALSE, hessian = FALSE, as_vector = TRUE, ...)
```

Arguments

`object`	An object of class `stanmodel`.
`data`	An object of class `list`, `environment` providing the data for the model, or a vector character strings for all the names of objects used as data in the working space. See the notes 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. 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. We can also specify a seed using a character string of digits, such as `"12345"`, which will be converted to integer.
`init`	One of digit `0`, string `"0"` or `"random"`, a function that returns a list, or a named list of initial parameter values. `"0"`: initialize all to be zero on the unconstrained support; `"random"`: randomly generated; `list`: a list specifying the initial values of parameters by name. `function`: a function that returns a list for specifying the initial values of parameters for a chain.
`check_data`	Logical: if `TRUE`, the data would be preprocessed; otherwise not. If the data is not checked and preprocessed, it is safe to leave it to be the default `TRUE`. See the notes 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" indicating which optimization algorithm is used. The default is LBFGS.
`verbose`	`TRUE` or `FALSE`: flag indicating whether to print intermediate output from Stan on the console.

`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 for the point estimate found. A list can be used instead by specifying it to be `FALSE`
`...`	Other optional parameters, refer to the manuals for both CmdStan and Stan. 1. `iter` (`integer`), the maximum number of iterations 2. `save_iterations` (logical), whether to save the iterations 3. `refresh` (`integer`) 4. `init_alpha` (`double`, default to 0.001), for BFGS and LBFGS, see manual of (Cmd)Stan 5. `tol_obj` (`double`, default to 1e-12), for BFGS and LBFGS, see the manual of (Cmd)Stan 6. `tol_grad` (`double`, default to 1e-8), for BFGS and LBFGS, see the manual of (Cmd)Stan 7. `tol_param` (`double`, default to 1e-8), for BFGS and LBFGS, see the manual of (Cmd)Stan 8. `tol_rel_obj` (`double`, default to 1e4), for BFGS and LBFGS, see the manual of (Cmd)Stan 9. `tol_rel_grad` (`double`, default to 1e7), for BFGS and LBFGS, see the manual of (Cmd)Stan 10. `history_size` (`integer`, default to 5), for LBFGS, see the manual of (Cmd)Stan

Value

A list with components if the optimization is done successfully:

`par`	The point estimate found. Its form (vector or list) is determined by argument `as_vector`.
`value`	The value of the log-posterior (up to an additive constant, the `"lp__"` in Stan) corresponding to `par`.

If the optimization is not finished 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 + has.diag/3, cex.labels = NULL, font.labels = 1,
    row1attop = TRUE, gap = 1, pars = NULL, condition = NULL)
```

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`

`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.

`condition` By default, `NULL`, which 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__"` or `"n_divergent__"`, which are the variables produced by `get_sampler_params`, in which case the lower panel will plot realizations that are below the median of the indicated variable (or are zero in the case of "n_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 "n_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.

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.

See Also

S4 class stanfit and its method extract as well as the pairs generic function

Examples

```
example(read_stan_csv)
pairs(fit, condition = list(1, 2:4))
pairs(fit, pars = c("mu", "sigma", "alpha", "lp__"), condition = 0.25)
pairs(fit, pars = c("mu", "sigma", "alpha", "lp__"), condition = "acc")
lp <- sapply(get_logposterior(fit), FUN = function(x) tail(x, nrow(fit)))
pairs(fit, pars = c("mu", "sigma", "alpha", "lp__"), condition = lp > median(lp))
```

---

plot-methods   plot*: plot an overview of summaries for the fitted model*

Description

Drawn an overview of parameter summaries for the fitted model. In the overview plot, we also indicate the values of Rhats for all parameters of interest using differnt colors. In addition to all the parameters, the log-posterior is also plotted.

Usage

```
  ## S4 method for signature 'stanfit,missing'
plot(x, pars, display_parallel = FALSE,
  ask = TRUE, npars_per_page = 6)
```

Arguments

`x`	An instance of class `stanfit`.
`pars`	A vector of character string specifying the parameters to be plotted. If not specified, all parameters are used.
`display_parallel`	`TRUE` or `FALSE`, indicating whether to plot the intervals with one line for each chain or one line for all chains. The default is `FALSE` so that only one interval line is drawn for each scalar parameter.
`ask`	`TRUE` or `FALSE`, to control (for the current device) whether the user is prompted before starting a new page of output in the case there are a lot of parameters (see `devAskNewPage`).
`npars_per_page`	An integer to specify the number of parameters that would be plotted per page.

