

# Package ‘greta’

November 12, 2024

**Type** Package

**Title** Simple and Scalable Statistical Modelling in R

**Version** 0.5.0

**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>, <https://github.com/greta-dev/greta>

**BugReports** <https://github.com/greta-dev/greta/issues>

**Depends** R (>= 4.1.0)

**Imports** abind, callr, cli (>= 3.4.1), coda, future (>= 1.22.1), glue (>= 1.5.1), methods, parallelly (>= 1.29.0), progress (>= 1.2.0), R6, reticulate (>= 1.19.0), rlang, tensorflow (== 2.16.0), tools, utils, whisker, yesno

**Suggests** bayesplot, covr, cramer, DiagrammeR, dplyr, DiagrammeRsvg, extraDistr, fields, ggplot2, knitr, lattice, MASS, MCMCpack, mockery, mvtnorm, purrr, rmarkdown, rmutl, rsvg, spelling, testthat (>= 3.1.0), tibble, tidyr, truncdist, withr, rstudioapi

**VignetteBuilder** knitr

**Config/testthat/edition** 3

**Encoding** UTF-8

**Language** en-US

**RoxygenNote** 7.3.2

**SystemRequirements** Python (>= 3.7.0) with header files and shared library; TensorFlow (>= v2.0.0; <https://www.tensorflow.org/>); TensorFlow Probability (v0.8.0; <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'  
 'data-deps-tf-tfp.R' 'greta\_model\_class.R' 'progress\_bar.R'  
 'inference\_class.R' 'samplers.R' 'optimisers.R'  
 'optimiser\_class.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' 'write-logfiles.R' 'zzz.R' 'internals.R'

**LazyData** true

**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** 2024-11-12 06:40:02 UTC

## Contents

are_null . . . . .	3
as.greta_model . . . . .	4
as.unknowns . . . . .	4
as_data . . . . .	5
calculate . . . . .	6
chol.greta_array . . . . .	9
chol2symm . . . . .	9
destroy_greta_deps . . . . .	10
dim.node . . . . .	11

dim<-unknowns . . . . .	11
distribution . . . . .	12
distributions . . . . .	13
extract-replace-combine . . . . .	17
functions . . . . .	18
gpu_cpu . . . . .	21
greta . . . . .	21
greta_create_conda_env . . . . .	22
greta_deps_receipt . . . . .	23
greta_deps_spec . . . . .	24
greta_deps_tf_tfp . . . . .	25
greta_install_miniconda . . . . .	25
greta_notes_tf_num_error . . . . .	26
greta_set_install_logfile . . . . .	26
greta_sitrep . . . . .	27
inference . . . . .	27
install_greta_deps . . . . .	32
internals . . . . .	34
is.greta_array . . . . .	35
is.greta_mcmc_list . . . . .	35
joint . . . . .	36
mixture . . . . .	37
model . . . . .	38
open_greta_install_log . . . . .	40
operators . . . . .	40
optimisers . . . . .	41
overloaded . . . . .	47
print.greta_deps_spec . . . . .	49
print.greta_mcmc_list . . . . .	49
reinstallers . . . . .	50
run_optimiser . . . . .	51
samplers . . . . .	51
simulate.greta_model . . . . .	52
structures . . . . .	53
transforms . . . . .	54
variable . . . . .	56
write_greta_install_log . . . . .	57

**Index****59**

are\_null

*Vectorised is.null***Description**

Vectorised is.null



---

as_data	<i>convert other objects to greta arrays</i>
---------	--

---

### 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)
arr <- array(sample(c(TRUE, FALSE), 2 * 2 * 2, replace = TRUE),
             dim = c(2, 2, 2)
            )
(a <- as_data(vec))
(b <- as_data(mat))
(c <- as_data(arr))

# dataframes can also be coerced, provided all the columns are numeric,
# integer or logical
df <- data.frame(
  x1 = rnorm(10),
  x2 = sample(1L:10L),
  x3 = sample(c(TRUE, FALSE), 10, replace = TRUE)
)
(d <- as_data(df))

## End(Not run)
```

---

 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,
  compute_options = cpu_only()
)
```

## 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 <code>mcmc()</code> .
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 <code>set.seed</code> immediately before the simulation so as to generate a reproducible sample
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
compute_options	Default is to use CPU only with <code>cpu_only()</code> . Use <code>gpu_only()</code> to use only GPU. In the future we will add more options for specifying CPU and GPU use. If setting GPU with <code>gpu_only()</code> then we cannot always guarantee that the random number seed will be respected. This is due to the way tensorflow interfaces with the GPU. If you must have reproducibility of all simulations we recommend using <code>cpu_only()</code> , which is the default. You can turn off the message about setting seed with GPU usage using <code>options(greta_gpu_message = FALSE)</code>

## 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)

```

---

chol.greta\_array      *Compute the Cholesky Factor of a Matrix*

---

### Description

Compute the Cholesky Factor of a Matrix

### Usage

```

## S3 method for class 'greta_array'
chol(x, ..., force_cholesky = FALSE)

```

### Arguments

**x**                    an object for which a method exists. The default method applies to numeric (or logical) symmetric, positive-definite matrices.

**...**                further arguments pass to or from methods.

**force\_cholesky**    Whether to force cholesky computation. Currently used as a workaround to ensure cholesky is calculated properly, and may result in code that uses chol() to be slow. Default is TRUE. Can change to FALSE, but may encounter issues in <https://github.com/greta-dev/greta/issues/585>.

---

chol2symm              *Cholesky Factor to Symmetric Matrix*

---

### Description

Evaluate  $t(x) \backslash \% \backslash x$  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**

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

## End(Not run)
```

