Package 'nimble'

October 11, 2016

Title Flexible BUGS-Compatible System for Hierarchical Statistical Modeling and Algorithm Development

Description Flexible application of algorithms to models specified in the BUGS language. Algorithms can be written in the NIMBLE language and made available to any model.

Version 0.6-1

Maintainer Christopher Paciorek <paciorek@stat.berkeley.edu>

Author Perry de Valpine, Christopher Paciorek, Daniel Turek, Cliff Anderson-Bergman, Duncan Temple Lang

Depends R (>= 3.1.2)

Imports methods,igraph,coda

Suggests testthat,R2WinBUGS,rjags,rstan,xtable,abind,ggplot2

URL http://r-nimble.org

SystemRequirements GNU make

License BSD_3_clause + file LICENSE

Note For convenience, the package includes the necessary header files for the Eigen distribution. (This is all that is needed to use that functionality.) You can use an alternative installation of Eigen on your system or the one we provide. The license for the Eigen code is very permissive and allows us to distribute it with this package. See http://eigen.tuxfamily.org/index.php?title=Main_Page and also the License section on that page.

LazyData false

Collate config.R all_utils.R options.R distributions_inputList.R distributions_processInputList.R distributions_implementations.R BUGS_BUGSdecl.R BUGS_contexts.R BUGS_nimbleGraph.R BUGS_modelDef.R BUGS_model.R BUGS_graphNodeMaps.R BUGS_readBUGS.R BUGS_testBUGS.R BUGS_getDependencies.R BUGS_utils.R BUGS_mathCompatibility.R genCpp_exprClass.R genCpp_operatorLists.R genCpp_RparseTree2exprClasses.R genCpp_initSizes.R genCpp_buildIntermediates.R genCpp_processSpecificCalls.R

genCpp_sizeProcessing.R genCpp_insertAssertions.R genCpp_maps.R
genCpp_liftMaps.R genCpp_eigenization.R genCpp_addDebugMarks.R
genCpp_generateCpp.R RCfunction_core.R RCfunction_compile.R
nimbleFunction_util.R nimbleFunction_core.R
nimbleFunction_nodeFunction.R nimbleFunction_nodeFunctionNew.R
nimbleFunction_Rexecution.R nimbleFunction_compile.R
nimbleFunction_keywordProcessing.R types_util.R
types_symbolTable.R types_modelValues.R
types_modelValuesAccessor.R types_modelVariableAccessor.R
types_nimbleFunctionList.R types_nodeFxnVector.R
types_numericLists.R cppDefs_utils.R cppDefs_variables.R
cppDefs_core.R cppDefs_namedObjects.R cppDefs_BUGSmodel.R
cppDefs_RCfunction.R cppDefs_nimbleFunction.R
cppDefs_modelValues.R cppDefs_cppProject.R
cppDefs_outputCppFromRparseTree.R cppInterfaces_utils.R
cppInterfaces_models.R cppInterfaces_modelValues.R
cppInterfaces_nimbleFunctions.R cppInterfaces_otherTypes.R
nimbleProject.R initializeModel.R MCMC_utils.R
MCMC_configuration.R MCMC_build.R MCMC_run.R MCMC_samplers.R
MCMC_conjugacy.R MCMC_suite.R MCMC_comparisons.R
MCMC_autoBlock.R MCEM_build.R filtering_auxiliary.R
filtering_liuwest.R filtering_enkf.R filtering_bootstrap.R
optimTools.R NF_utils.R miscFunctions.R makevars.R
setNimbleInternalFunctions.R registration.R nimble-package.r
zzz.R

RoxygenNote 5.0.1

NeedsCompilation yes

Repository CRAN

Date/Publication 2016-10-11 18:56:44

R topics documented:

`asRow` *Turn a numeric vector into a single-row or single-column matrix*

Description

Turns a numeric vector into a matrix that has 1 row or 1 column. Part of NIMBLE language.

Usage

```
asRow(x)

asCol(x)
```

Arguments

x	Numeric to be turned into a single row or column matrix

Details

In the NIMBLE language, some automatic determination of how to turn vectors into single-row or single-column matrices is done. For example, in `A %*% x`, where A is a matrix and x a vector, x will be turned into a single-column matrix unless it is known at compile time that A is a single column, in which case x will be turned into a single-row matrix. However, if it is desired that x be turned into a single row but A cannot be determined at compile time to be a single column, then one can use `A %*% asRow(x)` to force this conversion.

Author(s)

Perry de Valpine

---

autoBlock *Automated parameter blocking procedure for efficient MCMC sampling*

Description

Runs NIMBLE's automated blocking procedure for a given model object, to dynamically determine a blocking scheme of the continuous-valued model nodes. This blocking scheme is designed to produce efficient MCMC sampling (defined as number of effective samples generated per second of algorithm runtime). See Turek, et al (2015) for details of this algorithm. This also (optionally) compares this blocked MCMC against several static MCMC algorithms, including all univariate sampling, blocking of all continuous-valued nodes, NIMBLE's default MCMC configuration, and custom-specified blockings of parameters.

Usage

```
autoBlock(Rmodel, autoIt = 20000, run = list("all", "default"),
  setSeed = TRUE, verbose = FALSE, makePlots = FALSE, round = TRUE)
```

Arguments

Rmodel	A NIMBLE model object, created from nimbleModel.
autoIt	The number of MCMC iterations to run intermediate MCMC algorithms, through the course of the procedure. Default 20,000.
run	List of additional MCMC algorithms to compare against the automated blocking MCMC. These may be specified as: the character string 'all' to denote blocking all continuous-valued nodes; the character string 'default' to denote NIMBLE's default MCMC configuration; a named list element consisting of a quoted code block, which when executed returns an MCMC configuration object for comparison; a custom-specificed blocking scheme, specified as a named list element which itself is a list of character vectors, where each character vector specifies the nodes in a particular block. Default is c('all', 'default').

setSeed	Logical specificying whether to call set.seed(0) prior to beginning the blocking procedure. Default TRUE.
verbose	Logical specifying whether to output considerable details of the automated block procedure, through the course of execution. Default FALSE.
makePlots	Logical specifying whether to plot the hierarchical clustering dendrograms, through the course of execution. Default FALSE.
round	Logical specifying whether to round the final output results to two decimal places. Default TRUE.

Details

This method allows for fine-tuned usage of the automated blocking procedure. However, the main entry point to the automatic blocking procedure is intended to be through either buildMCMC(..., autoBlock = TRUE), or configureMCMC(..., autoBlock = TRUE).

Value

Returns a named list containing elements:

- summary: A data frame containing a numerical summary of the performance of all MCMC algorithms (including that from automated blocking)
- autoGroups: A list specifying the parameter blockings converged on by the automated blocking procedure
- conf: A NIMBLE MCMC configuration object corresponding to the results of the automated blocking procedure

Author(s)

Daniel Turek

References

Turek, D., de Valpine, P., Paciorek, C., and Anderson-Bergman, C. (2015). Automated Parameter Blocking for Efficient Markov-Chain Monte Carlo Sampling. arXiv: 1503.05621.

See Also

configureMCMC buildMCMC

BUGSdeclClass-class *BUGSdeclClass contains the information extracted from one BUGS declaration*

Description

BUGSdeclClass contains the information extracted from one BUGS declaration

`buildAuxiliaryFilter` *Create an auxiliary particle filter algorithm to estimate log-likelihood.*

Description

Create an auxiliary particle filter algorithm for a given NIMBLE state space model.

Usage

```
buildAuxiliaryFilter(model, nodes, control = list())
```

Arguments

`model`	A NIMBLE model object, typically representing a state space model or a hidden Markov model
`nodes`	A character vector specifying the latent model nodes over which the particle filter will stochastically integrate to estimate the log-likelihood function
`control`	A list specifying different control options for the particle filter. Options are described in the details section below.

Details

The `control` argument can be specified as follows.

lookahead The lookahead function used to calculate auxiliary weights. Can choose between `'mean'` and `'simulate'`. Defaults to `'simulate'`.

saveAll Indicates whether to save state samples for all time points (`TRUE`), or only for the most recent time point (`FALSE`)

smoothing Decides whether to save smoothed estimates of latent states, i.e., samples from f(x[1:t]|y[1:t]) if `smoothing = TRUE`, or instead to save filtered samples from f(x[t]|y[1:t]) if `smoothing = FALSE`. `smoothing = TRUE` only works if `saveAll = TRUE`.

timeIndex An integer used to manually specify which dimension of the latent state variable indexes time. This need only be set if the number of time points is less than or equal to the size of the latent state at each time point.

The auxiliary particle filter modifies the bootstrap filter (`buildBootstrapFilter`) by adding a lookahead step to the algorithm: before propagating particles from one time point to the next via the transition equation, the auxiliary filter calculates a weight for each pre-propogated particle by predicting how well the particle will agree with the next data point. These pre-weights are used to conduct an initial resampling step before propagation.

The resulting specialized particle filter algorithm will accept a single integer argument (`m`, default 10,000), which specifies the number of random \'particles\' to use for estimating the log-likelihood. The algorithm returns the estimated log-likelihood value, and saves unequally weighted samples from the posterior distribution of the latent states in the `mvWSamples` modelValues object, with corresponding logged weights in `mvWSamples['wts',]`. An equally weighted sample from the posterior can be found in the `mvEWsamp` modelValues object.

The auxiliary particle filter uses a lookahead function to select promising particles before propagation. This function can eithre be the expected value of the latent state at the next time point (`lookahead = 'mean'`) or a simulation from the distribution of the latent state at the next time point (`lookahead = 'simulate'`), conditioned on the current particle.

Author(s)

Nicholas Michaud

References

Pitt, M.K., and Shephard, N. (1999). Filtering via simulation: Auxiliary particle filters. *Journal of the American Statistical Association* 94(446): 590-599.

Examples

```
## Not run:
model <- nimbleModel(code = ...)
my_AuxF <- buildAuxiliaryFilter(model, 'x[1:100]',
   control = list(saveAll = TRUE, lookahead = 'mean'))
Cmodel <- compileNimble(model)
Cmy_AuxF <- compileNimble(my_AuxF, project = model)
logLike <- Cmy_AuxF(m = 100000)
hist(as.matrix(Cmy_Auxf$mvEWSamples, 'x'))

## End(Not run)
```

---

buildBootstrapFilter *Create a bootstrap particle filter algorithm to estimate log-likelihood.*

Description

Create a bootstrap particle filter algorithm for a given NIMBLE state space model.

Usage

```
buildBootstrapFilter(model, nodes, control = list())
```

Arguments

`model`	A nimble model object, typically representing a state space model or a hidden Markov model
`nodes`	A character vector specifying the latent model nodes over which the particle filter will stochastically integrate over to estimate the log-likelihood function
`control`	A list specifying different control options for the particle filter. Options are described in the details section below.

Details

Each of the `control()` list options are described in detail here:

thresh A number between 0 and 1 specifying when to resample: the resampling step will occur when the effective sample size is less than `thresh` times the number of particles. Defaults to 0.8.

saveAll Indicates whether to save state samples for all time points (TRUE), or only for the most recent time point (FALSE)

smoothing Decides whether to save smoothed estimates of latent states, i.e., samples from f(x[1:t]|y[1:t]) if `smoothing = TRUE`, or instead to save filtered samples from f(x[t]|y[1:t]) if `smoothing = FALSE`. `smoothing = TRUE` only works if `saveAll = TRUE`.

timeIndex An integer used to manually specify which dimension of the latent state variable indexes time. Only needs to be set if the number of time points is less than or equal to the size of the latent state at each time point.

The bootstrap filter starts by generating a sample of estimates from the prior distribution of the latent states of a state space model. At each time point, these particles are propagated forward by the model's transition equation. Each particle is then given a weight proportional to the value of the observation equation given that particle. The weights are used to draw an equally-weighted sample of the particles at this time point. The algorithm then proceeds to the next time point. Neither the transition nor the observation equations are required to be normal for the bootstrap filter to work.

The resulting specialized particle filter algorithm will accept a single integer argument (`m`, default 10,000), which specifies the number of random \'particles\' to use for estimating the log-likelihood. The algorithm returns the estimated log-likelihood value, and saves unequally weighted samples from the posterior distribution of the latent states in the `mvWSamples` modelValues object, with corresponding logged weights in `mvWSamples['wts',]`. An equally weighted sample from the posterior can be found in the `mvEWsamp` `modelValues` object.

Author(s)

Daniel Turek and Nicholas Michaud

References

Gordon, N.J., D.J. Salmond, and A.F.M. Smith. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEEE Proceedings F (Radar and Signal Processing)*. Vol. 140. No. 2. IET Digital Library, 1993.

Examples

```
## Not run:
model <- nimbleModel(code = ...)
my_BootF <- buildBootstrapFilter(model, 'x[1:100]')
Cmodel <- compileNimble(model)
Cmy_BootF <- compileNimble(my_BootF, project = model)
logLike <- Cmy_BootF(m = 100000)
boot_X <- as.matrix(Cmy_BootF$mvEWSamples)

## End(Not run)
```

`buildEnsembleKF` *Create an Ensemble Kalman filter algorithm to sample from latent states.*

Description

Create an Ensemble Kalman filter algorithm for a given NIMBLE state space model.

Usage

```
buildEnsembleKF(model, nodes, control = list())
```

Arguments

`model`	A NIMBLE model object, typically representing a state space model or a hidden Markov model
`nodes`	A character vector specifying the latent model nodes which the Ensemble Kalman filter will estimate.
`control`	A list specifying different control options for the particle filter. Options are described in the details section below.

Details

The `control()` list option is described in detail below:

saveAll Indicates whether to save state samples for all time points (TRUE), or only for the most recent time point (FALSE)

timeIndex An integer used to manually specify which dimension of the latent state variable indexes time. Only needs to be set if the number of time points is less than or equal to the size of the latent state at each time point.

Runs an Ensemble Kalman filter to estimate a latent state given observations at each time point. The ensemble Kalman filter is a Monte Carlo approximation to a Kalman filter that can be used when the model's transition euqations do not follow a normal distribution. Latent states (x[t]) and observations (y[t]) can be scalars or vectors at each time point, and sizes of observations can vary from time point to time point. In the BUGS model, the observations (y[t]) must be equal to some (possibly nonlinear) deterministic function of the latent state (x[t]) plus an additive error term. Currently only normal and multivariate normal error terms are supported. The transition from x[t] to x[t+1] does not have to be normal or linear. Output from the posterior distribution of the latent states is stored in `mvSamples`.

Author(s)

Nicholas Michaud

References

Houtekamer, P.L., and H.L. Mitchell. (1998). Data assimilation using an ensemble Kalman filter technique. *Monthly Weather Review*, 126(3), 796-811.

Examples

```
## Not run:
model <- nimbleModel(code = ...)
my_ENKFF <- buildEnsembleKF(model, 'x')
Cmodel <- compileNimble(model)
Cmy_ENKF <- compileNimble(my_ENKF, project = model)
Cmy_ENKF$run(m = 100000)
ENKF_X <- as.matrix(Cmy_ENKF$mvSamples, 'x')
hist(ENKF_X)

## End(Not run)
```

`buildLiuWestFilter` *Create a Liu and West particle filter algorithm.*

Description

Create a Liu and West particle filter algorithm for a given NIMBLE state space model.

Usage

```
buildLiuWestFilter(model, nodes, params = NULL, control = list())
```

Arguments

`model`	A NIMBLE model object, typically representing a state space model or a hidden Markov model
`nodes`	A character vector specifying the latent model nodes over which the particle filter will stochastically integrate over to estimate the log-likelihood function
`params`	A character vector specifying the top-level parameters to estimate the posterior distribution of. If unspecified, parameter nodes are specified as all stochastic top level nodes which are not in the set of latent nodes specified in `nodes`.
`control`	A list specifying different control options for the particle filter. Options are described in the details section below.

Details

Each of the `control()` list options are described in detail below:

d A discount factor for the Liu-West filter. Should be close to, but not above, 1.

saveAll Indicates whether to save state samples for all time points (TRUE), or only for the most recent time point (FALSE)

timeIndex An integer used to manually specify which dimension of the latent state variable indexes time. Only needs to be set if the number of time points is less than or equal to the size of the latent state at each time point.

The Liu and West filter samples from the posterior distribution of both the latent states and top-level parameters for a state space model. Each particle in the Liu and West filter contains values not only for latent states, but also for top level parameters. Latent states are propogated via an auxiliary step, as in the auxiliary particle filter (`buildAuxiliaryFilter`). Top-level parameters are propagated from one time point to the next through a smoothed kernel density based on previous particle values.

The resulting specialized particle filter algorthm will accept a single integer argument (`m`, default 10,000), which specifies the number of random \'particles\' to use for sampling from the posterior distributions. The algorithm saves unequally weighted samples from the posterior distribution of the latent states and top-level parameters in `mvWSamples`, with corresponding logged weights in `mvWSamples['wts',]`. An equally weighted sample from the posterior can be found in `mvEWSamples`.

Note that if `saveAll=TRUE`, the top-level parameter samples given in the `mvWSamples` output will correspond to the weights from the final time point.

Author(s)

Nicholas Michaud

References

Liu, J., and M. West. (2001). Combined parameter and state estimation in simulation-based filtering. *Sequential Monte Carlo methods in practice*. Springer New York, pages 197-223.

Examples

```
## Not run:
model <- nimbleModel(code = ...)
my_LWF <- buildLiuWestFilter(model, 'x[1:100]', params = 'sigma_x')
Cmodel <- compileNimble(model)
Cmy_LWF <- compileNimble(my_LWF, project = model)
Cmy_LWF$run(100000)
lw_X <- as.matrix(Cmy_LWF$mvEWSamples, 'x')

#  samples from posterior of a top level parameter named sigma_x:
lw_sigma_x <- as.matrix(Cmy_LWF$mvEWSamples, 'sigma_x')

## End(Not run)
```

buildMCEM *Builds an MCEM algorithm from a given NIMBLE model*

Description

Takes a NIMBLE model and builds an MCEM algorithm for it. The user must specify which latent nodes are to be integrated out in the E-Step. All other stochastic non-data nodes will be maximized over. If the nodes do not have positive density on the entire real line, then box constraints can be used to enforce this. The M-step is done by a nimble MCMC sampler. The E-step is done by a call to R's `optim` with `method = 'L-BFGS-B'`.

Usage

```
buildMCEM(model, latentNodes, burnIn = 500, mcmcControl = list(adaptInterval
  = 100), boxConstraints = list(), buffer = 10^-6, alpha = 0.01,
  beta = 0.01, gamma = 0.01, C = 0.001, numReps = 300, verbose = TRUE)
```

Arguments

`model`	a nimble model
`latentNodes`	character vector of the names of the stochastic nodes to integrated out. Names can be expanded, but don't need to be. For example, if the model contains `x[1]`, `x[2]` and `x[3]` then one could provide either `latentNodes = c('x[1]', 'x[2]', 'x[3]')` or `latentNodes = 'x'`.
`burnIn`	burn-in used for MCMC sampler in E step
`mcmcControl`	list passed to `configureMCMC`, which builds the MCMC sampler. See `help(configureMCMC)` for more details
`boxConstraints`	list of box constraints for the nodes that will be maximized over. Each constraint is a list in which the first element is a character vector of node names to which the constraint applies and the second element is a vector giving the lower and upper limits. Limits of `-Inf` or `Inf` are allowed.
`buffer`	A buffer amount for extending the boxConstraints. Many functions with boundary constraints will produce `NaN` or -Inf when parameters are on the boundary. This problem can be prevented by shrinking the boundary a small amount.
`alpha`	probability of a type one error - here, the probability of accepting a parameter estimate that does not increase the likelihood. Default is 0.01.
`beta`	probability of a type two error - here, the probability of rejecting a parameter estimate that does increase the likelihood. Default is 0.01.
`gamma`	probability of deciding that the algorithm has converged, that is, that the difference between two Q functions is less than C, when in fact it has not. Default is 0.01.
`C`	determines when the algorithm has converged - when C falls above a (1-gamma) confidence interval around the difference in Q functions from time point t-1 to time point t, we say the algorithm has converged. Default is 0.001.
`numReps`	number of bootstrap samples to use for asymptotic variance calculation
`verbose`	logical indicating whether to print additional logging information

Details

buildMCEM calls the NIMBLE compiler to create the MCMC and objective function as nimbleFunctions. If the given model has already been used in compiling other nimbleFunctions, it is possible you will need to create a new copy of the model for buildMCEM to use. Uses an ascent-based MCEM algorithm, which includes rules for automatically increasing the number of MC samples as iterations increase, and for determining when convergence has been reached.

Value

an R function that when called runs the MCEM algorithm. The function returned takes the arguments listed in Runtime Arguments.

Runtime Arguments

- initM starting number of iterations for the algorithm.

Author(s)

Clifford Anderson-Bergman and Nicholas Michaud

References

Caffo, Brian S., Wolfgang Jank, and Galin L. Jones. "Ascent-based Monte Carlo expectation-maximization." Journal of the Royal Statistical Society: Series B (Statistical Methodology) 67.2 (2005): 235-251.

