Package 'greta'

July 20, 2026

Type Package

Title Simple and 'Scalable' Statistical Modelling in R

Version 0.6.0

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. This work is discussed at Golding (2019) <doi:10.21105/joss.01601>.

License Apache License 2.0

URL https://greta-dev.github.io/greta/, 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), lifecycle, methods, parallelly (>= 1.29.0), progress (>= 1.2.0), R6, reticulate (>= 1.43.0), 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, rmutil, rsvg, spelling, testthat (>= 3.1.0), tibble, tidyr, truncdist, withr, rstudioapi

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

Language en-US

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' 'sampler_class.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'
'python_backend.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' 'greta-sitrep.R'

LazyData true

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Nick Golding [aut, cph] (ORCID:
<https://orcid.org/0000-0001-8916-5570>),
Nicholas Tierney [aut, cre] (ORCID:
<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 2026-07-20 12:20:30 UTC

Contents

as.greta_model *Convert object to a "greta_model" object*

Description

Convert object to a "greta_model" object

Usage

```
as.greta_model(x, ...)
```

Arguments

x	object to convert to greta model
...	extra arguments - not used.

Value

A greta_model object.

as.unknowns *Create objects of class 'unknowns' to nicely print ? valued arrays*

Description

Create objects of class 'unknowns' to nicely print ? valued arrays

Usage

```
as.unknowns(x)
```

Arguments

x	object to convert to "unknowns" class

Value

An object of class unknowns.

as_data *convert other objects to greta arrays*

Description

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

Usage

```
as_data(x)
```

Arguments

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

Details

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

Value

A data `greta_array` object.

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 `mcmc()`.
`nsim`	an optional positive integer scalar for the number of responses to simulate if stochastic greta arrays are present in the model - see Details.
`seed`	an optional seed to be used in set.seed immediately before the simulation so as to generate a reproducible sample
`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 `cpu_only()`. Use `gpu_only()` to use only GPU. In the future we will add more options for specifying CPU and GPU use. If setting GPU with `gpu_only()` 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 `cpu_only()`, which is the default. You can turn off the message about setting seed with GPU usage using `options(greta_gpu_message = FALSE)`

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.

Value

An upper-triangular `greta_array` representing the Cholesky factor of `x`.

`chol2symm` *Cholesky Factor to Symmetric Matrix*

Description

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

Usage

```
chol2symm(x)
```

Arguments

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

Value

A symmetric, positive-definite matrix or `greta_array`.

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)
```

`dim.node` *generic to grab dimensions of nodes*

Description

generic to grab dimensions of nodes

Usage

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

Arguments

x	greta node class

Value

An integer vector of the node's dimensions.

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

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

Value

The object x, of class unknowns, with its dimensions updated.

distribution *define a distribution over data*

Description

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

Usage

```
distribution(greta_array) <- value

distribution(greta_array)
```

Arguments

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

Details

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

Value

`distribution<-` returns `greta_array` (invisibly), the data greta array with the distribution assigned. `distribution()` returns the `greta_array` if it has a distribution, or `NULL` if it doesn't.

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` *probability distributions*

Description

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

Usage

```
uniform(min, max, dim = NULL)

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

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

bernoulli(prob, dim = NULL)

binomial(size, prob, dim = NULL)

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

```
negative_binomial(size, prob, dim = NULL)

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

poisson(lambda, dim = NULL)

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

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

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

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

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

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

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

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

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 dirichlet and dirichlet_multinomial.

truncation — a length-two vector giving values between which to truncate the distribution, similarly to the lower and upper arguments to variable()

prob — probability parameter (0 < prob < 1), must be a vector for multinomial and categorical

size, m, n, k — positive integer parameter

Sigma — positive definite variance-covariance matrix parameter

n_realisations — the number of independent realisation of a multivariate distribution

dimension — the dimension of a multivariate distribution

Details

The discrete probability distributions (bernoulli, binomial, negative_binomial, poisson, multinomial, categorical, dirichlet_multinomial) can be used when they have fixed values (e.g. defined as a likelihood using distribution(), but not as unknown variables.

For univariate distributions dim gives the dimensions of the greta array to create. Each element of the greta array will be (independently) distributed according to the distribution. dim can also be left at its default of NULL, in which case the dimension will be detected from the dimensions of the parameters (provided they are compatible with one another).

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

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

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

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

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

Value

A variable `greta_array` following the specified probability distribution.

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.

Usage

```
## S3 method for class 'greta_array'
x[...]

