# Package 'greta'

January 23, 2018

**Type** Package

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

**Version** 0.2.3

**Date** 2018-01-23

**Description** Write statistical models in R and fit them by MCMC on CPUs and GPUs, using Google TensorFlow (see <https://goldingn.github.io/greta> for more information).

**License** Apache License 2.0

**URL** https://github.com/greta-dev/greta

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

**SystemRequirements** Python (>= 2.7.0) with header files and shared library; TensorFlow (>= 1.0.0; https://www.tensorflow.org/)

**Encoding** UTF-8

**LazyData** true

**Depends** R (>= 3.0)

**Collate** 'package.R' 'utils.R' 'tf_functions.R' 'overloaded.R' 'node_class.R' 'node_types.R' 'variable.R' 'probability_distributions.R' 'unknowns_class.R' 'greta_array_class.R' 'as_data.R' 'distribution.R' 'operators.R' 'functions.R' 'transforms.R' 'structures.R' 'extract_replace_combine.R' 'dag_class.R' 'greta_model_class.R' 'progress_bar.R' 'samplers.R' 'inference.R' 'install_tensorflow.R' 'internals.R'

**Imports** R6, tensorflow, reticulate, progress, coda

**Suggests** knitr, rmarkdown, DiagrammeR, bayesplot, lattice, testthat, mvtnorm, MCMCpack, rmutil, extraDistr, truncdist

**VignetteBuilder** knitr

**RoxygenNote** 6.0.1

**NeedsCompilation** no

**Author** Nick Golding [aut, cre]

**Maintainer** Nick Golding <nick.golding.research@gmail.com>

**Repository** CRAN

**Date/Publication** 2018-01-23 04:32:35 UTC

# R topics documented:



---

`as_data` *convert other objects to greta arrays*

---

## Description

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

## Usage

```
as_data(x)
```

## Arguments

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

## Details

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

## Examples

```
## Not run:

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

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

---

`distribution` *define a distribution over data*

---

## Description

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

## Usage

```
distribution(greta_array) <- value

distribution(greta_array)
```

## Arguments

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

## Details

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

## Examples

```
## 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, 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, dim = 1)

wishart(df, Sigma)

lkj_correlation(eta, dim = 2)

multinomial(size, prob, dim = 1)

categorical(prob, dim = 1)

dirichlet(alpha, dim = 1)

dirichlet_multinomial(size, alpha, dim = 1)
```

## 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

## 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_normal()`, `multinomial()`, and `categorical()` `dim` must be a scalar giving the number of rows in the resulting greta array, each row being (independently) distributed according to the multivariate normal distribution. The number of columns will always be the dimension of the distribution, determined from the parameters specified. `wishart()` always returns a single square, 2D greta array, with dimension determined from the parameter `Sigma`.

`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 parameterisation and argument names of greta distributions matches 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 |

## 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
Sig <- diag(4)
```

```
Sig[3, 4] <- Sig[4, 3] <- 0.6
theta = multivariate_normal(rep(mu, 4), Sig)

# 10 independent replicates of that
theta = multivariate_normal(rep(mu, 4), Sig, dim = 10)

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


## End(Not run)
```

---

`extract-replace-combine`

*extract, replace and combine greta arrays*

---

## Description

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

## Arguments

| | |
|---|---|
| `x` | a greta array |
| `i, j` | indices specifying elements to extract or replace |
| `n` | a single integer, as in `utils::head()` and `utils::tail()` |
| `value` | for `` `[<-` `` a greta array to replace elements, for `` `dim<-` `` either NULL or a numeric vector of dimensions |
| `...` | either further indices specifying elements to extract or replace (`[`), or multiple greta arrays to combine (`cbind()`, `rbind()` & `c()`), or additional arguments (`rep()`, `head()`, `tail()`) |
| `drop, recursive` | generic arguments that are ignored for greta arrays |

