Package ‘nimble’

June 9, 2024

Title  MCMC, Particle Filtering, and Programmable Hierarchical Modeling

Description  A system for writing hierarchical statistical models largely compatible with 'BUGS' and 'JAGS', writing nimbleFunctions to operate models and do basic R-style math, and compiling both models and nimbleFunctions via custom-generated C++. 'NIMBLE' includes default methods for MCMC, Laplace Approximation, Monte Carlo Expectation Maximization, and some other tools. The nimbleFunction system makes it easy to do things like implement new MCMC samplers from R, customize the assignment of samplers to different parts of a model from R, and compile the new samplers automatically via C++ alongside the samplers 'NIMBLE' provides. 'NIMBLE' extends the 'BUGS/JAGS' language by making it extensible: New distributions and functions can be added, including as calls to external compiled code. Although most people think of MCMC as the main goal of the 'BUGS/JAGS' language for writing models, one can use 'NIMBLE' for writing arbitrary other kinds of model-generic algorithms as well. A full User Manual is available at <https://r-nimble.org>.

Version  1.2.0

Date  2024-06-07

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

Depends  R (>= 3.1.2)

Imports  methods,igraph,coda,R6,pracma,numDeriv

Suggests  testthat,mcmcs


BugReports  https://github.com/nimble-dev/nimble/issues

SystemRequirements  GNU make

License  BSD_3_clause + file LICENSE | GPL (>= 2)

Copyright  See COPYRIGHTS file.

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.
Encoding UTF-8

Collate config.R all_utils.R options.R distributions_inputList.R
distributions_processInputList.R
distributionsImplementations.R BUGS_BUGSDecl.R BUGS_contexts.R
BUGS_nimbleGraph.R BUGS_modelDef.R BUGS_model.R
BUGS_graphNodeMaps.R BUGS_readBUGS.R BUGS_macros.R
BUGS_testBUGS.R BUGS_getDependencies.R BUGS_utils.R
BUGS_mathCompatibility.R externalCalls.R genCpp_exprClass.R
genCpp_operatorLists.R genCpp_RParseTree2ExprClasses.R
genCpp_initSizes.R genCpp_buildIntermediates.R
genCpp_processSpecificCalls.R genCpp_sizeProcessing.R
nimbleFunction_util.R nimbleFunction_core.R
nimbleFunction_nodeFunction.R nimbleFunction_nodeFunctionNew.R
nimbleFunction_Rderivs.R nimbleFunction_Rexecution.R
nimbleFunction_compile.R nimbleFunction_keywordProcessing.R
nimbleList_core.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_nimbleFunction.R cppDefs_nimbleList.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
MCMC_RJ.R MCMC_WAIC.R MCEM_build.R crossValidation.R
makevars.R setNimbleInternalFunctions.R registration.R
nimble-package.r QuadratureGrids.R zzz.R

RoxygenNote 7.2.3

NeedsCompilation yes

Author Perry de Valpine [aut],
Christopher Paciorek [aut, cre],
Daniel Turek [aut],
Nick Michaud [aut],
Cliff Anderson-Bergman [aut],
Fritz Obermeyer [aut],
Claudia Wehrhahn Cortes [aut] (Bayesian nonparametrics system),
Abel Rodríguez [aut] (Bayesian nonparametrics system),
Duncan Temple Lang [aut] (packaging configuration),
Wei Zhang [aut] (Laplace approximation),
Sally Paganin [aut] (reversible jump MCMC),
Joshua Hug [aut] (WAIC),
Paul van Dam-Bates [aut] (AGHQ approximation, Pólya-Gamma sampler, nimIntegrate),
Jagadish Babu [ctb] (code for the compilation system for an early version of NIMBLE),
Lauren Ponisio [ctb] (contributions to the cross-validation code),
Peter Sujan [ctb] (multivariate t distribution code)

Repository CRAN

Date/Publication 2024-06-09 06:40:02 UTC

Contents

ADbreak . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
ADNimbleList . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
ADproxyModelClass-class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
any_na . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
as.carAdjacency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
as.carCM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
asRow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
autoBlock . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
BUGSdeclClass-class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
buildAGHQGrid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
buildAuxiliaryFilter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
buildBootstrapFilter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
buildEnsembleKF . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
buildIteratedFilter2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
buildLaplace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
buildLiuWestFilter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
buildMCEM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
buildMCMC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
calculateWAIC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
CAR-Normal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
CAR-Proper . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
carBounds . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
carMaxBound . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
carMinBound . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
CAR_calcNumIslands . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
Categorical . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
checkInterrupt . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
ChineseRestaurantProcess . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
clearCompiled . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
CmodelBaseClass-class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
CnimbleFunctionBase-class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
compileNimble . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>configureMCMC</td>
<td>52</td>
</tr>
<tr>
<td>configureRJ</td>
<td>54</td>
</tr>
<tr>
<td>Constraint</td>
<td>58</td>
</tr>
<tr>
<td>decide</td>
<td>59</td>
</tr>
<tr>
<td>decideAndJump</td>
<td>59</td>
</tr>
<tr>
<td>declare</td>
<td>60</td>
</tr>
<tr>
<td>deregisterDistributions</td>
<td>61</td>
</tr>
<tr>
<td>Dirichlet</td>
<td>62</td>
</tr>
<tr>
<td>distributionInfo</td>
<td>63</td>
</tr>
<tr>
<td>Double-Exponential</td>
<td>65</td>
</tr>
<tr>
<td>eigenNimbleList</td>
<td>66</td>
</tr>
<tr>
<td>Exponential</td>
<td>67</td>
</tr>
<tr>
<td>extractControlElement</td>
<td>68</td>
</tr>
<tr>
<td>flat</td>
<td>69</td>
</tr>
<tr>
<td>getBound</td>
<td>70</td>
</tr>
<tr>
<td>getBUGSexampleDir</td>
<td>70</td>
</tr>
<tr>
<td>getConditionallyIndependentSets</td>
<td>71</td>
</tr>
<tr>
<td>getDefinition</td>
<td>73</td>
</tr>
<tr>
<td>getMacroParameters</td>
<td>73</td>
</tr>
<tr>
<td>getNimbleOption</td>
<td>75</td>
</tr>
<tr>
<td>getParam</td>
<td>75</td>
</tr>
<tr>
<td>getSamplesDPmeasure</td>
<td>76</td>
</tr>
<tr>
<td>getSize</td>
<td>78</td>
</tr>
<tr>
<td>identityMatrix</td>
<td>79</td>
</tr>
<tr>
<td>initializeModel</td>
<td>79</td>
</tr>
<tr>
<td>Interval</td>
<td>80</td>
</tr>
<tr>
<td>Inverse-Gamma</td>
<td>81</td>
</tr>
<tr>
<td>Inverse-Wishart</td>
<td>83</td>
</tr>
<tr>
<td>is.nf</td>
<td>84</td>
</tr>
<tr>
<td>is.nl</td>
<td>85</td>
</tr>
<tr>
<td>LKJ</td>
<td>85</td>
</tr>
<tr>
<td>makeBoundInfo</td>
<td>86</td>
</tr>
<tr>
<td>makeModelDerivsInfo</td>
<td>87</td>
</tr>
<tr>
<td>makeParamInfo</td>
<td>88</td>
</tr>
<tr>
<td>MCMCConf-class</td>
<td>89</td>
</tr>
<tr>
<td>modelBaseClass-class</td>
<td>96</td>
</tr>
<tr>
<td>modelDefClass-class</td>
<td>104</td>
</tr>
<tr>
<td>modelInitialization</td>
<td>105</td>
</tr>
<tr>
<td>modelValues</td>
<td>105</td>
</tr>
<tr>
<td>modelValuesBaseClass-class</td>
<td>106</td>
</tr>
<tr>
<td>modelValuesConf</td>
<td>107</td>
</tr>
<tr>
<td>model_macro_builder</td>
<td>108</td>
</tr>
<tr>
<td>Multinomial</td>
<td>111</td>
</tr>
<tr>
<td>Multivariate-t</td>
<td>112</td>
</tr>
<tr>
<td>MultivariateNormal</td>
<td>114</td>
</tr>
<tr>
<td>nfMethod</td>
<td>115</td>
</tr>
<tr>
<td>nfVar</td>
<td>116</td>
</tr>
<tr>
<td>nimble-internal</td>
<td>117</td>
</tr>
</tbody>
</table>
nimble-math ................................................................. 117
nimble-R-functions ....................................................... 118
nimbleCode ................................................................. 119
nimbleExternalCall ......................................................... 120
nimbleFunction ............................................................. 122
nimbleFunctionBase-class ................................................. 123
nimbleFunctionList-class .................................................. 124
nimbleFunctionVirtual ..................................................... 124
nimbleList ................................................................. 125
nimbleMCMC ............................................................... 126
nimbleModel ............................................................... 130
nimbleOptions ............................................................. 132
nimbleRecall ............................................................... 133
nimbleType-class .......................................................... 134
nimCat .................................................................... 135
nimCopy ................................................................. 136
nimDerivs ................................................................. 138
nimDim ................................................................. 139
nimEigen ................................................................. 140
nimIntegrate .............................................................. 141
nimMatrix ............................................................... 143
nimNumeric ............................................................... 145
nimOptim ................................................................. 146
nimOptimDefaultControl ................................................ 148
nimOptimMethod .......................................................... 149
nimPrint ................................................................. 150
nimStop ................................................................. 151
nimSvd ................................................................. 152
nodeFunctions ............................................................. 153
optimControlNimbleList ................................................ 155
optimDefaultControl ..................................................... 155
optimResultNimbleList ................................................... 156
parameterTransform ....................................................... 156
pow_int ................................................................. 158
printErrors ............................................................... 159
rankSample .............................................................. 160
readBUGSmodel ........................................................... 161
registerDistributions ..................................................... 163
resize ................................................................. 166
Rmatrix2mvOneVar ........................................................ 167
RmodelBaseClass-class .................................................. 168
run.time ................................................................. 168
runCrossValidate ........................................................ 169
runLaplace ............................................................... 172
runMCMC ............................................................... 174
sampler_BASE ............................................................. 177
setAndCalculate .......................................................... 195
setAndCalculateOne ....................................................... 197
ADbreak

NIMBLE language function to break tracking of derivatives

Description

This function is used in a method of a nimbleFunction that has derivatives enabled. It returns its value but breaks tracking of derivatives.

Usage

```r
ADbreak(x)
```

Arguments

- `x` scalar value

Details

This function only works with scalars.
**ADNimbleList**

Data type for the return value of `nimDerivs`

---

**Description**

`nimbleList` definition for the type of `nimbleList` returned by `nimDerivs`.

**Usage**

`ADNimbleList`

**Format**

An object of class `list` of length 1.

**Fields**

- `value`: The value of the function evaluated at the given input arguments.
- `jacobian`: The Jacobian of the function evaluated at the given input arguments.
- `hessian`: The Hessian of the function evaluated at the given input arguments.