Value

`NULL`

Methods

plot `signature(x = "stanfit", y = "missing")` Plot an overview of parameter summaries for the fitted model.

Examples

```
## Not run:
library(rstan)
fit <- stan(model_code = "parameters {real y;} model {y ~ normal(0,1);}")
plot(fit)

## 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 {
    real y[20];
    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)
plot(exfit)
```

---

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, ...)
```

Arguments

`x`	An object of S4 class `stanfit`.
`pars`	Parameters in which the summaries are interest; defaults to all parameters for which samples are saved.
`probs`	Quantiles of interest; defaults to `"c(0.025,0.25,0.5,0.75,0.975)"`
`digits_summary`	The number of significant digits for printing out the summary; defaults to 1. The effective sample size is always rounded to integers.
`...`	Additional arguments that would be passed to method `summary` of `stanfit`.

Details

The information regarding the fitted model includes the number of iterations, the number of chains, the number of iterations that are saved in the `stanfit` object (including warmup); which sampler of NUTS1, NUTS2, HMC is used; and when the sampling is finished.

The summary about parameters includes the mean, the standard error of the mean (se_mean), the standard deviation (sd), quantiles, the effective sample size (n_eff), and the split Rhat. The summary is computed based on merging samples without warmup iterations from all chains.

In addition to parameters, the log-posterior (`lp__`) is also treated like a parameter in the printout.

See Also

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

`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)
```

Arguments

`f`	A character string providing the dump file name.

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]'), "xyz.Rdump")
l <- read_rdump('xyz.Rdump')
print(l)
```

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)
```

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. `rstawarmup_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.

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-methods — sampling: *draw samples from Stan model*

Description

Draw samples from the model defined by class `stanmodel`. This method is a generic function of the S4 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, diagnostic_file, verbose = FALSE,
    algorithm = c("NUTS", "HMC", "Fixed_param"),
    control = NULL,
    ...)
```

Arguments

`object`	An object of class `stanmodel`.
`data`	An object of class `list`, `environment` providing the data for the model, or a vector character strings for all the names of objects used as data in the working space. See the notes in `stan`.
`pars`	A vector of character strings specifying parameters of interest; defaults to `NA` indicating all parameters in the model. Only samples for parameters given in `pars` are stored in the fitted results.
`chains`	A positive integer specifying number of chains; defaults to 4.
`iter`	A positive integer specifying how many iterations for each chain (including warmup). The default is 2000.
`warmup`	A positive integer specifying the number of warmup (aka burnin) iterations. 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 the samples should not be used for inference). The number of warmup should not be larger than `iter` and the default is `iter/2`.
`thin`	A positive integer specifying the period for saving samples; defaults to 1.
`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. We can also specify a seed using a character string of digits, such as `"12345"`, which is converted to integer.

`init`	One of digit `0`, string `"0"` or `"random"`, a function that returns a list, or a list of initial parameter values with which to indicate how the initial values of parameters are specified. `"0"`: initialize all to be zero on the unconstrained support; `"random"`: randomly generated; `list`: a list of lists equal in length to the number of chains (parameter `chains`), where each list in the list of lists specifies the initial values of parameters by name for the corresponding chain. `function`: 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`.
`check_data`	Logical: if `TRUE`, the data would be preprocessed; otherwise not. If the data is not checked and preprocessed, it is safe to leave it to be the default `TRUE`. See the notes 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. When there are multiple chains, an underscore and chain number are appended to the file name.
`diagnostic_file`	A character string of file name for specifying where to write diagnostics data for *all* parameters. 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.
`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 algorithms that are implemented in Stan such as the No-U-Turn sampler (NUTS, Hoffman and Gelman 2011), static HMC, or Fixed_param.
`control`	See the argument `control` of function `stan`.
`...`	Additional arguments can be `chain_id`, `init_r`, `test_grad`, `append_samples`, `refresh`. See the document 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)
```

`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

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*. `http://mc-stan.org/`.