Examples

```
## Not run:
pumpCode <- nimbleCode({
 for (i in 1:N){
     theta[i] ~ dgamma(alpha,beta);
     lambda[i] <- theta[i]*t[i];
     x[i] ~ dpois(lambda[i])
 }
 alpha ~ dexp(1.0);
 beta ~ dgamma(0.1,1.0);
})

pumpConsts <- list(N = 10,
              t = c(94.3, 15.7, 62.9, 126, 5.24,
                31.4, 1.05, 1.05, 2.1, 10.5))

pumpData <- list(x = c(5, 1, 5, 14, 3, 19, 1, 1, 4, 22))

pumpInits <- list(alpha = 1, beta = 1,
             theta = rep(0.1, pumpConsts$N))
pumpModel <- nimbleModel(code = pumpCode, name = 'pump', constants = pumpConsts,
                  data = pumpData, inits = pumpInits)

# Want to maximize alpha and beta (both which must be positive) and integrate over theta
```

```
box = list( list(c('alpha','beta'), c(0, Inf)))

pumpMCEM <- buildMCEM(model = pumpModel, latentNodes = 'theta[1:10]',
                      boxConstraints = box)
pumpMCEM(initM = 1000)

## End(Not run)
# Could also use latentNodes = 'theta' and buildMCEM() would figure out this means 'theta[1:10]'
```

`buildMCMC` *Create an MCMC function, from an MCMCconf object*

Description

Accepts a single required argument, which may be of class MCMCconf, or inherit from class modelBaseClass (a NIMBLE model object). Returns an MCMC function; see details section.

Usage

```
buildMCMC(conf, ...)
```

Arguments

`conf`	An object of class MCMCconf that specifies the model, samplers, monitors, and thinning intervals for the resulting MCMC function. See `configureMCMC` for details of creating MCMCconf objects. Alternatively, `MCMCconf` may a NIMBLE model object, in which case an MCMC function corresponding to the default MCMC configuration for this model is returned.
`...`	Additional arguments to be passed to `configureMCMC` if `conf` is a NIMBLE model object

Details

Calling buildMCMC(conf) will produce an uncompiled (R) R mcmc function object, say 'Rmcmc'.

The uncompiled MCMC function will have arguments:

`niter`: The number of iterations to run the MCMC.

`reset`: Boolean specifying whether to reset the model and stored samples. This will simulate into any stochastic nodes with value NA, propagate values through any deterministic nodes, and calculate all model probabilities. This will also reset the internal stored MCMC samples. Specifying `reset=FALSE` allows the MCMC algorithm to continue running from where it left off. Generally, `reset=FALSE` should only be used when the MCMC has already been run (default = TRUE).

`simulateAll`: Boolean specifying whether to simulate into all stochastic nodes. This will overwrite the current values in all stochastic nodes (default = FALSE).

`time`: Boolean specifying whether to record runtimes of the individual internal MCMC samplers. When `time=TRUE`, a vector of runtimes (measured in seconds) can be extracted from the MCMC using the method `mcmc$getTimes()` (default = FALSE).

progressBar: Boolean specifying whether to display a progress bar during MCMC execution (default = TRUE).

Samples corresponding to the monitors and monitors2 from the MCMCconf are stored into the interval variables mvSamples and mvSamples2, respectively. These may be accessed and converted into R matrix objects via: as.matrix(mcmc$mvSamples) as.matrix(mcmc$mvSamples2)

The uncompiled (R) MCMC function may be compiled to a compiled MCMC object, taking care to compile in the same project as the R model object, using: Cmcmc <- compileNimble(Rmcmc, project=Rmodel)

The compiled function will function identically to the uncompiled object, except acting on the compiled model object.

Author(s)

Daniel Turek

Examples

```
## Not run:
code <- nimbleCode({
    mu ~ dnorm(0, 1)
    x ~ dnorm(mu, 1)
})
Rmodel <- nimbleModel(code)
conf <- configureMCMC(Rmodel)
Rmcmc <- buildMCMC(conf)
Cmodel <- compileNimble(Rmodel)
Cmcmc <- compileNimble(Rmcmc, project=Rmodel)
Cmcmc$run(10000)
samples <- as.matrix(Cmcmc$mvSamples)
head(samples)

## End(Not run)
```

Categorical *The Categorical Distribution*

Description

Density and random generation for the categorical distribution

Usage

```
dcat(x, prob, log = FALSE)

rcat(n = 1, prob)
```

Arguments

x	non-negative integer-value numeric value.
prob	vector of probabilities, summing to one.
log	logical; if TRUE, probability density is returned on the log scale.
n	number of observations.

Details

See the BUGS manual for mathematical details.

Value

dcat gives the density and rcat generates random deviates.

Author(s)

Christopher Paciorek

See Also

Distributions for other standard distributions

Examples

```
probs <- c(1/4, 1/10, 1 - 1/4 - 1/10)
x <- rcat(n = 30, probs)
dcat(x, probs)
```

---

checkInterrupt *Check for interrupt (e.g. Ctrl-C) during nimbleFunction execution. Part of the NIMBLE language.*

Description

Check for interrupt (e.g. Ctrl-C) during nimbleFunction execution. Part of the NIMBLE language.

Usage

```
checkInterrupt()
```

Details

During execution of nimbleFunctions that take a long time, it is nice to occassionally check if the user has entered an interrupt and bail out of execution if so. This function does that. During uncompiled nimbleFunction execution, it does nothing. During compiled execution, it calls R_checkUserInterrupt() of the R headers.

Author(s)

Perry de Valpine

`CmodelBaseClass-class` *Class* `CmodelBaseClass`

Description

Classes used internally in NIMBLE and not expected to be called directly by users.

`CnimbleFunctionBase-class`
Class `CnimbleFunctionBase`

Description

Classes used internally in NIMBLE and not expected to be called directly by users.

`codeBlockClass-class` *Class* `codeBlockClass`

Description

Classes used internally in NIMBLE and not expected to be called directly by users.

`combine_MCMC_comparison_results`
Combine multiple objects returned by compareMCMCs

Description

Useful for running different cases separately and combining them.

Usage

```
combine_MCMC_comparison_results(..., name = "MCMCresults")
```

Arguments

`...`	objects returned by `compareMCMCs`
`name`	(default "MCMCresults") name to be given to the resulting set of comparisons. This will be used in the html generated by `make_MCMC_comparison_pages`. It is simply a list name and thus can easily be modified later.

Value

An object in the same format as returned by compareMCMCs with summary = TRUE.

See Also

compareMCMCs, rename_MCMC_comparison_method, make_MCMC_comparison_pages, reshape_comparison_results.

---

compareMCMCs *Run multiple MCMCs (packages or NIMBLE cases) for multiple models and return summary results*

Description

Manages the input and output for multiple calls to MCMCsuite to generate comparisons among MCMCs

Usage

```
compareMCMCs(modelInfo, MCMCs = c("nimble"), MCMCdefs, BUGSdir, stanDir,
  stanInfo, doSamplePlots = FALSE, verbose = TRUE, summary = TRUE, ...)
```

Arguments

modelInfo	A set of model information for which one or more MCMCs should be run. Can take one of several formats: (1) a character vector of names of classic WinBUGS examples. (2) for one model, a list with elements code (containing the model code as returned by nimbleCode), data (containing a list of data that can be provided to nimbleModel, constants (containing a list of constants to be provided to nimbleModel), inits (containing a list of initial values that can be provided to nimbleModel; this is optional unless it is needed to match names in some of the other arguments, as described below), and name (a character name; this is optional). (3) for multiple models, a list of lists formatted as per option (2). See nimbleModel about handling of data vs. constants. In particular, if both are provided in data, then nimbleModel will try to determine which is which.
MCMCs	An object acceptable as the MCMCs argument to MCMCsuite. This specifies the set of MCMCs to be run. Valid entries include 'jags', 'nimble', 'nimble_RW', 'nimble_slice', 'autoBlock', 'stan', 'winbugs', 'openbugs', or a name provided in the MCMCdefs list.
MCMCdefs,	An optional object acceptable as the MCMCdefs argument to MCMCsuite.
BUGSdir	If models is a character vector of WinBUGS example names, BUGSdir can optionally provide the directory (as a character string) where to find them. If missing, they will be looked up in the installed nimble package using getBUGSexampleDir.
stanDir	Optional base directory in which Stan model code will be found (but stanInfo can override this).

stanInfo	A list of information for finding and using the Stan code for each model. If names of list elements are provided, they will be used to match either the character vector of names provided for models (option (1)) or the list names of models (options (2) or (3)). If names are not provided, the models will be used in order. Each element must be a list that can or must include (i) dir (optional: the subdirectory to use, instead of stanDir; often the subdirectory is the model name). (ii) codeFile (optional: the name of the stan code file, to which ".stan" will be appended if not already there; if omitted, this will be set to the value of modelName. (iii) data (optional: the name of a the stan data file, to which ".data.R" will be appended if not already there; if omitted, the system will try using stanCodeFile with ".data.R" appended). (iv) inits (optional: the name of a the stan inits file, to which ".init.R" will be appended if not already there; if omitted, the system will try using stanCodeFile with ".init.R" appended). (v) modelName (optional: a name for the model; if omitted, a value from the models argument will be used). (vi) stanParameterRules (optional: a list whose names are BUGS variable names, with each element a list with element StanSourceName to give the corresponding Stan variable name and transform to give a function that converts a value of the Stan variable to a value of the BUGS variable).
doSamplePlots	(default FALSE) This is passed to MCMCsuite as both the makePlots and savePlots argument.
verbose	(default FALSE) If TRUE, a message will be shown about which model is being worked on.
summary	(deafault TRUE) If TRUE, the return value will be a list with elements summary, timing and efficiency. If FALSE, the return value will be the return value generated by MCMCsuite. Among other things, the latter contains the actual samples.
...	additional arguments to be passed to MCMCsuite.

Details

compareMCMCs wraps a call to MCMCsuite

Value

If summary is FALSE, compareMCMCs returns the object returned by MCMCsuite, which comes from MCMCsuiteClass. If summary is TRUE, it returns a list with elements timing, efficiency, and summary.

See Also

MCMCsuiteClass, updateMCMCcomparisonWithHighOrderESS, make_MCMC_comparison_pages, reshape_comparison_results, combine_MCMC_comparison_results, rename_MCMC_comparison_method, reshape_comparison_results.

`compileNimble` *compile NIMBLE models and nimbleFunctions*

Description

compile a collection of models and nimbleFunctions: generate C++, compile the C++, load the result, and return an interface object

Usage

```
compileNimble(..., project, dirName = NULL, projectName = "",
  control = list(), resetFunctions = FALSE)
```

Arguments

`...`	An arbitrary set of NIMBLE models and nimbleFunctions, or lists of them. If given as named parameters, those names may be used in the return list.
`project`	Optional NIMBLE model or nimbleFunction already associated with a project, which the current units for compilation should join. If not provided, a new project will be created and the current compilation units will be associated with it.
`dirName`	Optional directory name in which to generate the C++ code. If not provided, a temporary directory will be generated using R's `tempdir` function.
`projectName`	Optional character name for labeling the project if it is new
`control`	A list mostly for internal use. See details.
`resetFunctions`	Logical value stating whether nimbleFunctions associated with an existing project should all be reset for compilation purposes. See details.

Details

This is the main function for calling the NIMBLE compiler. A set of compiler calls and output will be seen. Compiling in NIMBLE does 4 things: 1. It generates C++ code files for all the model and nimbleFunction components. 2. It calls the system's C++ compiler. 3. It loads the compiled object(s) into R using `dyn.load`. And 4. it generates R objects for using the compiled model and nimbleFunctions.

When the units for compilation provided in `...` include multiple models and/or nimbleFunctions, models are compiled first, in the order in which they are provided. Groups of nimbleFunctions that were specialized from the same nimbleFunction generator (the result of a call to `nimbleFunction`, which then takes setup arguments and returns a specialized nimbleFunction) are then compiled as a group, in the order of first appearance.

The behavior of adding new compilation units to an existing project is limited. For example, one can compile a model in one call to `compileNimble` and then compile a nimbleFunction that uses the model (i.e. was given the model as a setup argument) in a second call to `compileNimble`, with the model provided as the `project` argument. Either the uncompiled or compiled model can be provided. However, compiling a second nimbleFunction and adding it to the same project will only

work in limited circumstances. Basically, the limitations occur because it attempts to re-use already compiled pieces, but if these do not have all the necessary information for the new compilation, it gives up. An attempt has been made to give up in a controlled manner and provide somewhat informative messages.

When compilation is not allowed or doesn't work, try using `resetFunctions = TRUE`, which will force recompilation of all nimbleFunctions in the new call. Previously compiled nimbleFunctions will be unaffected, and their R interface objects should continue to work. The only cost is additional compilation time for the current compilation call. If that doesn't work, try re-creating the model and/or the nimbleFunctions from their generators. An alternative possible fix is to compile multiple units in one call, rather than sequentially in multiple calls.

The control list can contain the following named elements, each with `TRUE` or `FALSE`: debug, which sets a debug mode for the compiler for development purposes; debugCpp, which inserts an output message before every line of C++ code for debugging purposes; compileR, which determines whether the R-only steps of compilation should be executed; writeCpp, which determines whether the C++ files should be generated; compileCpp, which determines whether the C++ should be compiled; loadSO, which determines whether the DLL or shared object should be loaded and interfaced; and returnAsList, which determines whether calls to the compiled nimbleFunction should return only the returned value of the call (`returnAsList = FALSE`) or whether a list including the input arguments, possibly modified, should be returned in a list with the returned value of the call at the end (`returnAsList = TRUE`). The control list is mostly for developer use, although `returnAsArgs` may be useful to a user. An example of developer use is that one can have the compiler write the C++ files but not compile them, then modify them by hand, then have the C++ compiler do the subsequent steps without over-writing the files.

See NIMBLE User Manual for examples

Value

If there is only one compilation unit (one model or nimbleFunction), an R interface object is returned. This object can be used like the uncompiled model or nimbleFunction, but execution will call the corresponding compiled objects or functions. If there are multiple compilation units, they will be returned as a list of interface objects, in the order provided. If names were included in the arguments, or in a list if any elements of `...` are lists, those names will be used for the corresponding element of the returned list. Otherwise an attempt will be made to generate names from the argument code. For example `compileNimble(A = fun1, B = fun2, project = myModel)` will return a list with named elements A and B, while `compileNimble(fun1, fun2, project = myModel)` will return a list with named elements fun1 and fun2.

Author(s)

Perry de Valpine

`configureMCMC` *Build the MCMCconf object for construction of an MCMC object*

Description

Creates a defaut MCMC configuration for a given model. The resulting object is suitable as an argument to buildMCMC.

Usage

```
configureMCMC(model, nodes, control = list(), monitors, thin = 1,
  monitors2 = character(), thin2 = 1, useConjugacy = TRUE,
  onlyRW = FALSE, onlySlice = FALSE, multivariateNodesAsScalars = FALSE,
  print = FALSE, autoBlock = FALSE, oldConf, ...)
```

Arguments

model	A NIMBLE model object, created from nimbleModel
nodes	An optional character vector, specifying the nodes and/or variables for which samplers should be created. Nodes may be specified in their indexed form, y[1, 3]. Alternatively, nodes specified without indexing will be expanded fully, e.g., x will be expanded to x[1], x[2], etc. If missing, the default value is all non-data stochastic nodes. If NULL, then no samplers are added.
control	An optional list of control arguments to sampler functions. If a control list is provided, the elements will be provided to all sampler functions which utilize the named elements given. For example, the standard Metropolis-Hastings random walk sampler (sampler_RW) utilizes control list elements adaptive, adaptInterval, and scale. (Internally it also uses targetNode, but this should not generally be provided as a control list element). The default values for control list arguments for samplers (if not otherwise provided as an argument to configureMCMC()) are in the NIMBLE system option MCMCcontrolDefaultList.
monitors	A character vector of node names or variable names, to record during MCMC sampling. This set of monitors will be recorded with thinning interval thin, and the samples will be stored into the mvSamples object. The default value is all top-level stochastic nodes of the model – those having no stochastic parent nodes.
thin	The thinning interval for monitors. Default value is one.
monitors2	A character vector of node names or variable names, to record during MCMC sampling. This set of monitors will be recorded with thinning interval thin2, and the samples will be stored into the mvSamples2 object. The default value is an empty character vector, i.e. no values will be recorded.
thin2	The thinning interval for monitors2. Default value is one.
useConjugacy	A logical argument, with default value TRUE. If specified as FALSE, then no conjugate samplers will be used, even when a node is determined to be in a conjugate relationship.
onlyRW	A logical argument, with default value FALSE. If specified as TRUE, then Metropolis-Hastings random walk samplers (sampler_RW) will be assigned for all non-terminal continuous-valued nodes nodes. Discrete-valued nodes are assigned a slice sampler (sampler_slice), and terminal nodes are assigned a posterior_predictive sampler (sampler_posterior_predictive).

onlySlice	A logical argument, with default value FALSE. If specified as TRUE, then a slice sampler is assigned for all non-terminal nodes. Terminal nodes are still assigned a posterior_predictive sampler.
multivariateNodesAsScalars	A logical argument, with default value FALSE. If specified as TRUE, then non-terminal multivariate stochastic nodes will have scalar samplers assigned to each of the scalar components of the multivariate node. The default value of FALSE results in a single block sampler assigned to the entire multivariate node. Note, multivariate nodes appearing in conjugate relationships will be assigned the corresponding conjugate sampler (provided `useConjugacy == TRUE`), regardless of the value of this argument.
print	A logical argument, specifying whether to print the ordered list of default samplers.
autoBlock	A logical argument specifying whether to use an automated blocking procedure to determine blocks of model nodes for joint sampling. If TRUE, an MCMC configuration object will be created and returned corresponding to the results of the automated parameter blocking. Default value is FALSE.
oldConf	An optional MCMCconf object to modify rather than creating a new MCMCconf from scratch
...	Additional arguments to be passed to the `autoBlock()` function when `autoBlock = TRUE`

Details

See `MCMCconf` for details on how to manipulate the `MCMCconf` object

Author(s)

Daniel Turek

`Constraint` *Constraint calculations in NIMBLE*

Description

Calculations to handle censoring

Usage

```
dconstraint(x, cond, log = FALSE)

rconstraint(n = 1, cond)
```

Arguments

x	value indicating whether cond is TRUE or FALSE
cond	logical value
log	logical; if TRUE, probability density is returned on the log scale.
n	number of observations (only n=1 is handled currently).

Details

Used for working with constraints in BUGS code. See the NIMBLE manual for additional details.

Value

`dconstraint` gives the density and `rconstraint` generates random deviates, but these are unusual as the density is 1 if `x` matches `cond` and 0 otherwise and the deviates are simply the value of `cond`

Author(s)

Christopher Paciorek

See Also

Distributions for other standard distributions

Examples

```
constr <- 3 > 2 && 4 > 0
x <- rconstraint(1, constr)
dconstraint(x, constr)
dconstraint(0, 3 > 4)
dconstraint(1, 3 > 4)
rconstraint(1, 3 > 4)
```

`decide` *Makes the Metropolis-Hastings acceptance decision, based upon the input (log) Metropolis-Hastings ratio*

Description

This function returns a logical TRUE/FALSE value, indicating whether the proposed transition should be accepted (TRUE) or rejected (FALSE).

Usage

```
decide(logMetropolisRatio)
```

Arguments

`logMetropolisRatio`
: The log of the Metropolis-Hastings ratio, which is calculated from model probabilities and forward/reverse transition probabilities. Calculated as the ratio of the model probability under the proposal to that under the current values multiplied by the ratio of the reverse transition probability to the forward transition probability.

Details

The Metropolis-Hastings accept/reject decisions is made as follows. If `logMetropolisRatio` is greater than 0, accept (return `TRUE`). Otherwise draw a uniform random number between 0 and 1 and accept if it is less that `exp(logMetropolisRatio`. The proposed transition will be rejected (return `FALSE`). If `logMetropolisRatio` is NA, NaN, or -Inf, a reject (`FALSE`) decision will be returned.

Author(s)

Daniel Turek

`decideAndJump` *Creates a nimbleFunction for executing the Metropolis-Hastings jumping decision, and updating values in the model, or in a carbon copy modelValues object, accordingly.*

Description

This nimbleFunction generator must be specialized to three required arguments: a model, a modelValues, and a character vector of node names.

Usage

```
decideAndJump(model, mvSaved, calcNodes)
```

Arguments

`model`	An uncompiled or compiled NIMBLE model object.
`mvSaved`	A modelValues object containing identical variables and logProb variables as the model. Can be created by `modelValues(model)`.
`calcNodes`	A character vector representing a set of nodes in the model (and hence also the modelValues) object.