## Usage

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

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

# combine
```

```
cbind(...)
rbind(...)
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/replace
 x[1:3, ]
 x[, 2:4] <- 1:9

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

## End(Not run)
```

---

`functions` *functions for greta arrays*

---

## Description

This is a list of functions in 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()` and `solve()` will be ignored, see the TensorFlow documentation for details of these routines.

`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.

`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.

## 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)

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

# matrix operations
t(x)
chol(x, ...)
diag(x, nrow, ncol)
diag(x) <- value
solve(a, b, ...)

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

# cumulative operations
cumsum(x)
```

```
cumprod(x)

# miscellaneous operations
sweep(x, MARGIN, STATS, FUN = c('-', '+', '/', '*'))

# 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)
```

## 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)

e = diag(x)
diag(x) <- e + 1

z = t(a)

y = sweep(x, 1, e, '-')

## End(Not run)
```

---

`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.

## 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)
```

---

`inference` *statistical inference on greta models*

---

## Description

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

## Usage

```
stashed_samples()

mcmc(model, method = c("hmc"), n_samples = 1000, thin = 1, warmup = 100,
  verbose = TRUE, pb_update = 10, control = list(),
  initial_values = NULL)

opt(model, method = c("adagrad"), max_iterations = 100, tolerance = 1e-06,
  control = list(), initial_values = NULL)
```

## Arguments

| | |
|---|---|
| `model` | greta_model object |
| `method` | the method used to sample or optimise values. Currently only one method is available for each procedure: `hmc` and `adagrad` |
| `n_samples` | the number of MCMC samples to draw (after any warm-up, but before thinning) |
| `thin` | the MCMC thinning rate; every `thin` samples is retained, the rest are discarded |
| `warmup` | the number of samples to spend warming up the mcmc sampler. During this phase the sampler moves toward the highest density area and tunes sampler hyperparameters. |

| | |
|---|---|
| verbose | whether to print progress information to the console |
| pb_update | how regularly to update the progress bar (in iterations) |
| control | an optional named list of hyperparameters and options to control behaviour of the sampler or optimiser. See Details. |
| initial_values | an optional named vector of initial values for the free parameters in the model. These will be used as the starting point for sampling/optimisation |
| 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 |

## Details

If the sampler is aborted before finishing, the samples collected so far can be retrieved with `stashed_samples()`. Only samples from the sampling phase will be returned.

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). Updating the progress bar regularly slows down sampling, by as much as 9 seconds per 1000 updates. So if you want the sampler to run faster, you can change `pb_update` to increase the number of iterations between updates of the progress bar, or turn the progress bar off altogether by setting `verbose = FALSE`.

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 samples so far that were 'bad' (numerically unstable) and therefore rejected. If you're getting a lot of numerical instability, you might want to manually define starting values to move the sampler into a more reasonable part of the parameter space. Alternatively, you could redefine the model (via `model`) to have double precision, though this will slow down sampling.

Currently, the only implemented MCMC procedure is static Hamiltonian Monte Carlo (`method = "hmc"`). During the warmup iterations, the leapfrog stepsize hyperparameter `epsilon` is tuned to maximise the sampler efficiency. The `control` argument can be used to specify the initial value for epsilon, along with two other hyperparameters: `Lmin` and `Lmax`; positive integers (with `Lmax > Lmin`) giving the upper and lower limits to the number of leapfrog steps per iteration (from which the number is selected uniformly at random).

