Package: ggdmc (via r-universe)

September 12, 2024

Type Package

Title Cognitive Models

Version 0.2.8.1

Date 2022-06-13

Author Yi-Shin Lin [aut, cre], Andrew Heathcote [aut]

Maintainer Yi-Shin Lin <yishinlin001@gmail.com>

Description The package provides tools to fit the LBA, DDM, PM and 2-D diffusion models, using the population-based Markov Chain Monte Carlo.

License GPL-2

URL https://github.com/yxlin/ggdmc

BugReports https://github.com/yxlin/ggdmc/issues

LazyData TRUE

Imports Rcpp (>= 0.12.10), stats, utils, ggplot2, matrixStats, data.table (>= 1.10.4), loo (>= 2.1.0)

Depends R (>= 3.3.0)

LinkingTo Rcpp (>= 0.12.10), RcppArmadillo (>= 0.7.100.3.0)

Suggests testthat

RoxygenNote 6.1.1

Encoding UTF-8

NeedsCompilation yes

Repository https://yxlin.r-universe.dev

RemoteUrl https://github.com/yxlin/ggdmc

RemoteRef HEAD

RemoteSha de30bc2b1e4aabbf8487cd429555058cff8fcf0e

Contents

ac *Calculate the autocorrelation of a vector*

Description

Calculate the autocorrelation of a vector.

Usage

```
ac(x, nLags = 50)
```

Arguments

x	a vector storing parameter values
nLags	the maximum number of lags

Value

A data.frame

Examples

```
res <- ac(1:100)
## List of 2
## $ Lag            : int [1:50] 1 2 3 4 5 6 7 8 9 10 ...
## $ Autocorrelation: num [1:50] 1 1 1 1 1 1 1 1 1 1 ...

res <- ac(rnorm(100))
str(res)
## List of 2
## $ Lag            : int [1:50] 1 2 3 4 5 6 7 8 9 10 ...
## $ Autocorrelation: num [1:50] 1 -0.0485 0.0265 -0.1496 0.0437 ...
```

autocorr *Autocorrelation Plot*

Description

Plot the autocorrelation of posterior samples,

Usage

```
autocorr(x, start = 1, end = NA, nLags = 50, pll = TRUE,
  subchain = FALSE)
```

Arguments

x	posterior samples
start	start from which iteration.
end	end at which iteration
nLags	the maximum number of lags.
pll	a Boolean switch for plotting parameter values or posterior log likelihoods
subchain	a Boolean switch to plot a subset of chains.

Examples

```
## Model 1
## 27 elements with 20 levels
FR <- list(S = c("n","w","p"), cond=c("C","F", "H"), R=c("N", "W", "P"))
lev <- c("CnN","CwN", "CnW","CwW",
         "FnN","FwN","FpN", "FnW","FwW","FpW", "fa","FpP",
         "HnN","HwN","HpN", "HnW","HwW","HpW", "HpP",
         "FAKERATE")
map_mean_v <- ggdmc:::MakeEmptyMap(FR, lev)
map_mean_v[1:27] <- c(
  "CnN","CwN","FAKERATE", "FnN","FwN","FpN", "HnN","HwN","HpN",
  "CnW","CwW","FAKERATE", "FnW","FwW","FpW", "HnW","HwW","HpW",
  "FAKERATE","FAKERATE","FAKERATE", "fa","fa","FpP", "fa","fa","HpP")

model0 <- BuildModel(
  p.map     = list(A = "1", B = c("cond", "R"), t0 = "1", mean_v = c("MAPMV"),
                   sd_v = "1", st0 = "1", N = "cond"),
  match.map = list(M = list(n = "N", w = "W", p = "P"), MAPMV = map_mean_v),
  factors   = list(S = c("n","w","p"), cond = c("C","F", "H")),
  constants = c(N.C = 2, N.F = 3, N.H = 3, st0 = 0, B.C.P = Inf,
                mean_v.FAKERATE = 1, sd_v = 1),
  responses = c("N", "W", "P"),
  type      = "norm")

npar <- model0@npar

p.vector <- c(A = .3, B.C.N = 1.3,  B.F.N = 1.3,  B.H.N = 1.3,
              B.C.W = 1.3,  B.F.W = 1.4,  B.H.W = 1.5,
              B.F.P = 1.1,  B.H.P = 1.3,

              t0=.1,

              mean_v.CnN = 2.8,  mean_v.CwN = -0.3, mean_v.CnW=-1,
              mean_v.CwW = 2.9,  mean_v.FnN = 2.8,  mean_v.FwN=-.3,

              mean_v.FpN = -1.6, mean_v.FnW = -1,   mean_v.FwW = 2.9,
              mean_v.FpW = .5 ,  mean_v.fa = -2.4,  mean_v.FpP = 2.5,

              mean_v.HnN = 2.8, mean_v.HwN = -.5,   mean_v.HpN = -.6,
              mean_v.HnW = -.7, mean_v.HwW = 3.0,   mean_v.HpW = 1.6,
              mean_v.HpP = 2.3)
```

```
acc_tab0 <- TableParameters(p.vector, 1, model0, FALSE)
acc_tab1 <- TableParameters(p.vector, "w.C.N", model0, FALSE)
acc_tab2 <- TableParameters(p.vector, "w.F.P", model0, FALSE)
print(acc_tab0); print(acc_tab1); print(acc_tab2)

## Not run:
dat0 <- simulate(model0, nsim=50, ps=p.vector)
dmi0 <- BuildDMI(dat0, model0)

## End(Not run)
p1 <- rep(1, npar)
names(p1) <- model0@pnames

p.prior0 <- BuildPrior(
  dists = c(rep("tnorm", 9), "beta", rep("tnorm", 19)),
  p1    = p1,
  p2    = c(rep(2, 9), 1, rep(2, 19)),
  lower = c(rep(0, 10),  rep(NA, 19)),
  upper = c(rep(NA, 9), 1, rep(NA, 19)))

# plot(p.prior0, ps = p.vector)
## Sampling
## 18.4 & 36.17 s
## Not run:
fit0 <- StartNewsamples(dmi0, p.prior0, block = FALSE, thin=4)
fit0_correct  <- run(fit0, thin=4, block = FALSE)

hat <- gelman(fit0_correct, verbose=TRUE);
p0 <- autocorr(fit0_correct, subchain=1:3, pll=TRUE)

## End(Not run)
```

`BuildDMI` *Bind data and models*

Description

Binding a data set with an object of data-model instance. The function checks whether the data and the model are compatible and adds attributes to a data model instance.

Usage

```
BuildDMI(x, model)
```

Arguments

`x`	data formatted as a data frame
`model`	a model object

Value

a data model instance

`BuildModel` *Create a model object*

Description

A model object consists of arrays with model attributes.

Usage

```
BuildModel(p.map, responses, factors = list(A = "1"), match.map = NULL,
  constants = numeric(0), type = "norm", posdrift = TRUE,
  verbose = TRUE)
```

Arguments

`p.map`	parameter map. This option maps a particular factorial design to model parameters
`responses`	specifying the response names and levels
`factors`	specifying a list of factors and their treatment levels
`match.map`	match map. This option matches stimuli and responses
`constants`	specifying the parameters that you want to be fixed.
`type`	the model type defined in the package, "rd", "norm", or "cddm".
`posdrift`	a Boolean, switching between enforcing strict postive drift rates by using truncated normal distribution. This option is only useful in "norm" model type.
`verbose`	Print p.vector, constants and model type

Examples

```
## A diffusion decision model
model <- BuildModel(
p.map     = list(a ="1", v = "1",z = "1", d = "1", sz = "1", sv = "1",
                 t0 = "1", st0 = "1"),
match.map = list(M = list(s1 = "r1", s2 ="r2")),
factors   = list(S = c("s1", "s2")),
constants = c(st0 = 0, d = 0),
responses = c("r1", "r2"),
type      = "rd")

## A LBA model
model <- BuildModel(
  p.map     = list(A = "1", B = "1", t0 = "1", mean_v = "M", sd_v = "1",
                   st0 = "1"),
  match.map = list(M = list(s1 = 1, s2 = 2)),
```

```
  factors   = list(S = c("s1", "s2")),
  constants = c(st0 = 0, sd_v = 1),
  responses = c("r1", "r2"),
  type      = "norm")

## A circular diffusion decision model
model <- BuildModel(
  p.map     = list(v1 = "1", v2 = "1", a = "1", t0 = "1", sigma1="1",
                   sigma2="1", eta1="1", eta2="1", tmax="1", h="1"),
  match.map = list(M=list()),
  constants = c(sigma1 = 1, sigma2 = 1, eta1=0, eta2=0, tmax=6, h=1e-4),
  factors   = list(S = c("s1", "s2")),
  responses = paste0('theta_', letters[1:4]),
  type      = "cddm")
```

BuildPrior *Specifying Prior Distributions*

Description

BuildPrior sets up prior distributions for each model parameter. p1 and p2 refer to the first and second parameters a prior distribution. p1 must comes with parameter names.