Details

Calling decideAndJump(model, mvSaved, calcNodes) will generate a specialized nimbleFunction with four required numeric arguments:

modelLP1: The model log-probability associated with the newly proposed value(s)

modelLP0: The model log-probability associated with the original value(s)

propLP1: The log-probability associated with the proposal forward-transition

propLP0: The log-probability associated with the proposal reverse-tranisiton

Executing this function has the following effects: – Calculate the (log) Metropolis-Hastings ratio, as logMHR = modelLP1 - modelLP0 - propLP1 + propLP0 – Make the proposal acceptance decision based upon the (log) Metropolis-Hastings ratio – If the proposal is accepted, the values and associated logProbs of all calcNodes are copied from the model object into the mvSaved object –

If the proposal is rejected, the values and associated logProbs of all calcNodes are copied from the mvSaved object into the model object – Return a logical value, indicating whether the proposal was accepted

Author(s)

Daniel Turek

`declare` *Explicitly declare a variable in run-time code of a nimbleFunction*

Description

Explicitly declare a variable in run-time code of a nimbleFunction, for cases when its dimensions cannot be inferred before it is used. Works in R and NIMBLE.

Usage

```
declare(name, def)
```

Arguments

`name`	Name of a variable to declare, without quotes
`def`	NIMBLE type declaration, of the form `TYPE(nDim, sizes)`, where `TYPE` is `integer`, `double`, or `logical`, `nDim` is the number of dimensions, and `sizes` is an optional vector of sizes concatenated with `c`. If `nDim` is omitted, it defaults to 0, indicating a scalar. If sizes are provided, they should not be changed subsequently in the function, including by assignment. Omitting `nDim` results in a scalar. For `logical`, only scalar is currently supported.

Details

In a run-time function of a nimbleFunction (either the `run` function or a function provided in `methods` when calling `nimbleFunction`), the dimensionality and numeric type of a variable is inferred when possible from the statement first assigning into it. E.g. `A <- B + C` infers that `A` has numeric types, dimensions and sizes taken from `B + C`. However, if the first appearance of `A` is e.g. `A[i] <- 5`, `A` must have been explicitly declared. In this case, `declare(A, double(1))` would make `A` a 1-dimensional (i.e. vector) double.

When sizes are not set, they can be set by a call to `setSize` or by assignment to the whole object. Sizes are not automatically extended if assignment is made to elements beyond the current sizes. In compiled nimbleFunctions doing so can cause a segfault and crash the R session.

This part of the NIMBLE language is needed for compilation, but it also runs in R. When run in R, is works by the side effect of creating or modifying `name` in the calling environment.

Author(s)

NIMBLE development team

Examples

```
declare(A, logical())             ## scalar logical, the only kind allowed
declare(B, integer(2, c(10, 10))) ## 10 x 10 integer matrix
declare(C, double(3))             ## 3-dimensional double array with no sizes set.
```

`deregisterDistributions` *Remove user-supplied distributions from use in NIMBLE BUGS models*

Description

Deregister distributional information originally supplied by the user for use in BUGS model code

Usage

```
deregisterDistributions(distributionsNames)
```

Arguments

`distributionsNames`
: a character vector giving the names of the distributions to be dergistered

Author(s)

Christopher Paciorek

`Dirichlet` *The Dirichlet Distribution*

Description

Density and random generation for the Dirichlet distribution

Usage

```
ddirch(x, alpha, log = FALSE)

rdirch(n = 1, alpha)
```

Arguments

`x`	vector of values.
`alpha`	vector of parameters of same length as `x`
`log`	logical; if TRUE, probability density is returned on the log scale.
`n`	number of observations (only `n=1` is handled currently).

Details

See Gelman et al., Appendix A or the BUGS manual for mathematical details.

Value

`ddirch` gives the density and `rdirch` generates random deviates.

Author(s)

Christopher Paciorek

References

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. (2004) *Bayesian Data Analysis*, 2nd ed. Chapman and Hall/CRC.

See Also

Distributions for other standard distributions

Examples

```
alpha <- c(1, 10, 30)
x <- rdirch(1, alpha)
ddirch(x, alpha)
```

Exponential *The Exponential Distribution*

Description

Density, distribution function, quantile function and random generation for the exponential distribution with rate (i.e., mean of `1/rate`) or scale parameterizations.

Usage

```
dexp_nimble(x, rate = 1/scale, scale = 1, log = FALSE)

rexp_nimble(n = 1, rate = 1/scale, scale = 1)

pexp_nimble(q, rate = 1/scale, scale = 1, lower.tail = TRUE,
  log.p = FALSE)

qexp_nimble(p, rate = 1/scale, scale = 1, lower.tail = TRUE,
  log.p = FALSE)
```

Arguments

x	vector of values.
rate	vector of rate values.
scale	vector of scale values.
log	logical; if TRUE, probability density is returned on the log scale.
n	number of observations.
q	vector of quantiles.
lower.tail	logical; if TRUE (default) probabilities are $P[X \leq x]$; otherwise, $P[X > x]$.
log.p	logical; if TRUE, probabilities p are given by user as log(p).
p	vector of probabilities.

Details

NIMBLE's exponential distribution functions use Rmath's functions under the hood, but are parameterized to take both rate and scale and to use 'rate' as the core parameterization in C, unlike Rmath, which uses 'scale'. See Gelman et al., Appendix A or the BUGS manual for mathematical details.

Value

`dexp_nimble` gives the density, `pexp_nimble` gives the distribution function, `qexp_nimble` gives the quantile function, and `rexp_nimble` generates random deviates.

Author(s)

Christopher Paciorek

References

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. (2004) *Bayesian Data Analysis*, 2nd ed. Chapman and Hall/CRC.

See Also

Distributions for other standard distributions

Examples

```
x <- rexp_nimble(50, scale = 3)
dexp_nimble(x, scale = 3)
```

getBUGSexampleDir *Get the directory path to one of the classic BUGS examples installed with NIMBLE package*

Description

NIMBLE comes with some of the classic BUGS examples. `getBUGSexampleDir` looks up the location of an example from its name.

Usage

```
getBUGSexampleDir(example)
```

Arguments

`example`	The name of the classic BUGS example.

Value

Character string of the fully pathed directory of the BUGS example.

Author(s)

Christopher Paciorek

getDefinition *Get nimbleFunction definition*

Description

Returns a list containing the nimbleFunction definition components (setup function, run function, and other member methods) for the supplied nimbleFunction generator or specialized instance.

Usage

```
getDefinition(nf)
```

Arguments

`nf`	A nimbleFunction generator, or a compiled or un-compiled specialized nimbleFunction.

Author(s)

Daniel Turek

getLoadingNamespace *return the namespace in which a nimbleFunction is being loaded*

Description

nimbleFunction constructs and evals a reference class definition. When nimbleFunction is used in package source code, this can lead to problems finding things due to namespace issues. Using where = getLoadingNamespace() in a nimbleFunction in package source code should solve this problem.

Usage

```
getLoadingNamespace()
```

Details

nimbleFunctions defined in the NIMBLE source code use where = getLoadingNamespace(). Please let the NIMBLE developers know if you encounter problems with this.

getNimbleOption *Get NIMBLE Option*

Description

Allow the user to get the value of a global _option_ that affects the way in which NIMBLE operates

Usage

```
getNimbleOption(x)
```

Arguments

x	a character string holding an option name

Value

The value of the option.

Author(s)

Christopher Paciorek

Examples

```
getNimbleOption('verifyConjugatePosteriors')
```

getParam *Get value of a parameter of a stochastic node in a model*

Description

Part of the NIMBLE language

Usage

```
getParam(model, node, param, nodeFunctionIndex)
```

Arguments

model	A NIMBLE model object
node	The name of a stochastic node in the model
param	The name of a parameter for the node
nodeFunctionIndex	For internal NIMBLE use only

Details

For example, suppose node 'x[1:5]' follows a multivariate normal distribution (dmnorm) in a model declared by BUGS code. getParam(model, 'x[1:5]', 'mean') would return the current value of the mean parameter (which may be determined from other nodse). The parameter requested does not have to be part of the parameterization used to declare the node. Rather, it can be any parameter known to the distribution. For example, one can request the scale or rate parameter of a gamma distribution, regardless of which one was used to declare the node.

getsize *Returns number of rows of modelValues*

Description

Returns the number of rows of NIMBLE modelValues object. Works in R and NIMBLE.

Usage

```
getsize(container)
```

Arguments

container	modelValues object

Details

See the User Manual or `help(modelValuesBaseClass)` for information about modelValues objects

Author(s)

Clifford Anderson-Bergman

Examples

```
mvConf <- modelValuesConf(vars = 'a', types = 'double', sizes = list(a = 1) )
mv <- modelValues(mvConf)
 resize(mv, 10)
getsize(mv)
```

---

identityMatrix *Create an Identity matrix*

Description

Returns a d-by-d identity matrix (square matrix of 0's, with 1's on the main diagnol).

Usage

```
identityMatrix(d)
```

Arguments

d	The size of the identity matrix to return, will return a d-by-d matrix

Details

This function can be used in the NIMBLE DSL, i.e. in the run function and member methods of nimbleFunctions.

Value

A d-by-d identity matrix

Author(s)

Daniel Turek

Examples

```
Id <- identityMatrix(d = 3)
```

`initializeModel` *Performs initialization of nimble model node values and log probabilities*

Description

Performs initialization of nimble model node values and log probabilities

Usage

```
initializeModel(model, silent = FALSE)
```

Arguments

`model`	A setup argument, which specializes an instance of this nimble function to a particular model.
`silent`	logical indicating whether to suppress logging information

Details

This nimbleFunction may be used at the beginning of nimble algorithms to perform model initialization. The intended usage is to specialize an instance of this nimbleFunction in the setup function of an algorithm, then execute that specialied function at the beginning of the algorithm run function. The specialized function takes no arguments.

Executing this function ensures that all right-hand-side only nodes have been assigned real values, that all stochastic nodes have a real value, or otherwise have their simulate() method called, that all deterministic nodes have their simulate() method called, and that all log-probabilities have been calculated with the current model values. An error results if model initialization encounters a problem, for example a missing right-hand-side only node value.

Author(s)

Daniel Turek

Examples

```
myNewAlgorithm <- nimbleFunction(
   setup = function(model, ...) {
      my_initializeModel <- initializeModel(model)
      ....
   },
   run = function(...) {
      my_initializeModel()
      ....
   }
)
```

`Interval` *Interval calculations*

Description

Calculations to handle censoring

Usage

```
dinterval(x, t, c, log = FALSE)

rinterval(n = 1, t, c)
```

Arguments

`x`	vector of interval indices.
`t`	vector of values.
`c`	vector of one or more values delineating the intervals.
`log`	logical; if TRUE, probability density is returned on the log scale.
`n`	number of observations.

Details

Used for working with censoring in BUGS code. Taking `c` to define the endpoints of two or more intervals (with implicit endpoints of plus/minus infinity), `x` (or the return value of `rinterval`) gives the non-negative integer valued index of the interval in which `t` falls. See the NIMBLE manual for additional details.

Value

`dinterval` gives the density and `rinterval` generates random deviates, but these are unusual as the density is 1 if `x` indicates the interval in which `t` falls and 0 otherwise and the deviates are simply the interval(s) in which `t` falls.

Author(s)

Christopher Paciorek

See Also

Distributions for other standard distributions

Examples

```
endpoints <- c(-3, 0, 3)
vals <- c(-4, -1, 1, 5)
x <- rinterval(4, vals, endpoints)
dinterval(x, vals, endpoints)
dinterval(c(1, 5, 2, 3), vals, endpoints)
```

---

is.nf *check if a nimbleFunction*

Description

Checks an object to determine if it is a nimbleFunction (i.e., a function created by `nimbleFunction` using setup code).

Usage

```
is.nf(f)
```

Arguments

`f`	object to be tested

See Also

nimbleFunction for how to create a nimbleFunction

---

makeParamInfo *Make an object of information about a model-parameter pairing for getParam. Used internally*

Description

Creates a simple getParam_info object, which has a list with a paramID and a type

Usage

```
makeParamInfo(model, nodes, param)
```

Arguments

model	A model such as returned by nimbleModel.
nodes	A character string naming one one or more stochastic nodes, such as "mu", "c('mu', 'beta[2]')", or "eta[1:3, 2]". getParam only works for one node at a time, but if it is indexed (nodes[i]), then makeParamInfo sets up the information for the entire vector nodes. The processing pathway is used by the NIMBLE compiler.
param	A character string naming a parameter of the distribution followed by node, such as "mean", "rate", "lambda", or whatever parameter names are relevant for the distribution of the node.

Details

This is used internally by getParam. It is not intended for direct use by a user or even a nimbleFunction programmer.

---

make_MCMC_comparison_pages

Make html pages summarizing results from compareMCMCs

Description

This function is extensible: new html elements can be provided, but at the moment it is not very user-friendly to do so.

Usage

```
make_MCMC_comparison_pages(comparisonResults, dir = ".", pageComponents,
  modelNames = names(comparisonResults), control, plot = TRUE)
```

Arguments

comparisonResults	An object returned by compareMCMCs (or combine_MCMC_comparison_results or rename_MCMC_comparison_method).
dir	Directory in which to generate html output (default is current working directory)
pageComponents	An optional named list indicating which components to include in the html page for each MCMC method. Each element should be TRUE or FALSE. Options include: timing (default FALSE): a table of computation times, which is not very useful by itself without information about mixing. efficiencySummary (default FALSE): A summary of minimum and mean MCMC efficiencies (over all top-level parameters), defined as effective sample size / computation time. efficiencySummaryAllParams (default TRUE): Like efficiencySummary, but with the addition of a figure that shows the MCMC efficiency for each top-level parameter. efficiencySummaryAllParams (default TRUE): Like efficiencySummaryAllParams, but showing MCMC Pace, defined as 1/MCMC Efficiency. efficiencyDetails

(default TRUE) grid of efficiency barplots for each top-level parameter. `posteriorSummary` (default TRUE): grid of mean, median, and 95% credible interval for each top-level parameter.

`modelNames` names to use for the models. By default these will be taken from `comparisonResults`.

`control` Optional list with elements `makeTopPage` and/or `mainPageName`. `makeTopPage` has to do with whether an html index page should be generated to link to the results from each model in the `comparisonResults`. `makeTopPage` can take values 'yes', 'no', and 'if_needed', where the last case indicates to make an index page only if there is more than one model in `comparisonResults`. `mainPageName` gives a name for the html file if it is generated, with default 'main' (to which '.html') is appended.

`plot` If TRUE, the html files are actually generated. If FALSE, they are not generated, and instead a list with the figures is returned. Each element of the list can be rendered as the argument of `print` or `plot`.

Details

Effective sample size (ESS) is calculated using the `coda` packages `effectiveSize` function. In cases of poot mixing, this can over-estimate ESS by not using a sufficiently high-order autoregressive model in the estimation procudure. See `updateMCMCcomparisonWithHighOrderESS` for other options.

Value

See argument `plot`

`MCMCconf-class` *Class* `MCMCconf`

Description

Objects of this class configure an MCMC algorithm, specific to a particular model. Objects are normally created by calling configureMCMC. Given an MCMCconf object, the actual MCMC function can be built by calling buildMCMC`(conf)`. See documentation below for method initialize() for details of creating an MCMCconf object.

Methods

`addMonitors(vars, ind = 1, print = TRUE)` Adds variables to the list of monitors.

Arguments:

vars: A character vector of indexed nodes, or variables, which are to be monitored. These are added onto the current monitors list.

print: A logical argument, specifying whether to print all current monitors.

Details: See the initialize() function

`addMonitors2(vars, print = TRUE)` Adds variables to the list of monitors2.

Arguments:

vars: A character vector of indexed nodes, or variables, which are to be monitored. These are added onto the current monitors2 list.

print: A logical argument, specifying whether to print all current monitors.

Details: See the initialize() function

`addSampler(target, type = "RW", control = list(), print = FALSE, name)` Adds a sampler to the list of samplers contained in the MCMCconf object.

Arguments:

target: The target node or nodes to be sampled. This may be specified as a character vector of model node and/or variable names. This argument is required.

type: The type of sampler to add, specified as either a character string or a nimbleFunction object. If the character argument type='newSamplerType', then either newSamplerType or sampler_newSamplertype must correspond to a nimbleFunction (i.e. a function returned by nimbleFunction, not a specialized nimbleFunction). Alternatively, the type argument may be provided as a nimbleFunction itself rather than its name. In that case, the 'name' argument may also be supplied to provide a meaningful name for this sampler. The default value is 'RW' which specifies scalar adaptive Metropolis-Hastings sampling with a normal proposal distribution. This default will result in an error if 'target' specifies more than one target node.

control: A list of control arguments specific to the sampler function. These will override the defaults provided in the NIMBLE system option 'MCMCcontrolDefaultList', and any specified in the control list argument to configureMCMC(). An error results if the sampler function requires any control elements which are not present in this argument, the control list argument to configureMCMC(), or in the NIMBLE system option 'MCMCcontrolDefaultList'.

print: Logical argument, specifying whether to print the details of the newly added sampler, as well as its position in the list of MCMC samplers.

name: Optional character string name for the sampler, which is used by the printSamplers method. If 'name' is not provided, the 'type' argument is used to generate the sampler name.

Details: A single instance of the newly configured sampler is added to the end of the list of samplers for this MCMCconf object.

Invisibly returns a list of the current sampler configurations, which are samplerConf reference class objects.

`getMonitors()` Prints all current monitors and monitors2

Details: See the initialize() function

`getSamplerDefinition(ind)` Returns the nimbleFunction definition of an MCMC sampler.

Arguments:

ind: A numeric vector or character vector. A numeric vector may be used to specify the index of the sampler definition to return, or a character vector may be used to indicate a target node for which the sampler acting on this nodes will be printed. For example, getSamplerDefinition('x[2]') will return the definition of the sampler whose target is model node 'x[2]'. If more than one sampler function is specified, only the first is returned.

Returns a list object, containing the setup function, run function, and additional member methods for the specified nimbleFunction sampler.

`getSamplers(ind)` Returns a list of samplerConf objects.

Arguments:

ind: A numeric vector or character vector. A numeric vector may be used to specify the indices of the samplerConf objects to return, or a character vector may be used to indicate a set of target nodes and/or variables, for which all samplers acting on these nodes will be returned. For example, getSamplers('x') will return all samplerConf objects whose target is model node 'x', or whose targets are contained (entirely or in part) in the model variable 'x'. If omitted, then all samplerConf objects in this MCMC configuration object are returned.

`initialize(model, nodes, control = list(), monitors, thin = 1, monitors2 = character(), thin2 = 1,`

Creates a MCMC configuration for a given model. The resulting object is suitable as an argument to buildMCMC.

Arguments:

model: A NIMBLE model object, created from nimbleModel(...)

nodes: An optional character vector, specifying the nodes for which samplers should be created. Nodes may be specified in their indexed form, 'y[1, 3]', or nodes specified without indexing will be expanded fully, e.g., 'x' will be expanded to 'x[1]', 'x[2]', etc. If missing, the default value is all non-data stochastic nodes. If NULL, then no samplers are added.

control: An optional list of control arguments to sampler functions. If a control list is provided, the elements will be provided to all sampler functions which utilize the named elements given. For example, the standard Metropolis-Hastings random walk sampler (sampler_RW) utilizes control list elements 'adaptive', 'adaptInterval', 'scale', and also 'targetNode' however this should not generally be provided as a control list element to configureMCMC(). The default values for control list arguments for samplers (if not otherwise provided as an argument to configureMCMC) are in the NIMBLE system option 'MCMCcontrolDefaultList'.

monitors: A character vector of node names or variable names, to record during MCMC sampling. This set of monitors will be recorded with thinning interval 'thin', and the samples will be stored into the 'mvSamples' object. The default value is all top-level stochastic nodes of the model – those having no stochastic parent nodes.

monitors2: A character vector of node names or variable names, to record during MCMC sampling. This set of monitors will be recorded with thinning interval 'thin2', and the samples will be stored into the 'mvSamples2' object. The default value is an empty character vector, i.e. no values will be recorded.

thin: The thinning interval for 'monitors'. Default value is one.

thin2: The thinning interval for 'monitors2'. Default value is one.

useConjugacy: A logical argument, with default value TRUE. If specified as FALSE, then no conjugate samplers will be used, even when a node is determined to be in a conjugate relationship.

onlyRW: A logical argument, with default value FALSE. If specified as TRUE, then Metropolis-Hastings random walk samplers will be assigned for all non-terminal continuous-valued nodes nodes. Discrete-valued nodes are assigned a slice sampler, and terminal nodes are assigned a posterior_predictive sampler.

onlySlice: A logical argument, with default value FALSE. If specified as TRUE, then a slice sampler is assigned for all non-terminal nodes. Terminal nodes are still assigned a posterior_predictive sampler.

multivariateNodesAsScalars: A logical argument, with default value FALSE. If specified as TRUE, then non-terminal multivariate stochastic nodes will have scalar samplers assigned to each of the scalar components of the multivariate node. The default value of FALSE results in a single block sampler assigned to the entire multivariate node. Note, multivariate nodes

appearing in conjugate relationships will be assigned the corresponding conjugate sampler (provided useConjugacy == TRUE), regardless of the value of this argument.

print: A logical argument, specifying whether to print the ordered list of default samplers.

