Package ‘greta’

March 22, 2022

Type Package
Title Simple and Scalable Statistical Modelling in R
Version 0.4.2
Date 2022-03-22
Description Write statistical models in R and fit them by MCMC and
optimisation on CPUs and GPUs, using Google ‘TensorFlow’. greta lets
you write your own model like in BUGS, JAGS and Stan, except that you
write models right in R, it scales well to massive datasets, and it’s
easy to extend and build on. See the website for more information,
including tutorials, examples, package documentation, and the greta
forum.
License Apache License 2.0
URL https://greta-stats.org
BugReports https://github.com/greta-dev/greta/issues
Depends R (>= 3.1.0)
Imports abind, callr, cli (>= 3.0.0), coda, future (>= 1.22.1), glue
(>= 1.5.1), methods, parallelly (>= 1.29.0), progress (>= 1.2.0), R6, reticulate (>= 1.19.0), tensorflow (>= 2.7.0),
yesno
Suggests bayesplot, covr, cramer, DiagrammeR, extraDistr, fields,
ggplot2, knitr, lattice, MASS, MCMCpack, mockery, mvtnorm,
markdown, r mutil, spelling, testthat (>= 3.1.0), tidyverse,
truncdist, withr
VignetteBuilder knitr
Config/testthat/edition 3
Encoding UTF-8
Language en-GB
RoxygenNote 7.1.2
SystemRequirements Python (>= 2.7.0) with header files and shared
library; TensorFlow (v1.14; https://www.tensorflow.org/);
TensorFlow Probability (v0.7.0;
https://www.tensorflow.org/probability/)
Collate 'package.R' 'utils.R' 'greta_mcmc_list.R' 'tf_functions.R'
'overloaded.R' 'node_class.R' 'node_types.R' 'variable.R'
'probability_distributions.R' 'mixture.R' 'joint.R'
'unknowns_class.R' 'greta_array_class.R' 'as_data.R'
'distribution.R' 'operators.R' 'functions.R' 'transforms.R'
'structures.R' 'extract_replace_combine.R' 'dag_class.R'
'greta_model_class.R' 'progress_bar.R' 'inference_class.R'
'samplers.R' 'optimisers.R' 'inference.R'
'install_tensorflow.R' 'calculate.R' 'callbacks.R' 'simulate.R'
'chol2symm.R' 'install_greta_deps.R' 'conda_greta_env.R'
'greta_stash.R' 'greta_create_conda_env.R'
'greta_install_miniconda.R' 'greta_install_python_deps.R'
'new_install_process.R' 'reinstallers.R' 'checkers.R'
'test_if_forked_cluster.R' 'testthat-helpers.R' 'zzz.R'
'internals.R'

NeedsCompilation no

Author Nick Golding [aut] (<https://orcid.org/0000-0001-8916-5570>),
Nicholas Tierney [aut, cre] (<https://orcid.org/0000-0003-1460-8722>),
Simon Dirmeier [ctb],
Adam Fleischhacker [ctb],
Shirin Glander [ctb],
Martin Ingram [ctb],
Lee Hazel [ctb],
Lionel Hertzog [ctb],
Tiphaine Martin [ctb],
Matt Mulvahill [ctb],
Michael Quinn [ctb],
David Smith [ctb],
Paul Teetor [ctb],
Jian Yen [ctb]

Maintainer Nicholas Tierney <nicholas.tierney@gmail.com>

Repository CRAN

Date/Publication 2022-03-22 13:00:02 UTC

R topics documented:

- as_data .................................................. 3
- calculate ............................................... 4
- chol2symm ............................................. 7
- distribution ........................................... 8
- distributions .......................................... 9
- extract-replace-combine ............................. 13
- functions ............................................. 14
- greta ................................................... 17
- greta_notes_install_miniconda_output ............. 17
- greta_sitrep .......................................... 18
as_data

convert other objects to greta arrays

Description

define an object in an R session as a data greta array for use as data in a greta model.

Usage

as_data(x)

Arguments

x

an R object that can be coerced to a greta_array (see details).

Details

as_data() can currently convert R objects to greta_arrays if they are numeric or logical vectors, matrices or arrays; or if they are dataframes with only numeric (including integer) or logical elements. Logical elements are always converted to numerics. R objects cannot be converted if they contain missing (NA) or infinite (~Inf or Inf) values.

Examples

## Not run:

# numeric/integer/logical vectors, matrices and arrays can all be coerced to
# data greta arrays

vec <- rnorm(10)
mat <- matrix(seq_len(3 * 4), nrow = 3)
calculate

calculate greta arrays given fixed values

Description
Calculate the values that greta arrays would take, given temporary, or simulated values for the greta arrays on which they depend. This can be used to check the behaviour of your model, make predictions to new data after model fitting, or simulate datasets from either the prior or posterior of your model.

Usage
```
calculate(
  ..., 
  values = list(),
  nsim = NULL,
  seed = NULL,
  precision = c("double", "single"),
  trace_batch_size = 100
)
```

Arguments

- `...`: one or more `greta_arrays` for which to calculate the value
- `values`: a named list giving temporary values of the greta arrays with which target is connected, or a `greta_mcmc_list` object returned by `mcmc()`.
- `nsim`: an optional positive integer scalar for the number of responses to simulate if stochastic greta arrays are present in the model - see Details.
- `seed`: an optional seed to be used in `set.seed` immediately before the simulation so as to generate a reproducible sample
calculate

precision  the floating point precision to use when calculating values.
trace_batch_size  the number of posterior samples to process at a time when target is a greta_mcmc_list object; reduce this to reduce memory demands

Details

The greta arrays named in values need not be variables, they can also be other operations or even data.

At present, if values is a named list it must contain values for all of the variable greta arrays with which target is connected, even values are given for intermediate operations, or the target doesn’t depend on the variable. That may be relaxed in a future release.

If the model contains stochastic greta arrays; those with a distribution, calculate can be used to sample from these distributions (and all greta arrays that depend on them) by setting the nsim argument to a positive integer for the required number of samples. If values is specified (either as a list of fixed values or as draws), those values will be used, and remaining variables will be sampled conditional on them. Observed data with distributions (i.e. response variables defined with distribution()) can also be sampled, provided they are defined as greta arrays. This behaviour can be used for a number of tasks, like simulating datasets for known parameter sets, simulating parameters and data from a set of priors, or simulating datasets from a model posterior. See some examples of these below.

Value

Values of the target greta array(s), given values of the greta arrays on which they depend (either specified in values or sampled from their priors). If values is a greta_mcmc_list() and nsim = NULL, this will be a greta_mcmc_list object of posterior samples for the target greta arrays. Otherwise, the result will be a named list of numeric R arrays. If nsim = NULL the dimensions of returned numeric R arrays will be the same as the corresponding greta arrays, otherwise an additional dimension with nsim elements will be prepended, to represent multiple simulations.

Examples

## Not run:

# define a variable greta array, and another that is calculated from it
# then calculate what value y would take for different values of x
x <- normal(0, 1, dim = 3)
a <- lognormal(0, 1)
y <- sum(x^2) + a
calculate(y, values = list(x = c(0.1, 0.2, 0.3), a = 2))

# by setting nsim, you can also sample values from their priors
calculate(y, nsim = 3)

# you can combine sampling and fixed values
calculate(y, values = list(a = 2), nsim = 3)

# if the greta array only depends on data,
# you can pass an empty list to values (this is the default)
x <- ones(3, 3)
y <- sum(x)
calculate(y)

# define a model
alpha <- normal(0, 1)
beta <- normal(0, 1)
sigma <- lognormal(1, 0.1)
y <- as_data(iris$Petal.Width)
mu <- alpha + iris$Petal.Length * beta
distribution(y) <- normal(mu, sigma)
m <- model(alpha, beta, sigma)

# sample values of the parameters, or different observation data (y), from
# the priors (useful for prior # predictive checking) - see also
# simulate.greta_model
calculate(alpha, beta, sigma, nsim = 100)
calculate(y, nsim = 100)