---

destroy\_greta\_deps *Remove greta dependencies and remove miniconda*

---

**Description**

Sometimes when installing greta you might encounter an error and the best thing to do is start from a clean slate. This function does two things:

1. Removes the "greta-tf2-env" with [remove\\_greta\\_env\(\)](#)
2. Removes the miniconda installation with [remove\\_miniconda\(\)](#)

**Usage**

```
destroy_greta_deps()
```

**Value**

nothing

---

dim.node	<i>generic to grab dimensions of nodes</i>
----------	--

---

**Description**

generic to grab dimensions of nodes

**Usage**

```
## S3 method for class 'node'
dim(x)
```

**Arguments**

x	greta node class
---	------------------

---

dim<- .unknowns	<i>set dims like on a matrix/array</i>
-----------------	--

---

**Description**

set dims like on a matrix/array

**Usage**

```
## S3 replacement method for class 'unknowns'
dim(x) <- value
```

**Arguments**

x	matrix/array to set values to
value	values that are being set set

---

distribution	<i>define a distribution over data</i>
--------------	--

---

### 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

```
## 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)  
  
# define the distribution over y (the model likelihood)  
distribution(y) <- normal(mu, 1)  
  
# get the distribution over y  
distribution(y)  
  
## End(Not run)
```

---

distributions	<i>probability distributions</i>
---------------	----------------------------------

---

### 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))
```

```
cauchy(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 <code>dirichlet</code> and <code>dirichlet_multinomial</code> .
truncation	a length-two vector giving values between which to truncate the distribution, similarly to the lower and upper arguments to <code>variable()</code>
prob	probability parameter ( $0 < \text{prob} < 1$ ), must be a vector for <code>multinomial</code> and <code>categorical</code>
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:

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

dirichlet	<a href="#">extraDistr::ddirichlet</a>
dirichlet_multinomial	<a href="#">extraDistr::ddirmnom</a>
wishart	<a href="#">stats::rWishart</a>
lkj_correlation	<a href="#">rethinking::dlkcorr</a>

## 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,
eta <- 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)
```



---

 extract-replace-combine

*extract, replace and combine greta arrays*


---

## Description

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

## Arguments

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

## Details

`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. `diag(3)`, and then converted into a greta array.

Also note that since R 4.0.0, `head` and `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

```
# extract
x[i]
x[i, j, ..., drop = FALSE]
head(x, n = 6L, ...)
tail(x, n = 6L, ...)
diag(x, nrow, ncol)

# replace
x[i] <- value
x[i, j, ...] <- value
diag(x) <- 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

```
## 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  $\cos(x * \pi)$  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, ...)
kronecker(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)
```

---

gpu\_cpu

*Set GPU or CPU usage*

---

### Description

These functions set the use of CPU or GPU inside of greta. They simply return either "GPU" or "CPU", but in the future may handle more complexity. These functions are passed to `compute_options` inside of a few functions: `mcmc()`, `opt()`, and `calculate()`.

### Usage

```
gpu_only()
```

```
cpu_only()
```

---

greta

*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](#).

### Author(s)

**Maintainer:** Nicholas Tierney <nicholas.tierney@gmail.com> ([ORCID](#))

Authors:

- Nick Golding <nick.golding.research@gmail.com> ([ORCID](#))

Other contributors:

- Simon Dirmeier [contributor]
- Adam Fleischhacker [contributor]
- Shirin Glander [contributor]

- Martin Ingram [contributor]
- Lee Hazel [contributor]
- Lionel Hertzog [contributor]
- Tiphaine Martin [contributor]
- Matt Mulvahill [contributor]
- Michael Quinn [contributor]
- David Smith [contributor]
- Paul Teetor [contributor]
- Jian Yen [contributor]

### See Also

Useful links:

- <https://greta-stats.org>
- <https://github.com/greta-dev/greta>
- Report bugs at <https://github.com/greta-dev/greta/issues>

### 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\_create\_conda\_env

*Create conda environment for greta*

---

**Description**

This function runs `reticulate::conda_create()` inside `callr::r_process_options()`, to create the conda environment, "greta-env-tf2". This is used within `install_greta_deps()` as part of setting up python dependencies. It uses a version of python that is compatible with the versions of tensorflow and tensorflow-probability, which is established with `greta_deps_spec()`. We mostly recommend users use `install_greta_deps()` to manage their python dependency installation.

**Usage**

```
greta_create_conda_env(timeout = 5, deps = greta_deps_spec())
```

**Arguments**

timeout	time (minutes) until installation stops. Default is 5 minutes.
deps	dependency specification, see <code>greta_deps_spec()</code> for more details.

**Value**

nothing - creates a conda environment for a specific python version

---

greta_deps_receipt	<i>Capture greta python dependencies.</i>
--------------------	---

---

**Description**

To assist with capturing and sharing python dependencies, we provide a way to capture the dependencies currently used.