## S3 replacement method for class 'greta_array'
x[...] <- value

## S3 method for class 'greta_array'
```

```
cbind(...)

## S3 method for class 'greta_array'
rbind(...)

## S3 method for class 'greta_array'
c(...)

## S3 method for class 'greta_array'
rep(x, ...)

## S3 method for class 'greta_array'
dim(x)

## S3 method for class 'greta_array'
length(x)

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

## S3 method for class 'greta_array'
head(x, n = 6L, ...)

## S3 method for class 'greta_array'
tail(x, n = 6L, ...)

## S3 method for class 'greta_array'
diag(x = 1, nrow, ncol)
```

Arguments

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

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.

Value

A `greta_array`, with elements extracted, replaced, or combined as appropriate for the method called.

Syntax

These methods can be used with greta arrays as follows:

```
# 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`, `solve()`, and `log()` 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.

Value

A `greta_array`, with the function applied elementwise or as appropriate for the function called.

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()
```

Value

A single character string, "GPU" or "CPU".

Examples

```
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:

- Nicholas Tierney <nicholas.tierney@gmail.com> (ORCID)
- Nick Golding <nick.golding.research@gmail.com> (ORCID) [copyright holder]

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-dev.github.io/greta/
- 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 `greta_deps_spec()` for more details.

Value

nothing - creates a conda environment for a specific python version

---

greta_deps_receipt *Capture greta python dependencies.*

Description

To assist with capturing and sharing python dependencies, we provide a way to capture the dependencies currently used. Unlike `greta_deps_spec()`, the receipt records the versions actually installed and is **not** validated against the versions greta supports - so it will faithfully report, for example, a TensorFlow version newer than greta's supported range.

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 2.15.1, 0.23.0, and 3.11, respectively. greta checks it supports the TF version (greta does not support TF 2.16 or later, which ship Keras 3); compatible TFP and Python versions are resolved at install time by uv (or, for a conda environment, by conda/pip). The `greta_deps_tf_tfp` dataset lists known-good combinations of TF, TFP, and Python; inspect it with `View(greta_deps_tf_tfp)`.

Usage

```
greta_deps_spec(
  tf_version = "2.15.1",
  tfp_version = "0.23.0",
  python_version = "3.11"
)
```

Arguments

`tf_version`	character. TensorFlow version, in the format major.minor.patch. Default is 2.15.1.
`tfp_version`	Character. Tensorflow probability (TFP) version major.minor.patch. Default is 0.23.0.
`python_version`	Character. Python version in format major.minor.patch. Default is 3.11.

Details

Calling `greta_deps_spec()` with no arguments returns greta's current default (recommended) versions, and is the supported way to query them - for example `greta_deps_spec()$tf_version` for the default TensorFlow version.

Value

data frame of valid dependencies

Examples

```
greta_deps_spec()
greta_deps_spec(tf_version = "2.15.1")
greta_deps_spec(tf_version = "2.15.0")
greta_deps_spec(tf_version = "2.15.1", tfp_version = "0.23.0")
greta_deps_spec(tf_version = "2.15.0", tfp_version = "0.23.0")
greta_deps_spec(tf_version = "2.15.1", python_version = "3.10")
greta_deps_spec(tf_version = "2.15.0", python_version = "3.10")
greta_deps_spec(
  tf_version = "2.14.0",
  tfp_version = "0.22.1",
  python_version = "3.10"
  )
# this will fail: greta does not support TF 2.16+ (Keras 3)
## Not run:
greta_deps_spec(tf_version = "2.16.0")

## 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, 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_list_py_modules *List Python modules installed in greta env*

Description

List Python modules installed in greta env

Usage

```
greta_list_py_modules()
```

Value

matrix/data frame of Python modules that are installed in the greta environment - showing the name, version, build, and install channel.

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()
```

Value

Invisibly returns `NULL`; called for its side effect of printing the stored message.