Usage

```
BuildPrior(p1, p2, lower = rep(NA, length(p1)), upper = rep(NA,
  length(p1)), dists = rep("tnorm", length(p1)),
  untrans = rep("identity", length(p1)), types = c("tnorm", "beta",
  "gamma", "lnorm", "unif", "constant", "tnorm2", "cauchy", NA),
  lg = TRUE)
```

Arguments

p1	the first parameter of a distribution
p2	the second parameter of a distribution
lower	lower support (boundary)
upper	upper support (boundary)
dists	a vector of character string specifying a distribution.
untrans	whether to do log transformation. Default is not
types	available distribution types
lg	logical; if TRUE, probabilities p are given as log(p)

Details

Four distribution types are implemented:

1. Normal and truncated normal distribution, where: p1 = mean, p2 = sd. When the lower and upper are not provided, they are set to -Inf and Inf, rendering a normal distribution. Type name is "tnorm".
2. Beta distribution, where: p1 = shape1 and p2 = shape2 (see pbeta). Note the uniform distribution is a special case of the beta with p1 = 1 and p2 = 1. Type name is "beta".
3. Gamma distribution, where p1 = shape and p2 = scale (see pgamma). Note p2 is scale, not rate. Type name is "gamma".
4. Log-normal, where p1 = meanlog and p2 = sdlog (see plnorm).
5. Uniform distribution. The bounds are not c(0, 1). The option comes handy. Type name is "unif".

Value

a list of list

Examples

```
## Show using dbeta to visualise a uniform distribution with bound (0, 1)
x <- seq(-.1, 1.1, .001)
plot(x, dbeta(x, 1, 1), type="l", ylab="Density", xlab="x", lwd=2)

## BuildPrior
pop.mean  <- c(a=2,   v=4, z=0.5, t0=0.3)
pop.scale <- c(a=0.5, v=.5, z=0.1, t0=0.05)

pop.prior <- BuildPrior(
  dists = rep("tnorm", 4),
  p1    = pop.mean,
  p2    = pop.scale,
  lower = c(0,-5,  0, 0),
  upper = c(5, 7,  1, 1))

p.prior <- BuildPrior(
  dists = rep("tnorm", 4),
  p1    = pop.mean,
  p2    = pop.scale*5,
  lower = c(0,-5, 0, 0),
  upper = c(5, 7, 1, 1))

mu.prior <- BuildPrior(
  dists = rep("tnorm", 4),
  p1    = pop.mean,
  p2    = pop.scale*5,
  lower = c(0,-5,  0, 0),
  upper = c(5, 7,  1, 1))

sigma.prior <- BuildPrior(
```

```
  dists = rep("beta", 4),
  p1    = c(a=1, v=1, z=1, t0=1),
  p2    = rep(1, 4),
  upper = rep(1, 4))

## Bind three priors together for hierarchical modelling
priors <- list(pprior=p.prior, location=mu.prior, scale=sigma.prior)
```

CheckConverged *Convergence Diagnosis*

Description

These functions test whether Markov chains are converged .

Usage

```
CheckConverged(x)

CheckConverged(x)

PickStuck(x, ...)

## S4 method for signature 'posterior'
PickStuck(x, cut = 10, start = 1, end = NA,
  verbose = FALSE, digits = 2)

## S4 method for signature 'list'
PickStuck(x, cut = 10, start = 1, end = NA,
  verbose = FALSE, digits = 2)

## S4 method for signature 'hyper'
PickStuck(x, hyper = TRUE, cut = 10, start = 1,
  end = NA, verbose = FALSE, digits = 2)

isstuck(x, ...)

## S4 method for signature 'posterior'
isstuck(x, cut = 10, start = 1, end = NA,
  verbose = FALSE)

## S4 method for signature 'list'
isstuck(x, cut = 10, start = 1, end = NA,
  verbose = FALSE, digits = 2)

## S4 method for signature 'hyper'
isstuck(x, hyper = TRUE, cut = 10, start = 1,
```

```
  end = NA, verbose = FALSE, digits = 2)

isflat(x, ...)

## S4 method for signature 'posterior'
isflat(x, p1 = 1/3, p2 = 1/3, cut = 0.25,
  cut_scale = Inf, verbose = FALSE, digits = 2)

## S4 method for signature 'list'
isflat(x, p1 = 1/3, p2 = 1/3, cut = 0.25,
  cut_scale = Inf, verbose = FALSE, digits = 2)

ismixed(x, ...)

## S4 method for signature 'posterior'
ismixed(x, cut = 1.1, verbose = FALSE)
```

Arguments

x	posterior samples
...	other additional arguments
cut	a criterion for deciding whether chains get stuck (isstuck); whether chains are not flat (using median or IQR isflat); whether chains are well mixed ismixed.
start	start to evaluate from which iteration.
end	end at which iteration for evaeuation.
verbose	a boolean switch to print more information
digits	print how many digits. Default is 2
hyper	whether x are hierarhcial samples
p1	the range of the head of MCMC chains
p2	the range of the tail of the MCMC chains
cut_scale	Use IQR to decide whether chains are not flat

Details

isstuck tests whether a chain hovers around a region significantly deviates from other its peers.

PickStuck calculate each chain separately for the mean (across MC samples) of posterior log likelihood. If the difference of the means and the median (across chains) of the mean of posterior log likelihood is greater than the value set in cut, chains are considered stuck. The default value for cut is 10. The user should consider their situatin to set the cut value.

unstick removes stuck chains from posterior samples (not well tested).

ismixed tests whether the potential scale reduction factor for a model fit is lower than a criterion, defined by cut.

iseffective testes whether posterior samples are enough adjusted autocorrelation.

CheckConverged is a wrapper function running the four checking functions, isstuck, isflat, ismixed and iseffective.

Value

PickStuck gives an index vector; unstick gives a posterios samples.

Examples

```
model <- BuildModel(
p.map     = list(a = "1", v="1", z="1", d="1", sz="1", sv="1", t0="1",
                 st0="1"),
match.map = list(M = list(s1 = "r1", s2 = "r2")),
factors   = list(S = c("s1", "s2")),
responses = c("r1","r2"),
constants = c(st0 = 0, d = 0, sv = 0, sz = 0),
type      = "rd")

npar <- model@npar
pop.mean  <- c(a=2,   v=4, z=0.5, t0=0.3)
pop.scale <- c(a=0.5, v=.5, z=0.1, t0=0.05)
pop.prior <- BuildPrior(
  dists = rep("tnorm", npar),
  p1    = pop.mean,
  p2    = pop.scale,
  lower = c(0,-5,  0, 0),
  upper = c(5, 7,  1, 1))

dat <- simulate(model, nsub = 8, nsim = 30, prior = pop.prior)
dmi <- BuildDMI(dat, model)
ps <- attr(dat, "parameters")

p.prior <- BuildPrior(
  dists = rep("tnorm", npar),
  p1    = pop.mean,
  p2    = pop.scale*5,
  lower = c(0,-5, 0, 0),
  upper = c(5, 7, 1, 1))

mu.prior <- BuildPrior(
  dists = rep("tnorm", npar),
  p1    = pop.mean,
  p2    = pop.scale*5,
  lower = c(0,-5,  0, 0),
  upper = c(5, 7,  1, 1))

sigma.prior <- BuildPrior(
  dists = rep("beta", npar),
  p1    = c(a=1, v=1, z=1, t0=1),
  p2    = rep(1, npar),
  upper = rep(1, npar))

## Note the names are important
priors <- list(pprior=p.prior, location=mu.prior, scale=sigma.prior)

## Not run:
```

```
Fit hierarchical model ----##
fit0 <- StartNewsamples(dmi, priors)
fit  <- run(fit0)

PickStuck(fit, hyper=TRUE)
PickStuck(fit@individuals[[1]])
PickStuck(fit)

tmp <- PickStuck(fit, hyper=TRUE, verbose=T)
tmp <- PickStuck(fit@individuals[[1]], verbose=T)
tmp <- PickStuck(fit, verbose=T)
isstuck(fit0@individuals[[1]])
isstuck(fit@individuals[[1]])
isstuck(fit, hyper = TRUE)

tmp <- isflat(fit@individuals[[1]])
tmp <- isflat(fit@individuals[[1]], verbose = TRUE)

tmp <- isflat(fit@individuals[[1]], cut_scale = .25)
tmp <- isflat(fit@individuals[[1]], cut_scale = .25, verbose = TRUE)

## Test unstick
fit0 <- StartNewsamples(dmi, priors, nmc=50)
fit  <- run(fit0, nmc=200)
bad <- PickStuck(fit@individuals[[1]], verbose=T)
chain_removed <- unstick_one(fit@individuals[[1]], bad)
plot(tmp)

## End(Not run)
```