**Usage**

```
greta_deps_receipt()
```

**Value**

`greta_deps_spec()` object

**Examples**

```
## Not run:  
my_deps <- greta_deps_receipt()  
  
## End(Not run)
```

---

greta\_deps\_spec

*Specify python dependencies for greta*


---

## Description

A helper function for specifying versions of Tensorflow (TF), Tensorflow Probability (TFP), and Python. Defaulting to 2.15.0, 0.23.0, and 3.10, respectively. You can specify the version that you want to install, but it will check if these are compatible. That is, if you specify versions of TF/TFP/Python which do not work with each other, it will error and give a suggested version to install. It does this by using a dataset, `greta_deps_tf_tfp`, to check if the versions of TF, TFP, and Python specified are compatible on your operating system. You can inspect this dataset with `View(greta_deps_tf_tfp)`.

## Usage

```
greta_deps_spec(
  tf_version = "2.15.0",
  tfp_version = "0.23.0",
  python_version = "3.10"
)
```

## Arguments

<code>tf_version</code>	Character. Tensorflow (TF) version in format major.minor.patch. Default is "2.15.0".
<code>tfp_version</code>	Character. Tensorflow probability (TFP) version major.minor.patch. Default is "0.23.0".
<code>python_version</code>	Character. Python version in format major.minor.patch. Default is "3.10".

## Value

data frame of valid dependencies

## Examples

```
greta_deps_spec()
greta_deps_spec(tf_version = "2.15.0")
greta_deps_spec(tf_version = "2.15.0", tfp_version = "0.23.0")
greta_deps_spec(tf_version = "2.15.0", python_version = "3.10")
greta_deps_spec(
  tf_version = "2.15.0",
  tfp_version = "0.23.0",
  python_version = "3.10"
)
# this will fail
## Not run:
greta_deps_spec(
  tf_version = "2.11.0",
```



```

    tfp_version = "0.23.0",
    python_version = "3.10"
  )

  ## End(Not run)

```

---

greta\_deps\_tf\_tfp      *Suggested valid Python dependencies for greta*

---

### Description

This is a dataset that contains suggested valid versions of Tensorflow (TF), Tensorflow Probability (TFP), and Python for linux, mac, and windows machines. It was constructed from <https://www.tensorflow.org/install/source> and [https://www.tensorflow.org/install/source\\_windows](https://www.tensorflow.org/install/source_windows), and by inspecting <https://github.com/tensorflow/probability/releases>.

### Usage

```
greta_deps_tf_tfp
```

### Format

greta\_deps\_tf\_tfp:

A data frame with 63 rows and 5 columns:

**os** Operating System

**tfp\_version, tf\_version** numeric versions in format major.minor.patch for TFP and TF

**python\_version\_min, python\_version\_max** numeric versions range in format major.minor.patch for Python

### Details

We recommend using the default versions provided in `greta_deps_spec()`.

---

greta\_install\_miniconda  
*Installs miniconda*

---

### Description

This installs miniconda using `reticulate::install_miniconda()` inside `callr::r_process_options()`. Used internally by `install_greta_deps()`. We mostly recommend users use `install_greta_deps()` to manage their python dependency installation.

### Usage

```
greta_install_miniconda(timeout = 5)
```

**Arguments**

timeout                    time (minutes) until installation stops. Default is 5 minutes.

**Value**

nothing - installs miniconda.

---

greta\_notes\_tf\_num\_error

*Retrieve python messages.*

---

**Description**

These functions retrieve specific python error messages that might come up during greta use.

**Usage**

greta\_notes\_tf\_num\_error()

**Examples**

```
## Not run:
greta_notes_tf_error()

## End(Not run)
```

---

greta\_set\_install\_logfile

*Set logfile path when installing greta*

---

**Description**

To help debug greta installation, you can save all installation output to a single logfile.

**Usage**

greta\_set\_install\_logfile(path)

**Arguments**

path                    valid path to logfile - should end with .html so you can benefit from the html rendering.

**Value**

nothing - sets an environment variable for use with [install\\_greta\\_deps\(\)](#).

---

greta\_sitrep

*Greta Situation Report*

---

### Description

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

### Usage

```
greta_sitrep()
```

### Value

Message if greta is ready to use

### Examples

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

---

inference

*Statistical inference on greta models.*

---

### Description

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

### Usage

```
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,  
  compute_options = cpu_only()
```

```

)

stashd_samples()

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

initials(...)

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

```

### Arguments

model	greta_model object
sampler	sampler used to draw values in MCMC. See <a href="#">samplers()</a> 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).

<code>initial_values</code>	an optional <code>initials</code> object (or list of <code>initials</code> objects of length <code>chains</code> ) giving initial values for some or all of the variables in the model. These will be used as the starting point for sampling/optimisation.
<code>trace_batch_size</code>	the number of posterior samples to process at a time when tracing the parameters of interest; reduce this to reduce memory demands
<code>compute_options</code>	Default is to use CPU only with <code>cpu_only()</code> . Use <code>gpu_only()</code> to use only GPU. In the future we will add more options for specifying CPU and GPU use.
<code>draws</code>	a <code>greta_mcmc_list</code> object returned by <code>mcmc</code> or <code>stashed_samples</code>
<code>...</code>	named numeric values, giving initial values of some or all of the variables in the model (unnamed variables will be automatically initialised)
<code>optimiser</code>	an <code>optimiser</code> object giving the optimisation algorithm and parameters See <a href="#">optimisers()</a> .
<code>max_iterations</code>	the maximum number of iterations before giving up
<code>tolerance</code>	the numerical tolerance for the solution, the optimiser stops when the (absolute) difference in the joint density between successive iterations drops below this level
<code>adjust</code>	whether to account for Jacobian adjustments in the joint density. Set to <code>FALSE</code> (and do not use priors) for maximum likelihood estimates, or <code>TRUE</code> for maximum <i>a posteriori</i> estimates.
<code>hessian</code>	whether to return a list of <i>analytically</i> 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`)

**Note**

to set a seed with MCMC you must use `tensorflow::set_random_seed()`. This is due to an internal API with tensorflow. See <https://github.com/greta-dev/greta/issues/559> for a thread exploring this.