**See Also**

`nimDerivs`

---

**ADproxyModelClass-class**

*create an ADproxyModelClass object*

---

**Description**

create an ADproxyModelClass object. For internal use.

**Arguments**

- `Rmodel`: The name of an uncompiled model

**Details**

This is a proxy model for `model_AD`. The class needs just enough pieces to be used like a model for purposes of nodeFunction compilation. The model will contain an ADproxyModel and then the nodeFunction setup code will extract it. The model interface will populate the proxy model’s `CobjectInterface`

**Author(s)**

NIMBLE development team
any_na

Determines if any values in a vector are NA or NaN.

Usage

```r
any_na(x)
```

Arguments

- `x`: vector of values

Author(s)

NIMBLE Development Team

as.carAdjacency

Converts CAR structural parameters to adjacency, weights, num format.

Description

This will convert alternate representations of CAR process structure into (adj, weights, num) form required by `dcar_normal`.

Usage

```r
as.carAdjacency(...)```

Arguments

- `...`: Either: a symmetric matrix of unnormalized weights, or two lists specifying adjacency indices and the corresponding unnormalized weights.

Details

Two alternate representations are handled:

- A single matrix argument will be interpreted as a matrix of symmetric unnormalized weights.
- Two lists will be interpreted as (the first) a list of numeric vectors specifying the adjacency (neighboring) indices of each CAR process component, and (the second) a list of numeric vectors giving the unnormalized weights for each of these neighboring relationships.
as.carCM

Author(s)
Daniel Turek

See Also
CAR-Normal

as.carCM

Convert weights vector to parameters of \texttt{dcar_proper distribution}

**Description**

Convert weights vector to \( C \) and \( M \) parameters of \texttt{dcar_proper} distribution

**Usage**

\texttt{as.carCM(adj, weights, num)}

**Arguments**

- \texttt{adj}:
  - vector of indices of the adjacent locations (neighbors) of each spatial location.
  - This is a sparse representation of the full adjacency matrix.

- \texttt{weights}:
  - vector of symmetric unnormalized weights associated with each pair of adjacent locations, of the same length as \texttt{adj}.
  - This is a sparse representation of the full (unnormalized) weight matrix.

- \texttt{num}:
  - vector giving the number of neighbors of each spatial location, with length equal to the total number of locations.

**Details**

Given a symmetric matrix of unnormalized weights, this function will calculate corresponding values for the \( C \) and \( M \) arguments suitable for use in the \texttt{dcar_proper} distribution. This function can be used to transition between usage of \texttt{dcar_normal} and \texttt{dcar_proper}, since \texttt{dcar_normal} uses the \texttt{adj}, \texttt{weights}, and \texttt{num} arguments, while \texttt{dcar_proper} requires \texttt{adj}, \texttt{num}, and also the \( C \) and \( M \) as returned by this function.

Here, \( C \) is a sparse vector representation of the row-normalized adjacency matrix, and \( M \) is a vector containing the conditional variance for each region. The resulting values of \( C \) and \( M \) are guaranteed to satisfy the symmetry constraint imposed on \( C \) and \( M \), that \( M^{-1}C \) is symmetric, where \( M \) is a diagonal matrix and \( C \) is the row-normalized adjacency matrix.

**Value**

A named list with elements \( C \) and \( M \). These may be used as the \( C \) and \( M \) arguments to the \texttt{dcar_proper} distribution.
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

The automated parameter blocking algorithm is no longer actively maintained. In some cases, it may not operate correctly with more recent system features and/or distributions.

Usage

```r
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-specified 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 specifying 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

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.

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


See Also

configureMCMC buildMCMC
**buildAGHQuadGrid**

*Build Adaptive Gauss-Hermite Quadrature Grid*

**Description**

Create quadrature grid for use in AGHQuad methods in Nimble.

**Arguments**

- **d** Dimension of quadrature grid being requested.
- **nQuad** Number of quadrature nodes requested on build.

**Details**

This function is used by buildOneAGHQuad1D and buildOneAGHQuad to create the quadrature grid using adaptive Gauss-Hermite quadrature. Handles single or multiple dimension grids and computes both grid locations and weights. Additionally, acts as a cache system to do transformations, and return marginalized log density.

Any of the input node vectors, when provided, will be processed using `nodes <- model$expandNodeNames(nodes)`, where `nodes` may be `paramNodes`, `randomEffectsNodes`, and so on. This step allows any of the inputs to include node-name-like syntax that might contain multiple nodes. For example, `paramNodes = 'beta[1:10]'` can be provided if there are actually 10 scalar parameters, 'beta[1]' through 'beta[10]'. The actual node names in the model will be determined by the `expandNodeNames` step.

Available methods include:

- **buildAGHQ**. Builds a adaptive Gauss-Hermite quadrature grid in d dimensions. Calls `buildAGHQuad1D` to build the one dimensional grid and then expands in each dimension. Some numerical issues occur in Eigen decomposition making the grid weights only accurate up to 35 quadrature nodes.
- Options to get internally cached values are `getGridSize`, `getModeIndex` for when there are an odd number of quadrature nodes, `getLogDensity` for the cached values, `getAllNodes` for the quadrature grids, `getNodes` for getting a single indexed nodes, `getAllNodesTransformed` for nodes transformed to the parameter scale, `getNodesTransformed` for a single transformed node, `getAllWeights` to get all quadrature weights, and `getWeights` single indexed weight.
- **transformGrid(cholNegHess, inner_mode, method)** transforms the grid using either cholesky transformations, as default, or spectral that makes use of the Eigen decomposition. For a single dimension `transformGrid1D` is used.
- `setGridSize` builds the grid the initial time and is only run once in code. After, the user must choose to `setGridSize` to update the grid size.
- `check`. If TRUE (default), a warning is issued if `paramNodes`, `randomEffectsNodes` and/or `calcNodes` are provided but seek to have missing elements or unnecessary elements based on some default inspection of the model. If unnecessary warnings are emitted, simply set `check=FALSE`. 

• **innerOptimControl**. A list of control parameters for the inner optimization of Laplace approximation using `optim`. See 'Details' of `optim` for further information.

• **innerOptimMethod**. Optimization method to be used in `optim` for the inner optimization. See 'Details' of `optim`. Currently `optim` in NIMBLE supports: "Nelder-Mead", "BFGS", "CG", and "L-BFGS-B". By default, method "CG" is used when marginalizing over a single (scalar) random effect, and "BFGS" is used for multiple random effects being jointly marginalized over.

• **innerOptimStart**. Choice of starting values for the inner optimization. This could be "last", "last.best", or a vector of user provided values. "last" means the most recent random effects values left in the model will be used. When finding the MLE, the most recent values will be the result of the most recent inner optimization for Laplace. "last.best" means the random effects values corresponding to the largest Laplace likelihood (from any call to the `calcLaplace` or `calcLogLik` method, including during an MLE search) will be used (even if it was not the most recent Laplace likelihood). By default, the initial random effects values will be used for inner optimization.

• **outOptimControl**. A list of control parameters for maximizing the Laplace log-likelihood using `optim`. See 'Details' of `optim` for further information.

---

**References**


---

**buildAuxiliaryFilter**

**Placeholder for buildAuxiliaryFilter**

---

**Description**

This function has been moved to the 'nimbleSMC' package.

**Usage**

`buildAuxiliaryFilter(...)`

**Arguments**

... arguments
**buildBootstrapFilter**  
*Placeholder for buildBootstrapFilter*

---

**Description**

This function has been moved to the `nimbleSMC` package.

**Usage**

```r
buildBootstrapFilter(...)  
```

**Arguments**

```r
...  
```

---

**buildEnsembleKF**  
*Placeholder for buildEnsembleKF*

---

**Description**

This function has been moved to the `nimbleSMC` package.

**Usage**

```r
buildEnsembleKF(...)  
```

**Arguments**

```r
...  
```

---

**buildIteratedFilter2**  
*Placeholder for buildIteratedFilter2*

---

**Description**

This function has been moved to the `nimbleSMC` package.

**Usage**

```r
buildIteratedFilter2(...)  
```

**Arguments**

```r
...  
```
**Description**

Build a Laplace or AGHQ approximation algorithm for a given NIMBLE model.

**Usage**

```r
buildLaplace(
  model, 
  paramNodes, 
  randomEffectsNodes, 
  calcNodes, 
  calcNodesOther, 
  control = list()
)
```

```r
buildAGHQ(
  model, 
  nQuad = 1, 
  paramNodes, 
  randomEffectsNodes, 
  calcNodes, 
  calcNodesOther, 
  control = list()
)
```

**Arguments**

- `model`: a NIMBLE model object, such as returned by `nimbleModel`. The model must have automatic derivatives (AD) turned on, e.g. by using `buildDerivs=TRUE` in `nimbleModel`.
- `paramNodes`: a character vector of names of parameter nodes in the model; defaults are provided by `setupMargNodes`. Alternatively, `paramNodes` can be a list in the format returned by `setupMargNodes`, in which case `randomEffectsNodes`, `calcNodes`, and `calcNodesOther` are not needed (and will be ignored).
- `randomEffectsNodes`: a character vector of names of continuous unobserved (latent) nodes to marginalize (integrate) over using Laplace approximation; defaults are provided by `setupMargNodes`.
- `calcNodes`: a character vector of names of nodes for calculating the integrand for Laplace approximation; defaults are provided by `setupMargNodes`. There may be deterministic nodes between `paramNodes` and `calcNodes`. These will be included in calculations automatically and thus do not need to be included in `calcNodes` (but there is no problem if they are).
calcNodesOther  a character vector of names of nodes for calculating terms in the log-likelihood that do not depend on any randomEffectsNodes, and thus are not part of the marginalization, but should be included for purposes of finding the MLE. This defaults to stochastic nodes that depend on paramNodes but are not part of and do not depend on randomEffectsNodes. There may be deterministic nodes between paramNodes and calcNodesOther. These will be included in calculations automatically and thus do not need to be included in calcNodesOther (but there is no problem if they are).

control  a named list for providing additional settings used in Laplace approximation. See control section below. Most of these can be updated later with the ‘updateSettings’ method.

nQuad  number of quadrature points for AGHQ (in one dimension). Laplace approximation is AGHQ with ‘nQuad=1’. Only odd numbers of nodes really make sense. Often only one or a few nodes can achieve high accuracy. A maximum of 35 nodes is supported. Note that for multivariate quadratures, the number of nodes will be (number of dimensions)^nQuad.

buildLaplace creates an object that can run Laplace approximation and for a given model or part of a model. buildAGHQ creates an object that can run adaptive Gauss-Hermite quadrature (AGHQ, sometimes called "adaptive Gaussian quadrature") for a given model or part of a model. Laplace approximation is AGHQ with one quadrature point, hence ‘buildLaplace’ simply calls ‘buildAGHQ’ with ‘nQuad=1’. These methods approximate the integration over continuous random effects in a hierarchical model to calculate the (marginal) likelihood.

buildAGHQ and buildLaplace will by default (unless changed manually via ‘control$split’) determine from the model which random effects can be integrated over (marginalized) independently. For example, in a GLMM with a grouping factor and an independent random effect intercept for each group, the random effects can be marginalized as a set of univariate approximations rather than one multivariate approximation. On the other hand, correlated or nested random effects would require multivariate marginalization.