---

check_pvec *Does a model object specify a correct p.vector*

Description

Check a parameter vector

Usage

```
check_pvec(ps, model)
```

Arguments

ps	parameter vector
model	a model object

dbeta_lu *A modified dbeta function*

Description

A modified dbeta function

Usage

```
dbeta_lu(x, p1, p2, lower, upper, lg = FALSE)
```

Arguments

x	quantile
p1	shape1 parameter
p2	shape2 parameter
lower	lower bound
upper	upper bound
lg	logical; if TRUE, return log density.

dcauchy_l *A modified dcauchy functions*

Description

A modified dcauchy functions

Usage

```
dcauchy_l(x, p1, p2, lower, upper, lg = FALSE)
```

Arguments

x	quantile
p1	location parameter
p2	scale parameter
lower	lower bound
upper	upper bound
lg	log density?

dcircle *Two-dimension Diffusion Model*

Description

Density, random generation for the 2-D diffusion model.

This function generates one 1-D diffusion process.

Usage

```
dcircle(RT, A, P, tmax, kmax, sz, nw)

dcircle300(P, tmax, kmax, sz, nw)

rcircle(n, P, tmax, h, nw)

rcircle_process(P, tmax, h)

r1d(P, tmax, h)
```

Arguments

RT	a vector storing response times
A	a vector storing response angles.
P	is a parameter vector, c(v1, v2, a, t0, sigma1, sigma2, eta1, eta2). The sequence is important. v1 is the x-axis mean drift rate. v2 is the y-axis mean drift rate. sigma1 is the x-axis within-trial drift rate SD. sigma2 is the y-axix within-trial drift rate SD. a is decision threshold. sigma1 and sigma2 must be 1 and identical, because this is what has been thoroughly tested so far. Other values may return unknown results. t0 non-decision time.
tmax	maximum time of the model
kmax	the tuning parameter for Bessel function. Mostly 50.
nw	the number of theta steps (w = 2 * pi / nw)
n	number of observations
h, sz	sz is the number of time steps (h = tmax / sz). h is time step. Mostly .1 ms.
P	is a parameter vector, c(v, a, z, t0, s). The sequence must be followed. v is the drift rate a is decision threshold. t0 is the non-decision time.
tmax	maximum time allowed.
kmax	the tuning parameter for Bessel function. Mostly 50.
h, sz	sz is the number of time steps (h = tmax / sz). h is the size of one time step. We usually set h = 1e-4. That is .1 ms. So when tmax is 2 second and each time step is 0.1 ms, sz will be 2e4 steps.

Details

The model has the main parameters, v1, v2, eta1, eta2, a, sigma, and t0. tmax, kmax, sz and nw are tuning parameters for determining the set. dcircle300 produces PDF table and others.

The model has five parameters, v, a, z, t0, and s. tmax and h are tuning parameters for determining the set.

Value

rcircle returns a n x 2 matrix. Each row is an [RT R] trial. dcircle returns a n vector.

rcircle returns a n x 2 matrix. Each row is an [RT R] trial. dcircle returns a n vector.

Examples

```
## TODO examples
```

---

dconstant *A pseudo constant function to get constant densities*

Description

Used with constant prior

Usage

```
dconstant(x, p1, p2, lower, upper, lg = FALSE)
```

Arguments

x	quantile
p1	constant value
p2	unused argument
lower	dummy varlable
upper	dummy varlable
lg	log density?

deviance_model *Calculate the statistics of model complexity*

Description

Calculate deviance for a model object for which a log-likelihood value can be obtained, according to the formula -2*log-likelihood.

Usage

```
deviance_model(object, start, end, ...)
```

Arguments

object	posterior samples
start	start iteration
end	end iteration
...	other plotting arguments passing through dot dot dot.

References

Spiegelhalter, D. J., Best, N. G., Carlin, B. P., & van der Linde, A. (2002). Bayesian Measures of Model Complexity and Fit. Journal of the Royal Statistical Society, Series B (Statistical Methodology), 64(4), 583–639. doi:10.1111/1467-9868.00353

Ando, T. (2007). Bayesian predictive information criterion for the evaluation of hierarchical Bayesian and empirical Bayes models. Biometrika. 94(2), 443–458. doi:10.1093/biomet/asm017.

dgamma_l *A modified dgamma function*

Description

A modified dgamma function

Usage

```
dgamma_l(x, p1, p2, lower, upper, lg = FALSE)
```

Arguments

x	quantile
p1	shape parameter
p2	scale parameter
lower	lower bound
upper	upper bound
lg	log density?

DIC *Deviance Information Criteria*

Description

Calculate DIC and BPIC.

Usage

```
DIC(object, ...)

## S4 method for signature 'posterior'
DIC(object, start = 1, end = NA, BPIC = FALSE)

## S4 method for signature 'list'
DIC(object, start = 1, end = NA, BPIC = FALSE)

## S4 method for signature 'hyper'
DIC(object, start = 1, end = NA, BPIC = FALSE)
```

Arguments

object	posterior samples from one participant
...	other plotting arguments passing through dot dot dot.
start	start from which iteration.
end	end at which iteration. For example, set start = 101 and end = 1000, instructs the function to calculate from 101 to 1000 iteration.
BPIC	a Boolean switch to calculate BPIC, instead of DIC

Details

This function implements three different definitions of the "effective number of parameters of the model". First is from Spiegelhalter et al (2002, p. 587), "... that pD can be considered as a 'mean deviance minus the deviance of the means'". Second is from Gelman et al (2014, p. 173, equation 7.10), and third subtracts the minimal value of the deviance from the mean of the deviance.

Examples

```
## Calculate DIC from data of one participant
## Not run:
model <- BuildModel(
  p.map     = list(A = "1", B = "1", t0 = "1", mean_v = "M", sd_v = "1",
                   st0 = "1"),
  match.map = list(M = list(s1 = 1, s2 = 2)),
  factors   = list(S = c("s1", "s2")),
  constants = c(st0 = 0, sd_v = 1),
  responses = c("r1", "r2"),
  type      = "norm")

p.vector <- c(A = .75, B = 1.25, t0 = .15, mean_v.true = 2.5,
              mean_v.false = 1.5)
ntrial <- 50
dat <- simulate(model, nsim = ntrial, ps = p.vector)
dmi <- BuildDMI(dat, model)

p.prior <- BuildPrior(
  dists = c("tnorm", "tnorm", "beta", "tnorm", "tnorm"),
  p1    = c(A = 1, B = 1, t0 = 1, mean_v.true = 1, mean_v.false = 1),
  p2    = c(1,  1,  1, 1, 1),
  lower = c(rep(0, 3),  rep(NA, 2)),
  upper = c(rep(NA, 2), 1, rep(NA, 2)))

## Sampling
fit0 <- StartNewsamples(dmi, p.prior)
fit  <- run(fit0, thin = 8)

DIC(fit)
DIC(fit)
DIC(fit, start=100, end=200)
DIC(fit, BPIC=TRUE)
DIC(fit, BPIC=TRUE, start=201, end=400)

## End(Not run)

## Calculate DICs from data of 8 participant
## Not run:
model <- BuildModel(
p.map     = list(a = "1", v = "F", z = "1", d = "1", sz = "1", sv = "1",
                 t0 = "1", st0 = "1"),
match.map = list(M = list(s1 = "r1", s2 = "r2")),
factors   = list(S = c("s1", "s2"), F = c("f1", "f2")),
constants = c(st0 = 0, d = 0),
responses = c("r1", "r2"),
type      = "rd")
npar <- length(Get_pnames(model))

## Population distribution
pop.mean  <- c(a=2,   v.f1=4,  v.f2=3,  z=0.5, sz=0.3, sv=1,  t0=0.3)
pop.scale <- c(a=0.5, v.f1=.5, v.f2=.5, z=0.1, sz=0.1, sv=.3, t0=0.05)
```

```
pop.prior <- BuildPrior(
  dists = rep("tnorm", npar),
  p1    = pop.mean,
  p2    = pop.scale,
  lower = c(0,-5, -5, 0, 0, 0, 0),
  upper = c(5, 7,  7, 1, 2, 1, 1))

## Simulate some data
dat <- simulate(model, nsub = 8, nsim = 10, prior = pop.prior)
dmi <- BuildDMI(dat, model)
ps <- attr(dat, "parameters")

p.prior <- BuildPrior(
  dists = rep("tnorm", npar),
  p1    = pop.mean,
  p2    = pop.scale*5,
  lower = c(0,-5, -5, 0, 0, 0, 0),
  upper = c(5, 7,  7, 1, 2, 2, 1))

## Sampling
fit0 <- StartNewsamples(dmi, p.prior, ncore=1)
fit  <- run(fit0, ncore=4)  ## No printing when running in RStudio

## Calculate DIC for participant 1
DIC(fit[[1]])

## Calculate all participants
res <- DIC(fit)

## BPIC
res <- DIC(fit, BPIC = TRUE)

## End(Not run)
```

`dlnorm_l` *A modified dlnorm functions*

Description

A modified dlnorm functions

Usage

```
dlnorm_l(x, p1, p2, lower, upper, lg = FALSE)
```

Arguments

x	quantile
p1	meanlog parameter

p2	sdlog parameter
lower	lower bound
upper	upper bound
lg	log density?