Examples

```
## Not run:
greta_notes_tf_num_error()

## End(Not run)
```

greta_remove *Remove greta's Python dependencies*

Description

A single entry point for removing greta's Python bits, which is useful when debugging a greta installation and you want to get back to a "clean slate". Use the `what` argument to choose how much to remove.

Usage

```
greta_remove(
  what = c("all", "env", "miniconda", "uv_cache", "preference", "deps"),
  ask = interactive()
)
```

Arguments

`what`	What to remove. One of: • `"all"` (default): the `"greta-env-tf2"` conda environment, miniconda, reticulate's uv cache, and greta's stored preferences (the Python backend set via `greta_set_python()`, and the dependency versions set via `greta_set_deps()`). This is a "nuclear" reset that asks once, then removes everything it finds. • `"env"`: the `"greta-env-tf2"` conda environment. • `"miniconda"`: the miniconda installation. • `"uv_cache"`: reticulate's uv cache. Note this cache is shared by all R packages that use reticulate's uv (it is not greta-specific); a system-wide uv cache is left untouched. • `"preference"`: greta's stored Python backend preference (set via `greta_set_python()`). • `"deps"`: greta's stored dependency versions (set via `greta_set_deps()`).
`ask`	Ask for confirmation? Default is `interactive()`.

Value

Invisibly, `TRUE` if anything was removed, otherwise `FALSE`.

See Also

`reinstall_greta_deps()`, `greta_set_python()`

Examples

```
## Not run:
# remove everything (nuclear reset)
greta_remove()

# remove only the conda environment
```

```
greta_remove("env")

# clear the stored Python preference
greta_remove("preference")

# clear the stored dependency versions
greta_remove("deps")

## End(Not run)
```

`greta_set_deps` *Choose the dependency versions greta installs*

Description

Persistently choose which versions of TensorFlow, TensorFlow Probability, and Python greta uses, independently of *where* they are installed (see `greta_set_python()` for that). The stored versions are used by:

- the managed (uv) environment, which installs them automatically on first use after a restart, and
- `install_greta_deps()`, as its default `deps` argument when building a conda environment.

Most users never need this: greta's defaults (TensorFlow 2.15.1, TensorFlow Probability 0.23.0, Python 3.11) are the newest versions greta supports.

To clear the stored versions and return to the defaults, use `greta_remove()("deps")`.

Usage

```
greta_set_deps(deps = greta_deps_spec())
```

Arguments

`deps`	object created with `greta_deps_spec()`, which checks that the TensorFlow version is one greta supports.

Details

Your choice is stored under `tools::R_user_dir("greta", "config")` and applied the next time greta is loaded, so you will need to **restart R** for it to take effect. Changing versions on the managed (uv) backend may require internet access on the next load, to download the newly requested versions.

Value

Invisibly, the stored `greta_deps_spec()`.

See Also

greta_set_python(), greta_deps_spec(), install_greta_deps()

Examples

```
## Not run:
# pin an older TensorFlow for the managed (uv) environment
greta_set_deps(greta_deps_spec(
  tf_version = "2.14.0",
  tfp_version = "0.22.1",
  python_version = "3.10"
))

# clear the stored versions and return to greta's defaults
greta_remove("deps")

## 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_set_python` *Choose the Python environment greta uses*

Description

greta runs on Python (via TensorFlow and TensorFlow Probability). By default it uses uv (via the reticulate R package) to install a compatible Python, TensorFlow, and TensorFlow Probability automatically on first use. `greta_set_python()` persistently selects which Python environment greta uses: the managed (uv) environment, a conda environment (for example one created by `install_greta_deps()`), or your own Python. `greta_reset_python()` clears the stored choice, returning to greta's automatic resolution.

To choose which *versions* of TensorFlow and TensorFlow Probability the managed (uv) environment installs, see `greta_set_deps()` - dependency versions are separate from the choice of Python environment.

Usage