`printSamplers(ind)` Prints details of the MCMC samplers.

Arguments:

ind: A numeric vector or character vector. A numeric vector may be used to specify the indices of the samplers to print, or a character vector may be used to indicate a set of target nodes and/or variables, for which all samplers acting on these nodes will be printed. For example, printSamplers('x') will print all samplers whose target is model node 'x', or whose targets are contained (entirely or in part) in the model variable 'x'. If omitted, then all samplers are printed.

`removeSamplers(ind, print = FALSE)` Removes one or more samplers from an MCMCconf object.

Arguments:

ind: A numeric vector or character vector specifying the samplers to remove. A numeric vector may specify the indices of the samplers to be removed. Alternatively, a character vector may be used to specify a set of model nodes and/or variables, and all samplers whose 'target' is among these nodes will be removed. If omitted, then all samplers are removed.

print: A logical argument, default value FALSE, specifying whether to print the current list of samplers once the removal has been done.

`resetMonitors()` Resets the current monitors and monitors2 lists to nothing.

Details: See the initialize() function

`setSamplers(ind, print = FALSE)` Sets the ordering of the list of MCMC samplers.

Arguments:

ind: A numeric vector or character vector. A numeric vector may be used to specify the indicies for the new list of MCMC samplers, in terms of the current ordered list of samplers. For example, if the MCMCconf object currently has 3 samplers, then the ordering may be reversed by calling MCMCconf$setSamplers(3:1), or all samplers may be removed by calling MCMCconf$setSamplers(numeric(0)).

Alternatively, a character vector may be used to specify a set of model nodes and/or variables, and the sampler list will modified to only those samplers acting on these target nodes.

As another alternative, a list of samplerConf objects may be used as the argument, in which case this ordered list of samplerConf objects will define the samplers in this MCMC configuration object, completely over-writing the current list of samplers. No checking is done to ensure the validity of the contents of these samplerConf objects; only that all elements of the list argument are, in fact, samplerConf objects.

print: A logical argument, default value TRUE, specifying whether to print the new list of samplers.

`setThin(thin, print = TRUE)` Sets the value of thin.

Arguments:

thin: The new value for the thinning interval 'thin'.

print: A logical argument, specifying whether to print all current monitors.

Details: See the initialize() function

`setThin2(thin2, print = TRUE)` Sets the value of thin2.

Arguments:

thin2: The new value for the thinning interval 'thin2'.

print: A logical argument, specifying whether to print all current monitors.

Details: See the initialize() function

Author(s)

Daniel Turek

See Also

configureMCMC

Examples

```
code <- nimbleCode({
 mu ~ dnorm(0, 1)
 x ~ dnorm(mu, 1)
})
Rmodel <- nimbleModel(code)
conf <- configureMCMC(Rmodel)
conf$setSamplers(1)
conf$addSampler(target = 'x', type = 'slice', control = list(adaptInterval = 100))
conf$addMonitors('mu')
conf$addMonitors2('x')
conf$setThin(5)
conf$setThin2(10)
conf$getMonitors()
conf$printSamplers()
```

MCMCsuite *Executes multiple MCMC algorithms and organizes results.*

Description

Creates, runs, and organizes output from a suite of MCMC algorithms, all applied to the same model, data, and initial values. This can include WinBUGS, OpenBUGS, JAGS and Stan MCMCs, as well as NIMBLE MCMC algorithms. Trace plots and density plots for the MCMC samples may also be generated and saved.

Usage

```
MCMCsuite(code, constants = list(), data = list(), inits = list(),
  monitors = character(), niter = 10000, burnin = 2000, thin = 1,
  summaryStats = c("mean", "median", "sd", "CI95_low", "CI95_upp"),
  calculateEfficiency = FALSE, MCMCs = "nimble", MCMCdefs = list(),
  winbugs_directory = "C:/WinBUGS14", winbugs_program = "WinBUGS",
```

```
  openbugs_directory = "C:/OpenBUGS323", openbugs_program = "OpenBUGS",
  stan_model = "", stan_inits = NULL, stan_data = NULL,
  stanNameMaps = list(), makePlot = TRUE, savePlot = TRUE,
  plotName = "MCMCsuite", setSeed = TRUE,
  check = getNimbleOption("checkModel"), debug = FALSE)
```

Arguments

code	The quoted code expression representing the model, such as the return value from a call to nimbleCode). No default value, this is a required argument.
constants	A named list giving values of constants for the model. This is the same as the constants argument which would be passed to nimbleModel. Default value is list().
data	A named list giving the data values for the model. This is the same as the data argument which would be passed to nimbleModel or model$setData. Default value is list().
inits	A named list giving the initial values for the model. This is the same as the inits argument which would be passed to nimbleModel or model$setInits. Default value is list().
monitors	A character vector giving the node names or variable names to monitor. The samples corresponding to these nodes will be stored in the output samples, will have summary statistics calculated, and density and trace plots generated. Default value is all top-level stochastic nodes of the model.
niter	Number of MCMC iterations to run. This applies to all MCMC algorithms in the suite. Default value is 10,000.
burnin	Number of initial, post-thinning, MCMC iterations to discard. Default value is 2,000.
thin	Thinning interval for the MCMC samples. This applies to all MCMC algorithms in the suite. The thinning occurs prior to the burnin samples being discarded. Default value is 1.
summaryStats	A character vector, specifying the summary statistics to calculate on the MCMC samples. Each element may be the character name of an exisiting R function (possibly user-defined) which acts on a numeric vector and returns a scalar (e.g., mean or sd, or a character string which when parsed and evaluted will define such a function (e.g., function(x) mean(sqrt(x))). Default value is c('mean', 'median', 'sd', 'CI95_low', 'CI95_upp'), where the final two elements are functions which calculate the limits of a 95 percent Bayesian credible interval.
calculateEfficiency	A logical, specifying whether to calculate the efficiency for each MCMC algorithm. Efficiency is defined as the effective sample size (ESS) of each model parameter divided by the algorithm runtime (in seconds). Default is FALSE.
MCMCs	A character vector specifying the MCMC algorithms to run. 'winbugs' specifies WinBUGS; 'openbugs' specifies OpenBUGS; 'jags' specifies JAGS; 'stan' specifies Stan; in this case, must also provide the 'stan_model' argument; 'nimble' specifies NIMBLE's default MCMC algorithm; 'nimble_noConj'

specifies NIMBLE's default MCMC algorithm without the use of any conjugate Gibbs sampling; 'nimble_RW' specifies NIMBLE MCMC algorithm using only random walk Metropolis-Hastings ('RW') samplers; 'nimble_slice' specifies NIMBLE MCMC algorithm using only slice ('slice') samplers; 'autoBlock' specifies NIMBLE MCMC algorithm with block sampling of dynamically determined parameter groups attempting to maximize sampling efficiency; Anything else will be interpreted as NIMBLE MCMC algorithms, and must have associated entries in the MCMCdefs argument. Default value is 'nimble', which specifies NIMBLE's default MCMC algorithm.

MCMCdefs A named list of MCMC definitions. The names of list elements should correspond to any custom MCMC algorithms specified in the MCMCs argument. The list elements should be quoted expressions, enclosed in braces. When executed, the internal code must return an MCMC configuration object, specifying the corresponding MCMC algorithm; in particular, setting the appropriate samplers. The code may assume existance of the R model object Rmodel, and must *return* the MCMC configuration object. Therefore, the final line of such a code block would frequently be a standalone MCMCconf, to return this object.

winbugs_directory
A character string giving the directory of the executable WinBUGS program for the WinBUGS MCMC. This argument will be passed directly to the bugs(...) call, from the R2WinBUGS library. Default value is 'C:/WinBUGS14'.

winbugs_program
A character string giving the name of the WinBUGS program, for the WinBUGS MCMC. This argument will be passed directly to the bugs(...) call, from the R2WinBUGS library. Default value is 'WinBUGS'.

openbugs_directory
A character string giving the directory of the executable OpenBUGS program for the OpenBUGS MCMC. This argument will be passed directly to the bugs(...) call, from the R2WinBUGS library. Default value is 'C:/OpenBUGS323'.

openbugs_program
A character string giving the name of the OpenBUGS program, for the OpenBUGS MCMC. This argument will be passed directly to the bugs(...) call, from the R2WinBUGS library. Default value is 'OpenBUGS'.

stan_model A character string specifying the location and name of the model file ('modelName.stan') for use with the Stan MCMC program. This argument must include the '.stan' extension, and must be provided whenever the MCMCs argument includes 'stan'.

stan_inits A character string specifying the location and name of the inits file ('modelName.init.R') for use with the Stan MCMC program. This argument must include the '.init.R' extension, and must be provided whenever the MCMCs argument includes 'stan'. If omitted, it will attempt to locate an inits file in the same directory as the Stan model file.

stan_data A character string specifying the location and name of the data file (in the form 'modelName.data.R') for use with the Stan MCMC program. This argument must include the '.data.R' extension, and must be provided whenever the MCMCs argument includes 'stan'. If omitted, it will attempt to locate a data file in the same directory as the Stan model file.

stanNameMaps	A list specifying name mappings between Stan and WinBUGS/OpenBUGS. The syntax for list elements is list(BUGS_PARAM_NAME = list(StanSourceName = 'STAN_PARAM_NAME', transform = function(x) TRANSFORMATION_FUNCTION(x))). The transformation is optional.
makePlot	Logical argument, specifying whether to generate the trace plots and posterior density plots, for each monitored node. Default value is TRUE.
savePlot	Logical argument, specifying whether to save the trace plots and density plots. Plots will be saved into the current working directory. Only used when makePlot == TRUE. Default value is TRUE.
plotName	Character string, giving the file name for saving the trace plots and density plots. Only used when makePlot == TRUE and savePlot == TRUE. Default value is 'MCMCsuite'.
setSeed	Logical argument, specifying whether to set.seed(0) prior to MCMC sampling. Default value is TRUE.
check	Logical argument, specifying whether to check the model object for missing or invalid values. Default is given by the NIMBLE option 'checkModel', see help on nimbleOptions for details.
debug	Logical argument, specifying whether to enter a browser() at the onset of executing each MCMC algrithm. For use in debugging individual MCMC algorithms, if necessary. Default value is FALSE.
...	For internal use only

Details

Creates and runs an MCMC Suite. By default, this will execute the specified MCMCs, record all samples, generate summary statistics, and create and save trace plots and posterior density plots. This default behavior can ben altered via a variety of arguments. Following execution of the MCMC algorithms, returns a named list containing samples, summary, and timing elements. See the NIMBLE User Manual for more information about the organization of the return object.

Value

Returns a named list containing elements: samples: A 3-dimensional array containing samples from each MCMC algorithm. summary: A 3-dimensional array containing summary statistics for each variable and algorithm. timing: A numeric vector containing timing information. efficiency: Minimum and mean sampling efficiencies for each algorithm (only provided if option calculateEfficiency = TRUE). See the NIMBLE User Manual for more information about the organization of the return object.

Author(s)

Daniel Turek

Examples

```
## Not run:
code <- nimbleCode({
```

```
    mu ~ dnorm(0, 1)
    x ~ dnorm(mu, 1)
})
output <- MCMCsuite(code,
                    data = list(x=3),
                    inits = list(mu=0),
                    niter = 10000,
                    monitors = 'mu',
                    MCMCs = c('nimble', 'nimble_RW'),
                    summaryStats = c('mean', 'sd', 'max', 'function(x) max(abs(x))'),
                    makePlot = FALSE)

## End(Not run)
```

MCMCsuiteClass-class *Class* MCMCsuiteClass

Description

Objects of this class create, run, and organize output from a suite of MCMC algorithms, all applied to the same model, data, and initial values. This can include WinBUGS, OpenBUGS, JAGS and Stan MCMCs, as well as NIMBLE MCMC algorithms. Trace plots and density plots for the MCMC samples may also be generated and saved.

Author(s)

Daniel Turek

See Also

MCMCsuite

Examples

```
## Not run:
code <- nimbleCode({
    mu ~ dnorm(0, 1)
    x ~ dnorm(mu, 1)
})
output <- MCMCsuite(code,
                    data = list(x=3),
                    inits = list(mu=0),
                    niter = 10000,
                    monitors = 'mu',
                    MCMCs = c('nimble', 'nimble_RW'),
                    summaryStats = c('mean', 'sd', 'max', 'function(x) max(abs(x))'),
                    makePlot = FALSE)

## End(Not run)
```

`modelBaseClass-class` *Class* `modelBaseClass`

Description

This class underlies all NIMBLE model objects: both R model objects created from the return value of nimbleModel(), and compiled model objects. The model object contains a variety of member functions, for providing information about the model structure, setting or querying properties of the model, or accessing various internal components of the model. These member functions of the modelBaseClass are commonly used in the body of the `setup` function argument to nimbleFunction(), to aid in preparation of node vectors, nimbleFunctionLists, and other runtime inputs. See documentation for `nimbleModel` for details of creating an R model object.

Methods

`check()` Checks for errors in model specification and for missing values that prevent use of calculate/simulate on any nodes

`checkBasics()` Checks for size/dimension mismatches and for presence of NAs in model variables (the latter is not an error but a note of this is given to the user)

`checkConjugacy(nodeVector)` Determines whether or not the input nodes appear in conjugate relationships

Arguments:

nodeVector: A character vector specifying one or more node or variable names. If omitted, all stochastic non-data nodes are checked for conjugacy.

Details: The return value is a named list, with an element corresponding to each conjugate node. The list names are the conjugate node names, and list elements are the control list arguments required by the corresponding MCMC conjugate sampler functions. If no model nodes are conjugate, an empty list is returned.

`expandNodeNames(nodeNames, env = parent.frame(), returnScalarComponents = FALSE, returnType = "name` Takes a vector of nodeNames and returns the unique and expanded names in the model, i.e. 'x' expands to 'x[1]', 'x[2]', ...

Arguments:

nodeNames: a vector of characters of nodes to be expanded. Alternatively, can be a vector of integer graph IDs, but this use is intended only for advanced users

returnScalarComponents: should multivariate nodes (i.e. dmnorm or dmulti) be broken up into scalar components?

returnType: return type. Options are 'names' (character vector) or 'ids' (graph IDs)

sort: should names be topologically sorted before being returned?

`getDependencies(nodes, omit = character(), self = TRUE, determOnly = FALSE, stochOnly = FALSE, incl` Returns a character vector of the nodes dependent upon the input argument nodes, sorted topologically according to the model graph. Aditional input arguments provide flexibility in the values returned.

Arguments:

nodes: Character vector of node names, with index blocks allowed, and/or variable names, the dependents of which will be returned.

omit: Character vector of node names, which will be omitted from the nodes returned. In addition, dependent nodes subsequent to these omitted nodes will not be returned. The omitted nodes argument serves to stop the downward search within the hierarchical model struture, and excludes the specified node.

self: Logical argument specifying whether to include the input argument nodes in the return vector of dependent nodes. Default is TRUE.

determOnly: Logical argument specifying whether to return only deterministic nodes. Default is FALSE.

stochOnly: Logical argument specifying whether to return only stochastic nodes. Default is FALSE.

includeData: Logical argument specifying whether to include 'data' nodes (set via the member method setData). Default is TRUE.

dataOnly: Logical argument specifying whether to return only 'data' nodes. Default is FALSE.

includeRHSonly: Logical argument specifying whether to include right-hand-side-only nodes (model nodes which never appear on the left-hand-side of ~ or <- in the model code). These nodes are neither stochastic nor deterministic, but instead function as variable inputs to the model. Default is FALSE.

downstream: Logical argument specifying whether the downward search through the model hierarchical structure should continue beyond the first and subsequent stochastic nodes encountered, hence returning all nodes downstream of the input nodes. Default is FALSE.

returnType: Character argument specific type object returned. Options are 'names' (returns character vector) and 'ids' (returns numeric graph IDs for model)

returnScalar Componenets: Logical argument specifying whether multivariate nodes should return full node name (i.e. 'x[1:2]') or should break down into scalar componenets (i.e. 'x[1]' and 'x[2]')

Details: The downward search for dependent nodes propagates through deterministic nodes, but by default will halt at the first level of stochastic nodes encountered.

`getDependenciesList(returnNames = TRUE, sort = TRUE)` Returns a list of all neighbor relationships. Each list element gives the one-step dependencies of one vertex, and the element name is the vertex label (integer ID or character node name)

Arguments:

returnNames: If TRUE (default), list names and element contents are returns as character node names, e.g. 'x[1]'. If FALSE, everything is returned using graph IDs, which are unique integer labels for each node.

sort: If TRUE (default), each list element is returned in topologically sorted order. If FALSE, they are returned in arbitrary order.

Details: This provides a fairly raw representation of the graph (model) structure that may be useful for inspecting what NIMBLE has created from model code.

`getDownstream(...)` Identical to getDependencies(..., downstream = TRUE)

Details: See documentation for member method getDependencies.

`getNodeNames(determOnly = FALSE, stochOnly = FALSE, includeData = TRUE, dataOnly = FALSE, includeRH` Returns a character vector of all node names in the model, in topologically sorted order. A variety of logical arguments allow for flexible subsetting of all model nodes.

Arguments:

determOnly: Logical argument specifying whether to return only deterministic nodes. Default is FALSE.

stochOnly: Logical argument specifying whether to return only stochastic nodes. Default is FALSE.

includeData: Logical argument specifying whether to include 'data' nodes (set via the member method setData). Default is TRUE.

dataOnly: Logical argument specifying whether to return only 'data' nodes. Default is FALSE.

includeRHSonly: Logical argument specifying whether to include right-hand-side-only nodes (model nodes which never appear on the left-hand-side of ~ or <- in the model code). Default is FALSE.

topOnly: Logical argument specifying whether to return only top-level nodes from the hierarchical model structure.

latentOnly: Logical argument specifying whether to return only latent (mid-level) nodes from the hierarchical model structure.

endOnly: Logical argument specifying whether to return only end nodes from the hierarchical model structure.

returnType: Character argument specific type object returned. Options are 'names' (returns character vector) and 'ids' (returns numeric graph IDs for model)

returnScalar Componenets: Logical argument specifying whether multivariate nodes should return full node name (i.e. 'x[1:2]') or should break down into scalar componenets (i.e. 'x[1]' and 'x[2]')

Details: Multiple logical input arguments may be used simultaneously. For example, model$getNodeNames(endOnly = TRUE, dataOnly = TRUE) will return all end-level nodes from the model which are designated as 'data'.

`getVarNames(includeLogProb = FALSE, nodes, includeData = TRUE)` Returns the names of all variables in a model, optionally including the logProb variables

Arguments:

logProb: Logical argument specifying whether or not to include the logProb variables. Default is FALSE.

nodes: An optional character vector supplying a subset of nodes for which to extract the variable names and return the unique set of variable names

`isData(nodeNames)` Returns a vector of logical TRUE / FALSE values, corresponding to the 'data' flags of the input node names.

Arguments:

nodeNames: A character vector containing model variable or node names.

Details: The variable or node names specified is expanded into a vector of model node names. A logical vector is returned, indicating whether each model node has been flagged as containing 'data'.

`newModel(data = NULL, inits = NULL, modelName = character(), replicate = FALSE, check = getNimbleOp` Returns a new R model object, with the same model definiton (as defined from the original model code) as the existing model object.

Arguments:

data: A named list specifying data nodes and values, for use in the newly returned model. If not provided, the data argument from the creation of the original R model object will be used.

inits: A named list specifying initial values, for use in the newly returned model. If not provided, the inits argument from the creation of the original R model object will be used.

replicate: Logical specifying whether to repliate all current values and data flags from the current model in the new model. If TRUE, then the data and inits arguments are not used. Default value is FALSE.

check: A logical indicating whether to check the model object for missing or invalid values. Default is given by the NIMBLE option 'checkModel', see help on 'nimbleOptions' for details.

modelName: An optional character string, used to set the internal name of the model object. If provided, this name will propagate throughout the generated C++ code, serving to improve readability.

Details: The newly created model object will be identical to the original model in terms of structure and functionality, but entirely distinct in terms of the internal values.

`resetData()` Resets the 'data' property of ALL model nodes to FALSE. Subsequent to this call, the model will have no nodes flagged as 'data'.

`setData(..., warnAboutMissingNames = TRUE)` Sets the 'data' flag for specified nodes to TRUE, and also sets the value of these nodes to the value provided. This is the exclusive method for specifying 'data' nodes in a model object. When a 'data' argument is provided to 'nimbleModel()', it uses this method to set the data nodes.