`dmi-class` *An S4 class of the Data-model Instance*

Description

The class is to represent a data-model instance, which joins a model object with a data frame. The process of BuildDMI also generates cell.index and cell.empty.

Slots

`data` A data frame storing the would-be fit data set

`model` A 3-D model array. Dimension one stores the combinations of the factor levels and response types, dimension two stores parameters, and dimension three stores response types.

`cell.index` A ncell-element list. Each element represents one cell. Each element stores `nobs` Boolean indicators, showing whether a particular observation belongs to this cell.

`cell.empty` A ncell-element logical vector, indicating whether this cell has no observation.

`dtnorm` *Truncated Normal Distribution*

Description

Random number generation, probability density and cumulative density functions for truncated normal distribution.

Usage

```
dtnorm(x, p1, p2, lower, upper, lg = FALSE)

rtnorm(n, p1, p2, lower, upper)

ptnorm(q, p1, p2, lower, upper, lt = TRUE, lg = FALSE)
```

Arguments

x, q	vector of quantiles;
p1	mean (must be scalar).
p2	standard deviation (must be scalar).
lower	lower truncation value (must be scalar).
upper	upper truncation value (must be scalar).
lg	log probability. If TRUE (default is FALSE) probabilities p are given as log(p).
n	number of observations. n must be a scalar.
lt	lower tail. If TRUE (default) probabilities are P[X <= x], otherwise, P[X > x].

Value

a numeric vector.

Examples

```
## rtnorm example
dat1 <- rtnorm(1e5, 0, 1, 0, Inf)
hist(dat1, breaks = "fd", freq = FALSE, xlab = "",
     main = "Truncated normal distributions")

## dtnorm example
x <- seq(-5, 5, length.out = 1e3)
dat1 <- dtnorm(x, 0, 1, -2, 2, 0)
plot(x, dat1, type = "l", lwd = 2, xlab = "", ylab= "Density",
     main = "Truncated normal distributions")

## ptnorm example
x <- seq(-10, 10, length.out = 1e2)
mean <- 0
sd <- 1
lower <- 0
upper <- 5
dat1 <- ptnorm(x, 0, 1, 0, 5, lg = TRUE)
```

---

effectiveSize *Effective Sample Size*

Description

Posterior sample size adjusted for autocorrelation. The function is based on the effectiveSize function in coda package.

Usage

```
effectiveSize(x, ...)

## S4 method for signature 'hyper'
effectiveSize(x, hyper = TRUE, start = 1, end = NA,
  subchain = NA, digits = 2, verbose = FALSE)

## S4 method for signature 'list'
effectiveSize(x, start = 1, end = NA, subchain = NA,
  digits = 2, verbose = FALSE)

## S4 method for signature 'posterior'
effectiveSize(x, start = 1, end = NA,
  subchain = NA, digits = 2, verbose = FALSE)
```

Arguments

x	posterior samples
...	other additional arguments
hyper	a Boolean switch to calculate phi
start	start from iteration
end	end at which iteraton
subchain	calculate a subset of chains. This must be an integer vector
digits	printing how many digits
verbose	printing more information

Details

hyper argument does not work for list class (i.e., posterior samples from a fixed-effect model fit).

References

Plummer, M. Best, N., Cowles, K., Vines, K., Sarkar, D., Bates, D., Almond, R., & Magnusson, A. (2019). R package 'coda' https://cran.r-project.org/web/packages/coda/

Examples

```
##################################40
## effectiveSize example
##################################40
## Not run:
cat("Class:", class(fit), "\n")
es1 <- effectiveSize(fit, hyper=TRUE, verbose=FALSE)
es1 <- effectiveSize(fit, hyper=TRUE, verbose=TRUE)

es1 <- effectiveSize(fit, hyper=TRUE, verbose=FALSE, subchain=7:9)
es1 <- effectiveSize(fit, hyper=TRUE, verbose=TRUE, subchain=7:9)
```

```
es1 <- effectiveSize(fit, hyper=FALSE, verbose=FALSE)
es1 <- effectiveSize(fit, hyper=FALSE, verbose=TRUE)

es1 <- effectiveSize(fit, hyper=FALSE, verbose=FALSE, subchain=4:6)
es1 <- effectiveSize(fit, hyper=FALSE, verbose=TRUE, subchain=4:6)

cat("Starting a new fixed-effect model fit: \n")
fit0 <- StartNewsamples(dmi, p.prior, ncore=4)
fit  <- run(fit0, ncore=4)

cat("Class:", class(fit), "\n")
es1 <- effectiveSize(fit, verbose=FALSE)
es1 <- effectiveSize(fit, verbose=TRUE)
es1 <- effectiveSize(fit, verbose=FALSE, subchain=4:6)
es1 <- effectiveSize(fit, verbose=TRUE, subchain=4:6)


## End(Not run)
```

---

gelman *Potential scale reduction factor*

Description

gelman function calls the function, gelman.diag in the **coda** package to calculates PSRF.

Usage

```
gelman(x, ...)

## S4 method for signature 'posterior'
gelman(x, start = 1, end = NA, conf = 0.95,
  multivariate = TRUE, subchain = NA, digits = 2, verbose = FALSE)

## S4 method for signature 'list'
gelman(x, start = 1, end = NA, conf = 0.95,
  multivariate = TRUE, subchain = NA, digits = 2, verbose = FALSE)

## S4 method for signature 'hyper'
gelman(x, hyper = TRUE, start = 1, end = NA,
  conf = 0.95, multivariate = TRUE, subchain = NA, digits = 2,
  verbose = FALSE)
```

Arguments

x	posterior samples
...	other additional arguments

start	start iteration
end	end iteration
conf	confident inteval
multivariate	multivariate Boolean switch
subchain	whether only calculate a subset of chains
digits	print out how many digits
verbose	print more information
hyper	a Boolean switch, indicating posterior samples are from hierarchical modeling

Examples

```
## Not run:
rhat1 <- gelman(hsam);
rhat2 <- gelman(hsam, end = 51);
rhat3 <- gelman(hsam, conf = .90);
rhat7 <- gelman(hsam, subchain = TRUE);
rhat8 <- gelman(hsam, subchain = 1:4);
rhat9 <- gelman(hsam, subchain = 5:7, digits = 1, verbose = TRUE);

## End(Not run)
```

GetNsim *Get a n-cell matrix*

Description

Constructs a matrix, showing how many responses to in each cell. The function checks whether the format of n and ns conform.

Usage

```
GetNsim(ncell, n, ns)
```

Arguments

ncell	number of cells.
n	number of trials.
ns	number of subjects.

Details

n can be:

1. an integer for a balanced design,
2. a matrix for an unbalanced design, where rows are subjects and columns are cells. If the matrix is a row vector, all subjects have the same n in each cell. If it is a column vector, all cells have the same n. Otherwise each entry specifies the n for a particular subject x cell combination. See below for concrete examples.