See Also

stan

Examples

```
## Not run:
library(rstan)
scode <- "
data {
  int<lower=1> N;
}
parameters {
  real y1[N];
  real y2[N];
}
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 = 0)
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)

## this may not work on Windows
sflist1 <-
  mclapply(1:4, mc.cores = 4,
           function(i) stan(fit = foo, data = foo_data, seed = seed,
                    chains = 1, chain_id = i, refresh = -1))
f3 <- sflist2stanfit(sflist1)

## for Windows (this works on Unix-like OS as well)

CL = makeCluster(4)
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 using 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, diagnostic_file,
  save_dso = TRUE,
  verbose = FALSE, ...,
  boost_lib = NULL,
  eigen_lib = NULL)
```

Arguments

`file`	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 parameter `model_code` or through a previous fit using parameter `fit`. When `fit` is specified, parameter `file` is ignored.
`model_name`	A character string naming the model; defaults to `"anon_model"`. However, the model name would 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`	A character string either containing the model definition or the name of a character string object in the workspace. This parameter is used only if parameter `file` is not specified. When `fit` is specified, the model compiled previously is used so specifying `model_code` is ignored.
`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.
`data`	An object of class `list`, `environment` providing the data for the model, or a vector of character strings for all the names of objects used as data in the working space. See the notes below.

`pars`	A vector of character string specifying parameters of interest; defaults to `NA` indicating all parameters in the model. Only samples for parameters given in `pars` are stored in the fitted results.
`chains`	A positive integer specifying number of chains; defaults to 4.
`iter`	A positive integer specifying how many iterations for each chain (including warmup). The default is 2000.
`warmup`	A positive integer specifying number of warmup (aka burnin) iterations. This also specifies the number of iterations used for stepsize adaptation, so warmup samples should not be used for inference. The number of warmup should not be larger than `iter` and the default is `iter/2`.
`thin`	A positive integer specifying the period for saving sample; defaults to 1.
`init`	One of digit `0`, string `"0"` or `"random"`, a function that returns a named list, or a list of named list. `"0"`: initialize all to be zero on the unconstrained support; `"random"`: randomly generated by Stan: the seed of the random number generator in Stan can be specified by argument `seed`. So if seed for Stan is fixed, the same initial values are used. Additionally, optional parameter `init_r` controls the range for randomly generating the initial values for parameters in terms of their unconstrained support; `list`: a list of lists equal in length to the number of chains (parameter `chains`), where each named list (an element of the list of lists) specifies the initial values for parameters by names for a chain. `function`: 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` for which the `chain_id` if specified or the integers from 1 to `chains` will be supplied respectively to the function for generating initial values. See the examples below of defining such functions and using a list of lists for specify initial values. Additionally, see below notes on that RStan treats a vector of length 1 as a scalar.
`seed`	The seed, a positive integer, for random number generation of Stan. The default is generated from 1 to the maximum integer supported by R so fixing the seed of R's random number generator can essentially fix the seed of Stan. When multiple chains are used, only one seed is needed, with other chains' seeds being generated from the first chain's seed to prevent dependency among the random number streams for the chains. 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. We can also specify a seed using a character string of digits, such as `"12345"`, which is converted to integer.
`algorithm`	One of algorithms that are implemented in Stan such as the No-U-Turn sampler (NUTS, Hoffman and Gelman 2011) and static HMC.
`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. When there are multiple chains, an underscore and chain number are appended to the file name.
`diagnostic_file`	A character string of file name for specifying where to write diagnostics data for *all* parameters. 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.

1. `adapt_engaged` (`logical`)
2. `adapt_gamma` (`double`, positive, defaults to 0.05)
3. `adapt_delta` (`double`, between 0 and 1, defaults to 0.8)
4. `adapt_kappa` (`double`, positive, defaults to 0.75)
5. `adapt_t0` (`double`, positive, defaults to 10)
6. `adapt_init_buffer` (`integer`, positive, defaults to 75)
7. `adapt_term_buffer` (`integer`, positive, defaults to 50)
8. `adapt_window` (`integer`, positive, defaults to 25)

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

1. `stepsize` (`double`, positive)
2. `stepsize_jitter` (`double`, [0,1])
3. `metric` (`string`, one of "unit_e", "diag_e", "dense_e")

For algorithm HMC, we can also set

1. `int_time` (`double`, positive)

For algorithm NUTS, we can set

1. `max_treedepth` (`integer`, positive)

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

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

`...` Other optional parameters:

1. `chain_id` (`integer`)
2. `init_r` (`double`, positive)
3. `test_grad` (`logical`)
4. `append_samples` (`logical`)
5. `refresh(integer)`