Arguments:

...: Arguments may be provided as named elements with numeric values or as character names of model variables. These may be provided in a single list, a single character vector, or as multiple arguments. When a named element with a numeric value is provided, the size and dimension must match the corresponding model variable. This value will be copied to the model variable and any non-NA elements will be marked as data. When a character name is provided, the value of that variable in the model is not changed but any currently non-NA values are marked as data. Examples: setData('x', y = 1:10) will mark both x and y as data and will set the value of y to 1:10. setData(list('x', y = 1:10)) is equivalent. setData(c('x','y')) or setData('x','y') will mark both x and y as data.

Details: If a provided value (or the current value in the model when only a name is specified) contains some NA values, then the model nodes corresponding to these NAs will not have their value set, and will not be designated as 'data'. Only model nodes corresponding to numeric values in the argument list elements will be designated as data. Designating a deterministic model node as 'data' will result in an error. Designating part of a multivariate node as 'data' and part as non-data (NA) will result in an error; multivariate nodes must be entirely data, or entirely non-data.

`setInits(inits)` Sets initial values (or more generally, any named list of value elements) into the model

Arguments:

inits: A named list. The names of list elements must correspond to model variable names. The elements of the list must be of class numeric, with size and dimension each matching the corresponding model variable.

`topologicallySortNodes(nodeNames, returnType = "names")` Sorts the input list of node names according to the topological dependence ordering of the model structure.

Arguments:

nodeNames: A character vector of node names, which is to be topologically sorted. Alternatively can be a numeric vector of graphIDs

returnType: character vector indicating return type. Choices are "names" or "ids"

Details: This function merely reorders its input argument. This may be inportany prior to calls such as simulate(model, nodes) or calculate(model, nodes), to enforce that the operation is performed in topological order.

Author(s)

Daniel Turek

Examples

```
code <- nimbleCode({
    mu ~ dnorm(0, 1)
    x[1] ~ dnorm(mu, 1)
    x[2] ~ dnorm(mu, 1)
})
Rmodel <- nimbleModel(code)
modelVars <- Rmodel$getVarNames()   ## returns 'mu' and 'x'
modelNodes <- Rmodel$getNodeNames()   ## returns 'mu', 'x[1]' and 'x[2]'
Rmodel$resetData()
Rmodel$setData(list(x = c(1.2, NA)))   ## flags only 'x[1]' node as data
Rmodel$isData(c('mu', 'x[1]', 'x[2]'))   ## returns c(FALSE, TRUE, FALSE)
```

---

modelDefClass-class *Class for NIMBLE model definition*

Description

Class for NIMBLE model definition that is not usually needed directly by a user.

Details

See ?modelBaseClass for information about creating NIMBLE BUGS models.

---

modelValues *Create a NIMBLE modelValues Object*

Description

Builds modelValues object from a model values configuration object, which can include a NIMBLE model

Usage

```
modelValues(conf, m = 1)
```

Arguments

conf	An object which includes information for building modelValues. Can either be a NIMBLE model (see help(modelBaseClass)) or the object returned from modelValuesConf
m	The number of rows to create in the modelValues object. Can later be changed with resize

Details

See the User Manual or help(modelValuesBaseClass) for information about manipulating NIMBLE modelValues object returned by this function

Author(s)

NIMBLE development team

Examples

```
#From model object:
code <- nimbleCode({
 a ~ dnorm(0,1)
 for(i in 1:3){
for(j in 1:3)
b[i,j] ~ dnorm(0,1)
}
})
Rmodel <- nimbleModel(code)
Rmodel_mv <- modelValues(Rmodel, m = 2)
#Custom modelValues object:
mvConf <- modelValuesConf(vars = c('x', 'y'),
             types = c('double', 'int'),
             sizes = list(x = 3, y = c(2,2)))
custom_mv <- modelValues(mvConf, m = 2)
custom_mv['y',]
```

---

modelValuesBaseClass-class

Class modelValuesBaseClass

Description

modelValues are NIMBLE containers built to store values from models. They can either be built directly from a model or be custom built via the modelValuesConf function. They consist of rows, where each row can be thought of as a set of values from a model. Like most nimble objects, and unlike most R objects, they are passed by reference instead of by value.

See user manual for more details.

Examples

```
mvConf <- modelValuesConf(vars = c('a', 'b'),
types = c('double', 'double'),
sizes = list(a = 1, b = c(2,2) ) )
mv <- modelValues(mvConf)
as.matrix(mv)
resize(mv, 2)
as.matrix(mv)
mv['a',1] <- 1
mv['a',2] <- 2
mv['b',1] <- matrix(0, nrow = 2, ncol = 2)
mv['b',2] <- matrix(1, nrow = 2, ncol = 2)
mv['a',]
as.matrix(mv)
basicModelCode <- nimbleCode({
a ~ dnorm(0,1)
for(i in 1:4)
b[i] ~ dnorm(0,1)
})
basicModel <- nimbleModel(basicModelCode)
basicMV <- modelValues(basicModel, m = 2) # m sets the number of rows
basicMV['b',]
```

---

modelValuesConf *Create the confs for a custom NIMBLE modelValues object*

Description

Builds an R-based modelValues conf object

Usage

```
modelValuesConf(symTab, className, vars, types, sizes, modelDef = NA,
  where = globalenv())
```

Arguments

symTab	For internal use only
className	For internal use only
vars	A vector of character strings naming each variable in the modelValues object
types	A vector of character strings describing the type of data for the modelValues object. Options include 'double' (for real-valued variables) and 'int'.
sizes	A list in which the named items of the list match the var arguments and each item is a numeric vector of the dimensions
modelDef	For internal use only
where	For internal use only

Details

See the User Manual or `help(modelValuesBaseClass)` and `help(modelValues)` for information

Author(s)

Clifford Anderson-Bergman

Examples

```
#Custom modelValues object:
mvConf <- modelValuesConf(vars = c('x', 'y'),
types = c('double', 'int'),
sizes = list(x = 3, y = c(2,2)))
custom_mv <- modelValues(mvConf, m = 2)
custom_mv['y',]
```

Multinomial *The Multinomial Distribution*

Description

Density and random generation for the multinomial distribution

Usage

```
dmulti(x, size = sum(x), prob, log = FALSE)

rmulti(n = 1, size, prob)
```

Arguments

`x`	vector of values.
`size`	number of trials.
`prob`	vector of probabilities, summing to one, of same length as `x`
`log`	logical; if TRUE, probability density is returned on the log scale.
`n`	number of observations (only `n=1` is handled currently).

Details

See Gelman et al., Appendix A or the BUGS manual for mathematical details.

Value

`dmulti` gives the density and `rmulti` generates random deviates.

Author(s)

Christopher Paciorek

References

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. (2004) *Bayesian Data Analysis*, 2nd ed. Chapman and Hall/CRC.

See Also

Distributions for other standard distributions

Examples

```
size <- 30
probs <- c(1/4, 1/10, 1 - 1/4 - 1/10)
x <- rmulti(1, size, probs)
dmulti(x, size, probs)
```

Multivariate-t *The Multivariate t Distribution*

Description

Density and random generation for the multivariate t distribution, using the Cholesky factor of either the precision matrix (i.e., inverse scale matrix) or the scale matrix.

Usage

```
dmvt_chol(x, mu, cholesky, df, prec_param = TRUE, log = FALSE)

rmvt_chol(n = 1, mu, cholesky, df, prec_param = TRUE)
```

Arguments

x	vector of values.
mu	vector of values giving the location of the distribution.
cholesky	upper-triangular Cholesky factor of either the precision matrix (i.e., inverse scale matrix) (when prec_param is TRUE) or scale matrix (otherwise).
df	degrees of freedom.
prec_param	logical; if TRUE the Cholesky factor is that of the precision matrix; otherwise, of the scale matrix.
log	logical; if TRUE, probability density is returned on the log scale.
n	number of observations (only n=1 is handled currently).

Details

See Gelman et al., Appendix A or the BUGS manual for mathematical details. The 'precision' matrix as used here is defined as the inverse of the scale matrix, Σ^{-1}, given in Gelman et al.

Value

dmvt_chol gives the density and rmvt_chol generates random deviates.

Author(s)

Peter Sujan

References

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. (2004) *Bayesian Data Analysis*, 2nd ed. Chapman and Hall/CRC.

See Also

Distributions for other standard distributions

Examples

```
mu <- c(-10, 0, 10)
scalemat <- matrix(c(1, .9, .3, .9, 1, -0.1, .3, -0.1, 1), 3)
ch <- chol(scalemat)
x <- rmvt_chol(1, mu, ch, df = 1, prec_param = FALSE)
dmvt_chol(x, mu, ch, df = 1, prec_param = FALSE)
```

MultivariateNormal *The Multivariate Normal Distribution*

Description

Density and random generation for the multivariate normal distribution, using the Cholesky factor of either the precision matrix or the covariance matrix.

Usage

```
dmnorm_chol(x, mean, cholesky, prec_param = TRUE, log = FALSE)

rmnorm_chol(n = 1, mean, cholesky, prec_param = TRUE)
```

Arguments

x	vector of values.
mean	vector of values giving the mean of the distribution.
cholesky	upper-triangular Cholesky factor of either the precision matrix (when prec_param is TRUE) or covariance matrix (otherwise).
prec_param	logical; if TRUE the Cholesky factor is that of the precision matrix; otherwise, of the covariance matrix.
log	logical; if TRUE, probability density is returned on the log scale.
n	number of observations (only n=1 is handled currently).

Details

See Gelman et al., Appendix A or the BUGS manual for mathematical details. The rate matrix as used here is defined as the inverse of the scale matrix, S^{-1}, given in Gelman et al.

Value

`dmnorm_chol` gives the density and `rmnorm_chol` generates random deviates.

Author(s)

Christopher Paciorek

References

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. (2004) *Bayesian Data Analysis*, 2nd ed. Chapman and Hall/CRC.

See Also

Distributions for other standard distributions

Examples

```
mean <- c(-10, 0, 10)
covmat <- matrix(c(1, .9, .3, .9, 1, -0.1, .3, -0.1, 1), 3)
ch <- chol(covmat)
x <- rmnorm_chol(1, mean, ch, prec_param = FALSE)
dmnorm_chol(x, mean, ch, prec_param = FALSE)
```

---

nfMethod *access (call) a member function of a nimbleFunction*

Description

Internal function for accessing a member function (method) of a nimbleFunction. Normally a user will write `nf$method(x)` instead of `nfMethod(nf, method)(x)`.

Usage

```
nfMethod(nf, methodName)
```

Arguments

`nf`	a specialized nimbleFunction, i.e. one that has already had setup parameters processed
`methodName`	a character string giving the name of the member function to call

Details

nimbleFunctions have a default member function called `run`, and may have other member functions provided via the `methods` argument to `nimbleFunction`. As an internal step, the NIMBLE compiler turns `nf$method(x)` into `nfMethod(nf, method)(x)`, but a NIMBLE user or programmer would not normally need to use `nfMethod` directly.

Value

a function that can be called.

Author(s)

NIMBLE development team

---

nfVar *Access or set a member variable of a nimbleFunction*

Description

Access or set a member variable of a specialized nimbleFunction, i.e. a variable passed to or created during the `setup` function that is used in run code or preserved by `setupOutputs`. Works in R for any variable and in NIMBLE for numeric variables.

Usage

```
nfVar(nf, varName)

nfVar(nf, varName) <- value
```

Arguments

`nf`	a specialized nimbleFunction, i.e. a function returned by executing a function returned from `nimbleFunction` with `setup` arguments
`varName`	a character string naming a variable in the `setup` function.
`value`	value to set the variable to.

Details

Internal way to access or set a member variable of a nimbleFunction created during `setup`. Normally in NIMBLE code you would use `nf$var` instead of `nfVar(nf, var)`.

When `nimbleFunction` is called and a `setup` function is provided, then `nimbleFunction` returns a function. That function is a generator that should be called with arguments to the `setup` function and returns another function with `run` and possibly other member functions. The member functions can use objects created or passed to `setup`. During internal processing, the NIMBLE compiler turns some cases of `nf$var` into `nfVar(nf, var)`. These provide direct access to setup variables (member data). `nfVar` is not typically called by a NIMBLE user or programmer.

For internal access to methods of `nf`, see `nfMethod`.

For more information, see `?nimbleFunction` and the NIMBLE User Manual.

Value

whatever varName is in the nimbleFunction nf.

Author(s)

NIMBLE development team

Examples

```
nfGen1 <- nimbleFunction(
    setup = function(A) {
    B <- matrix(rnorm(4), nrow = 2)
    setupOutputs(B) ## preserves B even though it is not used in run-code
   },
   run = function() {
      print('This is A', A, '\n')
})

nfGen2 <- nimbleFunction(
  setup = function() {
    nf1 <- nfGen1(1000)
  },
  run = function() {
      print('accessing A:', nfVar(nf1, 'A'))
      nfVar(nf1, 'B')[2,2] <<- -1000
      print('accessing B:', nfVar(nf1, 'B'))
   })

nf2 <- nfGen2()
nf2$run()
Cnf2 <- compileNimble(nf2)
Cnf2$run()
```

---

nimArray *Creates a array object of arbitrary dimension for use in NIMBLE DSL functions*

Description

In a `nimbleFunction`, `array` is identical to `nimArray`

Usage

```
nimArray(value = 0, dim = c(1, 1), init = TRUE, type = "double")
```

Arguments

value	the initial value for each element of the array (default = 0)
dim	a vector specifying the dimensionality and sizes of the array, provided as c(size1, ...) (default = c(1, 1))
init	logical, whether to initialize elements of the matrix (default = TRUE)
type	character representing the data type, i.e. 'double' or 'integer' (default = 'double')

Details

See the User Manual for usage examples.

When used in a nimbleFunction (in run or other member function), array is a synonym for nimArray. When used with only the first two arguments, this behaves similarly to R's array function. NIMBLE provides additional arguments to control the initialization value, whether or not initialization will occur, and the type of scalar elements. Using init=FALSE when initialization is not necessary can make compiled nimbleFunctions a bit faster.

Author(s)

Daniel Turek

See Also

numeric integer matrix

---

nimble-internal *Functions and Classes Internal to NIMBLE*

Description

Functions and classes used internally in NIMBLE and not expected to be called directly by users. Some functions and classes not intended for direct use are documented and/or exported because they are used within Reference Class methods for classes programmatically generated by NIMBLE.

Author(s)

NIMBLE Development Team

nimble-math *Mathematical functions for BUGS and nimbleFunction programming*

Description

Mathematical functions for use in BUGS code and in nimbleFunction programming (i.e., nimbleFunction run code). See Chapter 5 of the User Manual for more details.

Author(s)

NIMBLE Development Team

nimbleCode *Turn BUGS model code into an object for use in* nimbleModel *or* readBUGSmodel

Description

Simply keeps model code as an R call object, the form needed by nimbleModel and optionally usable by readBUGSmodel

Usage

```
nimbleCode(code)
```

Arguments

code	expression providing the code for the model

Details

It is equivalent to use the R function quote. nimbleCode is simply provided as a more readable alternative for NIMBLE users not familiar with quote.

Author(s)

Daniel Turek

Examples

```
code <- nimbleCode({
    x ~ dnorm(mu, sd = 1)
    mu ~ dnorm(0, sd = prior_sd)
})
```

`nimbleFunction` *create a nimbleFunction*

Description

create a nimbleFunction from a setup function, run function, possibly other methods, and possibly inheritance via `contains`

Usage

```
nimbleFunction(setup = NULL, run = function() { }, methods = list(),
  globalSetup = NULL, contains = NULL, name = NA,
  where = getNimbleFunctionEnvironment())
```

Arguments

`setup`	An optional R function definition for setup processing.
`run`	An optional NIMBLE function definition the executes the primary job of the nimbleFunction
`methods`	An optional named list of NIMBLE function definitions for other class methods.
`globalSetup`	For internal use only
`contains`	An optional object returned from nimbleFunctionVirtual that defines arguments and returnTypes for `run` and/or methods, to which the current nimbleFunction must conform
`name`	An optional name used internally, for example in generated C++ code. Usually this is left blank and NIMBLE provides a name.
`where`	An optional `where` argument passed to `setRefClass` for where the reference class definition generated for this nimbleFunction will be stored. This is needed due to R package namespace issues but should never need to be provided by a user.

Details

This is the main function for defining nimbleFunctions. A lot of information is provided in the NIMBLE User Manual, so only a brief summary will be given here.

If a `setup` function is provided, then `nimbleFunction` returns a generator: a function that when called with arguments for the setup function will execute that function and return a specialized nimbleFunction. The `run` and other methods can be called using `$` like in other R classes, e.g. `nf$run()`. The methods can use objects that were created in or passed to the `setup` function.

If no `setup` function is provided, then `nimbleFunction` returns a function that executes the `run` function. It is not a generator in this case, and no other `methods` can be provided.

If one wants a generator but does not need any setup arguments or code, `setup = TRUE` can be used.

See the NIMBLE User Manual for examples.

For more information about the `contains` argument, see the section on nimbleFunctionLists.

Author(s)

NIMBLE development team

`nimbleFunctionBase-class` *Class* `nimbleFunctionBase`

Description

Classes used internally in NIMBLE and not expected to be called directly by users.

`nimbleFunctionList-class` *Create a list of nimbleFunctions*

Description

Create an empty list of nimbleFunctions that all will inherit from a base class.

Details

See the User Manual for information about creating and populating a `nimbleFunctionList`.

Author(s)

NIMBLE development team

`nimbleFunctionVirtual` *create a virtual nimbleFunction, a base class for other nimbleFunctions*

Description

define argument types and returnType for the `run` function and any `methods`, to be used in the `contains` argument of `nimbleFunction`

Usage

```
nimbleFunctionVirtual(contains = NULL, run = function() { },
  methods = list(), name = NA)
```

Arguments

contains	Not yet functional
run	A NIMBLE function that will only be used to inspect its argument types and returnType.
methods	An optional named list of NIMBLE functions that will also only be used for inspecting argument types and returnTypes.
name	An optional name used internally by the NIMBLE compiled. This is usually omitted and NIMBLE provides one.

Details

See the NIMBLE User Manual section on nimbleFunctionLists for explanation of how to use a virtual nimbleFunction.

Value

An object that can be passed as the `contains` argument to `nimbleFunction` or as the argument to `nimbleFunctionList`

Author(s)

NIMBLE development team

See Also

nimbleFunction

---

nimbleModel *Create a NIMBLE model from BUGS code*

Description

processes BUGS model code and optional constants, data, and initial values. Returns a NIMBLE model or model definition.

Usage

```
nimbleModel(code, constants = list(), data = list(), inits = list(),
  dimensions = list(), returnDef = FALSE, where = globalenv(),
  debug = FALSE, check = getNimbleOption("checkModel"), calculate = TRUE,
  name)
```

Arguments

code	code for the model in the form returned by nimbleCode or (equivalently) quote
constants	named list of constants in the model. Constants cannot be subsequently modified. For compatibility with JAGS and BUGS, one can include data values with constants and nimbleModel will automatically distinguish them based on what appears on the left-hand side of expressions in code.
data	named list of values for the data nodes. Data values can be subsequently modified. Providing this argument also flags nodes as having data for purposes of algorithms that inspect model structure. Values that are NA will not be flagged as data.
inits	named list of starting values for model variables. Unlike JAGS, should only be a single list, not a list of lists.
dimensions	named list of dimensions for variables. Only needed for variables used with empty indices in model code that are not provided in constants or data.
returnDef	logical indicating whether the model should be returned (FALSE) or just the model definition (TRUE).
where	argument passed to setRefClass, indicating the environment in which the reference class definitions generated for the model and its modelValues should be created. This is needed for managing package namespace issues during package loading and does not normally need to be provided by a user.
debug	logical indicating whether to put the user in a browser for debugging. Intended for developer use.
check	logical indicating whether to check the model object for missing or invalid values. Default is given by the NIMBLE option 'checkModel', see help on nimbleOptions for details.
calculate	logical indicating whether to run calculate on the model after building it; this will calculate all deterministic nodes and logProbability values given the current state of all nodes. Default is TRUE. For large models, one might want to disable this, but note that deterministic nodes, including nodes introduced into the model by NIMBLE, may be NA.
name	optional character vector giving a name of the model for internal use. If omitted, a name will be provided.

Details

See the User Manual or help(modelBaseClass) for information about manipulating NIMBLE models created by nimbleModel, including methods that operate on models, such as getDependencies.

The user may need to provide dimensions for certain variables as in some cases NIMBLE cannot automatically determine the dimensions and sizes of variables. See the User Manual for more information.

As noted above, one may lump together constants and data (as part of the constants argument (unlike R interfaces to JAGS and BUGS where they are provided as the data argument). One may not provide lumped constants and data as the data argument.

For variables that are a mixture of data nodes and non-data nodes, any values passed in via `inits` for components of the variable that are data will be ignored. All data values should be passed in through `data` (or `constants` as just discussed).