The default control options for HMC are: `control = list(Lmin = 10, Lmax = 20, epsilon = 0.005)`

Currently, the only implemented optimisation algorithm is Adagrad (`method = "adagrad"`). The `control` argument can be used to specify the optimiser hyperparameters: `learning_rate` (default 0.8), `initial_accumulator_value` (default 0.1) and `use_locking` (default `TRUE`). The are passed directly to TensorFlow's optimisers, see the TensorFlow docs for more information

## Value

`mcmc` & `stashed_samples` - an `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:

- parthe best set of parameters found

- valuethe log joint density of the model at the parameters par
- iterationsthe number of iterations taken by the optimiser
- convergencean integer code, 0 indicates successful completion, 1 indicates the iteration limit max_iterations had been reached

## Examples

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

# carry out mcmc on the model
draws <- mcmc(m,
              n_samples = 100,
              warmup = 10)

## End(Not run)
## Not run:
# find the MAP estimate
opt_res <- opt(m)

## End(Not run)
```

---

`internals` *internal greta methods*

---

## Description

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

## Details

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

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

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

### Usage

```
.internals$greta_arrays$unknowns        # greta array print methods
.internals$inference$progress_bar       # progress bar tools
                    samplers            # MCMC samplers
                    stash               # stashing MCMC samples
.internals$nodes$constructors           # node creation wrappers
                distribution_classes    # R6 distribution classes
                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
```

---

`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("single", "double"), n_cores = NULL,
  compile = TRUE)

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

## S3 method for class 'greta_model'
plot(x, y, ...)
```

### Arguments

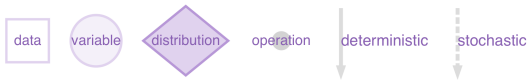
| | |
|---|---|
| `...` | for `model`: `greta_array` objects to be tracked by the model (i.e. those for which samples will be retained during mcmc). If not provided, all of the non-data `greta_array` objects defined in the calling environment will be tracked. For `print` and `plot`:further arguments passed to or from other methods (currently ignored). |
| `precision` | the floating point precision to use when evaluating this model. Switching from `'single'` (the default) to `'double'` should reduce the risk of numerical instability during sampling, but will also increase the computation time, particularly for large models. |

| | |
|---|---|
| n_cores | the number of cpu cores to use when evaluating this model. Defaults to and cannot exceed the number detected by parallel::detectCores. |
| 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 |

## 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::gdr_graph object (invisibly).

## Examples

```
## Not run:

# define a simple model
mu = variable()
sigma = lognormal(1, 0.1)
x = rnorm(10)
distribution(x) = normal(mu, sigma)

m <- model(mu, sigma)

plot(m)

## End(Not run)
```

---

# operators *arithmetic, logical and relational operators for greta arrays*

---

## Description

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

## Details

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

## Usage

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

# logical operators
!x
x & y
x | y

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

## Examples

```
## Not run:

x = as_data(-1:12)

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

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

# relational
```

```
h = x < 1
i = (-x) >= x
j = h == x

## End(Not run)
```

---

`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

diag(x = 1, nrow, ncol)

colMeans(x, na.rm = FALSE, dims = 1L)

rowMeans(x, na.rm = FALSE, dims = 1L)

colSums(x, na.rm = FALSE, dims = 1L)

rowSums(x, na.rm = FALSE, dims = 1L)

sweep(x, MARGIN, STATS, FUN = "-", check.margin = TRUE, ...)

backsolve(r, x, k = ncol(r), upper.tri = TRUE, transpose = FALSE)

forwardsolve(l, x, k = ncol(l), upper.tri = FALSE, transpose = FALSE)
```

## Arguments

`x, y, nrow, ncol, MARGIN, STATS, FUN, check.margin, ..., r, k, upper.tri, transpose, l, na.rm, dims`
: arguments as in original documentation

`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. it is equivalent to `as_data(array(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)
```

## 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') are therefore more likely to useful in modelling, and these are what are provided here.

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 $x > 1$, $y \approx x$

## Examples

```
## Not run:

 x = normal(1, 3, dim = 10)

 # transformation to the unit interval
 p1 <- iprobit(x)
 p2 <- ilogit(x)
```

```
p3 <- icloglog(x)
p4 <- icauchit(x)

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

## 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 = 1)
```

## Arguments

| | |
|---|---|
| `lower, upper` | scalar values giving 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. |

## 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.

## 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)
```

# Index