**Examples**

```
## 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)

# or find the MAP estimate
opt_res <- opt(m)

# get the MLE of the normal variance
mu <- variable()
variance <- variable(lower = 0)
distribution(x) <- normal(mu, sqrt(variance))
m2 <- model(variance)

# adjust = FALSE skips the jacobian adjustments used in MAP estimation, to
# give the true maximum likelihood estimates
o <- opt(m2, adjust = FALSE)

# the MLE corresponds to the *unadjusted* sample variance, but differs
# from the sample variance
o$par
```

```

mean((x - mean(x))^2) # same
var(x) # different

# initial values can also be passed to optimisers:
o <- opt(m2, initial_values = initials(variance = 1))

# and you can return a list of the Hessians for each of these parameters
o <- opt(m2, hessian = TRUE)
o$hessian

# to get a hessian matrix across multiple greta arrays, you must first
# combine them and then split them up for use in the model (so that the
# combined vector is part of the model) and pass that vector to model:
params <- c(variable(), variable(lower = 0))
mu <- params[1]
variance <- params[2]
distribution(x) <- normal(mu, sqrt(variance))
m3 <- model(params)
o <- opt(m3, hessian = TRUE)
o$hessian

## End(Not run)

```

---

install\_greta\_deps      *Install Python dependencies for greta*

---

## Description

This is a helper function to install Python dependencies needed. By default these are TF 2.15.0, TFP 0.23.0, and Python 3.10. These Python modules will be installed into a conda environment named "greta-env-tf2".

## Usage

```

install_greta_deps(
  deps = greta_deps_spec(),
  timeout = 5,
  restart = c("ask", "force", "no"),
  ...
)

reinstall_greta_deps(
  deps = greta_deps_spec(),
  timeout = 5,
  restart = c("ask", "force", "no")
)

```



## Arguments

deps	object created with <code>greta_deps_spec()</code> where you specify python, TF, and TFP versions. By default these are TF 2.15.0, TFP 0.23.0, and Python 3.10. These versions must be compatible with each other. If they are not, <code>greta_deps_spec()</code> will error with more information and suggestions. See <code>?greta_deps_spec()</code> for more information, and see the data object <code>greta_deps_tf_tfp</code> ( <code>'?greta_deps_tf_tfp'</code> ).
timeout	maximum time in minutes until the installation for each installation component times out and exits. Default is 5 minutes per installation component.
restart	character. Restart R after installation? Default is "ask". Other options are, "force", and "no". Using "force" will force a restart after installation. Using "no" will not restart. Note that this only restarts R during interactive sessions, and only in RStudio.
...	Optional arguments, reserved for future expansion.

## Details

You can specify an environment variable to write a logfile to a specific location with `GRETA_INSTALLATION_LOG` using `Sys.setenv('GRETA_INSTALLATION_LOG'='path/to/logfile.html')`. Or use `greta_set_install_logfile()` to set the path, e.g., `greta_set_install_logfile('path/to/logfile.html')`. By default it uses `tools::R_user_dir("greta")` as the directory to save a logfile named "greta-installation-logfile.html". To see installation notes or errors, after installation you can open the logfile with `open_greta_install_log()`, or you can navigate to the logfile and open it in a browser.

By default, if using RStudio, it will now ask you if you want to restart the R session. If the session is not interactive, or is not in RStudio, it will not restart. You can also override this with `restart = TRUE`.

## 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-tf2' 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-tf2" conda environment, you can install versions that you like, e.g., using `reticulate::py_install()`. If you want to see which versions of TF, TFP, and Python work with each other (at least according to information from tensorflow's website), see the data `greta_deps_tf_tfp`, which is provided with greta. Managing your own installation is not always straightforward, so we recommend installing the python packages using `install_greta_deps()` for most users.

## Examples

```
## Not run:
install_greta_deps()

## End(Not run)
## Not run:
# to help troubleshoot your greta installation, this can help resolve some
# issues with installing greta dependencies
reinstall_greta_deps()
```

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

---

internals

*internal greta methods*

---

## 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.

## 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

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

---

is.greta\_array      *Is object a greta array?*

---

**Description**

Is object a greta array?

**Usage**

```
is.greta_array(x, ...)
```

**Arguments**

x	object to test if is greta array
...	extra args (currently not used)

---

is.greta\_mcmc\_list      *Is object a greta\_mcmc\_list?*

---

**Description**

Is object a greta\_mcmc\_list?