Author(s)

NIMBLE development team

Examples

```
code <- nimbleCode({
    x ~ dnorm(mu, sd = 1)
    mu ~ dnorm(0, sd = prior_sd)
})
constants = list(prior_sd = 1)
data = list(x = 4)
Rmodel <- nimbleModel(code, constants = constants, data = data)
```

---

nimbleOptions *NIMBLE Options Settings*

Description

Allow the user to set and examine a variety of global _options_ that affect the way in which NIMBLE operates

Usage

```
nimbleOptions(...)
```

Arguments

`...`	any options to be defined as one or more 'name = value' pairs. Options can also be passed by giving a single unnamed argument that is a named list.

Details

`nimbleOptions` mimics `options`. Invoking `nimbleOptions()` with no arguments returns a list with the current values of the options. To access the value of a single option, one should use `getNimbleOption()`.

Value

When invoked with no arguments, a list with the current values of all options.

Author(s)

Christopher Paciorek

Examples

```
nimbleOptions(verifyConjugatePosteriors = FALSE)
```

`nimCat` *cat function for use in nimbleFunctions*

Description

cat function for use in nimbleFunctions

Usage

```
nimCat(...)
```

Arguments

`...`	an arbitrary set of arguments that will be printed in sequence.

Details

`cat` in nimbleFunction run-code imitates the R function `cat`. It prints its arguments in order. No newline is inserted, so include `"\n"` if one is desired.

When an uncompiled nimbleFunction is executed, R's `cat` is used. In a compiled nimbleFunction, a C++ output stream is used that will generally format output similarly to R's `cat`. Non-scalar numeric objects can be included, although their output will be formatted slightly different in uncompiled and compiled nimbleFunctions.

In nimbleFunction run-time code, `cat` is identical to `print` except the latter appends a newline at the end.

`nimCat` is the same as `cat`, and the latter is converted to the former when a nimbleFunction is defined.

See Also

`print`

Examples

```
ans <- matrix(1:4, nrow = 2) ## R code, not NIMBLE code
nimCat('Answer is ', ans) ## would work in R or NIMBLE
```

nimCopy *Copying function for NIMBLE*

Description

Copies values from a NIMBLE model or modelValues object to another NIMBLE model or modelValues. Work in R and NIMBLE. The NIMBLE keyword copy is identical to nimCopy

Usage

```
nimCopy(from, to, nodes = NULL, nodesTo = NULL, row = NA, rowTo = NA,
  logProb = FALSE)
```

Arguments

from	Either a NIMBLE model or modelValues object
to	Either a NIMBLE model or modelValues object
nodes	The nodes of object from which will be copied from
nodesTo	The nodes of object to which will be copied to. If nodesTo == NA, will automatically be set to nodes
row	If from is a modelValues, the row which will be copied from
rowTo	If to is a modelValues, the row which will be copied to. If rowTo == NA, will automatically be set to row
logProb	A logical value indicating whether the log probabilities of the given nodes should also be copied (i.e. if nodes = 'x' and logProb = TRUE, then both 'x' and 'logProb_x' will be copied)

Details

See the User Manual for more details

Author(s)

Clifford Anderson-Bergman

Examples

```
# Building model and modelValues object
simpleModelCode <- nimbleCode({
for(i in 1:100)
x[i] ~ dnorm(0,1)
})
rModel <- nimbleModel(simpleModelCode)
rModelValues <- modelValues(rModel)

#Setting model nodes
```

```
rModel$x <- rnorm(100)
#Using nimCopy in R.
nimCopy(from = rModel, to = rModelValues, nodes = 'x', rowTo = 1)

#Use of nimCopy in a simple nimbleFunction
cCopyGen <- nimbleFunction(
setup = function(model, modelValues, nodeNames){},
run = function(){
nimCopy(from = model, to = modelValues, nodes = nodeNames, rowTo = 1)
}
)

rCopy <- cCopyGen(rModel, rModelValues, 'x')
## Not run:
cModel <- compileNimble(rModel)
cCopy <- compileNimble(rCopy, project = rModel)
cModel[['x']] <- rnorm(100)

cCopy$run() ## execute the copy with the compiled function

## End(Not run)
```

`nimDim` *return sizes of an object whether it is a vector, matrix or array*

Description

R's regular `dim` function returns NULL for a vector. It is useful to have this function that treats a vector similarly to a matrix or array. Works in R and NIMBLE. In NIMBLE `dim` is identical to `nimDim`, not to R's `dim`

Usage

```
nimDim(obj)
```

Arguments

`obj`	objects for which the sizes are requested

Value

a vector of sizes in each dimension

Author(s)

NIMBLE development team

Examples

```
x <- rnorm(4)
dim(x)
nimDim(x)
y <- matrix(x, nrow = 2)
dim(y)
nimDim(y)
```

`nimInteger` *Creates an integer vector for use in NIMBLE DSL functions*

Description

In a `nimbleFunction`, `integer` is identical to `nimInteger`

Usage

```
nimInteger(length = 0, value = 0, init = TRUE)
```

Arguments

`length`	the length of the vector (default = 0)
`value`	the initial value for each element of the vector (default = 0L). Only used if `init` is `TRUE`.
`init`	logical, whether to initialize elements of the vector (default = TRUE).

Details

See the User Manual for usage examples.

When used in a `nimbleFunction` (in `run` or other member function), `integer` is a synonym for `nimInteger`. When used with only the `length` argument, this behaves similarly to R's `integer` function. NIMBLE provides additional arguments to control the initialization value and whether or not initialization will occur. Using `init=FALSE` when initialization is not necessary can make compiled nimbleFunctions a bit faster.

Author(s)

Daniel Turek

See Also

`numeric` `matrix` `array`

`nimMatrix` *Creates a matrix object for use in NIMBLE DSL functions*

Description

In a `nimbleFunction`, `matrix` is identical to `nimMatrix`

Usage

```
nimMatrix(value = 0, nrow = 1, ncol = 1, init = TRUE, type = "double")
```

Arguments

`value`	the initial value for each element of the matrix (default = 0)
`nrow`	the number of rows in the matrix (default = 1)
`ncol`	the number of columns in the matrix (default = 1)
`init`	logical, whether to initialize elements of the matrix (default = TRUE)
`type`	character representing the data type, i.e. 'double' or 'integer' (default = 'double')

Details

See the User Manual for usage examples.

When used in a `nimbleFunction` (in `run` or other member function), `matrix` is a synonym for `nimMatrix`. When used with only the first three arguments, this behaves similarly to R's `matrix` function. NIMBLE provides additional arguments to control the initialization value, whether or not initialization will occur, and the type of scalar elements. Using `init=FALSE` when initialization is not necessary can make compiled nimbleFunctions a bit faster.

Author(s)

Daniel Turek

See Also

`numeric` `integer` `array`

nimNumeric *Creates a numeric vector for use in NIMBLE DSL functions*

Description

In a nimbleFunction, numeric is identical to nimNumeric

Usage

```
nimNumeric(length = 0, value = 0, init = TRUE)
```

Arguments

length	the length of the vector (default = 0)
value	the initial value for each element of the vector (default = 0)
init	logical, whether to initialize elements of the vector (default = TRUE)

Details

See the User Manual for usage examples.

When used in a nimbleFunction (in run or other member function), numeric is a synonym for nimNumeric. When used with only the length argument, this behaves similarly to R's integer function. NIMBLE provides additional arguments to control the initialization value and whether or not initialization will occur. Using init=FALSE when initialization is not necessary can make compiled nimbleFunctions a bit faster.

Author(s)

Daniel Turek

See Also

integer matrix array

nimPrint *print function for use in nimbleFunctions*

Description

print function for use in nimbleFunctions

Usage

```
nimPrint(...)
```

Arguments

...	an abitrary set of arguments that will be printed in sequence.

Details

The keyword `print` in nimbleFunction run-time code will be automatically turned into `nimPrint`. This is a function that prints its arguments in order using `cat` in R, or using `std::cout` in C++ code generated by compiling nimbleFunctions. Non-scalar numeric objects can be included, although their output will be formatted slightly different in uncompiled and compiled nimbleFunctions.

See Also

`cat`

Examples

```
ans <- matrix(1:4, nrow = 2) ## R code, not NIMBLE code
nimPrint('Answer is ', ans) ## would work in R or NIMBLE
```

---

nimStop — *Halt execution of a nimbleFunction function method. Part of the NIMBLE language*

Description

Halt execution of a nimbleFunction function method. Part of the NIMBLE language

Usage

```
nimStop(msg)
```

Arguments

msg	Character object to be output as an error message

Details

The NIMBLE stop is similar to the native R stop, but it takes only one argument, the error message to be output. During uncompiled NIMBLE execution, nimStop simply calls R's stop funtion. During compiled execution it calls the error function from the R headers. stop is an alias for nimStop in the NIMBLE language

Author(s)

Perry de Valpine

nodeFunctions *calculate, calculateDiff, simulate, or get the current log probabilities (densities) a set of nodes in a NIMBLE model*

Description

calculate, calculateDiff, simulate, or get the current log probabilities (densities) of one or more nodes of a NIMBLE model and (for calculate and getLogProb) return the sum of their log probabilities (or densities). Part of R and NIMBLE.

Usage

```
calculate(model, nodes, nodeFxnVector, nodeFunctionIndex)

calculateDiff(model, nodes, nodeFxnVector, nodeFunctionIndex)

getLogProb(model, nodes, nodeFxnVector, nodeFunctionIndex)

simulate(model, nodes, includeData = FALSE, nodeFxnVector, nodeFunctionIndex)
```

Arguments

model	A NIMBLE model, either the compiled or uncompiled version
nodes	A character vector of node names, with index blocks allowed, such as 'x', 'y[2]', or 'z[1:3, 2:4]'
nodeFxnVector	An optional vector of nodeFunctions on which to operate, in lieu of model and nodes
nodeFunctionIndex	For internal NIMBLE use only
includeData	A logical argument specifying whether data nodes should be simulated into (only relevant for simulate)

Details

These functions expands the nodes and then process them in the model in the order provided. Expanding nodes means turning 'y[1:2]' into c('y[1]','y[2]') if y is a vector of scalar nodes. Calculation is defined for a stochastic node as executing the log probability (density) calculation and for a deterministic node as calculating whatever function was provided on the right-hand side of the model declaration.

Difference calculation (calculateDiff) executes the operation(s) on the model as calculate, but it returns the sum of the difference between the new log probabilities and the previous ones.

Simulation is defined for a stochastic node as drawing a random value from its distribution, and for deterministic node as equivalent to calculate.

getLogProb simply collects the sum of the log probabilities of nodes if they are known to have already been calculated.

These functions can be used from R or in NIMBLE run-time functions that will be compiled. When executed in R (including when an uncompiled nimbleFunction is executed), they can be slow because the nodes are expanded each time. When compiled in NIMBLE, the nodes are expanded only once during compilation, so execution will be much faster.

It is common to want the nodes to be provided in topologically sorted order, so that they will be calculated or simulated following the order of the model graph. Functions such as model$getDependencies(nodes, ...) return nodes in topologically sorted order. They can be directly sorted by model$topologicallySortNodes(nodes), but if so it is a good idea to expand names first by model$topologicallySortNodes(model$expandNodeNames(nodes))

Value

calculate and getLogProb return the sum of the log probabilities (densities) of the calculated nodes, with a contribution of 0 from any deterministic nodes

calculateDiff returns the sum of the difference between the new and old log probabilities (densities) of the calculated nodes, with a contribution of 0 from any deterministic nodes.

simulate returns NULL.

Author(s)

NIMBLE development team

`rankSample` *Generates a weighted sample (with replacement) of ranks*

Description

Takes a set of non-negative `weights` (do not need to sum to 1) and returns a sample with `size` elements of the integers `1:length(weights)`, where the probability of being sampled is proportional to the value of `weights`. An important note is that the output vector will be sorted in ascending order. Also, right now it works slightly odd syntax (see example below). Later releases of NIMBLE will contain more natural syntax.

Usage

```
rankSample(weights, size, output, silent = FALSE)
```

Arguments

`weights`	A vector of numeric weights. Does not need to sum to 1, but must be non-negative
`size`	Size of sample
`output`	An R object into which the values will be placed. See example below for proper use
`silent`	Logical indicating whether to suppress logging information

Details

If invalid weights provided (i.e. negative weights or weights sum to 1), sets output = rep(1, size) and prints warning. `rankSample` can be used inside nimble functions.

`rankSample` first samples from the joint distribution `size` uniform(0,1) distributions by conditionally sampling from the rank statistics. This leads to a sorted sample of uniform(0,1)'s. Then, a cdf vector is constructed from weights. Because the sample of uniforms is sorted, `rankSample` walks down the cdf in linear time and fills out the sample.

Author(s)

Clifford Anderson-Bergman

Examples

```
set.seed(1)
sampInts = NA #sampled integers will be placed in sampInts
rankSample(weights = c(1, 1, 2), size = 10, sampInts)
sampInts
# [1] 1 1 2 2 2 3 3 3 3 3
rankSample(weights = c(1, 1, 2), size = 10000, sampInts)
table(sampInts)
#sampInts
#   1    2    3
#2429 2498 5073

#Used in a nimbleFunction
sampGen <- nimbleFunction(setup = function(){
x = 1:2
},
run = function(weights = double(1), k = integer() ){
rankSample(weights, k, x)
returnType(integer(1))
return(x)
})
rSamp <- sampGen()
cSamp <- compileNimble(rSamp)
cSamp$run(1:4, 5)
#[1] 1 1 4 4 4
```

---

readBUGSmodel *Create a NIMBLE BUGS model from a variety of input formats, including BUGS model files*

Description

`readBUGSmodel` processes inputs providing the model and values for constants, data, initial values of the model in a variety of forms, returning a NIMBLE BUGS R model

Usage

```
readBUGSmodel(model, data = NULL, inits = NULL, dir = NULL,
  useInits = TRUE, debug = FALSE, returnComponents = FALSE,
  check = getNimbleOption("checkModel"), calculate = TRUE)
```

Arguments

model	one of (1) a character string giving the file name containing the BUGS model code, with relative or absolute path, (2) an R function whose body is the BUGS model code, or (3) the output of nimbleCode. If a file name, the file can contain a 'var' block and 'data' block in the manner of the JAGS versions of the BUGS examples but should not contain references to other input data files nor a const block. The '.bug' or '.txt' extension can be excluded.
data	(optional) (1) character string giving the file name for an R file providing the input constants and data as R code [assigning individual objects or as a named list], with relative or absolute path, or (2) a named list providing the input constants and data. If neither is provided, the function will look for a file named 'name_of_model-data' including extensions .R, .r, or .txt.
inits	(optional) (1) character string giving the file name for an R file providing starting values as R code [assigning individual objects or as a named list], with relative or absolute path, or (2) a named list providing the starting values. Unlike JAGS, this should provide a single set of starting values, and therefore if provided as a list should be a simple list and not a list of lists.
dir	(optional) character string giving the directory where the (optional) files are located
useInits	boolean indicating whether to set the initial values, either based on inits or by finding the '-inits' file corresponding to the input model file
debug	logical indicating whether to put the user in a browser for debugging when readBUGSmodel calls nimbleModel. Intended for developer use.
returnComponents	logical indicating whether to return pieces of model object without building the model. Default is FALSE.
check	logical indicating whether to check the model object for missing or invalid values. Default is given by the NIMBLE option 'checkModel', see help on nimbleOptions for details.
calculate	logical indicating whether to run calculate on the model after building it; this will calculate all deterministic nodes and logProbability values given the current state of all nodes. Default is TRUE. For large models, one might want to disable this, but note that deterministic nodes, including nodes introduced into the model by NIMBLE, may be NA.

Details

Note that readBUGSmodel should handle most common ways of providing information on a model as used in BUGS and JAGS but does not handle input model files that refer to additional files containing data. Please see the BUGS examples provided with JAGS (http://sourceforge.net/

projects/mcmc-jags/files/Examples/) for examples of supported formats. Also, readBUGSmodel takes both constants and data via the 'data' argument, unlike nimbleModel, in which these are distinguished. The reason for allowing both to be given via 'data' is for backwards compatibility with the BUGS examples, in which constants and data are not distinguished.

Value

returns a NIMBLE BUGS R model

Author(s)

Christopher Paciorek

Examples

```
code <- nimbleCode({
    x ~ dnorm(mu, sd = 1)
    mu ~ dnorm(0, sd = prior_sd)
})
data = list(prior_sd = 1, x = 4)
model <- readBUGSmodel(code, data = data, inits = list(mu = 0))
model$x
model[['mu']]
model$calculate('x')
```

---

registerDistributions *Add user-supplied distributions for use in NIMBLE BUGS models*

Description

Register distributional information so that NIMBLE can process user-supplied distributions in BUGS model code

Usage

```
registerDistributions(distributionsInputList)
```

Arguments

distributionsInputList	a list of lists in the form of that shown in distributionsInputList with each list having required field BUGSdist and optional fields Rdist, altParams, discrete, pqAvail, types. See Details for more information. If only one distribution is supplied it may be a list rather than a list containing a list.

Details

- `BUGSdist` a character string in the form of the density name (starting with 'd') followed by the names of the parameters in parentheses. When alternative parameterizations are given in `Rdist`, this should be an exhaustive list of the unique parameter names from all possible parameterizations, with the default parameters specified first.
- `Rdist` an optional character vector with one or more alternative specifications of the density; each alternative specification can be an alternative name for the density, a different ordering of the parameters, different parameter name(s), or an alternative parameterization. In the latter case, the character string in parentheses should provide a given reparameterization as comma-separated name = value pairs, one for each default parameter, where name is the name of the default parameter and value is a mathematical expression relating the default parameter to the alternative parameters or other default parameters. The default parameters should correspond to the input arguments of the nimbleFunctions provided as the density and random generation functions. The mathematical expression can use any of the math functions allowed in NIMBLE (see the User Manual) as well as user-supplied nimbleFunctions without setup code. The names of your nimbleFunctions for the distribution functions must match the function name in the `Rdist` entry (or if missing, the function name in the `BUGSdist` entry
- `discrete` a optional logical indicating if the distribution is that of a discrete random variable. If not supplied, distribution is assumed to be for a continuous random variable.
- `pqAvail` an optional logical indicating if distribution (CDF) and quantile (inverse CDF) functions are provided as nimbleFunctions. These are required for one to be able to use truncated versions of the distribution. Only applicable for univariate distributions. If not supplied, assumed to be FALSE.
- `altParams` a character vector of comma-separated 'name = value' pairs that provide the mathematical expressions relating non-canonical parameters to canonical parameters (canonical parameters are those passed as arguments to your distribution functions). These inverse functions are used for MCMC conjugacy calculations when a conjugate relationship is expressed in terms of non-default parameters (such as the precision for normal-normal conjugacy). If not supplied, the system will still function but with a possible loss of efficiency in certain algorithms.
- `types` a character vector of comma-separated 'name = input' pairs indicating the type and dimension of the random variable and parameters (including default and alternative parameters). 'input' should take the form 'double(d)' or 'integer(d)', where 'd' is 0 for scalars, 1 for vectors, 2 for matrices. Note that since NIMBLE uses doubles for numerical calculations and the default type is `double(0)`, one should generally use 'double' and one need only specify the type for non-scalars. 'name' should be either 'value' to indicate the random variable itself or the parameter name to indicate a given parameter.
- `range` a vector of two values giving the range of the distribution for possible use in future algorithms (not used currently). When the lower or upper limit involves a strict inequality (e.g., $x>0$), you should simply treat it as a non-strict inequality ($x>=0$, and set the lower value to 0). Also we do not handle ranges that are functions of parameters, so simply use the smallest/largest possible values given the possible parameter values. If not supplied this is taken to be `(-Inf, Inf)`.

Author(s)

Christopher Paciorek

Examples

```
dmyexp <- nimbleFunction(
   run = function(x = double(0), rate = double(0), log_value = integer(0)) {
       returnType(double(0))
       logProb <- log(rate) - x*rate
       if(log_value) {
           return(logProb)
       } else {
           return(exp(logProb))
       }
   })
rmyexp <- nimbleFunction(
   run = function(n = integer(0), rate = double(0)) {
       returnType(double(0))
       if(n != 1) nimPrint("rmyexp only allows n = 1; using n = 1.")
       dev <- runif(1, 0, 1)
       return(-log(1-dev) / rate)
   }
   )
registerDistributions(list(
    dmyexp = list(
               BUGSdist = "dmyexp(rate, scale)",
               Rdist = "dmyexp(rate = 1/scale)",
               altParams = "scale = 1/rate",
               pqAvail = FALSE)))
code <- nimbleCode({
    y ~ dmyexp(rate = r)
    r ~ dunif(0, 100)
})
m <- nimbleModel(code, inits = list(r = 1), data = list(y = 2))
calculate(m, 'y')
m$r <- 2
calculate(m, 'y')
m$resetData()
simulate(m, 'y')
m$y
```

---

rename_MCMC_comparison_method

Rename a method in an object returned by compareMCMCs

Description

Switches one label for a new one in the timing, efficiency, summary, and samples (if present) elements of a compareMCMCs result.