```
greta_set_python(backend = c("uv", "conda", "path"), path = NULL, name = NULL)

greta_reset_python()
```

Arguments

backend	Which Python environment to use. One of: • `"uv"` (default): the managed (uv) environment. reticulate installs a compatible Python, TensorFlow, and TensorFlow Probability automatically on first use. • `"conda"`: a conda environment, named by `name`. • `"path"`: a specific Python, given by `path`.
path	Only for `backend = "path"`. Path to a Python executable, or to an environment directory (a virtualenv or conda prefix) containing one. When given a directory, greta looks for `bin/python` (Unix) or `Scripts/python.exe` (Windows) inside it. Pointing at an already-installed environment on disk never downloads anything, which makes it useful for offline or restricted-network setups.
name	Only for `backend = "conda"`. Name of the conda environment to use. Defaults to `"greta-env-tf2"`, the environment created by `install_greta_deps()`.

Details

greta resolves which Python to use, in this order:

1. The `RETICULATE_PYTHON` environment variable, if set (usually in `~/.Renviron`, your `.Rprofile`, or your shell environment). This always wins: it takes precedence over any stored preference.
2. Your stored preference, set with `greta_set_python()`.
3. An auto-detected `"greta-env-tf2"` conda environment (created by `install_greta_deps()`) - kept so setups from older greta versions keep working after upgrading.

4. Otherwise, the managed (uv) environment (the default as of greta 0.6.0): reticulate installs a compatible Python, TensorFlow, and TensorFlow Probability automatically on first use. No setup is needed - this happens "automagically".

For the managed (uv) environment, greta automatically enables uv's offline mode once the environment is installed, so it no longer reaches out to PyPI. Set `UV_OFFLINE=0` yourself to force online resolution (for example, to refresh the environment), or `UV_OFFLINE=1` to force offline mode - greta never overrides a value you have already set.

To check which Python greta is currently using, and which it will use after a restart, call `greta_sitrep()`.

If a stored preference appears to be ignored, `RETICULATE_PYTHON` is usually why: remove it from wherever it is set (for example `~/.Renviron`), then restart R. Note that `Sys.unsetenv()` within a session is not enough, as the choice is applied when greta loads.

Your choice is stored under `tools::R_user_dir("greta", "config")` and applied the next time greta is loaded, so you will need to **restart R** for it to take effect.

Value

Invisibly, the stored preference (`NULL` for `greta_reset_python()`).

See Also

`greta_set_deps()`, `greta_sitrep()`, `install_greta_deps()`, `greta_remove()`

Examples

```
## Not run:
# use the managed (uv) environment (the default)
greta_set_python()

# use the conda environment from install_greta_deps()
greta_set_python("conda")

# use a differently-named conda environment
greta_set_python("conda", name = "my-tf-env")

# use a specific Python binary, or an environment directory
greta_set_python("path", path = "/path/to/python")
greta_set_python("path", path = "/opt/python-envs/greta")

# clear the stored choice and return to automatic resolution
greta_reset_python()

## End(Not run)
```

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(verbosity = c("minimal", "detailed", "quiet"))
```

Arguments

verbosity	character. How verbose the output of the situation report. Possible options: "minimal" (default), "detailed", and "quiet". "Minimal" provides just information in python version, tensorflow version, tensorflow probability, and whether greta conda environment is available. "Quiet" presents no information, but prepares greta to be used. "Detailed" gives information on the version and path for R, greta, python, tensorflow, tensorflow probability, the greta conda environment, and a statement on greta usability.

Value

Message on greta situation report. See "verbosity" parameter details above for more information.