	`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 only valid for `init="random"`. If specified, the initial values are simulated from [-init_r, init_r] rather than using the default interval (see the manual of (cmd)Stan). Argument `refresh` (`integer`) can be used to control how to indicate the progress during sampling (i.e. show the progress every `refresh` iterations). By default, `refresh = max(iter/10, 1)`. The progress indicator is turned off if `refresh <= 0`. Another parameter is `test_grad` (`TRUE` or `FALSE`). If `test_grad=TRUE`, Stan will not do any sampling. Instead, the gradient calculation is tested and printed out and the fitted `stanfit` object is in test gradient mode. By default, it is `FALSE`. When a new model is fitted starting from Stan model code, `...` is passed to `stan_model` and thus `stanc`.
`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

`stan` does all of the work of fitting a Stan model and returning the results as an instance of `stanfit`. First, it translates the Stan model to C++ code. Second, the C++ code is compiled into a binary shared object, which is loaded into the current R session (an object of S4 class `stanmodel` is created). Finally, sample are drawn and wrapped in an object of S4 class `stanfit`, which provides functions 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`) by providing argument `fit`. In this case, the compiled C++ code for the model is reused.

Value

Fitted results as an object of S4 class `stanfit`. If error occurs before or during sampling, and if `test_grad = TRUE`, the returned object would not contain samples. But the compiled binary object for the model is still included, so we can reuse the returned object for another sampling.

Note

The data passed to `stan` are preprocessed before passing to Stan. In general, each element of data should be either a numeric vector (including special case '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 examples, characters) 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, `"real y[1];"` 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.

The returned S4 class `stanfit` includes the compiled model if option `save_dso` is `TRUE`, so the compiled model can be saved for using in future R sessions.

The function accepts a previously fitted instance of `stanfit` through parameter `fit` in order to reuse the compiled model; other configuration may change.

The optimization level for compiling the C++ code generated for the model can be set by `set_cppo`. In general, the higher the optimization level is set, the faster the generated binary code for the model runs. However, the binary code generated for the model runs fast by using higher optimization level is at the cost of long time to compile the C++ code.

References

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

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

See Also

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

see `extract` and `as.array.stanfit` for extracting samples from `stanfit` objects.

see `set_cppo` for setting higher optimization level for compiling the C++ code.

Examples

```
## Not run:
#### example 1
library(rstan)
scode <- "
parameters {
  real y[2];
}
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)

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

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

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

excode <- '
  transformed data {
    real y[20];
    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 Stan model specification to C++ code, which can then be compiled and loaded for sampling.

Usage

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

Arguments

`file`	A character string or a connection that R supports specifying the Stan model specification in Stan's modeling language.
`model_code`	A character string either 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 empty string.

`model_name`	A character string naming the model. The default is `"anon_model"`. However, the model name would 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`	`TRUE` print out more intermediate information during the translation procedure; `FALSE` otherwise. The default is `FALSE`.
`...`	optional parameters including 1. `obfuscate_model_name` (logical), `TRUE` if not specified. If `FALSE`, the model name in the generated C++ code would not contain randomly generated character string so that if model names are given the same, the generated C++ code will have the same class names defining the model, and the same Rcpp module names, which is used for R to execute the C++ code for sampling. Generally, it is recommended not to specify this parameter or set it to `TRUE`.

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 (`TRUE/FALSE`) 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. Further, 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*. http://mc-stan.org/.

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

See Also

`stan_model` and `stan`

Examples

```
## Not run:
stanmodelcode <- "
data {
  int<lower=0> N;
  real y[N];
}
```

```
parameters {
  real mu;
}

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

r <- stanc(model_code = stanmodelcode, model_name = "normal1")
names(r)
cat(r$cppcode)