Usage

```
rename_MCMC_comparison_method(oldname, newname, comparison)
```

Arguments

oldname	(character) Existing name for the method (which would have been determined from inputs to compareMCMCs).
newname	(character) New name for the method
comparison	An object returned by compareMCMCs.

Details

One of the main purposes of renaming a method is that the method names are used in html results generated by make_MCMC_comparison_pages.

Value

An object in the same format as comparison.

See Also

compareMCMCs, codecombine_MCMC_comparison_results, make_MCMC_comparison_pages, reshape_comparison_resul

---

reshape_comparison_results

Convert comparison results to a more general format

Description

Useful for making new kinds of figures or other needs

Usage

```
reshape_comparison_results(oneComparisonResult, includeVars = TRUE,
  includeEfficiency = TRUE, includeTiming = TRUE)
```

Arguments

oneComparisonResult	An object returned by compareMCMCs (or combine_MCMC_comparison_results or rename_MCMC_comparison_method).
includeVars	(default TRUE): whether to include the summary elements for each variable
includeEfficiency	(default TRUE): whether to include the efficiency for each variable
includeTiming	(default TRUE): whether to include the timing for each variable (which is the same for all variables from the same MCMC method)

Details

This is used internally by make_MCMC_comparison_pages but could also be useful to users who want to do their own thing with results from compareMCMCs.

Value

A data frame with the content from oneComparisonResult reshaped

---

`resize` *Resizes a modelValues object*

Description

Adds or removes rows to a modelValues object. If rows are added to a modelValues object, the default values are `NA`. Works in both R and NIMBLE.

Usage

```
resize(container, k)
```

Arguments

`container`	modelValues object
`k`	number of rows that modelValues is set to

Details

See the User Manual or `help(modelValuesBaseClass)` for infomation about modelValues objects

Author(s)

Clifford Anderson-Bergman

Examples

```
mvConf <- modelValuesConf(vars = c('a', 'b'),
             types = c('double', 'double'),
             sizes = list(a = 1, b = c(2,2) ) )
mv <- modelValues(mvConf)
as.matrix(mv)
resize(mv, 3)
as.matrix(mv)
```

Rmatrix2mvOneVar *Set values of one variable of a modelValues object from an R matrix*

Description

Normally a modelValues object is accessed one "row" at a time. This function allows all rows for one variable to set from a matrix with one dimension more than the variable to be set.

Usage

```
Rmatrix2mvOneVar(mat, mv, varName, k)
```

Arguments

mat	Input matrix
mv	modelValues object to be modified.
varName	Character string giving the name of the variable on mv to be set
k	Number of rows to use

Details

This function may be deprecated in the future when a more natural system for interacting with modelValues objects is developed.

RmodelBaseClass-class *Class* RmodelBaseClass

Description

Classes used internally in NIMBLE and not expected to be called directly by users.

run.time *Time execution of NIMBLE code*

Description

Time execution of NIMBLE code

Usage

```
run.time(code)
```

Arguments

code	code to be timed

Details

Function for use in nimbleFunction run code; when nimbleFunctions are run in R, this simply wraps `system.time`.

Author(s)

NIMBLE Development Team

---

runMCMC *Run one or more chains of an MCMC algorithm and extract samples*

Description

Takes as input an MCMC algorithm (ideally a compiled one for speed) and runs the MCMC with one or more chains, automatically extracting the samples.

Usage

```
runMCMC(mcmc, niter = 10000, nburnin = 0, nchains = 1, inits,
  setSeed = FALSE, progressBar = TRUE, silent = FALSE,
  returnCodaMCMC = FALSE)
```

Arguments

mcmc	A NIMBLE MCMC algorithm. See details.
niter	Number of iterations to run each MCMC chain (default = 10000).
nburnin	Number of initial samples to discard from each MCMC chain (default = 0).
nchains	Number of MCMC chains to run (default = 1).
inits	Optional argument to specify initial values for each chain. See details.
setSeed	Logical argument. If TRUE, then R's random number seed is set to `i` (using `set.seed(i)`) at the onset of each MCMC chain number `i` (default = FALSE).
progressBar	Logical argument. If TRUE, an MCMC progress bar is displayed during execution of each MCMC chain (default = TRUE).
silent	Logical argument. If TRUE, then all output is suppressed during execution. This overrides the progressBar argument (default = FALSE).
returnCodaMCMC	Logical argument. If TRUE, then a coda mcmc object is returned instead of an R matrix of samples, or when `nchains > 1` a coda `mcmc.list` object is returned containing nchains mcmc objects (default = FALSE).

Details

The mcmc argument can be a compiled or uncompiled NIMBLE MCMC algorithm, which is generated using buildMCMC. Using a compiled algorithm will give substantially faster execution.

If provided, the inits argument can be one of three things: (1) a function to generate initial values, which will be executed to generate initial values at the beginning of each MCMC chain, (2) a single named list of initial values which, will be used for each chain, or (3) a list of length nchains, each element being a named list of initial values which be used for one MCMC chain. The inits argument may also be omitted, in which case the current values in the model object will be used as the initial values of the first chain, and subsequent chains will begin using starting values where the previous chain ended.

Other aspects of the MCMC algorithm, such as sampler assignments and thinning, must be specified in advance using the MCMC configuration object (created using configureMCMC), which is then used to build the MCMC algorithm (using buildMCMC) argument.

The niter argument specifies the number of pre-thinning MCMC iterations, and the nburnin argument will remove post-thinning samples.

The MCMC option mcmc$run(..., reset = FALSE), used to continue execution of an MCMC chain, is not available through runMCMC().

Value

When nchains = 1, a matrix of MCMC samples. When nchains > 1, a list of length nchains, where each list element is a matrix of MCMC samples. If returnCodaMCMC = TRUE, then a coda mcmc or mcmc.list object is returned instead.

Author(s)

Daniel Turek

See Also

configureMCMC buildMCMC

Examples

```
## Not run:
code <- nimbleCode({
    mu ~ dnorm(0, sd = 1000)
    sigma ~ dunif(0, 1000)
    for(i in 1:10) {
        x[i] ~ dnorm(mu, sd = sigma)
    }
})
Rmodel <- nimbleModel(code)
Rmodel$setData(list(x = c(2, 5, 3, 4, 1, 0, 1, 3, 5, 3)))
Rmcmc <- buildMCMC(Rmodel)
Cmodel <- compileNimble(Rmodel)
Cmcmc <- compileNimble(Rmcmc, project = Rmodel)
inits <- function() list(mu = rnorm(1,0,1), sigma = runif(1,0,10))
samplesList <- runMCMC(Cmcmc, niter = 10000, nchains = 3, inits = inits)
```

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

---

sampler_BASE *MCMC Sampling Algorithms*

Description

Details of the MCMC sampling algorithms provided with the NIMBLE MCMC engine

Usage

```
sampler_BASE()

sampler_posterior_predictive(model, mvSaved, target, control)

sampler_binary(model, mvSaved, target, control)

sampler_RW(model, mvSaved, target, control)

sampler_RW_block(model, mvSaved, target, control)

sampler_RW_llFunction(model, mvSaved, target, control)

sampler_slice(model, mvSaved, target, control)

sampler_ess(model, mvSaved, target, control)

sampler_crossLevel(model, mvSaved, target, control)

sampler_RW_llFunction_block(model, mvSaved, target, control)

sampler_RW_PF(model, mvSaved, target, control)

sampler_RW_PF_block(model, mvSaved, target, control)

sampler_RW_multinomial(model, mvSaved, target, control)
```

Arguments

model	(uncompiled) model on which the MCMC is to be run
mvSaved	modelValues object to be used to store MCMC samples
target	node(s) on which the sampler will be used
control	named list that controls the precise behavior of the sampler, with elements specific to samplertype. The default values for control list elements are determined by the NIMBLE system option 'MCMCcontrolDefaultList'. Descriptions of each sampling algorithm, and the possible customizations for each sampler (using the control argument) appear below.

`sampler_base`

base class for new samplers

When you write a new sampler for use in a NIMBLE MCMC (see User Manual), you must include `contains = sampler_BASE`.

binary sampler

The binary sampler performs Gibbs sampling for binary-valued (discrete 0/1) nodes. This can only be used for nodes following either a `dbern(p)` or `dbinom(p, size=1)` distribution.

The binary sampler accepts no control list arguments.

RW sampler

The RW sampler executes adaptive Metropolis-Hastings sampling with a normal proposal distribution (Metropolis, 1953), implementing the adaptation routine given in Shaby and Wells, 2011. This sampler can be applied to any scalar continuous-valued stochastic node, and can optionally sample on a log scale.

The RW sampler accepts the following control list elements:

- log. A logical argument, specifying whether the sampler should operate on the log scale. (default = FALSE)
- reflective. A logical argument, specifying whether the normal proposal distribution should reflect to stay within the range of the target distribution. (default = FALSE)
- adaptive. A logical argument, specifying whether the sampler should adapt the scale (proposal standard deviation) throughout the course of MCMC execution to achieve a theoretically desirable acceptance rate. (default = TRUE)
- adaptInterval. The interval on which to perform adaptation. Every adaptInterval MCMC iterations (prior to thinning), the RW sampler will perform its adaptation procedure. This updates the scale variable, based upon the sampler's achieved acceptance rate over the past adaptInterval iterations. (default = 200)
- scale. The initial value of the normal proposal standard deviation. If adaptive = FALSE, scale will never change. (default = 1)

The RW sampler cannot be used with options log=TRUE and reflective=TRUE, i.e. it cannot do reflective sampling on a log scale.

RW_block sampler

The RW_block sampler performs a simultaneous update of one or more model nodes, using an adaptive Metropolis-Hastings algorithm with a multivariate normal proposal distribution (Roberts and Sahu, 1997), implementing the adaptation routine given in Shaby and Wells, 2011. This sampler may be applied to any set of continuous-valued model nodes, to any single continuous-valued multivariate model node, or to any combination thereof.

The RW_block sampler accepts the following control list elements:

- adaptive. A logical argument, specifying whether the sampler should adapt the scale (a coefficient for the entire proposal covariance matrix) and propCov (the multivariate normal proposal covariance matrix) throughout the course of MCMC execution. If only the scale should undergo adaptation, this argument should be specified as TRUE. (default = TRUE)
- adaptScaleOnly. A logical argument, specifying whether adaption should be done only for scale (TRUE) or also for provCov (FALSE). This argument is only relevant when adaptive = TRUE. When adaptScaleOnly = FALSE, both scale and propCov undergo adaptation; the sampler tunes the scaling to achieve a theoretically good acceptance rate, and the proposal covariance to mimic that of the empirical samples. When adaptScaleOnly = FALSE, only the proposal scale is adapted. (default = FALSE)
- adaptInterval. The interval on which to perform adaptation. Every adaptInterval MCMC iterations (prior to thinning), the RW_block sampler will perform its adaptation procedure, based on the past adaptInterval iterations. (default = 200)
- scale. The initial value of the scalar multiplier for propCov. If adaptive = FALSE, scale will never change. (default = 1)
- propCov. The initial covariance matrix for the multivariate normal proposal distribution. This element may be equal to the character string 'identity', in which case the identity matrix of the appropriate dimension will be used for the initial proposal covariance matrix. (default = 'identity')

RW_llFunction sampler

Sometimes it is useful to control the log likelihood calculations used for an MCMC updater instead of simply using the model. For example, one could use a sampler with a log likelihood that analytically (or numerically) integrates over latent model nodes. Or one could use a sampler with a log likelihood that comes from a stochastic approximation such as a particle filter, allowing composition of a particle MCMC (PMCMC) algorithm (Andrieu et al., 2010). The RW_llFunction sampler handles this by using a Metropolis-Hastings algorithm with a normal proposal distribution and a user-provided log-likelihood function. To allow compiled execution, the log-likelihood function must be provided as a specialized instance of a nimbleFunction. The log-likelihood function may use the same model as the MCMC as a setup argument, but if so the state of the model should be unchanged during execution of the function (or you must understand the implications otherwise).

The RW_llFunction sampler accepts the following control list elements:

- adaptive. A logical argument, specifying whether the sampler should adapt the scale (proposal standard deviation) throughout the course of MCMC execution. (default = TRUE)
- adaptInterval. The interval on which to perform adaptation. (default = 200)
- scale. The initial value of the normal proposal standard deviation. (default = 1)
- llFunction. A specialized nimbleFunction that accepts no arguments and returns a scalar double number. The return value must be the total log-likelihood of all stochastic dependents of the target nodes – and, if includesTarget = TRUE, of the target node(s) themselves – or whatever surrogate is being used for the total log-likelihood. This is a required element with no default.
- includesTarget. Logical variable indicating whether the return value of llFunction includes the log-likelihood associated with target. This is a required element with no default.

slice sampler

The slice sampler performs slice sampling of the scalar node to which it is applied (Neal, 2003). This sampler can operate on either continuous-valued or discrete-valued scalar nodes. The slice sampler performs a 'stepping out' procedure, in which the slice is iteratively expanded to the left or right by an amount sliceWidth. This sampler is optionally adaptive, governed by a control list element, whereby the value of sliceWidth is adapted towards the observed absolute difference between successive samples.

The slice sampler accepts the following control list elements:

- adaptive. A logical argument, specifying whether the sampler will adapt the value of sliceWidth throughout the course of MCMC execution. (default = TRUE)
- adaptInterval. The interval on which to perform adaptation. (default = 200)
- sliceWidth. The initial value of the width of each slice, and also the width of the expansion during the iterative 'stepping out' procedure. (default = 1)
- sliceMaxSteps. The maximum number of expansions which may occur during the 'stepping out' procedure. (default = 100)

ess sampler

The ess sampler performs elliptical slice sampling of a single node, which must follow a multivariate normal distribution (Murray, 2010). The algorithm is an extension of slice sampling (Neal, 2003), generalized to the multivariate normal context. An auxilliary variable is used to identify points on an ellipse (which passes through the current node value) as candidate samples, which are accepted contingent upon a likelihood evaluation at that point. This algorithm requires no tuning parameters and therefore no period of adaptation, and may result in very efficient sampling from multivariate Gaussian distributions.

The ess sampler accepts no control list arguments.

crossLevel sampler

This sampler is constructed to perform simultaneous updates across two levels of stochastic dependence in the model structure. This is possible when all stochastic descendents of node(s) at one level have conjugate relationships with their own stochastic descendents. In this situation, a Metropolis-Hastings algorithm may be used, in which a multivariate normal proposal distribution is used for the higher-level nodes, and the corresponding proposals for the lower-level nodes undergo Gibbs (conjugate) sampling. The joint proposal is either accepted or rejected for all nodes involved based upon the Metropolis-Hastings ratio.

The requirement that all stochastic descendents of the target nodes must themselves have only conjugate descendents will be checked when the MCMC algorithm is built. This sampler is useful when there is strong dependence across the levels of a model that causes problems with convergence or mixing.

The crossLevel sampler accepts the following control list elements:

- adaptive. Logical argument, specifying whether the multivariate normal proposal distribution for the target nodes should be adaptived. (default = TRUE)
- adaptInterval. The interval on which to perform adaptation. (default = 200)

- scale. The initial value of the scalar multiplier for propCov. (default = 1)
- propCov. The initial covariance matrix for the multivariate normal proposal distribution. This element may be equal to the character string 'identity' or any positive definite matrix of the appropriate dimensions. (default = 'identity')

RW_llFunction_block sampler

Sometimes it is useful to control the log likelihood calculations used for an MCMC updater instead of simply using the model. For example, one could use a sampler with a log likelihood that analytically (or numerically) integrates over latent model nodes. Or one could use a sampler with a log likelihood that comes from a stochastic approximation such as a particle filter, allowing composition of a particle MCMC (PMCMC) algorithm (Andrieu et al., 2010) (but see samplers listed below for NIMBLE's direct implementation of PMCMC). The `RW_llFunctionBlock` sampler handles this by using a Metropolis-Hastings algorithm with a multivariate normal proposal distribution and a user-provided log-likelihood function. To allow compiled execution, the log-likelihood function must be provided as a specialized instance of a nimbleFunction. The log-likelihood function may use the same model as the MCMC as a setup argument, but if so the state of the model should be unchanged during execution of the function (or you must understand the implications otherwise).

The RW_llFunctionBlock sampler accepts the following control list elements:

- adaptive. A logical argument, specifying whether the sampler should adapt the proposal covariance throughout the course of MCMC execution. (default is TRUE)
- adaptScaleOnly. A logical argument, specifying whether adaption should be done only for scale (TRUE) or also for provCov (FALSE). This argument is only relevant when adaptive = TRUE. When adaptScaleOnly = FALSE, both scale and propCov undergo adaptation; the sampler tunes the scaling to achieve a theoretically good acceptance rate, and the proposal covariance to mimic that of the empirical samples. When adaptScaleOnly = FALSE, only the proposal scale is adapted. (default = FALSE)
- adaptInterval. The interval on which to perform adaptation. (default = 200)
- scale. The initial value of the scalar multiplier for propCov. If adaptive = FALSE, scale will never change. (default = 1)
- propCov. The initial covariance matrix for the multivariate normal proposal distribution. This element may be equal to the character string 'identity', in which case the identity matrix of the appropriate dimension will be used for the initial proposal covariance matrix. (default = 'identity')
- llFunction. A specialized nimbleFunction that accepts no arguments and returns a scalar double number. The return value must be the total log-likelihood of all stochastic dependents of the target nodes – and, if includesTarget = TRUE, of the target node(s) themselves – or whatever surrogate is being used for the total log-likelihood. This is a required element with no default.
- includesTarget. Logical variable indicating whether the return value of llFunction includes the log-likelihood associated with target. This is a required element with no default.

RW_PF sampler

The particle filter sampler allows the user to perform PMCMC (Andrieu et al., 2010), integrating over latent nodes in the model to sample top-level parameters. The `RW_PF` sampler uses a

Metropolis-Hastings algorithm with a univariate normal proposal distribution for a scalar parameter. Note that latent states can be sampled as well, but the top-level parameter being sampled must be a scalar. A bootstrap or auxiliary particle filter can be used to integrate over latent states.

The RW_PF sampler accepts the following control list elements:

- adaptive. A logical argument, specifying whether the sampler should adapt the scale (proposal standard deviation) throughout the course of MCMC execution to achieve a theoretically desirable acceptance rate. (default = TRUE)
- adaptInterval. The interval on which to perform adaptation. Every adaptInterval MCMC iterations (prior to thinning), the RW sampler will perform its adaptation procedure. This updates the scale variable, based upon the sampler's achieved acceptance rate over the past adaptInterval iterations. (default = 200)
- scale. The initial value of the normal proposal standard deviation. If `adaptive = FALSE`, scale will never change. (default = 1)
- pfNparticles. The number of particles to use in the approximation to the log likelihood of the data (default = 1000).
- latents. Character vector specifying the latent model nodes over which the particle filter will stochastically integrate over to estimate the log-likelihood function.
- pfType. Character argument specifying the type of particle filter that should be used for likelihood approximation. Choose from `"bootstrap"` and `"auxiliary"`. Defaults to `"bootstrap"`.
- pfLookahead. Optional character argument specifying the lookahead function for the auxiliary particle filter. Choose from `"simulate"` and `"mean"`. Only applicable if `pfType` is set to `"auxiliary"`.
- pfResample. A logical argument, specifying whether to resample log likelihood given current parameters at beginning of each MCMC step, or whether to use log likelihood from previous step.
- pfOptimizeNparticles. A logical argument, specifying whether to automatically determine the optimal number of particles to use, based on Pitt and Shephard (2011). This will override any value of `pfNparticles` specified above.

RW_PF_block sampler

The particle filter sampler allows the user to perform PMCMC (Andrieu et al., 2010), integrating over latent nodes in the model to sample top-level parameters. The `RW_PF_block` sampler uses a Metropolis-Hastings algorithm with a multivariate normal proposal distribution. A bootstrap or auxiliary particle filter can be used to integrate over latent states.