# calculate intermediate greta arrays, given some parameter values (useful
# for debugging models)
calculate(mu[1:5], values = list(alpha = 1, beta = 2, sigma = 0.5))
calculate(mu[1:5], values = list(alpha = -1, beta = 0.2, sigma = 0.5))

# simulate datasets given fixed parameter values
calculate(y, values = list(alpha = -1, beta = 0.2, sigma = 0.5), nsim = 10)

# you can use calculate in conjunction with posterior samples from MCMC, e.g.
# sampling different observation datasets, given a random set of these
# posterior samples - useful for posterior predictive model checks
draws <- mcmc(m, n_samples = 500)
calculate(y, values = draws, nsim = 100)

# you can use calculate on greta arrays created even after the inference on
# the model - e.g. to plot response curves
petal_length_plot <- seq(min(iris$Petal.Length),
                         max(iris$Petal.Length),
                         length.out = 100)
mu_plot <- alpha + petal_length_plot * beta
mu_plot_draws <- calculate(mu_plot, values = draws)
mu_est <- colMeans(mu_plot_draws[[1]])
plot(mu_est ~ petal_length_plot,
     type = "n",
     ylim = range(mu_plot_draws[[1]]))
apply(mu_plot_draws[[1]], 1, lines,
      x = petal_length_plot, col = grey(0.8))
lines(mu_est ~ petal_length_plot, lwd = 2)

# trace_batch_size can be changed to trade off speed against memory usage
# when calculating. These all produce the same result, but have increasing
# memory requirements:
mu_plot_draws_1 <- calculate(mu_plot,
  values = draws,
  trace_batch_size = 1
)
mu_plot_draws_10 <- calculate(mu_plot,
  values = draws,
  trace_batch_size = 10
)
mu_plot_draws_inf <- calculate(mu_plot,
  values = draws,
  trace_batch_size = Inf
)

## End(Not run)

---

**chol2symm**  
*Cholesky Factor to Symmetric Matrix*

**Description**

Evaluate $t(x) \%\% x$ efficiently, where $x$ is the (upper-triangular) Cholesky factor of a symmetric, positive definite square matrix. I.e. it is the inverse of chol.

**Usage**

`chol2symm(x)`

**Arguments**

- **x**
  
a square, upper triangular matrix representing the Cholesky factor of a symmetric, positive definite square matrix

**Examples**

```r
# a symmetric, positive definite square matrix
y <- rWishart(1, 4, diag(3))[, , 1]
# a symmetric, positive definite square matrix
u <- chol(y)
identical(y, chol2symm(u))
identical(chol2symm(u), t(u) %% u)
## Not run:
u_greta <- cholesky_variable(3)
y_greta <- chol2symm(u)
## End(Not run)
```
distribution

*define a distribution over data*

**Description**

distribution defines probability distributions over observed data, e.g. to set a model likelihood.

**Usage**

distribution(greta_array) <- value

distribution(greta_array)

**Arguments**

- **greta_array**: a data greta array. For the assignment method it must not already have a probability distribution assigned
- **value**: a greta array with a distribution (see `distributions()`)

**Details**

The extract method returns the greta array if it has a distribution, or NULL if it doesn’t. It has no real use-case, but is included for completeness

**Examples**

```r
## Not run:

# define a model likelihood

# observed data and mean parameter to be estimated
# (explicitly coerce data to a greta array so we can refer to it later)
y <- as_data(rnorm(5, 0, 3))
mu <- uniform(-3, 3)

distribution(y) <- normal(mu, 1)

distribution(y)
```

## End(Not run)
distributions

Description

These functions can be used to define random variables in a greta model. They return a variable
greta array that follows the specified distribution. This variable greta array can be used to represent
a parameter with prior distribution, combined into a mixture distribution using mixture(), or used
with distribution() to define a distribution over a data greta array.

Usage

uniform(min, max, dim = NULL)

normal(mean, sd, dim = NULL, truncation = c(-Inf, Inf))

lognormal(meanlog, sdlog, dim = NULL, truncation = c(0, Inf))

bernoulli(prob, dim = NULL)

binomial(size, prob, dim = NULL)

beta_binomial(size, alpha, beta, dim = NULL)

negative_binomial(size, prob, dim = NULL)

hypergeometric(m, n, k, dim = NULL)

poisson(lambda, dim = NULL)

gamma(shape, rate, dim = NULL, truncation = c(0, Inf))

inverse_gamma(alpha, beta, dim = NULL, truncation = c(0, Inf))

weibull(shape, scale, dim = NULL, truncation = c(0, Inf))

exponential(rate, dim = NULL, truncation = c(0, Inf))

pareto(a, b, dim = NULL, truncation = c(0, Inf))

student(df, mu, sigma, dim = NULL, truncation = c(-Inf, Inf))

laplace(mu, sigma, dim = NULL, truncation = c(-Inf, Inf))

beta(shape1, shape2, dim = NULL, truncation = c(0, 1))

cauchoy(location, scale, dim = NULL, truncation = c(-Inf, Inf))
chi_squared(df, dim = NULL, truncation = c(0, Inf))

logistic(location, scale, dim = NULL, truncation = c(-Inf, Inf))

f(df1, df2, dim = NULL, truncation = c(0, Inf))

multivariate_normal(mean, Sigma, n_realisations = NULL, dimension = NULL)

wishart(df, Sigma)

lkj_correlation(eta, dimension = 2)

multinomial(size, prob, n_realisations = NULL, dimension = NULL)

categorical(prob, n_realisations = NULL, dimension = NULL)

dirichlet(alpha, n_realisations = NULL, dimension = NULL)

dirichlet_multinomial(size, alpha, n_realisations = NULL, dimension = NULL)

Arguments

min, max scalar values giving optional limits to uniform variables. Like lower and upper, these must be specified as numerics, they cannot be greta arrays (though see details for a workaround). Unlike lower and upper, they must be finite. min must always be less than max.
dim the dimensions of the greta array to be returned, either a scalar or a vector of positive integers. See details.
mean, meanlog, location, mu unconstrained parameters
sd, sdlog, sigma, lambda, shape, rate, df, scale, shape1, shape2, alpha, beta, df1, df2, a, b, eta positive parameters, alpha must be a vector for dirichlet and dirichlet_multinomial.
truncation a length-two vector giving values between which to truncate the distribution, similarly to the lower and upper arguments to variable()
prob probability parameter (0 < prob < 1), must be a vector for multinomial and categorical
size, m, n, k positive integer parameter
Sigma positive definite variance-covariance matrix parameter
n_realisations the number of independent realisation of a multivariate distribution
dimension the dimension of a multivariate distribution

Details

The discrete probability distributions (bernoulli, binomial, negative_binomial, poisson, multinomial, categorical, dirichlet_multinomial) can be used when they have fixed values (e.g. defined as a likelihood using distribution()), but not as unknown variables.
For univariate distributions `dim` gives the dimensions of the greta array to create. Each element of the greta array will be (independently) distributed according to the distribution. `dim` can also be left at its default of `NULL`, in which case the dimension will be detected from the dimensions of the parameters (provided they are compatible with one another).

For multivariate distributions (`multivariate_normal()`, `multinomial()`, `categorical()`, `dirichlet()`, and `dirichlet_multinomial()`) each row of the output and parameters corresponds to an independent realisation. If a single realisation or parameter value is specified, it must therefore be a row vector (see example). `n_realisations` gives the number of rows/realisations, and `dimension` gives the dimension of the distribution. I.e. a bivariate normal distribution would be produced with `multivariate_normal(..., dimension = 2)`. The dimension can usually be detected from the parameters.

`multinomial()` does not check that observed values sum to `size`, and `categorical()` does not check that only one of the observed entries is 1. It's the user's responsibility to check their data matches the distribution!

The parameters of `uniform` must be fixed, not greta arrays. This ensures these values can always be transformed to a continuous scale to run the samplers efficiently. However, a hierarchical `uniform` parameter can always be created by defining a `uniform` variable constrained between 0 and 1, and then transforming it to the required scale. See below for an example.