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 = 2,
  n_cores = NULL,
  verbose = TRUE,
  pb_update = 50,
  one_by_one = FALSE,
  initial_values = initials(),
  trace_batch_size = 100,
  compute_options = cpu_only()
)

stashed_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 samplers() for options.

n_samples	number of MCMC samples to draw per chain (after any warm-up, but before thinning)
thin	MCMC thinning rate; every thin samples is retained, the rest are discarded
warmup	number of samples to spend warming up the mcmc sampler (moving chains toward the highest density area and tuning sampler hyperparameters).
chains	number of MCMC chains to run. Default is 2. We recommend using more chains as this helps improve convergence. However the number of chains specified can increase the CPU load, so we have to set a lower default value.
n_cores	the maximum number of CPU cores used by each sampler (see details). If NULL (default), it sets them to 2 cores.
verbose	whether to print progress information to the console
pb_update	how regularly to update the progress bar (in iterations). If pb_update is less than or equal to thin, it will be set to thin + 1 to ensure at least one saved iteration per pb_update iterations.
one_by_one	whether to run TensorFlow MCMC code one iteration at a time, so that greta can handle numerical errors as 'bad' proposals (see below).
initial_values	an optional initials object (or list of initials objects of length chains) giving initial values for some or all of the variables in the model. These will be used as the starting point for sampling/optimisation.
trace_batch_size	the number of posterior samples to process at a time when tracing the parameters of interest; reduce this to reduce memory demands
compute_options	Default is to use CPU only with cpu_only(). Use gpu_only() to use only GPU. In the future we will add more options for specifying CPU and GPU use.
draws	a greta_mcmc_list object returned by mcmc or stashed_samples
...	named numeric values, giving initial values of some or all of the variables in the model (unnamed variables will be automatically initialised)
optimiser	an optimiser object giving the optimisation algorithm and parameters See optimisers().
max_iterations	the maximum number of iterations before giving up
tolerance	the numerical tolerance for the solution, the optimiser stops when the (absolute) difference in the joint density between successive iterations drops below this level
adjust	whether to account for Jacobian adjustments in the joint density. Set to FALSE (and do not use priors) for maximum likelihood estimates, or TRUE for maximum *a posteriori* estimates.
hessian	whether to return a list of *analytically* differentiated Hessian arrays for the parameters

Details

For mcmc() if verbose = TRUE, the progress bar shows the number of iterations so far and the expected time to complete the phase of model fitting (warmup or sampling). Occasionally, a proposed set of parameters can cause numerical instability (I.e. the log density or its gradient is NA, Inf or

-Inf); normally because the log joint density is so low that it can't be represented as a floating point number. When this happens, the progress bar will also display the proportion of proposals so far that were 'bad' (numerically unstable) and therefore rejected. Some numerical instability during the warmup phase is normal, but 'bad' samples during the sampling phase can lead to bias in your posterior sample. If you only have a few bad samples (<10\%), you can usually resolve this with a longer warmup period or by manually defining starting values to move the sampler into a more reasonable part of the parameter space. If you have more samples than that, it may be that your model is misspecified. You can often diagnose this by using calculate() to evaluate the values of greta arrays, given fixed values of model parameters, and checking the results are what you expect.

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

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

If n_cores = NULL and mcmc samplers are being run sequentially, each sampler will be allowed to use only 2 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 can use `set.seed()`, or `tensorflow::set_random_seed()`. They both give identical results. See examples below.

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,
# othewise 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

# using set.seed or tensorflow::set_random_seed to set RNG for MCMC
a <- normal(0, 1)
y <- normal(a, 1)
m <- model(y)

set.seed(12345)
one <- mcmc(m, n_samples = 1, chains = 1)
set.seed(12345)
two <- mcmc(m, n_samples = 1, chains = 1)
# same
all.equal(as.numeric(one), as.numeric(two))
tensorflow::set_random_seed(12345)
one_tf <- mcmc(m, n_samples = 1, chains = 1)
tensorflow::set_random_seed(12345)
two_tf <- mcmc(m, n_samples = 1, chains = 1)
```

```
# same
all.equal(as.numeric(one_tf), as.numeric(two_tf))
# different
all.equal(as.numeric(one), as.numeric(one_tf))


## End(Not run)
```

`install_greta_deps` *Install Python dependencies for greta*

Description

This is a helper function to install specified versions of Python dependencies needed for greta. By default, greta version >= 0.6.0 now uses reticulate's managed (uv) Python environment to automatically identify dependencies. You can change over to this new approach with `greta_set_python()`, which is now what we recommend. This has changed from where we would previously use `install_greta_deps()`.

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"),
  ask = interactive()
)
```