The `RW_PF_block` sampler accepts the following control list elements:

- adaptive. A logical argument, specifying whether the sampler should adapt the proposal covariance throughout the course of MCMC execution. (default = TRUE)
- adaptScaleOnly. A logical argument, specifying whether adaption should be done only for `scale` (TRUE) or also for `provCov` (FALSE). This argument is only relevant when `adaptive = TRUE`. When `adaptScaleOnly = FALSE`, both `scale` and `propCov` undergo adaptation; the sampler tunes the scaling to achieve a theoretically good acceptance rate, and the proposal covariance to mimic that of the empirical samples. When `adaptScaleOnly = FALSE`, only the proposal scale is adapted. (default = FALSE)

- adaptInterval. The interval on which to perform adaptation. (default = 200)
- scale. The initial value of the scalar multiplier for `propCov`. If `adaptive = FALSE`, `scale` will never change. (default = 1)
- propCov. The initial covariance matrix for the multivariate normal proposal distribution. This element may be equal to the `'identity'`, in which case the identity matrix of the appropriate dimension will be used for the initial proposal covariance matrix. (default is `'identity'`)
- pfNparticles. The number of particles to use in the approximation to the log likelihood of the data (default = 1000).
- latents. Character vector specifying the latent model nodes over which the particle filter will stochastically integrate to estimate the log-likelihood function.
- pfResample. A logical argument, specifying whether to resample log likelihood given current parameters at beginning of each mcmc step, or whether to use log likelihood from previous step.
- pfType. Character argument specifying the type of particle filter that should be used for likelihood approximation. Choose from `"bootstrap"` and `"auxiliary"`. Defaults to `"bootstrap"`.
- pfLookahead. Optional character argument specifying the lookahead function for the auxiliary particle filter. Choose from `"simulate"` and `"mean"`. Only applicable if `pfType = "auxiliary"`.
- pfOptimizeNparticles. A logical argument, specifying whether to automatically determine the optimal number of particles to use, based on Pitt and Shephard (2011). This will override any value of `pfNparticles` specified above.

RW_multinomial sampler

This sampler is designed for sampling multinomial target distributions. The sampler performs a series of Metropolis-Hastings steps between pairs of groups. Proposals are generated via a draw from a binomial distribution, whereafter the proposed number density is moved from one group to another group. The acceptance or rejection of these proposals follows a standard Metropolis-Hastings procedure. Probabilities for the random binomial proposals are adapted to a target acceptance rate of 0.5.

The `RW_multinomial` sampler accepts the following control list elements:

- adaptive. A logical argument, specifying whether the sampler should adapt the binomial proposal probabilities throughout the course of MCMC execution. (default = TRUE)
- adaptInterval. The interval on which to perform adaptation. A minimum value of 100 is required. (default = 200)

posterior_predictive sampler

The posterior_predictive sampler is only appropriate for use on terminal stochastic nodes. Note that such nodes play no role in inference but have often been included in BUGS models to accomplish posterior predictive checks. NIMBLE allows posterior predictive values to be simulated independently of running MCMC, for example by writing a nimbleFunction to do so. This means that in many cases where terminal stochastic nodes have been included in BUGS models, they are not needed when using NIMBLE.

The posterior_predictive sampler functions by calling the simulate() method of relevant node, then updating model probabilities and deterministic dependent nodes. The application of a posterior_predictive

sampler to any non-terminal node will result in invalid posterior inferences. The posterior_predictive sampler will automatically be assigned to all terminal, non-data stochastic nodes in a model by the default MCMC configuration, so it is uncommon to manually assign this sampler.

The posterior_predictive sampler accepts no control list arguments.

Author(s)

Daniel Turek

References

Andrieu, C., Doucet, A., and Holenstein, R. (2010). Particle Markov Chain Monte Carlo Methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3), 269-342.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6), 1087-1092.

Neal, Radford M. (2003). Slice Sampling. *The Annals of Statistics*, 31(3), 705-741.

Murray, I., Prescott Adams, R., and MacKay, D. J. C. (2010). Elliptical Slice Sampling. *arXiv e-prints*, arXiv:1001.0175.

Pitt, M.K. and Shephard, N. (1999). Filtering via simulation: Auxiliary particle filters. *Journal of the American Statistical Association* 94(446), 590-599.

Roberts, G. O. and S. K. Sahu (1997). Updating Schemes, Correlation Structure, Blocking and Parameterization for the Gibbs Sampler. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 59(2), 291-317.

Shaby, B. and M. Wells (2011). *Exploring an Adaptive Metropolis Algorithm*. 2011-14. Department of Statistics, Duke University.

See Also

`configureMCMC` `addSampler` `buildMCMC` `runMCMC`

`setAndCalculate` *Creates a nimbleFunction for setting the values of one or more model nodes, calculating the associated deterministic dependents and logProb values, and returning the total sum log-probability.*

Description

This nimbleFunction generator must be specialized to any model object and one or more model nodes. A specialized instance of this nimbleFunction will set the values of the target nodes in the specified model, calculate the associated logProbs, calculate the values of any deterministic dependents, calculate the logProbs of any stochastic dependents, and return the sum log-probability associated with the target nodes and all stochastic dependent nodes.

Usage

```
setAndCalculate(model, targetNodes)

setAndCalculateDiff(model, targetNodes)
```

Arguments

model	An uncompiled or compiled NIMBLE model. This argument is required.
targetNodes	A character vector containing the names of one or more nodes or variables in the model. This argument is required.

Details

Calling setAndCalculate(model, targetNodes) or setAndCalculate(model, targetNodes) will return a nimbleFunction object whose run function takes a single, required argument:

targetValues: A vector of numeric values which will be put into the target nodes in the specified model object. The length of this numeric vector much exactly match the number of target nodes.

The difference between setAndCalculate and setAndCalculateDiff is the return value of their run functions. In the former, run returns the sum of the log probabilities of the targetNodes with the provided targetValues, while the latter returns the difference between that sum with the new targetValues and the previous values in the model.

Author(s)

Daniel Turek

Examples

```
code <- nimbleCode({ for(i in 1:3) { x[i] ~ dnorm(0,1); y[i] ~ dnorm(0, 1)}})
Rmodel <- nimbleModel(code)
my_setAndCalc <- setAndCalculate(Rmodel, c('x[1]', 'x[2]', 'y[1]', 'y[2]'))
lp <- my_setAndCalc$run(c(1.2, 1.4, 7.6, 8.9))
```

---

setAndCalculateOne *Creates a nimbleFunction for setting the value of a scalar model node, calculating the associated deterministic dependents and logProb values, and returning the total sum log-probability.*

Description

This nimbleFunction generator must be specialized to any model object and any scalar model node. A specialized instance of this nimbleFunction will set the value of the target node in the specified model, calculate the associated logProb, calculate the values of any deterministic dependents, calculate the logProbs of any stochastic dependents, and return the sum log-probability associated with the target node and all stochastic dependent nodes.

Usage

```
setAndCalculateOne(model, targetNode)
```

Arguments

model	An uncompiled or compiled NIMBLE model. This argument is required.
targetNode	The character name of any scalar node in the model object. This argument is required.

Details

Calling setAndCalculateOne(model, targetNode) will return a function with a single, required argument:

targetValue: The numeric value which will be put into the target node, in the specified model object.

Author(s)

Daniel Turek

Examples

```
code <- nimbleCode({ for(i in 1:3) x[i] ~ dnorm(0, 1) })
Rmodel <- nimbleModel(code)
my_setAndCalc <- setAndCalculateOne(Rmodel, 'x[1]')
lp <- my_setAndCalc$run(2)
```

---

setSize *set the size of a numeric variable in NIMBLE*

Description

set the size of a numeric variable in NIMBLE. This works in R and NIMBLE, but in R it usually has no effect.

Usage

```
setSize(numObj, ..., row)
```

Arguments

numObj	This is the object to be resized
...	sizes, provided as scalars, in order, with exactly as many as needed for the object
row	Optional argument that is not currently used

Details

This function is part of the NIMBLE language. Its purpose is to explicitly resize a multivariate object (vector, matrix or array), currently up to 4 dimensions. Explicit resizing is not needed when an entire object is assigned to. For example, in `Y <- A %*% B`, where A and B are matrices, `Y` will be resized automatically. Explicit resizing is necessary when assignment will be by indexed elements or blocks, if the object is not already an appropriate size for the assignment. E.g. prior to `Y[5:10] <- A %*% B`, one can use setSize to ensure that `Y` has a size (length) of at least 10.

This does work in uncompiled (R) and well as compiled execution, but in some cases it is only necessary for compiled execution. During uncompiled execution, it may not catch bugs due to resizing because some R objects will be dynamically resized during assignments anyway.

Author(s)

NIMBLE development team

`setupOutputs` *Explicitly declare objects created in setup code to be preserved and compiled as member data*

Description

Normally a `nimbleFunction` determines what objects from `setup` code need to be preserved for `run` code or other member functions. `setupOutputs` allows explicit declaration for cases when an object created in setup output is not use in member functions.

Arguments

...	An arbitrary set of names

Details

Normally any object created in `setup` code whose name appears in `run` or another member function is included in the save results of setup code. When the nimbleFunction is compiled, such objects will become member data of the resulting C++ class. If it is desired to force an object to become member data even if it does not appear in a member function, declare it using `setupOutputs`. E.g. `setupOutputs(a, b)` declares that `a` and `b` should be preserved.

The `setupOutputs` line will be removed from the setup code. It is really a marker during nimbleFunction creation of what should be preserved.

simNodes *Basic nimbleFunctions for calculate, simulate, and getLogProb with a set of nodes*

Description

simulate, calculate, or get existing log probabilities for the current values in a NIMBLE model

Usage

```
simNodes(model, nodes)

calcNodes(model, nodes)

getLogProbNodes(model, nodes)
```

Arguments

model	A NIMBLE model
nodes	A set of nodes. If none are provided, default is all `model$getNodeNames()`

Details

These are basic nimbleFunctions that take a model and set of nodes and return a function that will call `calculate`, `simulate`, or `getLogProb` on those nodes. Each is equivalent to a direct call from R, but in nimbleFunction form they can be be compiled and can be put into a nimbleFunctionList. For example, `myCalc <- calcNodes(model, nodes); ans <- myCalc()` is equivalent to `ans <- calculate(model, nodes)`, but one can also do `CmyCalc <- compileNimble(myCalc)` to get a faster version.

In nimbleFunctions, for only one set of nodes, it is equivalent or slightly better to simply use `calculate(model, nodes)` in the run-time code. The compiler will process the model-nodes combination in the same way as would occur by creating a specialized `calcNodes` in the setup code. However, if there are multiple sets of nodes, one can do the following:

Setup code: `myCalcs <- nimbleFunctionList(calcNodes); myCalcs[[1]] <- calcNodes(model, nodes[[1]]); myC`

Run code: `for(i in seq_along(myCalcs)) {ans[i] <- myCalcs[[i]]()}`

Author(s)

Perry de Valpine

simNodesMV *Basic nimbleFunctions for using a NIMBLE model with sets of stored values*

Description

simulate, calculate, or get the existing log probabilities for values in a modelValues object using a NIMBLE model

Usage

```
simNodesMV(model, mv, nodes)

calcNodesMV(model, mv, nodes)

getLogProbNodesMV(model, mv, nodes)
```

Arguments

model	A nimble model.
mv	A modelValues object in which multiple sets of model variables and their corresponding logProb values are or will be saved. mv must include the nodes provided
nodes	A set of nodes. If none are provided, default is all model$getNodeNames()

Details

simNodesMV simulates values in the given nodes and saves them in mv. calcNodesMV calculates these nodes for each row of mv and returns a vector of the total log probabilities (densities) for each row. getLogProbNodesMV is like calcNodesMV without actually doing the calculations.

Each of these will expand variables or index blocks and topologically sort them so that each node's parent nodes are processed before itself.

getLogProbMV should be used carefully. It is generally for situations where the logProb values are guaranteed to have already been calculated, and all that is needed is to query them. The risk is that a program may have changed the values in the nodes, in which case getLogProbMV would collect logProb values that are out of date with the node values.

Value

from simNodesMV: NULL. from calcNodesMV and getLogProbMV: a vector of the sum of log probabilities (densities) from any stochastic nodes in nodes.

Run time arguments

- m

  (simNodesMV only). Number of simulations requested.

- saveLP

  (calcNodesMVonly). Whether to save the logProb values in mv. Should be given as TRUE unless there is a good reason not to.

Author(s)

Clifford Anderson-Bergman

Examples

```
code <- nimbleCode({
for(i in 1:5)
x[i] ~ dnorm(0,1)
})

myModel <- nimbleModel(code)
myMV <- modelValues(myModel)

Rsim <- simNodesMV(myModel, myMV)
Rcalc <- calcNodesMV(myModel, myMV)
Rglp <- getLogProbNodesMV(myModel, myMV)
## Not run:
  cModel <- compileNimble(myModel)
  Csim <- compileNimble(Rsim, project = myModel)
  Ccalc <- compileNimble(Rcalc, project = myModel)
  Cglp <- compileNimble(Rglp, project = myModel)
  Csim$run(10)
  Ccalc$run(saveLP = TRUE)
  Cglp$run() #Gives identical answers to Ccalc because logProbs were saved
  Csim$run(10)
  Ccalc$run(saveLP = FALSE)
  Cglp$run()   #Gives wrong answers because logProbs were not saved

## End(Not run)
```

---

singleVarAccessClass-class

Class singleVarAccessClass

Description

Classes used internally in NIMBLE and not expected to be called directly by users.

t *The t Distribution*

Description

Density, distribution function, quantile function and random generation for the t distribution with df degrees of freedom, allowing non-zero location, mu, and non-unit scale, sigma

Usage

```
dt_nonstandard(x, df = 1, mu = 0, sigma = 1, log = FALSE)

rt_nonstandard(n, df = 1, mu = 0, sigma = 1)

pt_nonstandard(q, df = 1, mu = 0, sigma = 1, lower.tail = TRUE,
  log.p = FALSE)

qt_nonstandard(p, df = 1, mu = 0, sigma = 1, lower.tail = TRUE,
  log.p = FALSE)
```

Arguments

x	vector of values.
df	vector of degrees of freedom values.
mu	vector of location values.
sigma	vector of scale values.
log	logical; if TRUE, probability density is returned on the log scale.
n	number of observations.
q	vector of quantiles.
lower.tail	logical; if TRUE (default) probabilities are $P[X \leq x]$; otherwise, $P[X > x]$.
log.p	logical; if TRUE, probabilities p are given by user as log(p).
p	vector of probabilities.

Details

See Gelman et al., Appendix A or the BUGS manual for mathematical details.

Value

dt_nonstandard gives the density, pt_nonstandard gives the distribution function, qt_nonstandard gives the quantile function, and rt_nonstandard generates random deviates.

Author(s)

Christopher Paciorek

References

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. (2004) *Bayesian Data Analysis*, 2nd ed. Chapman and Hall/CRC.

See Also

Distributions for other standard distributions

Examples

```
x <- rt_nonstandard(50, df = 1, mu = 5, sigma = 1)
dt_nonstandard(x, 3, 5, 1)
```

`testBUGSmodel` *Tests BUGS examples in the NIMBLE system*

Description

`testBUGSmodel` builds a BUGS model in the NIMBLE system and simulates from the model, comparing the values of the nodes and their log probabilities in the uncompiled and compiled versions of the model

Usage

```
testBUGSmodel(example = NULL, dir = NULL, model = NULL, data = NULL,
  inits = NULL, useInits = TRUE, debug = FALSE)
```

Arguments

`example`	(optional) example character vector indicating name of BUGS example to test; can be null if `model` is provided
`dir`	(optional) character vector indicating directory in which files are contained, by default the classic-bugs directory if the installed package is used; to use the current working directory, set this to ""
`model`	(optional) one of (1) a character string giving the file name containing the BUGS model code, (2) an R function whose body is the BUGS model code, or (3) the output of `nimbleCode`. If a file name, the file can contain a 'var' block and 'data' block in the manner of the JAGS versions of the BUGS examples but should not contain references to other input data files nor a const block. The '.bug' or '.txt' extension can be excluded.
`data`	(optional) one of (1) character string giving the file name for an R file providing the input constants and data as R code [assigning individual objects or as a named list] or (2) a named list providing the input constants and data. If neither is provided, the function will look for a file named `example`-data including extensions .R, .r, or .txt.

inits	(optional) (1) character string giving the file name for an R file providing the initial values for parameters as R code [assigning individual objects or as a named list] or (2) a named list providing the values. If neither is provided, the function will look for a file named example-init or example-inits including extensions .R, .r, or .txt.
useInits	boolean indicating whether to test model with initial values provided via inits
debug	logical indicating whether to put the user in a browser for debugging when testBUGSmodel calls readBUGSmodel. Intended for developer use.

Details

Note that testing without initial values may cause warnings when parameters are sampled from improper or fat-tailed distributions

Author(s)

Christopher Paciorek

Examples

```
## Not run:
testBUGSmodel('pump')

## End(Not run)
```

---

updateMCMCcomparisonWithHighOrderESS

Re-estimate effective sample size from results of compareMCMCs

---

Description

By default the effective sample sizes (ESS) (and MCMC efficiencies) are estimated from the coda package. This function estimates them either using the same method as coda but allowing higher-order AR models or using the method provided in Stan via package rstan.

Usage

```
updateMCMCcomparisonWithHighOrderESS(mcmcResults, logVars = "",
  includeBurninTime = TRUE, StanESS = FALSE)
```

Arguments

mcmcResults	An object returned by compareMCMCs
logVars	Names of any variables for which ESS should be calculated after log transformation. This is useful if different methods use models in which the same variance component is set up either as standard deviation, variance, or precision. Calculating ESS on a log scale arguably makes these more comparable than converting one into another for comparisons.

includeBurninTime
: (default TRUE) If TRUE, the time spent on burnin (or warmup) is included in the denominator of MCMC efficiency. If false, only the time spent generating samples that are used for estimation of ESS is included.

StanESS
: (default FALSE) If FALSE, ESS will be estimated by the same method as in the code package, but allowing higher-order auto-regressive models to be considered. If TRUE, the method in Stan will be used via the `monitor` function in package rstan.

Value

A list of the same format as `mcmcResults`

`valueInCompiledNimbleFunction`
: *get or set value of member data from a compiled nimbleFunction using a multi-interface*

Description

Most nimbleFunctions written for direct user interaction allow standard R-object-like access to member data using `$` or `` `[[` ``. However, sometimes compiled nimbleFunctions contained within other compiled nimbleFunctions are interfaced with a light-weight system called a multi-interface. `valueInCompiledNimbleFunction` provides a way to get or set values in such cases.

Usage

```
valueInCompiledNimbleFunction(cnf, name, value)
```

Arguments

`cnf`	Compiled nimbleFunction object
`name`	Name of the member data
`value`	If provided, the value to assign to the member data. If omitted, the value of the member data is returned.

Details

The member data of a nimbleFunction are the objects created in `setup` code that are used in `run` code or other member functions.

Whether multi-interfaces are used for nested nimbleFunctions is controlled by the `buildInterfacesForCompiledNestedNim` option in nimbleOptions.

To see an example of a multi-interface, see `samplerFunctions` in a compiled MCMC interface object.

Author(s)

Perry de Valpine

values *Access or set values for a set of nodes in a model*

Description

Get or set values for a set of nodes in a model

Usage

```
values(model, nodes)

values(model, nodes) <- value
```

Arguments

model	a NIMBLE model object, either compiled or uncompiled
nodes	a vector of node names, allowing index blocks that will be expanded
value	value to set the node(s) to

Details

Access or set values for a set of nodes in a NIMBLE model.

Calling `values(model, nodes)` returns a vector of the concatenation of values from the nodes requested `P <- values(model, nodes)` is a newer syntax for `getValues(P, model, values)`, which still works and modifies P in the calling environment.

Calling `values(model, nodes) <- P` sets the value of the nodes in the model, in sequential order from the vector P.

In both uses, when requested nodes are from matrices or arrays, the values will be handled following column-wise order.

The older function `getValues(P, model, nodes)` is equivalent to `P <- values(model, nodes)`, and the older function `setValues(P, model, nodes)` is equivalent to `values(model, nodes) <- P`

These functions work in R and in NIMBLE run-time code that can be compiled.

Value

A vector of values concatenated from the provided nodes in the model

Author(s)

NIMBLE development team

`Wishart` *The Wishart Distribution*

Description

Density and random generation for the Wishart distribution, using the Cholesky factor of either the scale matrix or the rate matrix.

Usage

```
dwish_chol(x, cholesky, df, scale_param = TRUE, log = FALSE)

rwish_chol(n = 1, cholesky, df, scale_param = TRUE)
```

Arguments

`x`	vector of values.
`cholesky`	upper-triangular Cholesky factor of either the scale matrix (when `scale_param` is TRUE) or rate matrix (otherwise).
`df`	degrees of freedom.
`scale_param`	logical; if TRUE the Cholesky factor is that of the scale matrix; otherwise, of the rate matrix.
`log`	logical; if TRUE, probability density is returned on the log scale.
`n`	number of observations (only `n=1` is handled currently).

Details

See Gelman et al., Appendix A or the BUGS manual for mathematical details. The rate matrix as used here is defined as the inverse of the scale matrix, S^{-1}, given in Gelman et al.

Value

`dwish_chol` gives the density and `rwish_chol` generates random deviates.

Author(s)

Christopher Paciorek

References

Gelman, A., Carlin, J.B., Stern, H.S., and Rubin, D.B. (2004) *Bayesian Data Analysis*, 2nd ed. Chapman and Hall/CRC.

See Also

Distributions for other standard distributions

Examples

```
df <- 40
ch <- chol(matrix(c(1, .7, .7, 1), 2))
x <- rwish_chol(1, ch, df = df)
dwish_chol(x, ch, df = df)
```

Index