Maximum likelihood estimation is available for Laplace approximation (‘nQuad=1’) with univariate or multivariate integrations. With ‘nQuad > 1’, maximum likelihood estimation is available only if all integrations are univariate (e.g., a set of univariate random effects). If there are multivariate integrations, these can be calculated at chosen input parameters but not maximized over parameters. For example, one can find the MLE based on Laplace approximation and then increase ‘nQuad’ (using the ‘updateSettings’ method below) to check on accuracy of the marginal log likelihood at the MLE.

Beware that quadrature will use ‘nQuad^k’ quadrature points, where ‘k’ is the dimension of each integration. Therefore quadrature for ‘k’ greater than 2 or 3 can be slow. As just noted, ‘buildAGHQ’ will determine independent dimensions of quadrature, so it is fine to have a set of univariate random effects, as these will each have k=1. Multivariate quadrature (k>1) is only necessary for nested, correlated, or otherwise dependent random effects.

The recommended way to find the maximum likelihood estimate and associated outputs is by calling runLaplace or runAGHQ. The input should be the compiled Laplace or AGHQ algorithm object. This would be produced by running compileNimble with input that is the result of buildLaplace or buildAGHQ.
For more granular control, see below for methods `findMLE` and `summary`. See function `summaryLaplace` for an easier way to call the `summary` method and obtain results that include node names. These steps are all done within `runLaplace` and `runAGHQ`.

The NIMBLE User Manual at r-nimble.org also contains an example of Laplace approximation.

How input nodes are processed

`buildLaplace` and `buildAGHQ` make good tries at deciding what to do with the input model and any (optional) of the node arguments. However, random effects (over which approximate integration will be done) can be written in models in multiple equivalent ways, and customized use cases may call for integrating over chosen parts of a model. Hence, one can take full charge of how different parts of the model will be used.

Any of the input node vectors, when provided, will be processed using `nodes <- model$expandNodeNames(nodes)`, where nodes may be `paramNodes`, `randomEffectsNodes`, and so on. This step allows any of the inputs to include node-name-like syntax that might contain multiple nodes. For example, `paramNodes = 'beta[1:10]'` can be provided if there are actually 10 scalar parameters, 'beta[1]' through 'beta[10]'. The actual node names in the model will be determined by the `expandNodeNames` step.

In many (but not all) cases, one only needs to provide a NIMBLE model object and then the function will construct reasonable defaults necessary for Laplace approximation to marginalize over all continuous latent states (aka random effects) in a model. The default values for the four groups of nodes are obtained by calling `setupMargNodes`, whose arguments match those here (except for a few arguments which are taken from control list elements here).

`setupMargNodes` tries to give sensible defaults from any combination of `paramNodes`, `randomEffectsNodes`, `calcNodes`, and `calcNodesOther` that are provided. For example, if you provide only `randomEffectsNodes` (perhaps you want to marginalize over only some of the random effects in your model), `setupMargNodes` will try to determine appropriate choices for the others.

`setupMargNodes` also determines which integration dimensions are conditionally independent, i.e., which can be done separately from each other. For example, when possible, 10 univariate random effects will be split into 10 univariate integration problems rather than one 10-dimensional integration problem.

The defaults make general assumptions such as that `randomEffectsNodes` have `paramNodes` as parents. However, The steps for determining defaults are not simple, and it is possible that they will be refined in the future. It is also possible that they simply don’t give what you want for a particular model. One example where they will not give desired results can occur when random effects have no prior parameters, such as 'N(0,1)' nodes that will be multiplied by a scale factor (e.g. sigma) and added to other explanatory terms in a model. Such nodes look like top-level parameters in terms of model structure, so you must provide a `randomEffectsNodes` argument to indicate which they are.

It can be helpful to call `setupMargNodes` directly to see exactly how nodes will be arranged for Laplace approximation. For example, you may want to verify the choice of `randomEffectsNodes` or get the order of parameters it has established to use for making sense of the MLE and results from the summary method. One can also call `setupMargNodes`, customize the returned list, and then provide that to `buildLaplace` as `paramNodes`. In that case, `setupMargNodes` will not be called (again) by `buildLaplace`.

If `setupMargNodes` is emitting an unnecessary warning, simply use `control=list(check=FALSE)`. 


Managing parameter transformations that may be used internally

If any paramNodes (parameters) or randomEffectsNodes (random effects / latent states) have constraints on the range of valid values (because of the distribution they follow), they will be used on a transformed scale determined by parameterTransform. This means the Laplace approximation itself will be done on the transformed scale for random effects and finding the MLE will be done on the transformed scale for parameters. For parameters, prior distributions are not included in calculations, but they are used to determine valid parameter ranges and hence to set up any transformations. For example, if sigma is a standard deviation, you can declare it with a prior such as sigma ~ dhalfflat() to indicate that it must be greater than 0.

For default determination of when transformations are needed, all parameters must have a prior distribution simply to indicate the range of valid values. For a param p that has no constraint, a simple choice is p ~ dflat().

Understanding inner and outer optimizations

Note that there are two numerical optimizations when finding maximum likelihood estimates with a Laplace or (1D) AGHQ algorithm: (1) maximizing the joint log-likelihood of random effects and data given a parameter value to construct the approximation to the marginal log-likelihood at the given parameter value; (2) maximizing the approximation to the marginal log-likelihood over the parameters. In what follows, the prefix 'inner' refers to optimization (1) and 'outer' refers to optimization (2). Currently both optimizations default to using method "BFGS". However, one can use other optimizers or simply run optimization (2) manually from R; see the example below. In some problems, choice of inner and/or outer optimizer can make a big difference for obtaining accurate results, especially for standard errors. Hence it is worth experimenting if one is in doubt.

control list arguments

The control list allows additional settings to be made using named elements of the list. Most (or all) of these can be updated later using the 'updateSettings' method. Supported elements include:

- split. If TRUE (default), randomEffectsNodes will be split into conditionally independent sets if possible. This facilitates more efficient Laplace or AGHQ approximation because each conditionally independent set can be marginalized independently. If FALSE, randomEffectsNodes will be handled as one multivariate block, with one multivariate approximation. If split is a numeric vector, randomEffectsNodes will be split by calling split(randomEffectsNodes, control$split). The last option allows arbitrary control over how randomEffectsNodes are blocked.

- check. If TRUE (default), a warning is issued if paramNodes, randomEffectsNodes and/or calcNodes are provided but seem to have missing or unnecessary elements based on some default inspections of the model. If unnecessary warnings are emitted, simply set check=FALSE.

- innerOptimControl. A list (either an R list or a ‘optimControlNimbleList’) of control parameters for the inner optimization of Laplace approximation using nimOptim. See 'Details' of nimOptim for further information. Default is ‘nimOptimDefaultControl()’.

- innerOptimMethod. Optimization method to be used in nimOptim for the inner optimization. See 'Details' of nimOptim. Currently nimOptim in NIMBLE supports: "Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "nlminb", and user-provided optimizers. By default, method "BFGS" is used for both univariate and multivariate cases. For "nlminb" or user-provided
optimizers, only a subset of elements of the innerOptimControlList are supported. (Note that control over the outer optimization method is available as an argument to `findMLE`). Choice of optimizers can be important and so can be worth exploring.

- **innerOptimStart.** Method for determining starting values for the inner optimization. Options are:
  - "zero" (default): use all zeros;
  - "last": use the result of the last inner optimization;
  - "last.best": use the result of the best inner optimization so far for each conditionally independent part of the approximation;
  - "constant": always use the same values, determined by innerOptimStartValues;
  - "random": randomly draw new starting values from the model (i.e., from the prior);
  - "model": use values for random effects stored in the model, which are determined from the first call.

Note that "model" and "zero" are shorthand for "constant" with particular choices of innerOptimStartValues. Note that "last" and "last.best" require a choice for the very first values, which will come from innerOptimStartValues. The default is innerOptimStart="zero" and may change in the future.

- **innerOptimStartValues.** Values for some of innerOptimStart approaches. If a scalar is provided, that value is used for all elements of random effects for each conditionally independent set. If a vector is provided, it must be the length of *all* random effects. If these are named (by node names), the names will be used to split them correctly among each conditionally independent set of random effects. If they are not named, it is not always obvious what the order should be because it may depend on the conditionally independent sets of random effects. It should match the order of names returned as part of 'summaryLaplace'.

- **innerOptimWarning.** If FALSE (default), do not emit warnings from the inner optimization. Optimization methods may sometimes emit a warning such as for bad parameter values encountered during the optimization search. Often, a method can recover and still find the optimum. In the approximations here, sometimes the inner optimization search can fail entirely, yet the outer optimization see this as one failed parameter value and can recover. Hence, it is often desirable to silence warnings from the inner optimizer, and this is done by default. Set innerOptimWarning=TRUE to see all warnings.

- **useInnerCache.** If TRUE (default), use caching system for efficiency of inner optimizations. The caching system records one set of previous parameters and uses the corresponding results if those parameters are used again (e.g., in a gradient call). This should generally not be modified.

- **outerOptimControl.** A list of control parameters for maximizing the Laplace log-likelihood using nimOptim. See 'Details' of nimOptim for further information.

- **computeMethod.** There are three approaches available for internal details of how the approximations, and specifically derivatives involved in their calculation, are handled. These are labeled simply 1, 2, and 3, and the default is 2. The relative performance of the methods will depend on the specific model. Users wanting to explore efficiency can try switching from method 2 (default) to methods 1 or 3 and comparing performance. The first Laplace approximation with each method will be (much) slower than subsequent Laplace approximations. Further details are not provided at this time.
• gridType (relevant only nQuad>1). For multivariate AGHQ, a grid must be constructed based on the Hessian at the inner mode. Options include "cholesky" (default) and "spectral" (i.e., eigenvectors and eigenvalues) for the corresponding matrix decompositions on which the grid can be based.

# end itemize

Available methods

The object returned by buildLaplace is a nimbleFunction object with numerous methods (functions). Here these are described in three tiers of user relevance.

Most useful methods

The most relevant methods to a user are:

• calcLogLik(p, trans=FALSE). Calculate the approximation to the marginal log-likelihood function at parameter value p, which (if trans is FALSE) should match the order of paramNodes. For any non-scalar nodes in paramNodes, the order within the node is column-major. The order of names can be obtained from method getNodeNamesVec(TRUE). Return value is the scalar (approximate, marginal) log likelihood.