## End(Not run)
```

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

Description

The output derived from fitting a Stan model, including the samples, as returned by the top-level function `stan` or the lower-level sampling method `sampling` defined on class `stanmodel`. `print` and `plot` and other methods are provided for the summaries of the fitted results. Access methods allow the underlying data making up a fit to be retrieved. There are three modes; sampling mode, test gradient mode, error mode (no samples included). The model's functions for computing the log probability density `lp__` and the gradient are also exposed for a stanfit object.

Objects from the Class

Objects should be created by either calling function `stan` or `sampling` method in S4 class `stan_model`.

Slots

- `model_name`**:** The model name, object of type `character`
- `model_pars`**:** The names of parameters (or transformed parameters, derived quantities), object of type `character`
- `par_dims`**:** The dimensions for all parameters, object of type `list`
- `mode`**:** The mode of the fitted model, object of type `integer`. `0` indicates sampling mode; `1` indicates test gradient mode for which no sampling is done; `2`; error occurred before sampling. Most methods for `stanfit` are useful only for `mode=0`
- `sim`**:** Simulation results including samples for the model, object of type `list`
- `inits`**:** The initial values either specified or generated randomly for all chains, object of type `list` containing named lists corresponding to initial values in the chains.
- `stan_args`**:** The arguments used for sampling all chains, object of type `list`
- `stanmodel`**:** The instance of S4 class `stanmodel`
- `date`**:** The date the object is created
- `.MISC`**:** Miscellaneous helper information used for the fitted model, object of type `environment`

Methods

`show` `signature(object = "stanfit")`: print the default summary for the model.

`plot` `signature(x = "stanfit", y = "missing")`: plot an overview of summaries for all parameters (see `plot`).

`summary` `signature(object = "stanfit")`: summarizes the distributions of quantities using the samples: the quantiles (for example, 2.5%, 50%, 97.5%, which can be specified by using parameter `probs`), the mean, the standard deviation (sd), the effective sample size (n_eff), and the split Rhat (i.e., potential scale reduction derived from all chains after splitting each chain in half and treating the halves as chains). Returned is a named list with elements such as `summary` and `c_summary`, summaries for all chains merged without warmup and individual chains. For the summary of all chains merged, we also have `se_mean`, the standard error of the mean. In addition to parameters, the log-posterior (`lp__`) is also a quantity of interest in the summary. To specify parameters of interest, use parameter `pars`. The default for `pars` is all the parameters saved in the fitted results as well as the log-posterior. Another argument for `summary` is `use_cache`, which defaults to `TRUE`. When `use_cache=TRUE`, the summary quantities for all parameters are computed and cached for future use. So `use_cache=FALSE` can be used to avoid the computing of all parameters if `pars` is given as some specific parameters.

`extract` `signature(object = "stanfit")`: get the samples for all chains for all (or specified) parameters. (see `extract`). Also see as.array.stanfit for coercing samples without warmup to arrays or matrices.

`as.mcmc.list` `signature(object = "stanfit")`: return a list of all the chains that can be treated as an `mcmc.list` as in package **coda**. Parameter of interest can be specified using `pars`. The warmup samples are discarded.

`traceplot` `signature(object = "stanfit")`: plot the trace of chains (see `traceplot`).

`get_posterior_mean` `signature(object = "stanfit")`: get the posterior mean for parameters of interest (using `pars` to specify) among *all* parameters.

`get_stancode` `signature(object = "stanfit")`: get the Stan code for the fitted model.

`get_cppo_mode` `signature(object = "stanfit")`: get the optimization mode used for compiling the model associated with this fitted results. The returned string is one of `"fast"`, `"presentation2"`, `"presentation1"`, and `"debug"`.

`get_stanmodel` `signature(object = "stanfit")`: get the object of S4 class `stanmodel` of the fitted model.

`get_inits` `signature(object = "stanfit")`: get the initial values for parameters used in sampling all chains. For `mode=2`, it returns an empty list.

`get_seed` `signature(object = "stanfit")`: get the seed that used for sampling. When the fitted object is empty (`mode=2`), `NULL` might be returned. In the case the seeds for all chains are different, use `get_seeds`.

`get_seeds` `signature(object = "stanfit")`: get the seeds that used for all chains. When the fitted object is empty (`mode=2`), `NULL` might be returned.

`get_logposterior` `signature(object = "stanfit")`: get the log-posterior (up to an additive constant, which is up to a multiplicative constant on the linear scale) for all chains. Each element of the returned `list` is the log-posterior for a chain. Optional parameter `inc_warmup` indicates whether to include the warmup period.

`get_adaptation_info signature(object = "stanfit")`: obtain the adaptation information for sampler, which now only NUTS2 has. The results are returned as a list, each element of which is a character string for a chain.

`get_sampler_params signature(object = "stanfit")`: obtain the parameters used for the sampler such as `stepsize` and `treedepth`. The results are returned as a list, each element of which is an array for a 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 parameter `inc_warmup` indicates whether to include the warmup period.

`log_prob signature(object = "stanfit", "numeric")`: compute the log probability density (`lp__`) for a set of parameter values (on the *unconstrained* 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.

`grad_log_prob signature(object = "stanfit", "numeric")`: 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__`.

`get_num_upars signature(object = "stanfit")`: 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 signature(object = "stanfit", "list")`: transform the parameter to unconstrained space. The input is a named list as for specifying initial values for each parameter. A numeric vector is returned.

`constrain_pars signature(object = "stanfit", "numeric")`: 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`.

References

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

See Also

`stan` and `stanmodel`

Examples