Arguments

`deps`	object created with `greta_deps_spec()` where you specify python, TensorFlow (TF), and TensorFlow Probability (TFP) versions. By default these are TF 2.15.1, TFP 0.23.0, and Python 3.11. `greta_deps_spec()` checks that the TF version is one greta supports; compatible TFP and Python versions are resolved at install time. See ?`greta_deps_spec()` for more information, and the data object `greta_deps_tf_tfp` for known-good combinations. If you have stored a preference with `greta_set_deps()`, it is used when deps is not supplied.
`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 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.
ask	Logical; for reinstall_greta_deps(), whether to ask for confirmation before removing the existing greta conda environment. Defaults to interactive().

Details

This function, install_greta_deps(), is an alternative installation workflow. The default versions of the python modules are: TensorFlow 2.15.1, TensorFlow Probability 0.23.0, and Python 3.11. These Python modules will be installed into a conda environment named "greta-env-tf2".

It can be useful to identify installation notes, warnings, or errors that arise during install. You can do this by accessing the logfile with open_greta_install_log(), which opens your logfile in your default web browser. The logfile of the installation process is written to a user directory, by default to tools::R_user_dir("greta"), and is named: "greta-installation-logfile.html".

You can set the logfile location with greta_set_install_logfile(). E.g., greta_set_install_logfile('path/to/logf
You can also specify this with an environment variable, GRETA_INSTALLATION_LOG, e.g., Sys.setenv('GRETA_INSTALLATIO

By default, if using RStudio, it will 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.

Value

Invisibly returns NULL; called for its side effect of installing greta's Python dependencies into a conda environment.

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.

We now recommend using the new default method for installation, which uses uv (via the reticulate package) to install TensorFlow and TensorFlow Probability on first use. To make greta use the "greta-env-tf2" conda environment created here instead, use greta_set_python("conda") (or set the RETICULATE_PYTHON environment variable to its Python before loading greta). See the "Installing Dependencies" vignette and greta_set_python().

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 tensorflows website), see the data greta_deps_tf_tfp, which is provided with greta. Managing your own installation is not always straightforward, so proceed with caution.

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.

Value

A nested list (`module`) of internal greta functions and R6 generators.

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)

Value

A single logical value.

Examples

```
is.greta_array(1)
```

`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

Examples

```
is.greta_mcmc_list(1)
```

`joint` *define joint distributions*

Description

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

Usage

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

Arguments

`...`	scalar variable greta arrays following probability distributions (see `distributions()`); 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`.

Value

A variable `greta_array` following the joint distribution.

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` *mixtures of probability distributions*

Description

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

Usage

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

Arguments

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

Details

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

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

Value

A variable `greta_array` following the mixture distribution.

Examples

```
## Not run:
# a scalar variable following a strange bimodal distibution
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", ...)
```

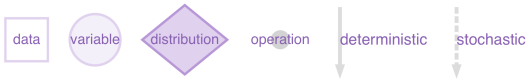
Arguments

...	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 XLA JIT compilation to the TensorFlow graph representing the model. This may slow down model definition, and speed up model evaluation.
x	a greta_model object
y	unused default argument
colour	base colour used for plotting. Defaults to greta colours in violet.

Details

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

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



Value

model - a greta_model object.

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

Examples

```
## Not run:

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

m <- model(mu, sigma)

plot(m)

## End(Not run)
```

`open_greta_install_log` *Read a greta logfile*

Description

This is a convenience function to facilitate reading logfiles. It opens a HTML 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.html')`. 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 HTML 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.

Value

A greta_array, the result of applying the operator.

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()

cobyla()

slsqp()

gradient_descent(learning_rate = 0.01, momentum = 0, nesterov = FALSE)

adadelta(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

`objective_function`
: 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 `batch_evaluate_objective` 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.

`initial_vertex`
: Tensor of real dtype and any shape that can be consumed by the `objective_function`. A single point in the domain that will be used to construct an axes aligned initial simplex.