If trans is TRUE, then p is the vector of parameters on the transformed scale, if any, described above. In this case, the parameters on the original scale (as the model was written) will be determined by calling the method pInverseTransform(p). Note that the length of the parameter vector on the transformed scale might not be the same as on the original scale (because some constraints of non-scalar parameters result in fewer free transformed parameters than original parameters).

• calcLaplace(p, trans). This is the same as calcLogLik but requires that the approximation be Laplace (i.e nQuad is 1), and results in an error otherwise.

• findMLE(pStart, method, hessian). Find the maximum likelihood estimates of parameters using the approximated marginal likelihood. This can be used if nQuad is 1 (Laplace case) or if nQuad>1 and all marginalizations involve only univariate random effects. Arguments include pStart: initial parameter values (defaults to parameter values currently in the model); method: (outer) optimization method to use in nimOptim (defaults to "BFGS", although some problems may benefit from other choices); and hessian: whether to calculate and return the Hessian matrix (defaults to TRUE, which is required for subsequent use of ‘summary’ method). Second derivatives in the Hessian are determined by finite differences of the gradients obtained by automatic differentiation (AD). Return value is a nimbleList of type optimResultNimbleList, similar to what is returned by R’s optim. See help(nimOptim). Note that parameters (‘par’) are returned for the natural parameters, i.e. how they are defined in the model. But the ‘hessian’, if requested, is computed for the parameters as transformed for optimization if necessary. Hence one must be careful interpreting ‘hessian’ if any parameters have constraints, and the safest next step is to use the ‘summary’ method or ‘summaryLaplace’ function.

• summary(MLEoutput, originalScale, randomEffectsStdError, jointCovariance). Summarize the maximum likelihood estimation results, given object MLEoutput that was returned by findMLE. The summary can include a covariance matrix for the parameters, the random effects, or both, and these can be returned on the original parameter scale or on the (potentially) transformed scale(s) used in estimation. It is often preferred instead to call function
(not method) 'summaryLaplace' because this will attach parameter and random effects names (i.e., node names) to the results.

In more detail, summary accepts the following optional arguments:

- `originalScale`. Logical. If TRUE, the function returns results on the original scale(s) of parameters and random effects; otherwise, it returns results on the transformed scale(s). If there are no constraints, the two scales are identical. Defaults to TRUE.
- `randomEffectsStdError`. Logical. If TRUE, standard errors of random effects will be calculated. Defaults to FALSE.
- `jointCovariance`. Logical. If TRUE, the joint variance-covariance matrix of the parameters and the random effects will be returned. If FALSE, the variance-covariance matrix of the parameters will be returned. Defaults to FALSE.

The object returned by `summary` is an `AGHQuad_summary` nimbleList with elements:

- `params`. A nimbleList that contains estimates and standard errors of parameters (on the original or transformed scale, as chosen by `originalScale`).
- `randomEffects`. A nimbleList that contains estimates of random effects and, if requested (`randomEffectsStdError=TRUE`) their standard errors, on original or transformed scale. Standard errors are calculated following the generalized delta method of Kass and Steffey (1989).
- `vcov`. If requested (i.e. `jointCovariance=TRUE`), the joint variance-covariance matrix of the parameters and random effects, on original or transformed scale. If `jointCovariance=FALSE`, the covariance matrix of the parameters, on original or transformed scale.
- `scale`. "original" or "transformed", the scale on which results were requested.

Methods for more advanced uses

Additional methods to access or control more details of the Laplace approximation include:

- `updateSettings`. This provides a single function through which many of the settings described above (mostly for the control list) can be later changed. Options that can be changed include: `innerOptimMethod`, `innerOptimStart`, `innerOptimStartValues`, `useInnerCache`, `nQuad`, `gridType`, `innerOptimControl`, `outerOptimControl`, and `computeMethod`. For `innerOptimStart`, method "zero" cannot be specified but can be achieved by choosing method "constant" with `innerOptimStartValues=0`. Only provided options will be modified. The exceptions are `innerOptimControl`, `outerOptimControl`, which are replaced only if `replace_innerOptimControl=TRUE` or `replace_outerOptimControl=TRUE`, respectively.
- `getNodeNamesVec(returnParams)`. Return a vector (>1) of names of parameters/random effects nodes, according to `returnParams = TRUE/FALSE`. Use this if there is more than one node.
- `getNodeNameSingle(returnParams)`. Return the name of a single parameter/random effect node, according to `returnParams = TRUE/FALSE`. Use this if there is only one node.
- `checkInnerConvergence(message)`. Checks whether all internal optimizers converged. Returns a zero if everything converged and one otherwise. If `message = TRUE`, it will print more details about convergence for each conditionally independent set.
- `gr_logLik(p, trans)`. Gradient of the (approximated) marginal log-likelihood at parameter value p. Argument `trans` is similar to that in `calcLaplace`. If there are multiple parameters, the vector `p` is given in the order of parameter names returned by `getNodeNamesVec(returnParams=TRUE)`.
• gr_Laplace(p, trans). This is the same as gr_logLik.
• otherLogLik(p). Calculate the calcNodesOther nodes, which returns the log-likelihood of
the parts of the model that are not included in the Laplace or AGHQ approximation.
• gr_otherLogLik(p). Gradient (vector of derivatives with respect to each parameter) of
otherLogLik(p). Results should match gr_otherLogLik_internal(p) but may be more
efficient after the first call.

Internal or development methods

Some methods are included for calculating the (approximate) marginal log posterior density by in-
cluding the prior distribution of the parameters. This is useful for finding the maximum a posteriori
probability (MAP) estimate. Currently these are provided for point calculations without estimation
methods.

• calcPrior_p(p). Log density of prior distribution.
• calcPrior_pTransformed(pTransform). Log density of prior distribution on transformed
scale, includes the Jacobian.
• calcPostLogDens(p). Marginal log posterior density in terms of the parameter p.
• calcPostLogDens_pTransformed (pTransform). Marginal log posterior density in terms of
the transformed parameter, which includes the Jacobian transformation.
• gr_postLogDens_pTransformed(pTransform). Gradient of marginal log posterior density
on the transformed scale. Other available options that are used in the derivative for more
flexible include logDetJacobian(pTransform) and gr_logDeJacobian(pTransform), as
well as gr_prior(p).

Finally, methods that are primarily for internal use by other methods include:

• gr_logLik_pTransformed. Gradient of the Laplace approximation (calcLogLik_pTransformed(pTransform))
at transformed (unconstrained) parameter value pTransform.
• pInverseTransform(pTransform). Back-transform the transformed parameter value pTransform
to original scale.
• derivs_pInverseTransform(pTransform, order). Derivatives of the back-transformation
(i.e. inverse of parameter transformation) with respect to transformed parameters at pTransform.
Derivative order is given by order (any of 0, 1, and/or 2).
• reInverseTransform(reTrans). Back-transform the transformed random effects value reTrans
to original scale.
• derivs_reInverseTransform(reTrans, order). Derivatives of the back-transformation
(i.e. inverse of random effects transformation) with respect to transformed random effects
at reTrans. Derivative order is given by order (any of 0, 1, and/or 2).
• optimRandomEffects(pTransform). Calculate the optimized random effects given trans-
formed parameter value pTransform. The optimized random effects are the mode of the
conditional distribution of random effects given data at parameters pTransform, i.e. the cal-
culation of calcNodes.
• inverse_negHess(p, reTransform). Calculate the inverse of the negative Hessian matrix of
the joint (parameters and random effects) log-likelihood with respect to transformed random
effects, evaluated at parameter value p and transformed random effects reTransform.
• **hess_logLik_wrt_p_wrt_re(p, reTransform)**. Calculate the Hessian matrix of the joint log-likelihood with respect to parameters and transformed random effects, evaluated at parameter value $p$ and transformed random effects $reTransform$.

• **one_time_fixes()**. Users never need to run this. Is called when necessary internally to fix dimensionality issues if there is only one parameter in the model.

• **calcLogLik_pTransformed(pTransform)**. Laplace approximation at transformed (unconstrained) parameter value $pTransform$. To make maximizing the Laplace likelihood unconstrained, an automated transformation via `parameterTransform` is performed on any parameters with constraints indicated by their priors (even though the prior probabilities are not used).

• **gr_otherLogLik_internal(p)**. Gradient (vector of derivatives with respect to each parameter) of `otherLogLik(p)`. This is obtained using automatic differentiation (AD) with single-taping. First call will always be slower than later calls.

• **cache_outer_logLik(logLikVal)**. Save the marginal log likelihood value to the inner Laplace marginalization functions to track the outer maximum internally.

• **reset_outer_inner_logLik()**. Reset the internal saved maximum marginal log likelihood.

• **get_inner_cholesky(atOuterMode = integer(0, default = 0))**. Returns the cholesky of the negative Hessian with respect to the random effects. If `atOuterMode = 1` then returns the value at the overall best marginal likelihood value, otherwise `atOuterMode = 0` returns the last.

• **get_inner_mode(atOuterMode = integer(0, default = 0))**. Returns the mode of the random effects for either the last call to the inner quadrature functions (`atOuterMode = 0`), or the last best value for the marginal log likelihood, `atOuterMode = 1`.

**Author(s)**

Wei Zhang, Perry de Valpine, Paul van Dam-Bates

**References**


**Examples**

```r
pumpCode <- nimbleCode(
  for (i in 1:N){
    theta[i] ~ dgamma(alpha, beta)
    lambda[i] <- theta[i] * t[i]
    x[i] ~ dpois(lambda[i])
  })
```


```r
pump <- nimbleModel(code = pumpCode, name = "pump", constants = pumpConsts, 
                    data = pumpData, inits = pumpInits, buildDerivs = TRUE)

# Build Laplace approximation
pumpLaplace <- buildLaplace(pump)

## Not run:
# Compile the model
Cpump <- compileNimble(pump)
CpumpLaplace <- compileNimble(pumpLaplace, project = pump)
# Calculate MLEs of parameters
MLERes <- CpumpLaplace$findMLE()
# Calculate estimates and standard errors for parameters and random effects on original scale
allres <- CpumpLaplace$summary(MLERes, randomEffectsStdError = TRUE)

# Change the settings and also illustrate runLaplace
CpumpLaplace$updateSettings(innerOptimMethod = "nlminb", outerOptimMethod = "nlminb")
newres <- runLaplace(CpumpLaplace)

# Illustrate use of the component log likelihood and gradient functions to
# run an optimizer manually from R.
# Use nlminb to find MLEs
MLERes.manual <- nlminb(c(0.1, 0.1),
                        function(x) -CpumpLaplace$calcLogLik(x),
                        function(x) -CpumpLaplace$gr_Laplace(x))

## End(Not run)
```

---

**buildLiuWestFilter**  
*Placeholder for buildLiuWestFilter*

### Description

This function has been moved to the 'nimbleSMC' package.