```
## Not run:
showClass("stanfit")
ecode <- '
  parameters {
    real<lower=0> y[2];
  }
  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 {
    real y[20];
    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)
plot(exfit)

adaptinfo <- get_adaptation_info(exfit)
seed <- get_seed(exfit)
sp <- get_sampler_params(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.

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.

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")`: return 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_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

`stanc`

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"), ...)
```

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) or call `optimizing` 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*. `http://mc-stan.org/`.

See Also

`sampling`, `optimizing`

Examples

```
  model_names <- gsub(".stan$", "", basename(stan_demo(0)))
  print(model_names)
  ## 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` can be used to draw samples from the model. If the model is supplied in the Stan modeling language, it is first translated to C++ code. 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, ...)
```

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 would 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`	A character string either containing the model specification or the name of a character string object in the workspace; an alternative is to specify the model with parameters `file` or `stanc_ret`.
`stanc_ret`	A named list returned from a previous call to function `stanc`. The list can be used to specify the model instead of using parameter `file` or `model_code`.
`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 with the default of `TRUE`: indication of 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`	`TRUE` or `FALSE`: indication of whether to report intermediate output to the console, which might be helpful for debugging.
`...`	passed to `stanc`.

Details

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

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

1. parameter `model_code`, containing character string to whose value is the Stan model specification,
2. parameter `file`, indicating a file (or a connection) from which to read the Stan model specification, or
3. parameter `stanc_ret`, indicating the re-use of a model generated in a previous call to `stanc`.

Value

An instance of S4 class `stanmodel`, which can be used later for drawing samples by calling its `sampling` function.

References

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

See Also

stanmodel and stan

Examples

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

## 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 (http://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*. `http://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 working directory
stan_rdump(c('a','b'), "ab.data.R")

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

See Also

`stan` and `stan_model`

Examples

```
stan_version()
```

`traceplot-methods` `traceplot`*: draw the traces of the sample*

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, inc_warmup = TRUE, ask = FALSE,
    nrow = 4, ncol = 2, window = NULL, ...)
```

Arguments

`object`	An instance of class `stanfit`.
`pars`	A vector of character string specifying the parameters to be plotted.
`inc_warmup`	`TRUE` or `FALSE`, indicating whether the warmup sample are included in the trace plot; defaults to `TRUE`
`ask`	`TRUE` or `FALSE`, to control (for the current device) whether the user is prompted before starting a new page of output in the case there are a lot of parameters (see `devAskNewPage`).
`nrow`	To specify the layout for the traceplots for multiple quantities: number of rows on every page. Together with `ncol`, the layout would be nrow * ncol. The layout by specifying `nrow` and `ncol` takes effects only when the total number is larger than `nrow` times `ncol`.
`ncol`	To specify the layout for the traceplots for multiple quantities: number of columns on every page.
`window`	To specify a window of all the iterations for plotting. Default of NULL means plotting all iterations. `window` needs to be an integer vector of at least length 1; smaller one specifies the starting iteration and the bigger one the last iteration. When there is only one integer, it just indicates the starting iteration. Examples of specifying window: `window=100` and `window=c(100,1500)`.
`...`	Additional parameters passed to the underlying function `plot`.

Value

`NULL`

Methods

traceplot `signature(object = "stanfit")` Plot the traces of sample for all chains.

See Also

`devAskNewPage`

Examples

```
## Not run:
library(rstan)
fit <- stan(model_code = "parameters {real y;} model {y ~ normal(0,1);}")
traceplot(fit)

## 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 {
    real y[20];
    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)
## only print some parameters
print(exfit, pars = c("sigma", "alpha"))
traceplot(exfit)
traceplot(exfit, pars = "sigma")

## make traceplots of z[1,1], z[1,2], z[2,1], z[2,2]
traceplot(exfit, pars = paste0("z[", c(1, 1, 2, 2), ",", c(1, 2, 1, 2), "]"))
```

Index