Wherever possible, the parameterisations and argument names of greta distributions match commonly used R functions for distributions, such as those in the `stats` or `extraDistr` packages. The following table states the distribution function to which greta's implementation corresponds:

<table>
<thead>
<tr>
<th>greta</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniform</td>
<td>stats::dunif</td>
</tr>
<tr>
<td>normal</td>
<td>stats::dnorm</td>
</tr>
<tr>
<td>lognormal</td>
<td>stats::dlnorm</td>
</tr>
<tr>
<td>bernoulli</td>
<td>extraDistr::dbern</td>
</tr>
<tr>
<td>binomial</td>
<td>stats::dbinom</td>
</tr>
<tr>
<td>beta_binomial</td>
<td>extraDistr::dbbinom</td>
</tr>
<tr>
<td>negative_binomial</td>
<td>stats::dnbinom</td>
</tr>
<tr>
<td>hypergeometric</td>
<td>stats::dhyper</td>
</tr>
<tr>
<td>poisson</td>
<td>stats::dpois</td>
</tr>
<tr>
<td>gamma</td>
<td>stats::dgamma</td>
</tr>
<tr>
<td>inverse_gamma</td>
<td>extraDistr::dinvgamma</td>
</tr>
<tr>
<td>weibull</td>
<td>stats::dweibull</td>
</tr>
<tr>
<td>exponential</td>
<td>stats::dexp</td>
</tr>
<tr>
<td>paretto</td>
<td>extraDistr::dpareto</td>
</tr>
<tr>
<td>student</td>
<td>extraDistr::dlst</td>
</tr>
<tr>
<td>laplace</td>
<td>extraDistr::dlaplace</td>
</tr>
<tr>
<td>beta</td>
<td>stats::dbeta</td>
</tr>
<tr>
<td>cauchy</td>
<td>stats::dcauchy</td>
</tr>
<tr>
<td>chi_squared</td>
<td>stats::dchisq</td>
</tr>
<tr>
<td>logistic</td>
<td>stats::dlogis</td>
</tr>
<tr>
<td>f</td>
<td>stats::df</td>
</tr>
<tr>
<td>multivariate_normal</td>
<td>mvtnorm::dmvnorm</td>
</tr>
<tr>
<td>multinomial</td>
<td>stats::dmultinom</td>
</tr>
<tr>
<td>categorical</td>
<td>stats::dmultinom (size = 1)</td>
</tr>
<tr>
<td>dirichlet</td>
<td>extraDistr::ddirichlet</td>
</tr>
</tbody>
</table>
distributions

extraDistr::ddirmnom
dirichlet_multinomial
extraDistr::ddirmnom
dirichlet_multinomial
extraDistr::ddirmnom
extraDistr::ddirmnom
extraDistr::ddirmnom
extraDistr::ddirmnom
extraDistr::ddirmnom
extraDistr::ddirmnom
extraDistr::ddirmnom
extraDistr::ddirmnom

Examples

## Not run:

# a uniform parameter constrained to be between 0 and 1
phi <- uniform(min = 0, max = 1)

# a length-three variable, with each element following a standard normal
distribution
alpha <- normal(0, 1, dim = 3)

# a length-three variable of lognormals
sigma <- lognormal(0, 3, dim = 3)

# a hierarchical uniform, constrained between alpha and alpha + sigma,
etta <- alpha + uniform(0, 1, dim = 3) * sigma

# a hierarchical distribution
mu <- normal(0, 1)
sigma <- lognormal(0, 1)
theta <- normal(mu, sigma)

# a vector of 3 variables drawn from the same hierarchical distribution
thetas <- normal(mu, sigma, dim = 3)

# a matrix of 12 variables drawn from the same hierarchical distribution
thetas <- normal(mu, sigma, dim = c(3, 4))

# a multivariate normal variable, with correlation between two elements
# note that the parameter must be a row vector
Sig <- diag(4)
Sig[3, 4] <- Sig[4, 3] <- 0.6
theta <- multivariate_normal(t(rep(mu, 4)), Sig)

# 10 independent replicates of that
theta <- multivariate_normal(t(rep(mu, 4)), Sig, n_realisations = 10)

# 10 multivariate normal replicates, each with a different mean vector,
# but the same covariance matrix
means <- matrix(rnorm(40), 10, 4)
theta <- multivariate_normal(means, Sig, n_realisations = 10)
dim(theta)

# a Wishart variable with the same covariance parameter
theta <- wishart(df = 5, Sigma = Sig)

## End(Not run)
Description

Generic methods to extract and replace elements of greta arrays, or to combine greta arrays.

Arguments

- \( x \) a greta array
- \( i, j \) indices specifying elements to extract or replace
- \( n \) a single integer, as in `utils::head()` and `utils::tail()`
- \( \text{nrow, ncol} \) optional dimensions for the resulting greta array when \( x \) is not a matrix.
- \( \text{value} \) for \( [\gets \) a greta array to replace elements, for \( \text{dim}\gets \) either NULL or a numeric vector of dimensions
- \( \ldots \) either further indices specifying elements to extract or replace \( (\mathbb{1}) \), or multiple greta arrays to combine \( (\text{cbind()}, \text{rbind()} \& \text{c()} \), or additional arguments \( (\text{rep()}, \text{head()}, \text{tail()} \)
- \( \text{drop, recursive} \) generic arguments that are ignored for greta arrays

Details

- \( \text{diag()} \) can be used to extract or replace the diagonal part of a square and two-dimensional greta array, but it cannot be used to create a matrix-like greta array from a scalar or vector-like greta array. A static diagonal matrix can always be created with e.g. \( \text{diag(3)} \), and then converted into a greta array.
- Also note that since R 4.0.0, \( \text{head} \) and \( \text{tail} \) methods for arrays changed to print a vector rather than maintain the array structure. The greta package supports both methods, and will do so based on which version of R you are using.

Usage

```r
# extract
dx[i]
x[i, j, \ldots, \text{drop} = \text{FALSE}]
\text{head}(x, n = \text{6L}, \ldots)
\text{tail}(x, n = \text{6L}, \ldots)
\text{diag}(x, \text{nrow}, \text{ncol})

# replace
x[i] <- \text{value}
x[i, j, \ldots] <- \text{value}
\text{diag}(x) <- \text{value}
```
# combine
cbind(...)
rbind(...)
abind(...)
c(..., recursive = FALSE)
rep(x, times, ..., recursive = FALSE)

# get and set dimensions
length(x)
dim(x)
dim(x) <- value

Examples

```r
## Not run:

x <- as_data(matrix(1:12, 3, 4))

# extract and replace
x[1:3, ]
x[, 2:4] <- 1:9
e <- diag(x)
diag(x) <- e + 1

# combine
cbind(x[, 2], x[, 1])
rbind(x[1, ], x[3, ])  
abind(x[1, ], x[3, ], along = 1)
c(x[, 1], x)
rep(x[, 2], times = 3)

## End(Not run)
```

---

**functions**

**functions for greta arrays**

---

**Description**

This is a list of functions (mostly from base R) that are currently implemented to transform greta arrays. Also see operators and transforms.

**Details**

TensorFlow only enables rounding to integers, so `round()` will error if `digits` is set to anything other than 0.

Any additional arguments to `chol()`, `chol2inv`, and `solve()` will be ignored, see the TensorFlow documentation for details of these routines.
sweep() only works on two-dimensional greta arrays (so MARGIN can only be either 1 or 2), and
only for subtraction, addition, division and multiplication.

tapply() works on column vectors (2D greta arrays with one column), and INDEX cannot be a greta
array. Currently five functions are available, and arguments passed to... are ignored.

cospi(), sinpi(), and tanpi() do not use the computationally more stable routines to compute
\(\cos(x \times \pi)\) etc. that are available in R under some operating systems. Similarly trigamma() uses
tensorFlow’s polygamma function, resulting in lower precision than R’s equivalent.