### Usage

```r
buildLiuWestFilter(...)```

### Arguments

```r
... arguments```
buildMCEM

Builds an MCEM algorithm for a given NIMBLE model

Description

Takes a NIMBLE model (with some missing data, aka random effects or latent state nodes) and builds a Monte Carlo Expectation Maximization (MCEM) algorithm for maximum likelihood estimation. The user can specify which latent nodes are to be integrated out in the E-Step, or default choices will be made based on model structure. All other stochastic non-data nodes will be maximized over. The E-step is done with a sample from a nimble MCMC algorithm. The M-step is done by a call to optim.

Usage

buildMCEM(
  model,
  paramNodes,
  latentNodes,
  calcNodes,
  calcNodesOther,
  control = list(),
  ...
)

Arguments

model
  a NIMBLE model object, either compiled or uncompiled.

paramNodes
  a character vector of names of parameter nodes in the model; defaults are provided by setupMargNodes. Alternatively, paramNodes can be a list in the format returned by setupMargNodes, in which case latentNodes, calcNodes, and calcNodesOther are not needed (and will be ignored).

latentNodes
  a character vector of names of unobserved (latent) nodes to marginalize (sum or integrate) over; defaults are provided by setupMargNodes (as the randomEffectsNodes in its return list).

calcNodes
  a character vector of names of nodes for calculating components of the full-data likelihood that involve latentNodes; defaults are provided by setupMargNodes. There may be deterministic nodes between paramNodes and calcNodes. These will be included in calculations automatically and thus do not need to be included in calcNodes (but there is no problem if they are).

calcNodesOther
  a character vector of names of nodes for calculating terms in the log-likelihood that do not depend on any latentNodes, and thus are not part of the marginalization, but should be included for purposes of finding the MLE. This defaults to stochastic nodes that depend on paramNodes but are not part of and do not depend on latentNodes. There may be deterministic nodes between paramNodes and calcNodesOther. These will be included in calculations automatically and thus do not need to be included in calcNodesOther (but there is no problem if they are).
buildMCEM

control

- a named list for providing additional settings used in MCEM. See control section below.
- provided only as a means of checking if a user is using the deprecated interface to `buildMCEM` in nimble versions < 1.2.0.

Details

buildMCEM is a nimbleFunction that creates an MCEM algorithm for a model and choices (perhaps default) of nodes in different roles in the model. The MCEM can then be compiled for fast execution with a compiled model.

Note that `buildMCEM` was re-written for nimble version 1.2.0 and is not backward-compatible with previous versions. The new version is considered to be in beta testing.

Denote data by Y, latent states (or missing data) by X, and parameters by T. MCEM works by the following steps, starting from some T:

1. Draw a sample of size M from P(X | Y, T) using MCMC.
2. Update T to be the maximizer of E[log P(X, Y | T)] where the expectation is approximated as a Monte Carlo average over the sample from step(1)
3. Repeat until converged.

The default version of MCEM is the ascent-based MCEM of Caffo et al. (2015). This attempts to update M when necessary to ensure that step 2 really moves uphill given that it is maximizing a Monte Carlo approximation and could accidently move downhill on the real surface of interest due to Monte Carlo error. The main tuning parameters include alpha, beta, gamma, Mfactor, C, and tol (tolerance).

If the model supports derivatives via nimble’s automatic differentiation (AD) (and buildDerivs=TRUE in nimbleModel), the maximization step can use gradients from AD. You must manually set useDerivs=FALSE in the control list if derivatives aren’t supported or if you don’t want to use them.

In the ascent-based method, after maximization in step 2, the Monte Carlo standard error of the uphill movement is estimated. If the standardized uphill step is bigger than 0 with Type I error rate alpha, the iteration is accepted and the algorithm continues. Otherwise, it is not certain that step 2 really moved uphill due to Monte Carlo error, so the MCMC sample size M is incremented by a fixed factor (e.g. 0.33 or 0.5, called Mfactor in the control list), the additional samples are added by continuing step 1, and step 2 is tried again. If the Monte Carlo noise still overwhelms the magnitude of uphill movement, the sample size is increased again, and so on. alpha should be between 0 and 0.5. A larger value than usually used for inference is recommended so that there is an easy threshold to determine uphill movement, which avoids increasing M prematurely. M will never be increased above maxM.

Convergence is determined in a similar way. After a definite move uphill, we determine if the uphill increment is less than tol, with Type I error rate gamma. (But if M hits a maximum value, the convergence criterion changes. See below.)

beta is used to help set M to a minimal level based on previous iterations. This is a desired Type II error rate, assuming an uphill move and standard error based on the previous iteration. Set adjustM=FALSE in the control list if you don’t want this behavior.

There are some additional controls on convergence for practical purposes. Set C in the control list to be the number of times the convergence criterion must be satisfied in order to actually stop. E.g setting C=2 means there will always be a restart after the first convergence.
One problem that can occur with ascent-based MCEM is that the final iteration can be very slow if \( M \) must become very large to satisfy the convergence criterion. Indeed, if the algorithm starts near the MLE, this can occur. Set \( \text{maxM} \) in the control list to set the MCMC sample size that should never be exceeded.

If \( M = \text{maxM} \), a softer convergence criterion is used. This second convergence criterion is to stop if we can’t be sure we moved uphill using Type I error rate delta. This is a soft criterion because for small delta, Type II errors will be common (e.g. if we really did move uphill but can’t be sure from the Monte Carlo sample), allowing the algorithm to terminate. One can continue the algorithm from where it stopped, so it is helpful to not have it get stuck when having a very hard time with the first (stricter) convergence criterion.

All of \( \alpha \), \( \beta \), \( \delta \), and \( \gamma \) are utilized based on asymptotic arguments but in practice must be chosen heuristically. In other words, their theoretical basis does not exactly yield practical advice on good choices for efficiency and accuracy, so some trial and error will be needed.

It can also be helpful to set a minimum and maximum of allowed iterations (of steps 1 and 2 above). Setting \( \text{minIter} > 1 \) in the control list can sometimes help avoid a false convergence on the first iteration by forcing at least one more iteration. Setting \( \text{maxIter} \) provides a failsafe on a stuck run.

If you don’t want the ascent-based method at all and simply want to run a set of iterations, set \( \text{ascent} = \text{FALSE} \) in the control list. This will use the second (softer) convergence criterion.

Parameters to be maximized will by default be handled in an unconstrained parameter space, transformed if necessary by a \texttt{parameterTransform} object. In that case, the default \texttt{optim} method will be "BFGS" and can be changed by setting \texttt{optimMethod} in the control list. Set \texttt{useTransform=FALSE} in the control list if you don’t want the parameters transformed. In that case the default \texttt{optimMethod} will be "L-BFGS-B" if there are any actual constraints, and you can provide a list of \texttt{boxConstraints} in the control list. (Constraints may be determined by priors written in the model for parameters, even though their priors play no other role in MLE. E.g. \( \text{sigma} \sim \text{halfflat}() \) indicates \( \text{sigma} > 0 \).

Most of the control list elements can be overridden when calling the \texttt{findMLE} method. The \texttt{findMLE} argument \texttt{continue=TRUE} results in attempting to continue the algorithm where the previous call finished, including whatever settings were in use.

See \texttt{setupMargNodes} (which is called with the given arguments for \texttt{paramNodes}, \texttt{calcNodes}, and \texttt{calcNodesOther}; and with \texttt{allowDiscreteLatent=TRUE}, \texttt{randomEffectsNodes=latentNodes}, and \texttt{check=check}) for more about how the different groups of nodes are determined. In general, you can provide none, one, or more of the different kinds of nodes and \texttt{setupMargNodes} will try to determine the others in a sensible way. However, note that this cannot work for all ways of writing a model. One key example is that if random (latent) nodes are written as top-level nodes (e.g. following \( N(0,1) \)), they appear structurally to be parameters and you must tell \texttt{buildMCEM} that they are \texttt{latentNodes}. The various "Nodes" arguments will all be passed through \texttt{model$expandNodeNames}, allowing for example simply "x" to be provided when there are many nodes within "x".

Estimating the Monte Carlo standard error of the uphill step is not trivial because the sample was obtained by MCMC and so likely is autocorrelated. This is done by calling whatever function in R’s global environment is called "MCEM_mcse". which is required to take two arguments: \texttt{samples} (which will be a vector of the differences in \( \log(P(Y, X | T)) \) between the new and old values of \( T \), across the sample of \( X \)) and \( m \), the sample size. It must return an estimate of the standard error of the mean of the sample. \texttt{NIMBLE} provides a default version (exported from the package namespace),
which calls mcmcse::mcse with method "obm". Simply provide a different function with this name in your R session to override NIMBLE’s default.

Control list details
The control list accepts the following named elements:

- **initM** initial MCMC sample size, M. Default=1000.
- **Mfactor** Factor by which to increase MCMC sample size when step 2 results in noise overwhelming the uphill movement. The new M will be \(1+\text{Mfactor} \times M\) (rounded up). Mfactor is \(1/k\) of Caffo et al. (2015). Default=1/3.
- **maxM** Maximum allowed value of M (see above). Default=initM*20.
- **burnin** Number of burn-in iterations for the MCMC in step 1. Note that the initial states of one MCMC will be the last states from the previous MCMC, so they will often be good initial values after multiple iterations. Default=500.
- **thin** Thinning interval for the MCMC in step 1. Default=1.
- **alpha** Type I error rate for determining when step 2 has moved uphill. See above. Default=0.25.
- **beta** Used for determining a minimal value of $MS$ based on previous iteration, if adjustM is TRUE. beta is a desired Type II error rate for determining uphill moves. Default=0.25.
- **delta** Type I error rate for the soft convergence approach (second approach above). Default=0.25.
- **gamma** Type I error rate for determining when step 2 has moved less than tol uphill, in which case ascent-based convergence is achieved (first approach above). Default=0.05.
- **buffer** A small amount added to lower box constraints and subtracted from upper box constraints for all parameters, relevant only if useTransform=FALSE and some parameters do have boxConstraints set or have bounds that can be determined from the model. Default=1e-6.
- **tol** Ascent-based convergence tolerance. Default=0.001.
- **ascent** Logical to determine whether to use the ascent-based method of Caffo et al. Default=TRUE.
- **C** Number of convergences required to actually stop the algorithm. Default = 1.
- **maxIter** Maximum number of MCEM iterations to run.
- **minIter** Minimum number of MCEM iterations to run.
- **adjustM** Logical indicating whether to see if M needs to be increased based on statistical power argument in each iteration (using beta). Default=TRUE.
- **verbose** Logical indicating whether verbose output is desired. Default=TRUE.
- **MCMCprogressBar** Logical indicating whether MCMC progress bars should be shown for every iteration of step 1. This argument is passed to configureMCMC, or to config if provided. Default=TRUE.
- **derivsDelta** If AD derivatives are not used, then the method vcov must use finite difference derivatives to implement the method of Louis (1982). The finite differences will be delta or delta/2 for various steps. This is the same for all dimensions. Default=0.0001.
• `mcmcControl` This is passed to `configureMCMC`, or `configureMCMC` if provided, as the `control` argument, i.e. `control=mcmcControl`.