`step_sizes`
: Tensor of real dtype and shape broadcasting compatible with `initial_vertex`. Supplies the simplex scale along each axes.

`func_tolerance`
: 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.

`position_tolerance`
: Single numeric number. The algorithm stops if the largest absolute difference between the coordinates of the vertices is below this threshold.

`reflection`
: (optional) Positive Scalar Tensor of same dtype as `initial_vertex`. This parameter controls the scaling of the reflected vertex. See, Press et al(2007) for details. If not specified, uses the dimension dependent prescription of Gao and Han (2012) doi:10.1007/s1058901093293

`expansion`
: (optional) Positive Scalar Tensor of same dtype as `initial_vertex`. Should be greater than 1 and reflection. This parameter controls the expanded scaling of a reflected vertex.See, Press et al(2007) for details. If not specified, uses the

dimension dependent prescription of Gao and Han (2012) doi:10.1007/s10589-01093293

`contraction` (optional) Positive scalar Tensor of same dtype as `initial_vertex`. 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, Press et al(2007) for details. If not specified, uses the dimension dependent prescription of Gao and Han (2012) doi:10.1007/s1058901093293

`shrinkage` (Optional) Positive scalar Tensor of same dtype as `initial_vertex`. 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, Press et al(2007) for details. If not specified, uses the dimension dependent prescription of Gao and Han (2012) doi:10.1007/s1058901093293

`value_and_gradients_function`
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 `[..., 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 `[...]`. The second component (the gradient) should also be of shape `[..., n]` like the input value to the function.

`initial_position`
real Tensor of shape `[..., 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.

`tolerance` 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.

`x_tolerance` 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.

`f_relative_tolerance`
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.

`initial_inverse_hessian_estimate`
Optional Tensor of the same dtype as the components of the output of the value_and_gradients_function. If specified, the shape should broadcastable to shape `[..., 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.

`stopping_condition`
(Optional) A function that takes as input two Boolean tensors of shape `[...]`, and returns a Boolean scalar tensor. The input tensors are converged and failed, indicating the current status of each respective batch member; the return value

states whether the algorithm should stop. The default is `tfp$optimizer.converged_all` which only stops when all batch members have either converged or failed. An alternative is `tfp$optimizer.converged_any` which stops as soon as one batch member has converged, or when all have failed.

`validate_args` Logical, default TRUE. When TRUE, optimizer parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs.

`max_line_search_iterations`
Python int. The maximum number of iterations for the hager_zhang line search algorithm.

`f_absolute_tolerance`
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.

`learning_rate` 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 McMahan et al 2013. 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. This 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, ...)

outer(X, Y, FUN = "*", ...)

X %o% Y

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, Y, 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.

Value

An object of the same type as the base R equivalent (typically a greta_array).

---

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

Value

Invisibly returns x; called for its side effect of printing x as a data frame.

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

run_optimiser *Dispatch optimisation method to right class*

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.

Value

Invisibly returns NULL; called for its side effect of running the optimiser and updating its internal state.

samplers *MCMC samplers*

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 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

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

Details

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

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 -Inf to Inf) greta array to transform to a constrained value

Details

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

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

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

Value

A `greta_array` with the transformation applied.

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` *create greta variables*

Description

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

Usage

```
variable(lower = -Inf, upper = Inf, dim = NULL)

cholesky_variable(dim, correlation = FALSE)

simplex_variable(dim)

ordered_variable(dim)
```

Arguments

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

Details

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

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

Value

A variable greta_array satisfying the requested constraints.

Examples

```
## Not run:

# a scalar variable
a <- variable()

# a positive length-three variable
b <- variable(lower = 0, dim = 3)

# a 2x2x2 variable bounded between 0 and 1
c <- variable(lower = 0, upper = 1, dim = c(2, 2, 2))

# create a variable, with lower and upper defined by greta arrays
min <- as_data(iris$Sepal.Length)
max <- min^2
```

```
d <- min + variable(0, 1, dim = nrow(iris)) * (max - min)

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

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

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

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

`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)
```

Arguments

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

Value

nothing - writes to file

Index