Usage

# logarithms and exponentials
log(x)
exp(x)
log1p(x)
expm1(x)

# miscellaneous mathematics
abs(x)
mean(x)
sqrt(x)
sign(x)

# rounding of numbers
ceiling(x)
floor(x)
round(x, digits = 0)

# trigonometry
cos(x)
sin(x)
tan(x)
acos(x)
asin(x)
atan(x)
cosh(x)
sinh(x)
tanh(x)
acosh(x)
asinh(x)
atanh(x)
cospi(x)
sinpi(x)
tanpi(x)

# special mathematical functions
lgamma(x)
digamma(x)
trigamma(x)
choose(n, k)
lchoose(n, k)

# matrix operations
t(x)
chol(x, ...)
chol2inv(x, ...)
cov2cor(V)
solve(a, b, ...)
kroncker(X, Y, FUN = c('*', '/', '+', '-'))

# reducing operations
sum(..., na.rm = TRUE)
prod(..., na.rm = TRUE)
min(..., na.rm = TRUE)
max(..., na.rm = TRUE)

# cumulative operations
cumsum(x)
cumprod(x)
cummax(x)
cummin(x)

# solve an upper or lower triangular system
backsolve(r, x, k = ncol(r), upper.tri = TRUE,
          transpose = FALSE)
forwardsolve(l, x, k = ncol(l), upper.tri = FALSE,
             transpose = FALSE)

# miscellaneous operations
aperm(x, perm)
apply(x, MARGIN, FUN = c("sum", "max", "mean", "min",
                         "prod", "cumsum", "cumprod"))
sweep(x, MARGIN, STATS, FUN = c('-', '+', '/', '*'))
tapply(X, INDEX, FUN = c("sum", "max", "mean", "min", "prod"), ...)

Examples

## Not run:

x <- as_data(matrix(1:9, nrow = 3, ncol = 3))
a <- log(exp(x))
b <- log1p(expm1(x))
c <- sign(x - 5)
d <- abs(x - 5)

z <- t(a)
y <- sweep(x, 1, e, "-")

## End(Not run)

greta: simple and scalable statistical modelling in R

**Description**

`greta` lets you write statistical models interactively in native R code, then sample from them efficiently using Hamiltonian Monte Carlo.

The computational heavy lifting is done by TensorFlow, Google’s automatic differentiation library. So `greta` is particularly fast where the model contains lots of linear algebra, and `greta` models can be run across CPU clusters or on GPUs.

See the simple example below, and take a look at the `greta` website for more information including tutorials and examples.

**Examples**

## Not run:

# a simple Bayesian regression model for the iris data

# priors
int <- normal(0, 5)
coef <- normal(0, 3)
sd <- lognormal(0, 3)

# likelihood
mean <- int + coef * iris$Petal.Length
distribution(iris$Sepal.Length) <- normal(mean, sd)

# build and sample
m <- model(int, coef, sd)
draws <- mcmc(m, n_samples = 100)

## End(Not run)

**greta_notes_install_miniconda_output**

*Retrieve python installation or error details* These functions retrieve installation or error information output by python when running `install_miniconda()`, `conda_create()`, `conda_install()`, or when encountering a TensorFlow numerical problem.
Description

Retrieve python installation or error details

These functions retrieve installation or error information output by python when running `install_miniconda()`, `conda_create()`, `conda_install()`, or when encountering a TensorFlow numerical problem.

Usage

```python
    greta_notes_install_miniconda_output()
    greta_notes_install_miniconda_error()
    greta_notes_conda_create_output()
    greta_notes_conda_create_error()
    greta_notes_conda_install_output()
    greta_notes_conda_install_error()
    greta_notes_tf_num_error()
```

Examples

```r
## Not run:
greta_notes_install_miniconda()
greta_notes_conda_create()
greta_notes_conda_install()
greta_notes_tf_num_error()
greta_notes_tf_error()
## End(Not run)
```

Description

This checks if Python, Tensorflow, Tensorflow Probability, and the greta conda environment are available, and also loads and initialises python.

Usage

```r
greta_sitrep()
```

Value

Message if greta is ready to use
## inference

**statistical inference on greta models**

### Examples

```r
## Not run:
greta_sitrep()

## End(Not run)
```

### Description

Carry out statistical inference on greta models by MCMC or likelihood/posterior optimisation.

### Usage

```r
mcmc(
    model,
    sampler = hmc(),
    n_samples = 1000,
    thin = 1,
    warmup = 1000,
    chains = 4,
    n_cores = NULL,
    verbose = TRUE,
    pb_update = 50,
    one_by_one = FALSE,
    initial_values = initials(),
    trace_batch_size = 100
)
```

```r
stashed_samples()
```

```r
extra_samples(
    draws,
    n_samples = 1000,
    thin = 1,
    n_cores = NULL,
    verbose = TRUE,
    pb_update = 50,
    one_by_one = FALSE,
    trace_batch_size = 100
)
```

```r
initials(...)
```

```r
opt(
    model,
```
optimiser = bfgs(),
max_iterations = 100,
tolerance = 1e-06,
initial_values = initials(),
adjust = TRUE,
hessian = FALSE
)

Arguments

model  
greta_model object

sampler  
sampler used to draw values in MCMC. See samplers() for options.

n_samples  
number of MCMC samples to draw per chain (after any warm-up, but before thinning)

thin  
MCMC thinning rate; every thin samples is retained, the rest are discarded

warmup  
number of samples to spend warming up the mcmc sampler (moving chains toward the highest density area and tuning sampler hyperparameters).

chains  
number of MCMC chains to run

n_cores  
the maximum number of CPU cores used by each sampler (see details).

verbose  
whether to print progress information to the console

pb_update  
how regularly to update the progress bar (in iterations). If pb_update is less than or equal to thin, it will be set to thin + 1 to ensure at least one saved iteration per pb_update iterations.

one_by_one  
whether to run TensorFlow MCMC code one iteration at a time, so that greta can handle numerical errors as 'bad' proposals (see below).

initial_values  
an optional initials object (or list of initials objects of length chains) giving initial values for some or all of the variables in the model. These will be used as the starting point for sampling/optimisation.

trace_batch_size  
the number of posterior samples to process at a time when tracing the parameters of interest; reduce this to reduce memory demands

draws  
a greta_mcmc_list object returned by mcmc or stashed_samples

...  
named numeric values, giving initial values of some or all of the variables in the model (unnamed variables will be automatically initialised)

optimiser  
an optimiser object giving the optimisation algorithm and parameters See optimisers().

max_iterations  
the maximum number of iterations before giving up

tolerance  
the numerical tolerance for the solution, the optimiser stops when the (absolute) difference in the joint density between successive iterations drops below this level

adjust  
whether to account for Jacobian adjustments in the joint density. Set to FALSE (and do not use priors) for maximum likelihood estimates, or TRUE for maximum a posteriori estimates.

hessian  
whether to return a list of analytically differentiated Hessian arrays for the parameters
Details

For `mcmc()` if `verbose = TRUE`, the progress bar shows the number of iterations so far and the expected time to complete the phase of model fitting (warmup or sampling). Occasionally, a proposed set of parameters can cause numerical instability (i.e. the log density or its gradient is `NA`, `Inf` or `~Inf`); normally because the log joint density is so low that it can’t be represented as a floating point number. When this happens, the progress bar will also display the proportion of proposals so far that were ‘bad’ (numerically unstable) and therefore rejected. Some numerical instability during the warmup phase is normal, but ‘bad’ samples during the sampling phase can lead to bias in your posterior sample. If you only have a few bad samples (<10%), you can usually resolve this with a longer warmup period or by manually defining starting values to move the sampler into a more reasonable part of the parameter space. If you have more samples than that, it may be that your model is misspecified. You can often diagnose this by using `calculate()` to evaluate the values of greta arrays, given fixed values of model parameters, and checking the results are what you expect.

greta runs multiple chains simultaneously with a single sampler, vectorising all operations across the chains. E.g. a scalar addition in your model is computed as an elementwise vector addition (with vectors having length chains), a vector addition is computed as a matrix addition etc. TensorFlow is able to parallelise these operations, and this approach reduced computational overheads, so this is the most efficient of computing on multiple chains.