**Usage**

```
is.greta_mcmc_list(x, ...)
```

**Arguments**

x	An object that may be a greta_mcmc_list
...	extra args (not currently used)

**Value**

logical TRUE/FALSE

---

joint	<i>define joint distributions</i>
-------	-----------------------------------

---

### 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 <a href="#">distributions()</a> ); the components of the joint distribution.
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

```
## 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	<i>mixtures of probability distributions</i>
---------	--

---

## 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 <a href="#">distributions()</a> ); 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

```
## 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)
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
)

## 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

```

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

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

```

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

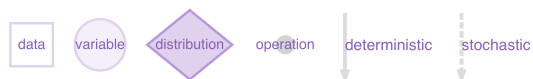
### Arguments

...	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 <b>XLA JIT compilation</b> 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)
```

---

```
open_greta_install_log
```

*Read a greta logfile*

---

### Description

This is a convenience function to facilitate reading logfiles. It opens a browser using `utils::browseURL()`. It will search for the environment variable "GRETA\_INSTALLATION\_LOG" or default to `tools::R_user_dir("greta")`. To set "GRETA\_INSTALLATION\_LOG" you can use `Sys.setenv('GRETA_INSTALLATION_LOG'='path/to/logfile.htm')`. Or use `greta_set_install_logfile()` to set the path, e.g., `greta_set_install_logfile('path/to/logfile.html')`.

### Usage

```
open_greta_install_log()
```

### Value

opens a URL in your default browser

---

```
operators
```

*arithmetic, logical and relational operators for greta arrays*

---

### Description

This is a list of currently implemented arithmetic, logical and relational operators to combine greta arrays into probabilistic models. Also see [functions](#) and [transforms](#).

### Details

greta's operators are used just like R's the standard arithmetic, logical and relational operators, but they return other greta arrays. Since the operations are only carried during sampling, the greta array objects have unknown values.

### 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)
```

---

optimisers

*optimisation methods*

---

## 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 [TensorFlow optimiser docs](#), or the [Tensorflow Probability optimiser docs](#).

**Usage**

```
nelder_mead(  
  objective_function = NULL,  
  initial_vertex = NULL,  
  step_sizes = NULL,  
  func_tolerance = 1e-08,  
  position_tolerance = 1e-08,  
  reflection = NULL,  
  expansion = NULL,  
  contraction = NULL,  
  shrinkage = NULL  
)  
  
bfgs(  
  value_and_gradients_function = NULL,  
  initial_position = NULL,  
  tolerance = 1e-08,  
  x_tolerance = 0L,  
  f_relative_tolerance = 0L,  
  initial_inverse_hessian_estimate = NULL,  
  stopping_condition = NULL,  
  validate_args = TRUE,  
  max_line_search_iterations = 50L,  
  f_absolute_tolerance = 0L  
)  
  
powell()  
  
momentum()  
  
cg()  
  
newton_cg()  
  
l_bfgs_b()  
  
tnc()  
  
cobyala()  
  
slsqp()  
  
gradient_descent(learning_rate = 0.01, momentum = 0, nesterov = FALSE)  
  
adadelat(learning_rate = 0.001, rho = 1, epsilon = 1e-08)  
  
adagrad(learning_rate = 0.8, initial_accumulator_value = 0.1, epsilon = 1e-08)
```

```
adagrad_da(  
    learning_rate = 0.8,  
    global_step = 1L,  
    initial_gradient_squared_accumulator_value = 0.1,  
    l1_regularization_strength = 0,  
    l2_regularization_strength = 0  
)  
  
adam(  
    learning_rate = 0.1,  
    beta_1 = 0.9,  
    beta_2 = 0.999,  
    amsgrad = FALSE,  
    epsilon = 1e-08  
)  
  
adamax(learning_rate = 0.001, beta_1 = 0.9, beta_2 = 0.999, epsilon = 1e-07)  
  
ftrl(  
    learning_rate = 1,  
    learning_rate_power = -0.5,  
    initial_accumulator_value = 0.1,  
    l1_regularization_strength = 0,  
    l2_regularization_strength = 0,  
    l2_shrinkage_regularization_strength = 0,  
    beta = 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  
)  
  
nadam(learning_rate = 0.001, beta_1 = 0.9, beta_2 = 0.999, epsilon = 1e-07)  
  
rms_prop(  
    learning_rate = 0.1,  
    rho = 0.9,  
    momentum = 0,  
    epsilon = 1e-10,
```

```

    centered = FALSE
)

```

### Arguments

<code>objective_function</code>	A function that accepts a point as a real Tensor and returns a Tensor of real dtype containing the value of the function at that point. The function to be minimized. If <code>batch_evaluate_objective</code> is TRUE, the function may be evaluated on a Tensor of shape $[n+1] + s$ where $n$ is the dimension of the problem and $s$ is the shape of a single point in the domain (so $n$ is the size of a Tensor representing a single point). In this case, the expected return value is a Tensor of shape $[n+1]$ . Note that this method does not support univariate functions so the problem dimension $n$ must be strictly greater than 1.
<code>initial_vertex</code>	Tensor of real dtype and any shape that can be consumed by the <code>objective_function</code> . A single point in the domain that will be used to construct an axes aligned initial simplex.
<code>step_sizes</code>	Tensor of real dtype and shape broadcasting compatible with <code>initial_vertex</code> . Supplies the simplex scale along each axes.
<code>func_tolerance</code>	Single numeric number. The algorithm stops if the absolute difference between the largest and the smallest function value on the vertices of the simplex is below this number. Default is $1e-08$ .
<code>position_tolerance</code>	Single numeric number. The algorithm stops if the largest absolute difference between the coordinates of the vertices is below this threshold.
<code>reflection</code>	(optional) Positive Scalar Tensor of same dtype as <code>initial_vertex</code> . This parameter controls the scaling of the reflected vertex. See, <a href="#">Press et al(2007)</a> for details. If not specified, uses the dimension dependent prescription of Gao and Han (2012) <a href="https://doi.org/10.1007/s1058901093293">doi:10.1007/s1058901093293</a>
<code>expansion</code>	(optional) Positive Scalar Tensor of same dtype as <code>initial_vertex</code> . Should be greater than 1 and reflection. This parameter controls the expanded scaling of a reflected vertex. See, <a href="#">Press et al(2007)</a> for details. If not specified, uses the dimension dependent prescription of Gao and Han (2012) <a href="https://doi.org/10.1007/s10589-01093293">doi:10.1007/s10589-01093293</a>
<code>contraction</code>	(optional) Positive scalar Tensor of same dtype as <code>initial_vertex</code> . Must be between 0 and 1. This parameter controls the contraction of the reflected vertex when the objective function at the reflected point fails to show sufficient decrease. See, <a href="#">Press et al(2007)</a> for details. If not specified, uses the dimension dependent prescription of Gao and Han (2012) <a href="https://doi.org/10.1007/s1058901093293">doi:10.1007/s1058901093293</a>
<code>shrinkage</code>	(Optional) Positive scalar Tensor of same dtype as <code>initial_vertex</code> . Must be between 0 and 1. This parameter is the scale by which the simplex is shrunk around the best point when the other steps fail to produce improvements. See, <a href="#">Press et al(2007)</a> for details. If not specified, uses the dimension dependent prescription of Gao and Han (2012) <a href="https://doi.org/10.1007/s1058901093293">doi:10.1007/s1058901093293</a>
<code>value_and_gradients_function</code>	A function that accepts a point as a real Tensor and returns a tuple of Tensors of real dtype containing the value of the function and its gradient at that point. The