Examples

```
model <- BuildModel(
  p.map     = list(A = "1", B = "R", t0 = "1", mean_v = "M", sd_v = "M",
                   st0 = "1"),
  match.map = list(M = list(s1 = 1, s2 = 2)),
  constants = c(sd_v.false = 1, st0 = 0),
  factors   = list(S = c("s1","s2")),
  responses = c("r1", "r2"),
  type      = "norm")

#######################30
## Example 1
#######################30
cells <- as.numeric(sapply(model@factors, length))
ncell <- prod(cells)
GetNsim(ncell, ns = 2, n = 1)
#      [,1] [,2]
# [1,]    1    1
# [2,]    1    1

#######################30
## Example 2
#######################30
n <- matrix(c(1:2), ncol = 1)
#      [,1]
# [1,]    1  ## subject 1 has 1 response for each cell
# [2,]    2  ## subject 2 has 2 responses for each cell

GetNsim(ncell, ns = 2, n = n)
#      [,1] [,2]
# [1,]    1    1
# [2,]    2    2

#######################30
## Example 3
#######################30
n <- matrix(c(1:2), nrow = 1)
#      [,1] [,2]
# [1,]    1    2
GetNsim(ncell, ns = 2, n = n)
#     [,1] [,2]
# [1,]   1    2 ## subject 1 has 1 response for cell 1 and 2 responses for cell 2
# [2,]   1    2 ## subject 2 has 1 response for cell 1 and 2 responses for cell 2

#######################30
## Example 4
#######################30
n <- matrix(c(1:4), nrow=2)
#      [,1] [,2]
# [1,]    1    3
# [2,]    2    4
GetNsim(ncell, ns = 2, n = n)
```

```
#      [,1] [,2]
# [1,]    1    3 ## subject 1 has 1 response for cell 1 and 3 responses for cell 2
# [2,]    2    4 ## subject 2 has 2 responses for cell 1 and 4 responses for cell 2
```

`GetParameterMatrix` *Constructs a ns x npar matrix,*

Description

The matrix is used to simulate data. Each row represents one set of parameters for a participant.

Usage

```
GetParameterMatrix(object, nsub, prior, ps, seed = NULL)
```

Arguments

`object`	a model object
`nsub`	number of subjects.
`prior`	a prior object
`ps`	a vector or a matirx.
`seed`	an integer specifying a random seed.

Details

One must enter either a vector or a matrix as true parameters to the argument, `ps`, when presuming to simulate data based on a fixed-effect model. When the assumption is to simulate data based on a random-effect model, one must enter a prior object to the argument, `prior` to first randomly generate a true parameter matrix.

Value

a ns x npar matrix

Examples

```
model <- BuildModel(
p.map     = list(a ="1", v = "1",z = "1", d = "1", sz = "1", sv = "1",
            t0 = "1", st0 = "1"),
match.map = list(M = list(s1 = "r1", s2 ="r2")),
factors   = list(S = c("s1", "s2")),
constants = c(st0 = 0, d = 0),
responses = c("r1", "r2"),
type      = "rd")

p.prior <- BuildPrior(
  dists = c("tnorm", "tnorm", "beta", "beta", "tnorm", "beta"),
  p1    = c(a = 1, v = 0, z = 1, sz = 1, sv = 1, t0 = 1),
```

```
  p2    = c(a = 1, v = 2, z = 1, sz = 1, sv = 1, t0 = 1),
  lower = c(0, -5, NA, NA, 0, NA),
  upper = c(2,  5, NA, NA, 2, NA))

## Example 1: Randomly generate 2 sets of true parameters from
## parameter priors (p.prior)
GetParameterMatrix(model, nsub=2, p.prior)
##            a         v         z        sz       sv        t0
## [1,] 1.963067  1.472940 0.9509158 0.5145047 1.344705 0.0850591
## [2,] 1.512276 -1.995631 0.6981290 0.2626882 1.867853 0.1552828

## Example 2: Use a user-selected true parameters
true.vector  <- c(a=1, v=1, z=0.5, sz=0.2, sv=1, t0=.15)
GetParameterMatrix(model, nsub=2, ps = true.vector)
##   a v   z  sz sv   t0
## 1 1 1 0.5 0.2  1 0.15
## 2 1 1 0.5 0.2  1 0.15

## Example 3: When a user enter arbritary sequence of parameters.
## Note sv is before sz. It should be sz before sv
## See correct sequence, by entering "model@pnames"
## GetParameterMatrix will rearrange the sequence.
true.vector  <- c(t0=15, a=1, v=1, z=0.5, sv=1, sz = .2)
GetParameterMatrix(model, nsub=2, ps= true.vector)
##   a v   z  sz sv   t0
## 1 1 1 0.5 0.2  1 0.15
## 2 1 1 0.5 0.2  1 0.15
```

`GetPNames` *Extract parameter names from a model object*

Description

GetPNames will be deprecated. Please extract pnames directly via S4 slot 'model@pnames'

Usage

```
GetPNames(x)
```

Arguments

x	a model object

get_os *Retrieve information of operating system*

Description

A wrapper function to extract system information from `Sys.info` and `.Platform`

Usage

```
get_os()
```

Examples

```
get_os()
## sysname
## "linux"
```

ggdmc *Cognitive Multilevel Models*

Description

ggdmc provides tools for conducting Bayesian inference in cognitive models.

Author(s)

Yi-Shin Lin <yishinlin001@gmail.com>
Andrew Heathcote <andrew.heathcote@utas.edu.au>

References

Lin, Y.-S. & Strickland, L., (2019). Evidence accumulation models with R: A practical guide to hierarchical Bayesian methods. *The Quantitative Method in Psychology*.

Heathcote, A., Lin, Y.-S., Reynolds, A., Strickland, L., Gretton, M. & Matzke, D., (2018). Dynamic model of choice. *Behavior Research Methods*. https://doi.org/10.3758/s13428-018-1067-y.

Turner, B. M., & Sederberg P. B. (2012). Approximate Bayesian computation with differential evolution, *Journal of Mathematical Psychology*, 56, 375–385.

Ter Braak (2006). A Markov Chain Monte Carlo version of the genetic algorithm Differential Evolution: easy Bayesian computing for real parameter spaces. *Statistics and Computing*, 16, 239-249.

hyper-class *An S4 class to represent an object storing posterior samples at the participant and hyper level*

Description

An S4 class to represent an object storing posterior samples at the participant and hyper level

Slots

`phi_loc` posterior samples for the location parameters

`phi_sca` posterior samples for the scale parameters

`summed_log_prior` summed log prior likelihoods for phi.

`log_likelihoods` log likelihoods for phi

`prior_loc` a S4 prior object for the location parameters

`prior_sca` a S4 prior object for the scale parameters

`start` the index of starting sample

`npar` number of parameters

`pnames` parameter names

`nmc` number of Monte Carlo samples

`thin` thinning length

`nchain` number of Markov chains

`individuals` a list storing posterior samples for each individual participant

`snames` names of individual participants

See Also

`posterior-class`

iseffective *Model checking functions*

Description

The function tests whether we have drawn enough samples.

The function tests whether we have drawn enough samples.

Usage

```
iseffective(x, minN, nfun, verbose = FALSE)
```

```
iseffective(x, minN, nfun, verbose = FALSE)
```

Arguments

x	posterior samples
minN	specify the size of minimal effective samples
nfun	specify to use the mean or median function to calculate effective samples
verbose	print more information
x	posterior samples
minN	specify the size of minimal effective samples
nfun	specify to use the mean or median function to calculate effective samples
verbose	print more information

---

likelihood *Calculate log likelihoods*

Description

These function calculate log likelihoods. likelihood_rd implements the equations in Voss, Rothermund, and Voss (2004). These equations calculate diffusion decision model (Ratcliff & Mckoon, 2008). Specifically, this function implements Voss, Rothermund, and Voss's (2004) equations A1 to A4 (page 1217) in C++.

Usage

```
likelihood(pvector, data, min_lik = 1e-10, precision = 3)
```

Arguments

pvector	a parameter vector
data	data model instance
min_lik	minimal likelihood.
precision	a tuning parameter for the precision of DDM likelihood. The larger the value is, the more precise the likelihood is and the slower the computation would be.

Value

a vector

References

Voss, A., Rothermund, K., & Voss, J. (2004). Interpreting the parameters of the diffusion model: An empirical validation. *Memory & Cognition*, **32(7)**, 1206-1220.

Ratcliff, R. (1978). A theory of memory retrival. *Psychological Review*, **85**, 238-255.

Examples

```
model <- BuildModel(
p.map     = list(A = "1", B = "1", t0 = "1", mean_v = "M", sd_v = "1",
            st0 = "1"),
match.map = list(M = list(s1 = 1, s2 = 2)),
factors   = list(S = c("s1", "s2")),
constants = c(st0 = 0, sd_v = 1),
responses = c("r1", "r2"),
type      = "norm")

p.vector <- c(A = .25, B = .35,  t0 = .2, mean_v.true = 1, mean_v.false = .25)
dat <- simulate(model, 1e3,  ps = p.vector)
dmi <- BuildDMI(dat, model)
den <- likelihood(p.vector, dmi)

model <- BuildModel(
p.map     = list(a = "1", v = "1", z = "1", d = "1", t0 = "1", sv = "1",
            sz = "1", st0 = "1"),
constants = c(st0 = 0, d = 0),
match.map = list(M = list(s1 = "r1", s2 = "r2")),
factors   = list(S = c("s1", "s2")),
responses = c("r1", "r2"),
type      = "rd")

p.vector <- c(a = 1, v = 1, z = 0.5, sz = 0.25, sv = 0.2, t0 = .15)
dat <- simulate(model, 1e2, ps = p.vector)
dmi <- BuildDMI(dat, model)
den <- likelihood (p.vector, dmi)
```

---

logLik *Extract Posterior Log-Likelihood*

Description

This function is to extract posterior log-likelihood in a "model" object.