Multiple `mcmc` samplers (each of which can simultaneously run multiple chains) can also be run in parallel by setting the execution plan with the `future` package. Only `plan(multisession)` futures or `plan(cluster)` futures that don’t use fork clusters are allowed, since forked processes conflict with TensorFlow’s parallelism. Explicitly parallelising chains on a local machine with `plan(multisession)` will probably be slower than running multiple chains simultaneously in a single sampler (with `plan(sequential)`, the default) because of the overhead required to start new sessions. However, `plan(cluster)` can be used to run chains on a cluster of machines on a local or remote network. See `future::cluster()` for details, and the `future.batchtools` package to set up plans on clusters with job schedulers.

If `n_cores = NULL` and `mcmc` samplers are being run sequentially, each sampler will be allowed to use all CPU cores (possibly to compute multiple chains sequentially). If samplers are being run in parallel with the `future` package, `n_cores` will be set so that `n_cores * [future::nbrOfWorkers]` is less than the number of CPU cores.

After carrying out `mcmc` on all the model parameters, `mcmc()` calculates the values of (i.e. traces) the parameters of interest for each of these samples, similarly to `calculate()`. Multiple posterior samples can be traced simultaneously, though this can require large amounts of memory for large models. As in calculate, the argument `trace_batch_size` can be modified to trade-off speed against memory usage.

If the sampler is aborted before finishing (and `future` parallelism isn’t being used), the samples collected so far can be retrieved with `stashed_samples()`. Only samples from the sampling phase will be returned.

Samples returned by `mcmc()` and `stashed_samples()` can be added to with `extra_samples()`. This continues the chain from the last value of the previous chain and uses the same sampler and model as was used to generate the previous samples. It is not possible to change the sampler or extend the warmup period.

Because `opt()` acts on a list of greta arrays with possibly varying dimension, the `par` and `hessian` objects returned by `opt()` are named lists, rather than a vector (`par`) and a matrix (`hessian`), as returned by `stats::optim()`. Because greta arrays may not be vectors, the Hessians may not be
matrices, but could be higher-dimensional arrays. To return a Hessian matrix covering multiple model parameters, you can construct your model so that all those parameters are in a vector, then split the vector up to define the model. The parameter vector can then be passed to model. See example.

**Value**

`mcmc, stashed_samples & extra_samples` - a `greta_mcmc_list` object that can be analysed using functions from the coda package. This will contain mcmc samples of the greta arrays used to create model.

`opt` - a list containing the following named elements:

- `par` a named list of the optimal values for the greta arrays specified in model
- `value` the (unadjusted) negative log joint density of the model at the parameters 'par'
- `iterations` the number of iterations taken by the optimiser
- `convergence` an integer code, 0 indicates successful completion, 1 indicates the iteration limit `max_iterations` had been reached
- `hessian` (if `hessian = TRUE`) a named list of hessian matrices/arrays for the parameters (w.r.t. value)

**Examples**

```r
## Not run:
# define a simple Bayesian model
x <- rnorm(10)
mu <- normal(0, 5)
sigma <- lognormal(1, 0.1)
distribution(x) <- normal(mu, sigma)
m <- model(mu, sigma)

# carry out mcmc on the model
draws <- mcmc(m, n_samples = 100)

# add some more samples
draws <- extra_samples(draws, 200)

# initial values can be passed for some or all model variables
draws <- mcmc(m, chains = 1, initial_values = initials(mu = -1))

# if there are multiple chains, a list of initial values should be passed,
# otherwise the same initial values will be used for all chains
inits <- list(initials(sigma = 0.5), initials(sigma = 1))
draws <- mcmc(m, chains = 2, initial_values = inits)

# you can auto-generate a list of initials with something like this:
inits <- replicate(4,
    initials(mu = rnorm(1), sigma = runif(1)),
    simplify = FALSE
)
draws <- mcmc(m, chains = 4, initial_values = inits)
```
install_greta_deps

Install Python dependencies for greta

Description

This is a helper function to install Python dependencies needed. This includes Tensorflow version 1.14.0, Tensorflow Probability 0.7.0, and numpy version 1.16.4. These Python modules will be installed into a virtual or conda environment, named "greta-env". Note that "virtualenv" is not available on Windows.
Usage

install_greta_deps(
  method = c("auto", "virtualenv", "conda"),
  conda = "auto",
  timeout = 5,
  ...
)

Arguments

- method: Installation method ("virtualenv" or "conda")
- conda: The path to a conda executable. Use "auto" to allow reticulate to automatically find an appropriate conda binary. See Finding Conda for more details.
- timeout: maximum time in minutes until the installation for each installation component times out and exits. Default is 5 minutes per installation component.
- ... Optional arguments, reserved for future expansion.

Note

This will automatically install Miniconda (a minimal version of the Anaconda scientific software management system), create a 'conda' environment for greta named 'greta-env' with required python and python package versions, and forcibly switch over to using that conda environment.

If you don't want to use conda or the "greta-env" conda environment, you can install these specific versions of tensorflow (version 1.14.0), and tensorflow-probability (version 0.7.0), and ensure that the python environment that is initialised in this R session has these versions installed. This is now always straightforward, so we recommend installing the python packages using `install_greta_deps()` for most users.

Examples

```r
## Not run:
install_greta_deps()
## End(Not run)
```

Description

A list of functions and R6 class objects that can be used to develop extensions to greta. Most users will not need to access these methods, and it is not recommended to use them directly in model code.

---

internal greta methods

---

internals

---
Details

This help file lists the available internals, but they are not fully documented and are subject to change and deprecation without warning (though care will be taken not to break dependent packages on CRAN). For an overview of how greta works internally, see the technical details vignette. See https://github.com/greta-dev for examples of R packages extending and building on greta.

Please get in contact via GitHub if you want to develop an extension to greta and need more details of how to use these internal functions.

You can use `attach()` to put a sublist in the search path. E.g. `attach(.internals$nodes$constructors)` will enable you to call `op()`, `vble()` and `distrib()` directly.

Usage

```
.joint internals$greta_arrays$unknowns # greta array print methods
.joint internals$inference$progress_bar # progress bar tools
   samplers    # MCMC samplers
   stash       # stashing MCMC samples
.joint internals$nodes$constructors # node creation wrappers
   distribution_classes # R6 distribution classes
   mixture_classes # R6 mixture distribution classes
   node_classes # R6 node classes
.joint internals$tensors # functions on tensors
.joint internals$utils$checks # checking function inputs
   colours       # greta colour scheme
   dummy_arrays  # mocking up extract/replace
   misc          # code simplification etc.
   samplers      # mcmc helpers
.joint internals$greta_stash # internal information storage
```

---

Joint

**Description**

`joint` combines univariate probability distributions together into a multivariate (and *a priori* independent between dimensions) joint distribution, either over a variable, or for fixed data.

**Usage**

```
joint(..., dim = NULL)
```

**Arguments**

- `...` scalar variable greta arrays following probability distributions (see `distributions()`);
- `dim` the dimensions of the greta array to be returned, either a scalar or a vector of positive integers. The final dimension of the greta array returned will be determined by the number of component distributions.
Details

The component probability distributions must all be either continuous or discrete, and must have
the same dimensions.

This functionality is unlikely to be useful in most models, since the same result can usually be
achieved by combining variables with separate distributions. It is included for situations where it
is more convenient to consider these as a single distribution, e.g. for use with distribution or
mixture.

Examples

```r
## Not run:
# an uncorrelated bivariate normal
x <- joint(normal(-3, 0.5), normal(3, 0.5))
m <- model(x)
plot(mcmc(m, n_samples = 500))

# joint distributions can be used to define densities over data
x <- cbind(rnorm(10, 2, 0.5), rbeta(10, 3, 3))
mu <- normal(0, 10)
sd <- normal(0, 3, truncation = c(0, Inf))
a <- normal(0, 3, truncation = c(0, Inf))
b <- normal(0, 3, truncation = c(0, Inf))
distribution(x) <- joint(normal(mu, sd), beta(a, b),
                     dim = 10)
m <- model(mu, sd, a, b)
plot(mcmc(m))
## End(Not run)
```

mixture

Mixtures of probability distributions

Description

mixture combines other probability distributions into a single mixture distribution, either over a
variable, or for fixed data.