• `boxConstraints` List of box constraints for the nodes that will be maximized over, only relevant if `useTransform=FALSE` and `forceNoConstraints=FALSE` (and ignored otherwise). 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. Any nodes that are not given constrains will have their constraints automatically determined by NIMBLE. See above. Default=list().

• `forceNoConstraints` Logical indicating whether to force ignoring constraints even if they might be necessary. Default=FALSE.

• `useTransform` Logical indicating whether to use a parameter transformation (see `parameterTransform`) to create an unbounded parameter space for the `paramNodes`. This allows unconstrained maximization algorithms to be used. Default=TRUE.

• `check` Logical passed as the `check` argument to `setupMargNodes`. Default=TRUE.

• `useDerivs` Logical indicating whether to use AD. If TRUE, the model must have been build with ‘`buildDerivs=TRUE`’. It is not automatically determined from the model whether derivatives are supported. Default=TRUE.

• `config` Function to create the MCMC configuration used for step 1. The MCMC configuration is created by calling

\[
\text{config(model, nodes = latentNodes, monitors = latentNodes, thin = thinDefault, control = mcmcControl, print = FALSE)}
\]

The default for `config` (if it is missing) is `configureMCMC`, which is nimble’s general default MCMC configuration function.

### Methods in the returned algorithm

The object returned by `buildMCEM` is a `nimbleFunction` object with the following methods

• `findMLE` is the main method of interest, launching the MCEM algorithm. It takes the following arguments:
  – `pStart`. Vector of initial parameter values. If omitted, the values currently in the model object are used.
  – `returnTrans`. Logical indicating whether to return parameters in the transformed space, if a parameter transformation is in use. Default=FALSE.
  – `continue`. Logical indicating whether to continue the MCEM from where the last call stopped. In addition, if TRUE, any other control setting provided in the last call will be used again. If FALSE, all control settings are reset to the values provided when `buildMCEM` was called. Any control settings provided in the same call as `continue=TRUE` will over-ride these behaviors and be used in the continued run.
  – All run-time control settings available in the `control` list for `buildMCEM` (except for ‘`buffer`, `boxConstraints`, `forceNoConstraints`, `useTransform`, and `useDerivs`) are accepted as individual arguments to over-ride the values provided in the control list.

`findMLE` returns an object of class `optimResultNimbleList` with the results of the final optimization of step 2. The `par` element of this list is the vector of maximum likelihood (MLE) parameters.
vcov computes the approximate variance-covariance matrix of the MLE using the method of Louis (1982). It takes the following arguments:

- params. Vector of parameters at which to compute the Hessian matrix used to obtain the vcov result. Typically this will be \texttt{MLE$par}, if \texttt{MLE} is the output of \texttt{findMLE}.
- trans. Logical indicating whether params is on the transformed parameter scale, if a parameter transformation is in use. Typically this should be the same as the \texttt{returnTrans} argument to \texttt{findMLE}. Default=FALSE.
- returnTrans. Logical indicating whether the vcov result should be for the transformed parameter space. Default matches trans.
- M. Number of MCMC samples to obtain if \texttt{resetSamples=TRUE}. Default is the final value of M from the last call to \texttt{findMLE}. It can be helpful to increase M to obtain a more accurate vcov result (i.e. with less Monte Carlo noise).
- resetSamples. Logical indicating whether to generate a new MCMC sample from P(X | Y, T), where T is params. If FALSE, the last sample from \texttt{findMLE} will be used. If MLE convergence was reasonable, this sample can be used. However, if the last MCEM step made a big move in parameter space (e.g. if convergence was not achieved), the last MCMC sample may not be accurate for obtaining vcov. Default=FALSE.
- atMLE. Logical indicating whether you believe the params represents the MLE. If TRUE, one part of the computation will be skipped because it is expected to be 0 at the MLE. If there are parts of the model that are not connected to the latent nodes, i.e. of \texttt{calcNodesOther} is not empty, then atMLE will be ignored and set to FALSE. Default=FALSE. It is not really worth using TRUE unless you are confident and the time saving is meaningful, which is not very likely. In other words, this argument is provided for technical completeness.

vcov returns a matrix that is the inverse of the negative Hessian of the log likelihood surface, i.e. the usual asymptotic approximation of the parameter variance-covariance matrix.

- doMCMC. This method runs the MCMC to sample from P(X | Y, T). One does not need to call this, as it is called via the MCEM algorithm in \texttt{findMLE}. This method is provided for users who want to use the MCMC for latent states directly. Samples should be retrieved by \texttt{as.matrix(MCEM$mvSamples)}, where \texttt{MCEM} is the (compiled or uncompiled) MCEM algorithm object. This method takes the following arguments:
  - M. MCMC sample size.
  - thin. MCMC thinning interval.
  - reset. Logical indicating whether to reset the MCMC (passed to the MCMC run method as reset).
- transform and inverseTransform. Convert a parameter vector to an unconstrained parameter space and vice-versa, if \texttt{useTransform=TRUE} in the call to \texttt{buildDerivs}.
- resetControls. Reset all control arguments to the values provided in the call to \texttt{buildMCEM}. The user does not normally need to call this.

Author(s)

Perry de Valpine, Clifford Anderson-Bergman and Nicholas Michaud

References


### Examples

```r
## 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$N))
pumpModel <- nimbleModel(code = pumpCode, name = 'pump',
  constants = pumpConsts, data = pumpData, inits = pumpInits,
  buildDerivs=TRUE)
pumpMCEM <- buildMCEM(model = pumpModel)
CpumpModel <- compileNimble(pumpModel)
CpumpMCEM <- compileNimble(pumpMCEM, project=pumpModel)
MLE <- CpumpMCEM$findMLE()
vcov <- CpumpMCEM$vcov(MLE$par)
## End(Not run)
```

---

**buildMCMC**

Create an MCMC object from a NIMBLE model, or an MCMC configuration object.

**Description**

First required argument, which may be of class `MCMCconf` (an MCMC configuration object), or inherit from class `modelBaseClass` (a NIMBLE model object). Returns an uncompiled executable MCMC object. See details.
Usage

buildMCMC(conf, print, ...)

Arguments

conf  Either an MCMC configuration object of class MCMCconf or a NIMBLE model object. An MCMC configuration object would be returned from configureMCMC and contains information on the model, samplers, monitors, and thinning intervals to be used. Alternatively, conf may a NIMBLE model object, in which case default configuration from calling configureMCMC(model, ...) will be used.

print  A logical argument, specifying whether to print details of the MCMC samplers and monitors.

...  Additional arguments to be passed to configureMCMC if conf is a NIMBLE model object (see help(configureMCMC)).

Details

Calling buildMCMC(conf) will produce an uncompiled MCMC object. The object contains several methods, including the main run function for running the MCMC, a getTimes function for determining the computation time spent in each sampler (see 'getTimes' section below), and functions related to WAIC (getWAIC, getWAICdetails, calculateWAIC (see help(waic)).

The uncompiled run function will have arguments:

niter: The number of iterations to run the MCMC.

nburnin: Number of initial, pre-thinning, MCMC iterations to discard (default = 0).

thin: The thinning interval for the monitors that were specified in the MCMC configuration. If this argument is provided at MCMC runtime, it will take precedence over the thin interval that was specified in the MCMC configuration. If omitted, the thin interval from the MCMC configuration will be used.

thin2: The thinning interval for the second set of monitors (monitors2) that were specified in the MCMC configuration. If this argument is provided at MCMC runtime, it will take precedence over the thin2 interval that was specified in the MCMC configuration. If omitted, the thin2 interval from the MCMC configuration will be used.

reset: Boolean specifying whether to reset the internal MCMC sampling algorithms to their initial state (in terms of self-adapting tuning parameters), and begin recording posterior sample chains anew. Specifying reset = FALSE allows the MCMC algorithm to continue running from where it left off, appending additional posterior samples to the already existing sample chains. Generally, reset = FALSE should only be used when the MCMC has already been run (default = TRUE).

resetMV: Boolean specifying whether to begin recording posterior sample chains anew. This argument is only considered when using reset = FALSE. Specifying reset = FALSE, resetMV = TRUE allows the MCMC algorithm to continue running from where it left off, but without appending the new posterior samples to the already existing samples, i.e. all previously obtained samples will be erased. This option can help reduce memory usage during burn-in (default = FALSE).

resetWAIC: Boolean specifying whether to reset the WAIC summary statistics to their initial states and thereby begin the WAIC calculation anew (default = TRUE). Specifying resetWAIC = FALSE allows the WAIC calculation to continue running from where it left off.
initializeModel: Boolean specifying whether to run the initializeModel routine on the underlying model object, prior to beginning MCMC sampling (default = TRUE).

chain: Integer specifying the MCMC chain number. The chain number is passed to each MCMC sampler's before_chain and after_chain methods. The value for this argument is specified automatically from invocation via runMCMC, and generally need not be supplied when calling mcmc$run (default = 1).

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). The progress bar can be permanently disabled by setting the system option nimbleOptions(MCMCprogressBar = FALSE).

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 or list objects via: as.matrix(mcmc$mvSamples) as.list(mcmc$mvSamples) as.matrix(mcmc$mvSamples2) as.list(mcmc$mvSamples2)

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

Timing the MCMC samplers

If you want to obtain the computation time spent in each sampler, you can set time=TRUE as a run-time argument to run() and then use the method getTimes() to obtain the times.

Calculating WAIC

Please see help(waic) for more information.

Author(s)

Daniel Turek

References


See Also

configureMCMC runMCMC nimbleMCMC
## Examples

```r
## Not run:
code <- nimbleCode(
  mu ~ dnorm(0, 1)
  x ~ dnorm(mu, 1)
  y ~ dnorm(x, 1)
)
Rmodel <- nimbleModel(code, data = list(y = 0))
conf <- configureMCMC(Rmodel, monitors = c('mu', 'x'), enableWAIC = TRUE)
Rmcmc <- buildMCMC(conf)
Cmodel <- compileNimble(Rmodel)
Cmcmc <- compileNimble(Rmcmc, project=Rmodel)

## Running the MCMC with `run`
Cmcmc$run(10000)
samples <- as.matrix(Cmcmc$mvSamples)
samplesAsList <- as.list(Cmcmc$mvSamples)
head(samples)

## Getting WAIC
waicInfo <- Cmcmc$getWAIC()
daicInfo$WAIC
daicInfo$pWAIC

## Timing the samplers (must set `time = TRUE` when running the MCMC)
Cmcmc$run(10000, time = TRUE)
Cmcmc$getTimes()

## End(Not run)
```

### calculateWAIC

**Calculating WAIC using an offline algorithm**

#### Description

In addition to the core online algorithm, NIMBLE implements an offline WAIC algorithm that can be computed on the results of an MCMC. In contrast to NIMBLE’s built-in online WAIC, offline WAIC can compute only conditional WAIC and does not allow for grouping data nodes.