Usage

```
logLik(object, ...)

## S4 method for signature 'posterior'
logLik(object, start = 1, end = NA)

## S4 method for signature 'list'
logLik(object, start = 1, end = NA)

## S4 method for signature 'hyper'
logLik(object, start = 1, end = NA)
```

Arguments

object	posterior samples
...	other arguments passing through dot dot dot.
start	start from which iteration.
end	end at which iteration. For example, set start = 101 and end = 1000, instructs the function to calculate from 101 to 1000 iteration.
hyper	whether to summarise hyper parameters

---

model-class *An S4 class of the process model.*

Description

The class is to represent a process model, e.g., a DDM, a LBA model, a PM model, or a CDDM.

Slots

model A 3-D model array. Dimension one stores the combinations of the factor levels and response types (when discrete), dimension two stores parameters, and dimension three stores response types.

all.par all parameters

p.vector parameter vector, excluding constant parameters

par.names parameter names / labels

type model type

factors a list of factors and their levels

responses response types

constants constant parameters

posdrift a Boolean switch indicating whether drift rates must be positive

n1.order node 1 ordering. This is only for the LBA model

match.cell an indicator matrix storing whether a particular trial matches a cell

match.map a mapping mechanism for calculating whether a trial matches a positive boundary / accumulator or a negative boundary / accumulator.

dimnames dimension names of the model array

pnames parameter names

npar number of parameters

names,prior-method *The Parameter Names in a Prior Object*

Description

Extract parameter names from a prior object. This function extends the names funciton in the base package.

Usage

```
## S4 method for signature 'prior'
names(x)
```

Arguments

x	a prior object.

Value

a string vector

plot *ggdmc Plotting Methods*

Description

The function plots prior distributions or posterior samples depending on whether the first argument x is a prior object or an object storing posterior samples.

Usage

```
plot(x, y = NULL, ...)

## S4 method for signature 'prior'
plot(x, y = NULL, ps = NULL, save = FALSE, ...)

## S4 method for signature 'posterior'
plot(x, y = NULL, hyper = FALSE, start = 1,
  end = NA, pll = TRUE, save = FALSE, den = FALSE,
  subchain = FALSE, nsubchain = 3, chains = NA, ...)

## S4 method for signature 'hyper'
plot(x, y = NULL, hyper = TRUE, start = 1,
  end = NA, pll = TRUE, save = FALSE, den = FALSE,
  subchain = FALSE, nsubchain = 3, chains = NA, ...)
```

```
## S4 method for signature 'list'
plot(x, y = NULL, start = 1, end = NA, pll = TRUE,
  save = FALSE, den = FALSE, subchain = FALSE, nsubchain = 3,
  chains = NA, ...)
```

Arguments

x	a prior object or posterior samples.
y	NULL
...	Additional argument passing via dot dot dot.
ps	a parameter vector
save	a Boolean switch whether to save plotting data
hyper	a Boolean switch, indicating posterior samples are from hierarchical modeling
start	start from iteration
end	end at which iteraton
pll	a Boolean switch whether to plot posterior log likelihoods
den	a Boolean switch whether for density plots
subchain	a Boolean switch whether to plot a subset of chains.
nsubchain	number of subchain
chains	indicate the subchains to plot. This must be an integer vector

Examples

```
p.prior <- BuildPrior(
           dists = rep("tnorm", 7),
           p1    = c(a = 2,   v.f1 = 4,  v.f2 = 3,  z = 0.5, sv = 1,
                     sz = 0.3, t0 = 0.3),
           p2    = c(a = 0.5, v.f1 = .5, v.f2 = .5, z = 0.1, sv = .3,
                     sz = 0.1, t0 = 0.05),
           lower = c(0,-5, -5, 0, 0, 0, 0),
           upper = c(5, 7,  7, 1, 2, 1, 1))
plot(p.prior)
```

---

posterior-class

An S4 class to represent an object storing posterior samples at the participant level. Posterior samples storing both the participant and the hyper lever are represented by an S4 class hyper

---

Description

An S4 class to represent an object storing posterior samples at the participant level. Posterior samples storing both the participant and the hyper lever are represented by an S4 class hyper

Slots

`theta` posterior samples for one-participant fit.

`summed_log_prior` summed log prior likelihoods.

`log_likelihoods` log likelihoods

`dmi` a S4 object of data model instance

`prior` a S4 prior object

`start` the index of starting sample

`npar` number of parameters

`pnames` parameter names

`nmc` number of Monte Carlo samples

`thin` thinning length

`nchain` number of Markov chains

See Also

`hyper-class`

---

print *ggdmc Printing Methods*

Description

The function is an extension of the print function in `base` pacakge. It prints a model object set up by `BuildModel` and a prior object set up by `BuildPrior`.

Usage

```
print(x, ...)

## S4 method for signature 'model'
print(x, ps = NULL, ...)

## S4 method for signature 'prior'
print(x, ...)
```

Arguments

`x`	a model object.
`...`	Additional argument passing via dot dot dot.
`ps`	a parameter vector

Details

The print method for a prior object merely rearranges a prior object as a data frame for the inspection convenience.

Value

The original model object, a list of parameter matrices or a prior matrix

Examples

```
model <- BuildModel(
          p.map     = list(A = "1", B = "1", t0 = "1", mean_v = "M",
                           sd_v = "1", st0 = "1"),
          match.map = list(M = list(s1 = 1, s2 = 2)),
          factors   = list(S = c("s1", "s2")),
          constants = c(st0 = 0, sd_v = 1),
          responses = c("r1", "r2"),
          type      = "norm")

p.vector <- c(A = .75, B = 1.25, t0 = .15, mean_v.true = 2.5,
              mean_v.false = 1.5)

print(model)
print(model, ps=p.vector)

dat <- simulate(model, nsim = 10, ps = p.vector);
dmi <- BuildDMI(dat, model)
p.prior <- BuildPrior(
  dists = c("tnorm", "tnorm", "beta", "tnorm", "tnorm"),
  p1    = c(A = 1, B = 1, t0 = 1, mean_v.true = 1, mean_v.false = 1),
  p2    = c(1,  1,  1, 1, 1),
  lower = c(rep(0, 3),  rep(NA, 2)),
  upper = c(rep(NA, 2), 1, rep(NA, 2)))

print(p.prior)

## A different example printing a prior object
pop.mean  <- c(a=1,  v.f1=1,  v.f2=.2, z=.5, sz=.3,  sv.f1=.25, sv.f2=.23,
               t0=.3)
pop.scale <- c(a=.2, v.f1=.2, v.f2=.2, z=.1, sz=.05, sv.f1=.05, sv.f2=.05,
               t0=.05)

p.prior <- BuildPrior(
  dists = rep("tnorm", 8),
  p1    = pop.mean,
  p2    = pop.scale,
  lower = c(0, -5, -5, 0, 0, 0, 0, 0),
  upper = c(2,  5,  5, 1, 2, 2, 1, 1))

print(p.prior)
```

prior-class *An S4 class to represent an object storing prior distributions*

Description

An S4 class to represent an object storing prior distributions

Slots

`npar` the number of parameters

`pnames` the names of parameters

`priors` a list storing the location parameter, scale parameter, upper bound, lower bound, log indicator (0=FALSE, 1=TRUE), distribution type and transform information.

random *Random number generation*

Description

A wrapper function for generating random numbers from different model types, `rd`, `norm`, `norm_pda`, `norm_pda_gpu`, or `cddm`. `pmat` is generated usually by `TableParameter`.

Usage

```
random(type, pmat, n, seed = NULL, ...)
```

Arguments

`type`	a character string of the model type
`pmat`	a matrix of response x parameter
`n`	number of observations. This must be an integer.
`seed`	an integer specifying a random seed
`...`	other arguments

Details

Note PM model uses `norm` type.