Usage

```
mixture(..., weights, dim = NULL)
```

Arguments

```
... variable greta arrays following probability distributions (see distributions());
the component distributions in a mixture distribution.
```
weights a column vector or array of mixture weights, which must be positive, but need not sum to one. The first dimension must be the number of distributions, the remaining dimensions must either be 1 or match the distribution dimension.

dim the dimensions of the greta array to be returned, either a scalar or a vector of positive integers.

Details

The weights are rescaled to sum to one along the first dimension, and are then used as the mixing weights of the distribution. I.e. the probability density is calculated as a weighted sum of the component probability distributions passed in via `dots`

The component probability distributions must all be either continuous or discrete, and must have the same dimensions.

Examples

```r
## Not run:
# a scalar variable following a strange bimodal distribution
weights <- uniform(0, 1, dim = 3)
a <- mixture(normal(-3, 0.5),
              normal(3, 0.5),
              normal(0, 3),
              weights = weights)

m <- model(a)
plot(mcmc(m, n_samples = 500))

# simulate a mixture of poisson random variables and try to recover the
# parameters with a Bayesian model
x <- c(
rpois(800, 3),
rpois(200, 10)
)
weights <- uniform(0, 1, dim = 2)
rates <- normal(0, 10, truncation = c(0, Inf), dim = 2)
distribution(x) <- mixture(poisson(rates[1]),
                          poisson(rates[2]),
                          weights = weights)

m <- model(rates)
draws_rates <- mcmc(m, n_samples = 500)

# check the mixing probabilities after fitting using calculate()
# (you could also do this within the model)
normalized_weights <- weights / sum(weights)
draws_weights <- calculate(normalized_weights, draws_rates)

# get the posterior means
summary(draws_rates)$statistics[, "Mean"]
summary(draws_weights)$statistics[, "Mean"]
```
# weights can also be an array, giving different mixing weights
# for each observation (first dimension must be number of components)

```r
dim <- c(5, 4)
weights <- uniform(0, 1, dim = c(2, dim))
b <- mixture(normal(1, 1, dim = dim),
              normal(-1, 1, dim = dim),
              weights = weights)
```

```r
## End(Not run)
```

---

**model**  
**greta model objects**

### Description

Create a `greta_model` object representing a statistical model (using `model`), and plot a graphical representation of the model. Statistical inference can be performed on `greta_model` objects with `mcmc()`

### Usage

```r
model(..., precision = c("double", "single"), compile = TRUE)
```

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

```r
## S3 method for class 'greta_model'
plot(x, y, colour = "#996bc7", ...)
```

### Arguments

- `...` for `model`: `greta_array` objects to be tracked by the model (i.e. those for which samples will be retained during `mcmc`). If not provided, all of the non-data `greta_array` objects defined in the calling environment will be tracked. For `print` and `plot`: further arguments passed to or from other methods (currently ignored).
- `precision` the floating point precision to use when evaluating this model. Switching from "double" (the default) to "single" may decrease the computation time but increase the risk of numerical instability during sampling.
- `compile` whether to apply XLA JIT compilation to the TensorFlow graph representing the model. This may slow down model definition, and speed up model evaluation.
- `x` a `greta_model` object
- `y` unused default argument
- `colour` base colour used for plotting. Defaults to `greta` colours in violet.
Details

model() takes greta arrays as arguments, and defines a statistical model by finding all of the other
greta arrays on which they depend, or which depend on them. Further arguments to model can be
used to configure the TensorFlow graph representing the model, to tweak performance.

The plot method produces a visual representation of the defined model. It uses the DiagrammeR
package, which must be installed first. Here’s a key to the plots:

Value

model - a greta_model object.

plot - a DiagrammeR::grViz() object, with the DiagrammeR::dgr_graph() object used to create
it as an attribute "dgr_graph".

Examples

## Not run:

# define a simple model
mu <- variable()
sigma <- normal(0, 3, truncation = c(0, Inf))
x <- rnorm(10)
distribution(x) <- normal(mu, sigma)

m <- model(mu, sigma)
plot(m)

## End(Not run)
Usage

# arithmetic operators
-x
x + y
x - y
x * y
x / y
x ^ y
x %% y
x %/% y
x %*% y

# logical operators
!x
x & y
x | y

# relational operators
x < y
x > y
x <= y
x >= y
x == y
x != y

Examples

## Not run:

x <- as_data(-1:12)

# arithmetic
a <- x + 1
b <- 2 * x + 3
c <- x %% 2
d <- x %/% 5

# logical
e <- (x > 1) | (x < 1)
f <- e & (x < 2)
g <- !f

# relational
h <- x < 1
i <- (-x) >= x
j <- h == x

## End(Not run)
Description

Functions to set up optimisers (which find parameters that maximise the joint density of a model) and change their tuning parameters, for use in `opt()`. For details of the algorithms and how to tune them, see the SciPy optimiser docs or the TensorFlow optimiser docs.

Usage

```r
nelder_mead()
powell()
cg()
bfgs()
newton_cg()
l_bfgs_b(maxcor = 10, maxls = 20)
tnc(max_cg_it = -1, stepmx = 0, rescale = -1)
cobyla(rhobeg = 1)
slsqp()
gradient_descent(learning_rate = 0.01)
adadelta(learning_rate = 0.001, rho = 1, epsilon = 1e-08)
adagrad(learning_rate = 0.8, initial_accumulator_value = 0.1)
adagrad_da(
  learning_rate = 0.8,
  global_step = 1L,
  initial_gradient_squared_accumulator_value = 0.1,
  l1_regularization_strength = 0,
  l2_regularization_strength = 0
)
momentum(learning_rate = 0.001, momentum = 0.9, use_nesterov = TRUE)
adam(learning_rate = 0.1, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-08)
```
ftrl(
    learning_rate = 1,
    learning_rate_power = -0.5,
    initial_accumulator_value = 0.1,
    l1_regularization_strength = 0,
    l2_regularization_strength = 0
)
proximal_gradient_descent(
    learning_rate = 0.01,
    l1_regularization_strength = 0,
    l2_regularization_strength = 0
)
proximal_adagrad(
    learning_rate = 1,
    initial_accumulator_value = 0.1,
    l1_regularization_strength = 0,
    l2_regularization_strength = 0
)
rms_prop(learning_rate = 0.1, decay = 0.9, momentum = 0, epsilon = 1e-10)

Arguments

maxcor maximum number of 'variable metric corrections' used to define the approximation to the hessian matrix
maxls maximum number of line search steps per iteration
max_cg_it maximum number of hessian * vector evaluations per iteration
stepmx maximum step for the line search
rescale log10 scaling factor used to trigger rescaling of objective
rhobeg reasonable initial changes to the variables
learning_rate the size of steps (in parameter space) towards the optimal value
rho the decay rate
epsilon a small constant used to condition gradient updates
initial_accumulator_value initial value of the 'accumulator' used to tune the algorithm
global_step the current training step number
initial_gradient_squared_accumulator_value initial value of the accumulators used to tune the algorithm
l1_regularization_strength L1 regularisation coefficient (must be 0 or greater)
l2_regularization_strength L2 regularisation coefficient (must be 0 or greater)
momentum the momentum of the algorithm
Details

The optimisers powell(), cg(), newton_cg(), l_bfgs_b(), tnc(), cobyla(), and slsqp() are deprecated. They will be removed in greta 0.4.0, since they will no longer be available in TensorFlow 2.0, on which that version of greta will depend.

The cobyla() does not provide information about the number of iterations nor convergence, so these elements of the output are set to NA

Value

an optimiser object that can be passed to opt().