	function to be minimized. The input should be of shape $[\dots, n]$ , where $n$ is the size of the domain of input points, and all others are batching dimensions. The first component of the return value should be a real Tensor of matching shape $[\dots]$ . The second component (the gradient) should also be of shape $[\dots, n]$ like the input value to the function.
<code>initial_position</code>	real Tensor of shape $[\dots, n]$ . The starting point, or points when using batching dimensions, of the search procedure. At these points the function value and the gradient norm should be finite.
<code>tolerance</code>	Scalar Tensor of real dtype. Specifies the gradient tolerance for the procedure. If the supremum norm of the gradient vector is below this number, the algorithm is stopped. Default is $1e-08$ .
<code>x_tolerance</code>	Scalar Tensor of real dtype. If the absolute change in the position between one iteration and the next is smaller than this number, the algorithm is stopped. Default of $0L$ .
<code>f_relative_tolerance</code>	Scalar Tensor of real dtype. If the relative change in the objective value between one iteration and the next is smaller than this value, the algorithm is stopped.
<code>initial_inverse_hessian_estimate</code>	Optional Tensor of the same dtype as the components of the output of the <code>value_and_gradients_function</code> . If specified, the shape should broadcastable to shape $[\dots, n, n]$ ; e.g. if a single $[n, n]$ matrix is provided, it will be automatically broadcasted to all batches. Alternatively, one can also specify a different hessian estimate for each batch member. For the correctness of the algorithm, it is required that this parameter be symmetric and positive definite. Specifies the starting estimate for the inverse of the Hessian at the initial point. If not specified, the identity matrix is used as the starting estimate for the inverse Hessian.
<code>stopping_condition</code>	(Optional) A function that takes as input two Boolean tensors of shape $[\dots]$ , and returns a Boolean scalar tensor. The input tensors are <code>converged</code> and <code>failed</code> , indicating the current status of each respective batch member; the return value states whether the algorithm should stop. The default is <code>tfp\$optimizer.converged_all</code> which only stops when all batch members have either converged or failed. An alternative is <code>tfp\$optimizer.converged_any</code> which stops as soon as one batch member has converged, or when all have failed.
<code>validate_args</code>	Logical, default <code>TRUE</code> . When <code>TRUE</code> , optimizer parameters are checked for validity despite possibly degrading runtime performance. When <code>FALSE</code> invalid inputs may silently render incorrect outputs.
<code>max_line_search_iterations</code>	Python int. The maximum number of iterations for the <code>hager_zhang</code> line search algorithm.
<code>f_absolute_tolerance</code>	Scalar Tensor of real dtype. If the absolute change in the objective value between one iteration and the next is smaller than this value, the algorithm is stopped.
<code>learning_rate</code>	the size of steps (in parameter space) towards the optimal value. Default value $0.01$

momentum	hyperparameter that accelerates gradient descent in the relevant direction and dampens oscillations. Defaults to 0, which is vanilla gradient descent.
nesterov	Whether to apply Nesterov momentum. Defaults to FALSE.
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)
beta_1	exponential decay rate for the 1st moment estimates
beta_2	exponential decay rate for the 2nd moment estimates
amsgrad	Boolean. Whether to apply AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and beyond". Defaults to FALSE.
learning_rate_power	power on the learning rate, must be 0 or less
l2_shrinkage_regularization_strength	A float value, must be greater than or equal to zero. This differs from L2 above in that the L2 above is a stabilization penalty, whereas this L2 shrinkage is a magnitude penalty. When input is sparse shrinkage will only happen on the active weights.
beta	A float value, representing the beta value from the paper by <a href="#">McMahan et al 2013</a> . Defaults to 0
centered	Boolean. If TRUE, gradients are normalized by the estimated variance of the gradient; if FALSE, by the uncentered second moment. Setting this to TRUE may help with training, but is slightly more expensive in terms of computation and memory. Defaults to FALSE.