Examples

```
model <- BuildModel(
p.map     = list(a = "1", v="1", z="1", d="1", sz="1", sv="1", t0="1", st0="1"),
match.map = list(M = list(s1 = "r1", s2 = "r2")),
factors   = list(S = c("s1", "s2")),
responses = c("r1","r2"),
constants = c(st0 = 0, d = 0, sv = 0, sz = 0),
type      = "rd")

p.vector <- c(a=1, v=1.5, z=0.6, t0=.15)

pmat <- TableParameters(p.vector, 1, model, FALSE)
type <- model@type;
res1 <- random(type, pmat, 1)
res2 <- random(type, pmat, 10)

model <- BuildModel(
p.map     = list(A = "1", B = "R", t0 = "1", mean_v = c("D", "M"),
                 sd_v = "M", st0 = "1"),
match.map = list(M = list(s1 = 1, s2 = 2)),
factors   = list(S = c("s1", "s2"), D = c("d1", "d2")),
constants = c(sd_v.false = 1, st0 = 0),
responses = c("r1", "r2"),
type      = "norm")

p.vector <- c(A=.51, B.r1=.69, B.r2=.88, t0=.24, mean_v.d1.true=1.1,
              mean_v.d2.true=1.0, mean_v.d1.false=.34, mean_v.d2.false=.02,
              sd_v.true=.11)

pmat <- TableParameters(p.vector, 1, model, FALSE)
type <- model@type;
res1 <- random(type, pmat, 1)
res2 <- random(type, pmat, 10)
```

`rlba_norm` *Generate Random Deviates of the LBA Distribution*

Description

`rlba_norm`, only slightly faster than `maker`, calls C++ function directly.

Usage

```
rlba_norm(n, A, b, mean_v, sd_v, t0, st0, posdrift)
```

Arguments

`n`	is the numbers of observation.
`A`	start point upper bound, a vector of a scalar.

b	decision threshold, a vector or a scalar.
mean_v	mean drift rate vector
sd_v	standard deviation of drift rate vector
t0	nondecision time, a vector.
st0	nondecision time variation, a vector.
posdrift	if exclude negative drift rates

Value

a n x 2 matrix of RTs (first column) and responses (second column).

`rprior` *Generate Random Numbers*

Description

Random number generation based on a prior object

Usage

```
rprior(x, ...)

## S4 method for signature 'prior'
rprior(x, n = 1)
```

Arguments

x	a prior object.
...	Additional argument passing via dot dot dot.
n	number of observations

Examples

```
p.prior <- BuildPrior(
 dists = c("tnorm", "tnorm", "beta", "tnorm", "beta", "beta"),
 p1    = c(a = 1, v = 0, z = 1, sz = 1, sv = 1, t0 = 1),
 p2    = c(a = 1, v = 2, z = 1, sz = 1, sv = 1, t0 = 1),
 lower = c(0,-5, NA, NA, 0, NA),
 upper = c(2, 5, NA, NA, 2, NA))

rprior(p.prior, 9)
##               a           v         z         sz        sv         t0
## [1,] 0.97413686  0.78446178 0.9975199 -0.5264946 0.5364492 0.55415052
## [2,] 0.72870190  0.97151662 0.8516604  1.6008591 0.3399731 0.96520848
## [3,] 1.63153685  1.96586939 0.9260939  0.7041254 0.4138329 0.78367440
## [4,] 1.55866180  1.43657110 0.6152371  0.1290078 0.2957604 0.23027759
## [5,] 1.32520281 -0.07328408 0.2051155  2.4040387 0.9663111 0.06127237
```

```
## [6,] 0.49628528 -0.19374770 0.5142829  2.1452972 0.4335482 0.38410626
## [7,] 0.03655549  0.77223432 0.1739831  1.4431507 0.6257398 0.63228368
## [8,] 0.71197612 -1.15798082 0.8265523  0.3813370 0.4465184 0.23955415
## [9,] 0.38049166  3.32132034 0.9888108  0.9684292 0.8437480 0.13502154
```

`rvonmises` *Generate random deviates from a von Mises distribution*

Description

This function generates random numbers in radian unit from a von Mises distribution using the location (ie mean) parameter, mu and the concentration (ie precision) parameter kappa.

Usage

```
rvonmises(n, mu, kappa)

dvonmises(x, mu, kappa)

pvonmises(q, mu, kappa, tol = 1e-20)
```

Arguments

`n`	number of observations
`mu`	mean direction of the distribution. Must be a scalar.
`kappa`	concentration parameter. A positive value for the concentration parameter of the distribution. Must be a scalar.
`x`, `q`	x and q are the quantiles. These must be one a scalar.
`tol`	the tolerance imprecision for von Mist distribution function.

Details

A random number for a circular normal distribution has the form:

$$f(theta; mu, kappa) = 1/(2 * pi * I0(kappa)) * exp(kappa * cos(theta - mu))$$

theta is between 0 and 2*pi.

`I0(kappa)` in the normalizing constant is the modified Bessel function of the first kind and order zero.

Value

a column vector

References

Ulric Lund, Claudio Agostinelli, et al's (2017). R package 'circular': Circular Statistics (version 0.4-91). https://r-forge.r-project.org/projects/circular/

Examples

```
n  <- 1e2
mu <- 0
k  <- 10

## Not run:
vm1 <- circular:::RvonmisesRad(n, mu, k)
vm2 <- rvm(n, mu, k)
vm3 <- circular:::conversion.circular(circular:::circular(vm1))
vm4 <- circular:::conversion.circular(circular:::circular(vm2))
plot(vm3)
plot(vm4)

## End(Not run)
```

---

simulate,model-method *Simulate Choice Responses*

Description

The function is an extension of the simulate function in `stats` pacakge. It simulates the data from either two-alternative force choice tasks, multiple-alternative force choice task, or continuous report tasks.

Usage

```
## S4 method for signature 'model'
simulate(object, nsim = 1, seed = NULL, nsub,
  prior = NA, ps = NA)
```

Arguments

`object`	a model object.
`nsim`	number of observations. `nsim` can be a single number for a balanced design or a matrix for an unbalanced design, where rows are participants and columns are design cells. If the matrix has one row than all participants have the same `nsim` in each cell, if it has one column then all cells have the same `nsim`; Otherwise each entry specifies the `nsim` for a particular participant x design cell combination.
`seed`	a user specified random seed.
`nsub`	number of participants
`prior`	a prior object
`ps`	a true parameter vector or matrix.
`...`	additional optional arguments.

Details

The function simulates data either for one participant or multiple participants. The simulation process is based on the model object, entering via `object` argument. For simulating one participant, one must supply a true parameter vector to the `ps` argument.

For simulating multiple participants, one can enter a matrix or a row vector as true parameters. Each row is used to generate the data for a participant. This process is usually dubbed the fixed-effect modelling. To generate data via the random-effect modelling, one must supply a set of prior distributions. In this case, `ps` argument is unused. Note in some cases, a random-effect modelling may fail to draw data from the model, because true parameters are randomly drawn from prior distributions. This would happen sometimes for example in the diffusion decision model, because certain parameter combinations are considered invalid (e.g., t0 < 0, zr > a) for obvious reasons.

`ps` can be a row vector, in which case each participant has one set of identical parameters. It can also be a matrix with one row per participant, in which case it must have `ns` rows. The true values will be saved as `parameters` attribute in the output.

Value

a data frame

Examples

```
model <- BuildModel(
  p.map     = list(A = "1", B = "1", t0 = "1", mean_v = "M", sd_v = "1",
                   st0 = "1"),
  match.map = list(M = list(s1 = 1, s2 = 2)),
  factors   = list(S = c("s1", "s2")),
  constants = c(st0 = 0, sd_v = 1),
  responses = c("r1", "r2"),
  type      = "norm")

p.vector <- c(A = .75, B = 1.25, t0 = .15, mean_v.true = 2.5,
              mean_v.false = 1.5)
ntrial <- 100
dat <- simulate(model, nsim = ntrial, ps = p.vector)
```

StartNewsamples *Start new model fits*

Description

Fit a hierarchical or a fixed-effect model, using Bayeisan optimisation. We use a specific type of pMCMC algorithm, the DE-MCMC. This particular sampling method includes crossover and two different migration operators. The migration operators are similar to random-walk algorithm. They would be less efficient to find the target parameter space, if been used alone.

Usage

```
StartNewsamples(dmi, prior, nmc = 200, thin = 1, nchain = NULL,
  report = 100, rp = 0.001, gammamult = 2.38, pm0 = 0.05,
  pm1 = 0.05, block = TRUE, ncore = 1)

run(samples, nmc = 500, thin = 1, report = 100, rp = 0.001,
  gammamult = 2.38, pm0 = 0, pm1 = 0, block = TRUE, ncore = 1,
  add = FALSE, prior = NULL)
```

Arguments

dmi	a data model instance or a list of data model instances
prior	prior objects. For hierarchical model, this must be a list with three sets of prior distributions. Each is respectively named, "pprior", "location", and "scale".
nmc	number of Monte Carlo samples
thin	thinning length
nchain	number of chains
report	progress report interval
rp	tuning parameter 1
gammamult	tuning parameter 2. This is the step size.
pm0	probability of migration type 0 (Hu & Tsui, 2010)
pm1	probability of migration type 1 (Turner et al., 2013)
block	Only for hierarchical modeling. A Boolean switch for update one parameter at a time
ncore	Only for non-hierarchical, fixed-effect models with many subjects.
samples	posterior samples.
add	Boolean whether to add new samples

---

summary *ggdmc Summary Methods*

Description

Summarise posterior samples. Note when recovery = TRUE, the prob vector will be fixed at the default values.