Examples

```r
## Not run:
# use optimisation to find the mean and sd of some data
x <- rnorm(100, -2, 1.2)
mu <- variable()
sd <- variable(lower = 0)
distribution(x) <- normal(mu, sd)
m <- model(mu, sd)

# configure optimisers & parameters via 'optimiser' argument to opt
opt_res <- opt(m, optimiser = bfgs())

# compare results with the analytic solution
opt_res$par
c(mean(x), sd(x))

## End(Not run)
```

overloaded  Functions overloaded by greta

Description

greta provides a wide range of methods to apply common R functions and operations to greta_array objects. A few of these functions and operators are not associated with a class system, so they are overloaded here. This should not affect normal use of these functions, but they need to be documented to satisfy CRAN’s check.
Usage

```r
x %*% y
chol2inv(x, size = NCOL(x), LINPACK = FALSE)
cov2cor(V)
identity(x)
colMeans(x, na.rm = FALSE, dims = 1L)
rowMeans(x, na.rm = FALSE, dims = 1L)
colSums(x, na.rm = FALSE, dims = 1L)
rowSums(x, na.rm = FALSE, dims = 1L)
sweep(x, MARGIN, STATS, FUN = "-", check.margin = TRUE, ...)
backsolve(r, x, k = ncol(r), upper.tri = TRUE, transpose = FALSE)
forwardsolve(l, x, k = ncol(l), upper.tri = FALSE, transpose = FALSE)
apply(X, MARGIN, FUN, ...)
tapply(X, INDEX, FUN, ...)
eigen(x, symmetric, only.values, EISPACK)
rdist(x1, x2 = NULL, compact = FALSE)
abind(
  ..., 
  along = N,
  rev.along = NULL,
  new.names = NULL,
  force.array = TRUE,
  make.names = use.anon.names,
  use.anon.names = FALSE,
  use.first.dimnames = FALSE,
  hier.names = FALSE,
  use.dnns = FALSE
)

diag(x = 1, nrow, ncol)
```
Arguments

x, y, size, LINPACK, V, na.rm, dims, MARGIN, STATS, FUN, check.margin, ..., r, k, upper.tri, transpose, l, X
arguments as in original documentation

Details

Note that, since R 3.1, the LINPACK argument is defunct and silently ignored. The argument is
only included for compatibility with the base functions that call it.
To find the original help file for these overloaded functions, search for the function, e.g., ?cov2cor
and select the non-greta function.

reinstallers

Helpers to remove, and reinstall python environments and miniconda

Description

This can be useful when debugging greta installation to get to "clean slate". There are four functions:

Usage

remove_greta_env()
reinstall_greta_env(timeout = 5)
remove_miniconda()
reinstall_miniconda(timeout = 5)

Arguments

timeout time in minutes to wait until timeout (default is 5 minutes)

Details

• remove_greta_env() removes the 'greta-env' conda environment
• remove_miniconda() removes miniconda installation
• reinstall_greta_env() remove 'greta-env' and reinstall it using greta_create_conda_env()
  (which is used internally).
• reinstall_miniconda() removes miniconda and reinstall it using greta_install_miniconda()
  (which is used internally)

Value

invisible
Examples

```r
## Not run:
remove_greta_env()
remove_miniconda()
reinstall_greta_env()
reinstall_miniconda()

## End(Not run)
```

## Description

Functions to set up MCMC samplers and change the starting values of their parameters, for use in `mcmc()`.

## Usage

```r
hmc(Lmin = 5, Lmax = 10, epsilon = 0.1, diag_sd = 1)
rwmh(proposal = c("normal", "uniform"), epsilon = 0.1, diag_sd = 1)
slice(max_doublings = 5)
```

## Arguments

- **Lmin**: minimum number of leapfrog steps (positive integer, Lmin > Lmax)
- **Lmax**: maximum number of leapfrog steps (positive integer, Lmax > Lmin)
- **epsilon**: leapfrog stepsize hyperparameter (positive, will be tuned)
- **diag_sd**: estimate of the posterior marginal standard deviations (positive, will be tuned).
- **proposal**: the probability distribution used to generate proposal states
- **max_doublings**: the maximum number of iterations of the 'doubling' algorithm used to adapt the size of the slice

## Details

During the warmup iterations of `mcmc`, some of these sampler parameters will be tuned to improve the efficiency of the sampler, so the values provided here are used as starting values.

For `hmc()`, the number of leapfrog steps at each iteration is selected uniformly at random from between `Lmin` and `Lmax`. `diag_sd` is used to rescale the parameter space to make it more uniform, and make sampling more efficient.

`rwmh()` creates a random walk Metropolis-Hastings sampler; a gradient-free sampling algorithm. The algorithm involves a proposal generating step `proposal_state = current_state + perturb`
by a random perturbation, followed by Metropolis-Hastings accept/reject step. The class is imple-
mented for uniform and normal proposals.

slice() implements a multivariate slice sampling algorithm. Currently this algorithm can only be
used with single-precision models (set using the precision argument to model()). The parameter
max_doublings is not tuned during warmup.

Value

a sampler object that can be passed to mcmc().

---

**simulate.greta_model**  
*Simulate Responses From greta_model Object*

**Description**

Simulate values of all named greta arrays associated with a greta model from the model priors,
including the response variable.

**Usage**

```r
## S3 method for class 'greta_model'
simulate(object, nsim = 1, seed = NULL, precision = c("double", "single"), ...)
```

**Arguments**

- **object** a greta_model() object
- **nsim** positive integer scalar - the number of responses to simulate
- **seed** an optional seed to be used in set.seed immediately before the simulation so as
to generate a reproducible sample
- **precision** the floating point precision to use when calculating values.
- **...** optional additional arguments, none are used at present

**Details**

This is essentially a wrapper around calculate() that finds all relevant greta arrays. See that func-
tion for more functionality, including simulation conditional on fixed values or posterior samples.

To simulate values of the response variable, it must be both a named object (in the calling envi-
ronment) and be a greta array. If you don’t see it showing up in the output, you may need to use
as_data to convert it to a greta array before defining the model.

**Value**

A named list of vectors, matrices or arrays containing independent samples of the greta arrays
associated with the model. The number of samples will be prepended as the first dimension of
the greta array, so that a vector of samples is returned for each scalar greta array, and a matrix is
returned for each vector greta array, etc.
Examples

```r
## Not run:
# build a greta model
n <- 10
y <- rnorm(n)
y <- as_data(y)

library(greta)
sd <- lognormal(1, 2)
mu <- normal(0, 1, dim = n)
distribution(y) <- normal(mu, sd)
m <- model(mu, sd)

# simulate one random draw of y, mu and sd from the model prior:
sims <- simulate(m)

# 100 simulations of y, mu and sd
sims <- simulate(m, nsim = 100)
## End(Not run)
```

structures

These structures can be used to set up more complex models. For example, scalar parameters can be embedded in a greta array by first creating a greta array with `zeros()` or `ones()`, and then embedding the parameter value using greta’s replacement syntax.

Usage

- `zeros(...)`
- `ones(...)`
- `greta_array(data = 0, dim = length(data))`

Arguments

- `...` dimensions of the greta arrays to create
- `data` a vector giving data to fill the greta array. Other object types are coerced by `as.vector()`.
- `dim` an integer vector giving the dimensions for the greta array to be created.
Details

greta_array is a convenience function to create an R array with array() and then coerce it to a
greta array. I.e. when passed something that can be coerced to a numeric array, it is equivalent to
as_data(array(data,dim)).

If data is a greta array and dim is different than dim(data), a reshaped greta array is returned. This
is equivalent to: dim(data) <-dim.

Value

a greta array object

Examples

## Not run:

# a 3 row, 4 column greta array of 0s
z <- zeros(3, 4)

# a 3x3x3 greta array of 1s
z <- ones(3, 3, 3)

# a 2x4 greta array filled with pi
z <- greta_array(pi, dim = c(2, 4))

# a 3x3x3 greta array filled with 1, 2, and 3
z <- greta_array(1:3, dim = c(3, 3, 3))

## End(Not run)

transforms

transformation functions for greta arrays

Description

transformations for greta arrays, which may also be used as inverse link functions. Also see operators and functions.

Usage

iprobit(x)

ilogit(x)

icloglog(x)

icauchit(x)
log1pe(x)

imultilogit(x)

Arguments

x a real-valued (i.e. values ranging from -Inf to Inf) greta array to transform to a constrained value

Details

greta does not allow you to state the transformation/link on the left hand side of an assignment, as is common in the BUGS and STAN modelling languages. That’s because the same syntax has a very different meaning in R, and can only be applied to objects that are already in existence. The inverse forms of the common link functions (prefixed with an ‘i’) can be used instead.