#### Usage

```r
calculateWAIC(mcmc, model, nburnin = 0, thin = 1)
```

#### Arguments

- **mcmc**: An MCMC object (compiled or uncompiled) or matrix or dataframe of MCMC samples as the first argument of calculateWAIC.
model A model (compiled or uncompiled) as the second argument of calculateWAIC. Only required if mcmc is a matrix/dataframe of samples.

nburnin The number of pre-thinning MCMC samples to remove from the beginning of the posterior samples for offline WAIC calculation via calculateWAIC (default = 0). These samples are discarded in addition to any burn-in specified when running the MCMC.

thin Thinning factor interval to apply to the samples for offline WAIC calculation using calculateWAIC (default = 1, corresponding to no thinning).

Details

The ability to calculate WAIC post hoc after all MCMC sampling has been done has certain advantages (e.g., allowing a user to calculate WAIC from MCMC chains run separately) in addition to providing compatibility with versions of NIMBLE before 0.12.0. This functionality includes the ability to call the calculateWAIC function on an MCMC object or matrix of samples after running an MCMC and without setting up the MCMC initially to use WAIC.

Important: The necessary variables to compute WAIC (all stochastic parent nodes of the data nodes) must have been monitored when setting up the MCMC.

See help(waic) for details on using NIMBLE’s recommended online algorithm for WAIC.

Offline WAIC (WAIC computed after MCMC sampling)

As an alternative to online WAIC, NIMBLE also provides a function, calculateWAIC, that can be called on an MCMC object or a matrix of samples, after running an MCMC. This function does not require that one set enableWAIC = TRUE nor WAIC = TRUE when calling runMCMC. The function checks that the necessary variables were monitored in the MCMC and returns an error if they were not. This function behaves identically to the calculateWAIC method of an MCMC object. Note that to use this function when using nimbleMCMC one would need to build the model outside of nimbleMCMC.

The calculateWAIC function requires either an MCMC object or a matrix (or dataframe) of posterior samples plus a model object. In addition, one can provide optional burnin and thin arguments.

In addition, for compatibility with older versions of NIMBLE (prior to v0.12.0), one can also use the calculateWAIC method of the MCMC object to calculate WAIC after all sampling has been completed.

The calculateWAIC() method accepts a single argument, nburnin, equivalent to the nburnin argument of the calculateWAIC function described above.

The calculateWAIC method can only be used if the enableWAIC argument to configureMCMC or to buildMCMC is set to TRUE, or if the NIMBLE option enableWAIC is set to TRUE. If a user attempts to call calculateWAIC without having set enableWAIC = TRUE (either in the call to configureMCMC, or buildMCMC, or as a NIMBLE option), an error will occur.

The calculateWAIC function and method calculate the WAIC based on Equations 5, 12, and 13 in Gelman et al. (2014) (i.e., using pWAIC2).

Note that there is not a unique value of WAIC for a model. The calculateWAIC function and method only provide the conditional WAIC, namely the version of WAIC where all parameters directly involved in the likelihood are treated as theta for the purposes of Equation 5 from Gelman et al. (2014). As a result, the user must set the MCMC monitors (via the monitors argument) to
include all stochastic nodes that are parents of any data nodes; by default the MCMC monitors are only the top-level nodes of the model. For more detail on the use of different predictive distributions, see Section 2.5 from Gelman et al. (2014) or Ariyo et al. (2019). Also note that WAIC relies on a partition of the observations, i.e., 'pointwise' prediction. In calculateWAIC the sum over log pointwise predictive density values treats each data node as contributing a single value to the sum. When a data node is multivariate, that data node contributes a single value to the sum based on the joint density of the elements in the node. Note that if one wants the WAIC calculation via calculateWAIC to be based on the joint predictive density for each group of observations (e.g., grouping the observations from each person or unit in a longitudinal data context), one would need to use a multivariate distribution for the observations in each group (potentially by writing a user-defined distribution).

For more control over and flexibility in how WAIC is calculated, see help(waic).

Author(s)
Joshua Hug and Christopher Paciorek

References


See Also

waic configureMCMC buildMCMC runMCMC nimbleMCMC

Examples

```r
code <- nimbleCode(
  for(j in 1:J) {
    for(i in 1:n) {
      y[j, i] ~ dnorm(mu[j], sd = sigma)
      mu[j] ~ dnorm(mu0, sd = tau)
    }
    tau ~ dunif(0, 10)
    sigma ~ dunif(0, 10)
  })
J <- 5
n <- 10
y <- matrix(rnorm(J*n), J, n)
```
CAR-Normal

The CAR-Normal Distribution

Description

Density function and random generation for the improper (intrinsic) Gaussian conditional autoregressive (CAR) distribution.

Usage

dcar_normal(
  x,
  adj,
  weights = adj/adj,
  num,
  tau,
  c = CAR_calcNumIslands(adj, num),
  zero_mean = 0,
  log = FALSE
)

rcar_normal(
  n = 1,
  adj,
  weights = adj/adj,
  num,
  tau,
  c = CAR_calcNumIslands(adj, num),
  zero_mean = 0
)
Arguments

x | vector of values.
adj | vector of indices of the adjacent locations (neighbors) of each spatial location. This is a sparse representation of the full adjacency matrix.
weights | vector of symmetric unnormalized weights associated with each pair of adjacent locations, of the same length as adj. If omitted, all weights are taken to be one.
num | vector giving the number of neighboring locations of each spatial location, with length equal to the total number of locations.
tau | scalar precision of the Gaussian CAR prior.
c | integer number of constraints to impose on the improper density function. If omitted, c is calculated as the number of disjoint groups of spatial locations in the adjacency structure, which implicitly assumes a first-order CAR process for each group. Note that c should be equal to the number of eigenvalues of the precision matrix that are zero. For example, if the neighborhood structure is based on a second-order Markov random field in one dimension then the matrix has two zero eigenvalues and in two dimensions it has three zero eigenvalues. See Rue and Held (2005) and the NIMBLE User Manual for more information.
zero_mean | integer specifying whether to set the mean of all locations to zero during MCMC sampling of a node specified with this distribution in BUGS code (default 0). This argument is used only in BUGS model code when specifying models in NIMBLE. If 0, the overall process mean is included implicitly in the value of each location in a BUGS model; if 1, then during MCMC sampling, the mean of all locations is set to zero at each MCMC iteration, and a separate intercept term should be included in the BUGS model. Note that centering during MCMC as implemented in NIMBLE follows the ad hoc approach of WinBUGS and does not sample under the constraint that the mean is zero as discussed on p. 36 of Rue and Held (2005). See ‘Details’.
log | logical; if TRUE, probability density is returned on the log scale.
n | number of observations.

Details

When specifying a CAR distribution in BUGS model code, the zero_mean parameter should be specified as either 0 or 1 (rather than TRUE or FALSE).

Note that because the distribution is improper, rcar_normal does not generate a sample from the distribution. However, as discussed in Rue and Held (2005), it is possible to generate a sample from the distribution under constraints imposed based on the eigenvalues of the precision matrix that are zero.

Value

dcar_normal gives the density, while rcar_normal returns the current process values, since this distribution is improper.

Author(s)

Daniel Turek
References

See Also
CAR-Proper, Distributions for other standard distributions

Examples
```r
x <- c(1, 3, 3, 4)
um <- c(1, 2, 2, 1)
adj <- c(2, 1, 3, 2, 4, 3)
weights <- c(1, 1, 1, 1, 1, 1)
lp <- dcar_normal(x, adj, weights, num, tau = 1)
```

---

CAR-Proper  
*The CAR-Proper Distribution*

Description
Density function and random generation for the proper Gaussian conditional autoregressive (CAR) distribution.

Usage
dcar_proper(
x, 
mu, 
C = CAR_calcC(adj, num), 
adj, 
um, 
M = CAR_calcM(num), 
tau, 
gamma, 
evs = CAR_calcEVs3(C, adj, num), 
log = FALSE
)

rcar_proper(
n = 1, 
mu, 
C = CAR_calcC(adj, num), 
adj, 
um, 
M = CAR_calcM(num),
```
\[ \\begin{align*} 
\tau, \\
\gamma, \\
ev = \text{CAR\_calcEVs3}(C, \text{adj}, \text{num}) 
\end{align*} \]

**Arguments**

- **x**: vector of values.
- **mu**: vector of the same length as \( x \), specifying the mean for each spatial location.
- **C**: vector of the same length as \( \text{adj} \), giving the weights associated with each pair of neighboring locations. See ‘Details’.
- **adj**: vector of indices of the adjacent locations (neighbors) of each spatial location. This is a sparse representation of the full adjacency matrix.
- **num**: vector giving the number of neighboring locations of each spatial location, with length equal to the number of locations.
- **M**: vector giving the diagonal elements of the conditional variance matrix, with length equal to the number of locations. See ‘Details’.
- **tau**: scalar precision of the Gaussian CAR prior.
- **gamma**: scalar representing the overall degree of spatial dependence. See ‘Details’.
- **evs**: vector of eigenvalues of the adjacency matrix implied by \( C, \text{adj}, \text{num} \). This parameter should not be provided; it will always be calculated using the adjacency information.
- **log**: logical; if TRUE, probability density is returned on the log scale.
- **n**: number of observations.

**Details**

If both \( C \) and \( M \) are omitted, then all weights are taken as one, and corresponding values of \( C \) and \( M \) are generated.

The \( C \) and \( M \) parameters must jointly satisfy a symmetry constraint: that \( M^{-1} \cdot C \) is symmetric, where \( M \) is a diagonal matrix and \( C \) is the full weight matrix that is sparsely represented by the parameter vector \( C \).

For a proper CAR model, the value of \( \gamma \) must lie within the inverse minimum and maximum eigenvalues of \( M^{-0.5} \cdot C \cdot M^{0.5} \), where \( M \) is a diagonal matrix and \( C \) is the full weight matrix. These bounds can be calculated using the deterministic functions \( \text{carMinBound}(C, \text{adj}, \text{num}, M) \) and \( \text{carMaxBound}(C, \text{adj}, \text{num}, M) \), or simultaneously using \( \text{carBounds}(C, \text{adj}, \text{num}, M) \). In the case where \( C \) and \( M \) are omitted (all weights equal to one), the bounds on \( \gamma \) are necessarily (-1, 1).

**Value**

\( \text{dcar\_proper} \) gives the density, and \( \text{rcar\_proper} \) generates random deviates.

**Author(s)**

Daniel Turek
carBounds

Calculate bounds for the autocorrelation parameter of the dcar_proper distribution

Description

Calculate the lower and upper bounds for the gamma parameter of the dcar_proper distribution

Usage

carBounds(C, adj, num, M)
Arguments

C vector of the same length as adj, giving the normalized weights associated with each pair of neighboring locations.

adj vector of indices of the adjacent locations (neighbors) of each spatial location. This is a sparse representation of the full adjacency matrix.

num vector giving the number of neighboring locations of each spatial location, with length equal to the number of locations.

M vector giving the diagonal elements of the conditional variance matrix, with length equal to the number of locations.

Details

Bounds for gamma are the inverse of the minimum and maximum eigenvalues of: $M^{(-0.5)}CM^{(0.5)}$. The lower and upper bounds are returned in a numeric vector.

Value

A numeric vector containing the bounds (minimum and maximum allowable values) for the gamma parameter of the dcar_proper distribution.

Author(s)

Daniel Turek

See Also

CAR-Proper, carMinBound, carMaxBound

carMaxBound(C, adj, num, M)

Description

Calculate the upper bound for the autocorrelation parameter of the dcar_proper distribution

Usage

carMaxBound(C, adj, num, M)
Arguments

C vector of the same length as adj, giving the normalized weights associated with each pair of neighboring locations.

adj vector of indices of the adjacent locations (neighbors) of each spatial location. This is a sparse representation of the full adjacency matrix.

num vector giving the number of neighboring locations of each spatial location, with length equal to the number of locations.

M vector giving the diagonal elements of the conditional variance matrix, with length equal to the number of locations.

Details

Bounds for gamma are the inverse of the minimum and maximum eigenvalues of \( M^{(-0.5)}CM^{(0.5)}. \)

Value

The upper bound (maximum allowable value) for the gamma parameter of the dcar_proper distribution.

Author(s)

Daniel Turek

See Also

CAR- Proper, carMinBound, carBounds

---

**carMinBound**

*Calculate the lower bound for the autocorrelation parameter of the dcar_proper distribution*

Description

Calculate the lower bound for the gamma parameter of the dcar_proper distribution

Usage

carMinBound(C, adj, num, M)

Arguments

C vector of the same length as adj, giving the normalized weights associated with each pair of neighboring locations.

adj vector of indices of the adjacent locations (neighbors) of each spatial location. This is a sparse representation of the full adjacency matrix.
CAR_calcNumIslands

\[ \text{num} \]
vector giving the number of neighboring locations of each spatial location, with length equal to the number of locations.

\[ M \]
vector giving the diagonal elements of the conditional variance matrix, with length equal to the number of locations.

**Details**

Bounds for \( \gamma \) are the inverse of the minimum and maximum eigenvalues of: \( M(-0.5)CM(0.5) \).

**Value**

The lower bound (minimum allowable value) for the \( \gamma \) parameter of the \( \text{dcar\_proper} \) distribution.

**Author(s)**

Daniel Turek

**See Also**

`CAR-Proper, carMaxBound, carBounds`

---

### CAR_calcNumIslands

*Calculate number of islands based on a CAR adjacency matrix.*

**Description**

Calculate number of islands (distinct connected groups) based on a CAR adjacency matrix.

**Usage**