Usage

```
summary(object, ...)

## S4 method for signature 'posterior'
summary(object, start = 1, end = NA,
  prob = c(0.025, 0.25, 0.5, 0.75, 0.975), recovery = FALSE, ps = NA,
  verbose = FALSE, digits = max(3, getOption("digits") - 3))

## S4 method for signature 'list'
summary(object, start = 1, end = NA, prob = c(0.025,
  0.25, 0.5, 0.75, 0.975), recovery = FALSE, ps = NA,
  verbose = FALSE, digits = max(3, getOption("digits") - 3))

## S4 method for signature 'hyper'
summary(object, hyper = TRUE, start = 1, end = NA,
  prob = c(0.025, 0.25, 0.5, 0.75, 0.975), recovery = FALSE, ps = NA,
  type = 1, verbose = FALSE, digits = max(3, getOption("digits") -
  3))
```

Arguments

object	an object storing posterior samples.
...	Additional argument passing via dot dot dot.
start	start from which iteration.
end	end at which iteration. For example, set start = 101 and end = 1000, instructs the function to calculate from 101st to 1000th iteration.
prob	a numeric vector, indicating the quantiles to calculate
recovery	a Boolean switch indicating if samples are from a recovery study.
ps	true parameter values. This is only for recovery studies
verbose	print more information
digits	printing digits
hyper	a Boolean switch to plot hyper parameters
type	calculate type 1 = location or type 2 = scale hyper parameters

Examples

```
## Not run:
model <- BuildModel(
     p.map    = list(a = "1", v = "F", z = "1", d = "1", sz = "1", sv = "1",
                t0 = "1", st0 = "1"),
    match.map = list(M = list(s1 = "r1", s2 = "r2")),
    factors   = list(S = c("s1", "s2"), F = c("f1", "f2")),
    constants = c(st0 = 0, d = 0),
    responses = c("r1", "r2"),
    type      = "rd")
npar <- model@npar
```

```
## Population distribution
pop.mean  <- c(a=2,   v.f1=4,  v.f2=3,  z=0.5, sz=0.3, sv=1,  t0=0.3)
pop.scale <- c(a=0.5, v.f1=.5, v.f2=.5, z=0.1, sz=0.1, sv=.3, t0=0.05)
pop.prior <- BuildPrior(
  dists = rep("tnorm", npar),
  p1    = pop.mean,
  p2    = pop.scale,
  lower = c(0,-5, -5, 0, 0, 0, 0),
  upper = c(5, 7,  7, 1, 2, 1, 1))

## Simulate some data
dat <- simulate(model, nsub = 30, nsim = 30, prior = pop.prior)
dmi <- BuildDMI(dat, model)
ps <- attr(dat, "parameters")

p.prior <- BuildPrior(
  dists = rep("tnorm", npar),
  p1    = pop.mean,
  p2    = pop.scale*5,
  lower = c(0,-5, -5, 0, 0, 0, 0),
  upper = c(5, 7,  7, 1, 2, 1, 1))

mu.prior <- ggdmc::BuildPrior(
  dists = rep("tnorm", npar),
  p1    = pop.mean,
  p2    = pop.scale*5,
  lower = c(0,-5, -5, 0, 0, 0, 0),
  upper = c(5, 7,  7, 1, 2, 1, 1)
)
sigma.prior <- BuildPrior(
  dists = rep("beta", npar),
  p1    = c(a=1, v.f1=1,v.f2 = 1, z=1, sz=1, sv=1, t0=1),
  p2    = rep(1, npar),
  upper = rep(2, npar))

priors <- list(pprior=p.prior, location=mu.prior, scale=sigma.prior)

## Sampling
## Processing time: 394.37 secs.
fit0 <- StartNewsamples(dmi, priors, thin = 2)
fit  <- run(fit0)
fit  <- run(fit, 1e2, add=TRUE)

## By default the type = 1 for location parameters
## When recovery = TRUE, one must enter the true parameter to ps
est0 <- summary(fit, recovery = TRUE, ps = pop.mean, verbose = TRUE)
## Explicitly enter type = 1
est0 <- summary(fit, recovery = TRUE, ps = pop.mean,  type=1, verbose = TRUE)
est0 <- summary(fit, recovery = TRUE, ps = pop.scale, type=2, verbose = TRUE)

## When recovery = FALSE (default), the function return parameter estimates
est0 <- summary(fit, verbose = TRUE, type=1)
```

```
est0 <- summary(fit, verbose = TRUE, type=2)

## To estimate individual participants, one must enter hyper = FALSE for a
## hierarchical model fit
est0 <- summary(fit, hyper=FALSE, verbose = TRUE)

## End(Not run)
```

`TableParameters` *Table response and parameter*

Description

`TableParameters` arranges the values in a parameter vector and creates a response x parameter matrix. The matrix is used by the likelihood function, assigning a trial to a cell for calculating probability densities.

Usage

```
TableParameters(p.vector, cell, model, n1order)
```

Arguments

`p.vector`	a parameter vector
`cell`	a string or an integer indicating a design cell, e.g., `s1.f1.r1` or 1. Note the integer cannot exceed the number of cell. One can check this by entering `length(dimnames(model))`.
`model`	a model object
`n1order`	a Boolean switch, indicating using node 1 ordering. This is only for LBA-like models and its n1PDF likelihood function.

Value

each row corresponding to the model parameter for a response. When `n1.order` is FALSE, TableParameters returns a martix without rearranging into node 1 order. For example, this is used in the `simulate` function. By default `n1.order` is TRUE.

Examples

```
m1 <- BuildModel(
  p.map     = list(a = "1", v = "F", z = "1", d = "1", sz = "1", sv = "F",
                   t0 = "1", st0 = "1"),
  match.map = list(M = list(s1 = "r1", s2 = "r2")),
  factors   = list(S = c("s1", "s2"), F = c("f1","f2")),
  constants = c(st0 = 0, d = 0),
  responses = c("r1","r2"),
  type      = "rd")
```

```
m2 <- BuildModel(
  p.map = list(A = "1", B = "1", mean_v = "M", sd_v = "1",
    t0 = "1", st0 = "1"),
  constants = c(st0 = 0, sd_v = 1),
  match.map = list(M = list(s1 = 1, s2 = 2)),
  factors   = list(S = c("s1", "s2")),
  responses = c("r1", "r2"),
  type      = "norm")

pvec1 <- c(a = 1.15, v.f1 = -0.10, v.f2 = 3, z = 0.74, sz = 1.23,
           sv.f1 = 0.11, sv.f2 = 0.21, t0 = 0.87)
pvec2 <- c(A = .75, B = .25, mean_v.true = 2.5, mean_v.false = 1.5,
           t0 = .2)

print(m1, pvec1)
print(m2, pvec2)

accMat1 <- TableParameters(pvec1, "s1.f1.r1", m1, FALSE)
accMat2 <- TableParameters(pvec2, "s1.r1",    m2, FALSE)

##    a    v   t0    z d   sz   sv st0
## 1.15 -0.1 0.87 0.26 0 1.23 0.11   0
## 1.15 -0.1 0.87 0.26 0 1.23 0.11   0

##    A b  t0 mean_v sd_v st0
## 0.75 1 0.2    2.5    1   0
## 0.75 1 0.2    1.5    1   0
```

`trial_loglik_hier` *Extract trial log likelihoods*

Description

This function simply run trial_loglik to loop through one subject after another to extracts trial_log_likes from a list of subject fits and concatanates the result into an array.

Usage

```
trial_loglik_hier(samples, thin = 1, verbose = FALSE)
```

Arguments

`samples`	posterior samples
`thin`	thinnng length
`verbose`	whether print information

`unstick_one` *Unstick posterios samples (One subject)*

Description

Unstick posterios samples (One subject)

Unstick posterios samples (One subject)

Usage

```
unstick_one(x, bad)
```


```
unstick_one(x, bad)
```


Arguments

`x`	posterior samples
`bad`	a numeric vector, indicating which chains to remove
`x`	posterior samples
`bad`	a numeric vector, indicating which chains to remove

Index