The log1pe inverse link function is equivalent to log(1 + exp(x)), yielding a positive transformed parameter. Unlike the log transformation, this transformation is approximately linear for x > 1. i.e. when x > 1, y is approximately x

imultilogit expects an n-by-m greta array, and returns an n-by-(m+1) greta array of positive reals whose rows sum to one. This is equivalent adding a final column of 0s and then running the softmax function widely used in machine learning.

Examples

## Not run:

```r
x1 <- normal(1, 3, dim = 10)

# transformation to the unit interval
p1 <- iprobit(x1)
p2 <- ilogit(x1)
p3 <- icloglog(x1)
p4 <- icauchit(x1)

# and to positive reals
y <- log1pe(x1)

# transform from 10x3 to 10x4, where rows are a complete set of probabilities
x2 <- normal(1, 3, dim = c(10, 3))
z <- imultilogit(x2)
```

## End(Not run)
variable

create greta variables

Description

variable() creates greta arrays representing unknown parameters, to be learned during model fitting. These parameters are not associated with a probability distribution. To create a variable greta array following a specific probability distribution, see distributions().

Usage

variable(lower = -Inf, upper = Inf, dim = NULL)
cholesky_variable(dim, correlation = FALSE)
simplex_variable(dim)
ordered_variable(dim)

Arguments

lower, upper optional limits to variables. These must be specified as numerics, they cannot be greta arrays (though see details for a workaround). They can be set to -Inf (lower) or Inf (upper), though lower must always be less than upper.
dim the dimensions of the greta array to be returned, either a scalar or a vector of positive integers. See details.
correlation whether to return a cholesky factor corresponding to a correlation matrix (diagonal elements equalling 1, off-diagonal elements between -1 and 1).

Details

lower and upper must be fixed, they cannot be greta arrays. This ensures these values can always be transformed to a continuous scale to run the samplers efficiently. However, a variable parameter with dynamic limits can always be created by first defining a variable constrained between 0 and 1, and then transforming it to the required scale. See below for an example.

The constraints in simplex_variable() and ordered_variable() operate on the final dimension, which must have more than 1 element. Passing in a scalar value for dim therefore results in a row-vector.

Examples

## Not run:

# a scalar variable
a <- variable()

# a positive length-three variable
b <- variable(lower = 0, dim = 3)
# a 2x2x2 variable bounded between 0 and 1

c <- variable(lower = 0, upper = 1, dim = c(2, 2, 2))
# create a variable, with lower and upper defined by greta arrays

min <- as_data(iris$Sepal.Length)
max <- min^2
d <- min + variable(0, 1, dim = nrow(iris)) * (max - min)

## End(Not run)
# 4x4 cholesky factor variables for covariance and correlation matrices
e_cov <- cholesky_variable(dim = 4)
e_correl <- cholesky_variable(dim = 4, correlation = TRUE)

# these can be converted to symmetic matrices with chol2symm
# (equivalent to t(e_cov) %*% e_cov, but more efficient)
cov <- chol2symm(e_cov)
correl <- chol2symm(e_correl)
# a 4D simplex (sums to 1, all values positive)
f <- simplex_variable(4)

# a 4D simplex on the final dimension
g <- simplex_variable(dim = c(2, 3, 4))
# a 2D variable with each element higher than the one in the cell to the left
h <- ordered_variable(dim = c(3, 4))

# more constraints can be added with monotonic transformations, e.g. an
# ordered positive variable
i <- exp(ordered_variable(5))
Index

.%internals (internals), 24
%*% (overloaded), 33
abind (overloaded), 33
adadelta (optimisers), 31
adagrad (optimisers), 31
adagrad_da (optimisers), 31
adam (optimisers), 31
apply (overloaded), 33
array (), 39
as.vector (), 38
as_data, 3
backsolve (overloaded), 33
bernoulli (distributions), 9
beta (distributions), 9
beta_binomial (distributions), 9
bfgs (optimisers), 31
binomial (distributions), 9
c (extract-replace-combine), 13
calculate, 4
calculate (), 21, 37
categorical (distributions), 9
caucho (distributions), 9
cbind (extract-replace-combine), 13
cg (optimisers), 31
chi_squared (distributions), 9
chol2inv (overloaded), 33
chol2symm, 7
cholesky_variable (variable), 41
cobyta (optimisers), 31
colMeans (overloaded), 33
colSums (overloaded), 33
cov2cor (overloaded), 33
diag (overloaded), 33
DiagrammeR::dgr_graph (), 29
DiagrammeR::grViz (), 29
dirichlet (distributions), 9
dirichlet_multinomial (distributions), 9
distribution, 8
distribution (), 9, 10
distribution<- (distribution), 8
distributions, 9
distributions (), 8, 25, 26, 41
eigen (overloaded), 33
exponential (distributions), 9
extra_samples (inference), 19
extract (extract-replace-combine), 13
extract-replace-combine, 13
extraDistr::dbinom, 11
extraDistr::dbern, 11
extraDistr::ddichlet, 11
extraDistr::ddirmnom, 12
extraDistr::dinvgamma, 11
extraDistr::dlaplace, 11
extraDistr::dlst, 11
extraDistr::dpareto, 11
f (distributions), 9
forwardsolve (overloaded), 33
ftrl (optimisers), 31
functions, 14, 29, 39
future::cluster (), 21
gamma (distributions), 9
gradient_descent (optimisers), 31
greta, 17
greta_array (structures), 38
greta_mcmc_list (), 5
greta_model (), 37
greta_notes_conda_create_error
  (greta_notes_install_miniconda_output), 17
greta_notes_conda_create_output
  (greta_notes_install_miniconda_output), 17
greta_notes_conda_install_error
(greta_notes_install_miniconda_output
\texttt{\textbackslash m})
17

multivariate_normal (distributions), 9

-index

negative_binomial (distributions), 9

-greta_notes_install_miniconda_output
\texttt{\textbackslash m})
17

normal (distributions), 9

-nelder_mead (optimisers), 31

-newton_cg (optimisers), 31

-opt (inference), 19

-opt (), 31, 33

-optimisers (), 20

-ordered_variable (variable), 41

-overloaded, 33

-pareto (distributions), 9

-plot.greta_model (model), 28

-poisson (distributions), 9

-powell (optimisers), 31

-print.greta_model (model), 28

-proximal_adagrad (optimisers), 31

-proximal_gradient_descent (optimisers), 31

-rbind (extract-replace-combine), 13

-rdist (overloaded), 33

-reinstall_greta_env (reinstallers), 35

-reinstall_miniconda (reinstallers), 35

-reinstallers, 35

-remove_greta_env (reinstallers), 35

-remove_miniconda (reinstallers), 35

-rep (extract-replace-combine), 13

-replace (extract-replace-combine), 13

-rms_prop (optimisers), 31

-rowMeans (overloaded), 33

-rowSums (overloaded), 33

-rwmh (samplers), 36

-samplers, 36

-samplers(), 20

-simplex_variable (variable), 41

simulate.greta_model, 37

-slice (samplers), 36

-slsqp (optimisers), 31

-stash-notes

-stash-notes

(greta_notes_install_miniconda_output),
17
stashed_samples (inference), 19
stats::dbeta, 11
stats::dbinom, 11
stats::dcauchy, 11
stats::dchisq, 11
stats::dexp, 11
stats::df, 11
stats::dgamma, 11
stats::dhyper, 11
stats::dlnorm, 11
stats::dlogis, 11
stats::dmultinom, 11
stats::dnbinom, 11
stats::dnorm, 11
stats::dpois, 11
stats::dunif, 11
stats::dweibull, 11
stats::optim(), 21
stats::rWishart, 12
structures, 38
student (distributions), 9
sweep (overloaded), 33
tapply (overloaded), 33
tnc (optimisers), 31
transforms, 14, 29, 39
uniform (distributions), 9
variable, 41
variable(), 10
weibull (distributions), 9
wishart (distributions), 9
zeros (structures), 38