### Details

The optimisers `powell()`, `cg()`, `newton_cg()`, `l_bfgs_b()`, `tnc()`, `cobyla()`, and `slsqp()` are now defunct. They will error when called in `greta 0.5.0`. These are removed because they are no longer available in TensorFlow 2.0. Note that optimiser `momentum()` has been replaced with `gradient_descent()`

### Value

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

**Note**

This optimizer isn't supported in TF2, so proceed with caution. See the [TF docs on AdagradDAOptimiser](#) for more detail.

This optimizer isn't supported in TF2, so proceed with caution. See the [TF docs on ProximalGradientDescentOptimizer](#) for more detail.

This optimizer isn't supported in TF2, so proceed with caution. See the [TF docs on ProximalAdagradOptimizer](#) for more detail.

**Examples**

```
## 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**

```
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`, `INDEX`, `symmetric`, `only.values`, `EISPACK`, `x1`, `x2`, `compact`, `along`, `rev.along`, `new.names`, `force.array`, `make.names`, `use.anon.names`, `use.first.dimnames`, `hier.names`, `use.dnns`, `nrow`, `ncol`  
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.

---

print.greta\_deps\_spec *Print method for greta python deps*

---

### Description

Print method for greta python deps

### Usage

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

### Arguments

x	greta python deps
...	extra args, not used

---

print.greta\_mcmc\_list *Print method for greta MCMC list*

---

### Description

Print method for greta MCMC list

### Usage

```
## S3 method for class 'greta_mcmc_list'  
print(x, ..., n = 5)
```

### Arguments

x	greta mcmc list
...	extra args (currently not used)
n	number of lines to print

### Value

printed MCMC output

---

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-tf2' conda environment
- [remove\\_miniconda\(\)](#) removes miniconda installation
- [reinstall\\_greta\\_env\(\)](#) remove 'greta-env-tf2' and reinstall it using [greta\\_create\\_conda\\_env\(\)](#) (which is used internally).
- [reinstall\\_miniconda\(\)](#) removes miniconda and reinstalls it using [greta\\_install\\_miniconda\(\)](#) (which is used internally)

### Value

invisible

### See Also

[destroy\\_greta\\_deps\(\)](#)

### Examples

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

## End(Not run)
```

---

run_optimiser	<i>Dispatch optimisation method to right class</i>
---------------	--

---

**Description**

Should also allow for building other methods in the future

**Usage**

```
run_optimiser(self)
```

**Arguments**

self                    optimiser of class: tf\_optimiser, tfp\_optimiser, or tf\_compat\_optimiser.

---

samplers	<i>MCMC samplers</i>
----------	----------------------

---

**Description**

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

**Usage**

```
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 implemented for uniform and normal proposals.

`slice()` implements a multivariate slice sampling algorithm. 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**

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

**Arguments**

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

**Details**

This is essentially a wrapper around `calculate()` that finds all relevant greta arrays. See that function 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 environment) 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**

```
## 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)
# nolint start
```

---

structures	<i>create data greta arrays</i>
------------	---------------------------------

---

**Description**

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  $-\infty$  to  $\infty$ ) 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:

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	<i>create greta variables</i>
----------	-------------------------------

---

### 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 <code>-Inf</code> (lower) or <code>Inf</code> (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 symmetric 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))
```

---

```
write_greta_install_log
```

*Write greta dependency installation log file*

---

**Description**

This can only be run after installation has happened with `install_greta_deps()`, and before restarting R.

**Usage**

```
write_greta_install_log(path = greta_logfile)
```

58

*write\_greta\_install\_log*

**Arguments**

path                    a path with an HTML (.html) extension.

**Value**

nothing - writes to file

# Index

## \* datasets

greta\_deps\_tf\_tfp, 25  
.internals (internals), 34  
%% (overloaded), 47

abind (overloaded), 47  
adadelta (optimisers), 41  
adagrad (optimisers), 41  
adagrad\_da (optimisers), 41  
adam (optimisers), 41  
adamax (optimisers), 41  
apply (overloaded), 47  
are\_null, 3  
array(), 54  
as.greta\_model, 4  
as.unknowns, 4  
as.vector(), 54  
as\_data, 5

backsolve (overloaded), 47  
bernoulli (distributions), 13  
beta (distributions), 13  
beta\_binomial (distributions), 13  
bfgs (optimisers), 41  
binomial (distributions), 13

c (extract-replace-combine), 17  
calculate, 6  
calculate(), 21, 29, 30, 52  
callr::r\_process\_options(), 23, 25  
categorical (distributions), 13  
cauchy (distributions), 13  
cbind (extract-replace-combine), 17  
cg (optimisers), 41  
chi\_squared (distributions), 13  
chol.greta\_array, 9  
chol2inv (overloaded), 47  
chol2symm, 9  
cholesky\_variable (variable), 56  
cobyala (optimisers), 41

colMeans (overloaded), 47  
colSums (overloaded), 47  
cov2cor (overloaded), 47  
cpu\_only (gpu\_cpu), 21

destroy\_greta\_deps, 10  
destroy\_greta\_deps(), 50  
diag (overloaded), 47  
DiagrammeR::dgr\_graph(), 39  
DiagrammeR::grViz(), 39  
dim.node, 11  
dim<- .unknowns, 11  
dirichlet (distributions), 13  
dirichlet\_multinomial (distributions),  
13  
distribution, 12  
distribution(), 13, 14  
distribution<- (distribution), 12  
distributions, 13  
distributions(), 12, 36, 37, 56

eigen (overloaded), 47  
exponential (distributions), 13  
extra\_samples (inference), 27  
extract (extract-replace-combine), 17  
extract-replace-combine, 17  
extraDistr::dbbinom, 15  
extraDistr::dbern, 15  
extraDistr::ddirichlet, 16  
extraDistr::ddirmnom, 16  
extraDistr::dinvgamma, 15  
extraDistr::dlaplace, 15  
extraDistr::dlst, 15  
extraDistr::dpareto, 15

f (distributions), 13  
forwardsolve (overloaded), 47  
ftrl (optimisers), 41  
functions, 18, 40, 54  
future::cluster(), 30

- gamma (distributions), 13
- gpu\_cpu, 21
- gpu\_only (gpu\_cpu), 21
- gradient\_descent (optimisers), 41
- greta, 21
- greta-package (greta), 21
- greta\_array (structures), 53
- greta\_create\_conda\_env, 22
- greta\_create\_conda\_env(), 50
- greta\_deps\_receipt, 23
- greta\_deps\_spec, 24
- greta\_deps\_spec(), 23, 33
- greta\_deps\_tf\_tfp, 25
- greta\_install\_miniconda, 25
- greta\_install\_miniconda(), 50
- greta\_mcmc\_list(), 7
- greta\_model(), 52
- greta\_notes\_tf\_num\_error, 26
- greta\_set\_install\_logfile, 26
- greta\_set\_install\_logfile(), 33, 40
- greta\_sitrep, 27
  
- hmc (samplers), 51
- hypergeometric (distributions), 13
  
- icauchit (transforms), 54
- icloglog (transforms), 54
- identity (overloaded), 47
- ilogit (transforms), 54
- imultilogit (transforms), 54
- inference, 27
- initials (inference), 27
- install\_greta\_deps, 32
- install\_greta\_deps(), 23, 25, 26, 57
- internals, 34
- inverse-links (transforms), 54
- inverse\_gamma (distributions), 13
- iprobit (transforms), 54
- is.greta\_array, 35
- is.greta\_mcmc\_list, 35
  
- joint, 36
  
- l\_bfgs\_b (optimisers), 41
- laplace (distributions), 13
- lkj\_correlation (distributions), 13
- log1pe (transforms), 54
- logistic (distributions), 13
- lognormal (distributions), 13
  
- mcmc (inference), 27
- mcmc(), 6, 21, 38, 51, 52
- mixture, 37
- mixture(), 13
- model, 38
- momentum (optimisers), 41
- multinomial (distributions), 13
- multivariate\_normal (distributions), 13
- mvtnorm::dmvnorm, 15
  
- nadam (optimisers), 41
- negative\_binomial (distributions), 13
- nelder\_mead (optimisers), 41
- newton\_cg (optimisers), 41
- normal (distributions), 13
  
- ones (structures), 53
- open\_greta\_install\_log, 40
- open\_greta\_install\_log(), 33
- operators, 18, 40, 54
- opt (inference), 27
- opt(), 21, 41, 46
- optimisers, 41
- optimisers(), 29
- ordered\_variable (variable), 56
- overloaded, 47
  
- pareto (distributions), 13
- plot.greta\_model (model), 38
- poisson (distributions), 13
- powell (optimisers), 41
- print.greta\_deps\_spec, 49
- print.greta\_mcmc\_list, 49
- print.greta\_model (model), 38
- proximal\_adagrad (optimisers), 41
- proximal\_gradient\_descent (optimisers), 41
  
- rbind (extract-replace-combine), 17
- rdist (overloaded), 47
- reinstall\_greta\_deps
  - (install\_greta\_deps), 32
- reinstall\_greta\_env (reinstallers), 50
- reinstall\_greta\_env(), 50
- reinstall\_miniconda (reinstallers), 50
- reinstall\_miniconda(), 50
- reinstallers, 50
- remove\_greta\_env (reinstallers), 50
- remove\_greta\_env(), 10, 50

remove\_miniconda (reinstallers), 50  
remove\_miniconda(), 10, 50  
rep (extract-replace-combine), 17  
replace (extract-replace-combine), 17  
reticulate::conda\_create(), 23  
reticulate::install\_miniconda(), 25  
reticulate::py\_install(), 33  
rms\_prop (optimisers), 41  
rowMeans (overloaded), 47  
rowSums (overloaded), 47  
run\_optimiser, 51  
rwmh (samplers), 51

samplers, 51  
samplers(), 28  
simplex\_variable (variable), 56  
simulate.greta\_model, 52  
slice (samplers), 51  
slsqp (optimisers), 41  
stashed\_samples (inference), 27  
stats::dbeta, 15  
stats::dbinom, 15  
stats::dcauchy, 15  
stats::dchisq, 15  
stats::dexp, 15  
stats::df, 15  
stats::dgamma, 15  
stats::dhyper, 15  
stats::dlnorm, 15  
stats::dlogis, 15  
stats::dmultinom, 15  
stats::dnbinom, 15  
stats::dnorm, 15  
stats::dpois, 15  
stats::dunif, 15  
stats::dweibull, 15  
stats::optim(), 30  
stats::rWishart, 16  
structures, 53  
student (distributions), 13  
sweep (overloaded), 47

tapply (overloaded), 47  
tensorflow::set\_random\_seed(), 31  
tnc (optimisers), 41  
transforms, 18, 40, 54

uniform (distributions), 13  
utils::browseURL(), 40

variable, 56  
variable(), 14

weibull (distributions), 13  
wishart (distributions), 13  
write\_greta\_install\_log, 57

zeros (structures), 53