```r
CAR_calcNumIslands(adj, num)
```

**Arguments**

- **adj**
  vector of indices of the adjacent locations (neighbors) of each spatial location. This is a sparse representation of the full adjacency matrix.

- **num**
  vector giving the number of neighbors of each spatial location, with length equal to the total number of locations.

**Author(s)**

Daniel Turek

**See Also**

`CAR-Normal`
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, internally normalized to sum 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

```r
checkInterrupt()
```

Details

During execution of `nimbleFunction`s that take a long time, it is nice to occasionally 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

ChineseRestaurantProcess

The Chinese Restaurant Process Distribution

Description

Density and random generation for the Chinese Restaurant Process distribution.

Usage

```r
dCRP(x, conc = 1, size, log = 0)
rCRP(n, conc = 1, size)
```

Arguments

- `x` vector of values.
- `conc` scalar concentration parameter.
- `size` integer-valued length of `x` (required).
- `log` logical; if TRUE, probability density is returned on the log scale.
- `n` number of observations (only `n = 1` is handled currently).
The Chinese restaurant process distribution is a distribution on the space of partitions of the positive integers. The distribution with concentration parameter $\alpha$ equal to $\text{conc}$ has probability function

$$f(x_i | x_1, \ldots, x_{i-1}) = \frac{1}{i-1 + \alpha} \sum_{j=1}^{i-1} \delta_{x_j} + \frac{\alpha}{i-1 + \alpha} \delta_{x_{\text{new}}},$$

where $x_{\text{new}}$ is a new integer not in $x_1, \ldots, x_{i-1}$.

If $\text{conc}$ is not specified, it assumes the default value of 1. The $\text{conc}$ parameter has to be larger than zero. Otherwise, NaN are returned.

**Value**

$d\text{CRP}$ gives the density, and $r\text{CRP}$ gives random generation.

**Author(s)**

Claudia Wehrhahn

**References**


**Examples**

```r
x <- rCRP(n=1, conc = 1, size=10)
dCRP(x, conc = 1, size=10)
```

---

**clearCompiled**

Clear compiled objects from a project and unload shared library

**Description**

Clear all compiled objects from a project and unload the shared library produced by the C++ compiler. Has no effect on Windows.

**Usage**

`clearCompiled(obj)`
Arguments

obj A compiled nimbleFunction or nimble model

Details

This will clear all compiled objects associated with your NIMBLE project. For example, if `cModel` is a compiled model, `clearCompiled(cModel)` will clear both the model and all associated nimbleFunctions such as compiled MCMCs that use that model.

Use of this function can be dangerous. There is some risk that if you have copies of the R objects that interfaced to compiled C++ objects that have been removed, and you attempt to use those R objects after clearing their compiled counterparts, you will crash R. We have tried to minimize that risk, but we can’t guarantee safe behavior.
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

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

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.

`showCompilerOutput` Logical value indicating whether details of C++ compilation should be printed.

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 the NIMBLE User Manual 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.
configureMCMC

Build the MCMCconf object for construction of an MCMC object

Description

Creates a default MCMC configuration for a given model.

Usage

configureMCMC(
  model,
  nodes,
  control = list(),
  monitors,
  thin = 1,
  monitors2 = character(),
  thin2 = 1,
  useConjugacy = getNimbleOption("MCMCuseConjugacy"),
  onlyRW = FALSE,
  onlySlice = FALSE,
  multivariateNodesAsScalars = getNimbleOption("MCMCmultivariateNodesAsScalars"),
  enableWAIC = getNimbleOption("MCMCenableWAIC"),
  controlWAIC = list(),
  print = getNimbleOption("verbose"),
  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 setup code of the sampling algorithms.

**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.

**enableWAIC**
A logical argument, specifying whether to enable WAIC calculations for the resulting MCMC algorithm. Defaults to the value of `nimbleOptions('MCMCenableWAIC')`, which in turn defaults to FALSE. Setting `nimbleOptions('enableWAIC' = TRUE)` will ensure that WAIC is enabled for all calls to `configureMCMC` and `buildMCMC`.

**controlWAIC**
A named list of inputs that control the behavior of the WAIC calculation. See `help(waic)`.

**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.
configureRJ

**oldConf**
An optional MCMCconf object to modify rather than creating a new MCMC-conf from scratch

... Additional named control list elements for default samplers, or 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

**See Also**
`buildMCMC runMCMC nimbleMCMC`

---

**configureRJ**

*Configure Reversible Jump for Variable Selection*

**Description**
Modifies an MCMC configuration object to perform a reversible jump MCMC sampling for variable selection, using a univariate normal proposal distribution. Users can control the mean and scale of the proposal. This function supports two different types of model specification: with and without indicator variables.

**Usage**

```r
configureRJ(
  conf,
  targetNodes,
  indicatorNodes = NULL,
  priorProb = NULL,
  control = list(mean = NULL, scale = NULL, fixedValue = NULL)
)
```

**Arguments**

- `conf` An MCMCconf object.
- `targetNodes` A character vector, specifying the nodes and/or variables for which variable selection is to be performed. Nodes may be specified in their indexed form, e.g., 'y[1, 3]'. Alternatively, nodes specified without indexing will be expanded, e.g., 'x' will be expanded to 'x[1]', 'x[2]', etc.
**configureRJ**

indicatorNodes  An optional character vector, specifying the indicator nodes and/or variables paired with targetNodes. Nodes may be specified in their indexed form, 'y[1, 3]'. Alternatively, nodes specified without indexing will be expanded, e.g., 'x' will be expanded to 'x[1]', 'x[2]', etc. Nodes must be provided consistently with targetNodes. See details.

priorProb  An optional value or vector of prior probabilities for each node to be in the model. See details.

control  An optional list of control arguments:
  • mean. The mean of the normal proposal distribution (default = 0).
  • scale. The standard deviation of the normal proposal distribution (default = 1).
  • fixedValue. Value for the variable when it is out of the model, which can be used only when priorProb is provided (default = 0). If specified when indicatorNodes is passed, a warning is given and fixedValue is ignored.

**Details**

This function modifies the samplers in MCMC configuration object for each of the nodes provided in the targetNodes argument. To these elements two samplers are assigned: a reversible jump sampler to transition the variable in/out of the model, and a modified version of the original sampler, which performs updates only when the target node is already in the model.

configureRJ can handle two different ways of writing a NIMBLE model, either with or without indicator variables. When using indicator variables, the indicatorNodes argument must be provided. Without indicator variables, the priorProb argument must be provided. In the latter case, the user can provide a non-zero value for fixedValue if desired.

Note that this functionality is intended for variable selection in regression-style models but may be useful for other situations as well. At the moment, setting a variance component to zero and thereby removing a set of random effects that are explicitly part of a model will not work because MCMC sampling in that case would need to propose values for multiple parameters (the random effects), whereas the current functionality only proposes adding/removing a single model node.

**Value**

NULL configureRJ modifies the input MCMC configuration object in place.

**Author(s)**

Sally Paganin, Perry de Valpine, Daniel Turek

**References**


**See Also**

samplers configureMCMC
Examples

## Not run:

## Linear regression with intercept and two covariates, using indicator variables

code <- nimbleCode(
  beta0 ~ dnorm(0, sd = 100)
  beta1 ~ dnorm(0, sd = 100)
  beta2 ~ dnorm(0, sd = 100)
  sigma ~ dunif(0, 100)
  z1 ~ dbern(psi)  ## indicator variable associated with beta1
  z2 ~ dbern(psi)  ## indicator variable associated with beta2
  psi ~ dunif(0, 1) ## hyperprior on inclusion probability
  for(i in 1:N) {
    Ypred[i] <- beta0 + beta1 * z1 * x1[i] + beta2 * z2 * x2[i]
    Y[i] ~ dnorm(Ypred[i], sd = sigma)
  }
)

## simulate some data
set.seed(1)
N <- 100
x1 <- runif(N, -1, 1)
x2 <- runif(N, -1, 1)  ## this covariate is not included
Y <- rnorm(N, 1 + 2.5 * x1, sd = 1)

## build the model
rIndicatorModel <- nimbleModel(code, constants = list(N = N),
  data = list(Y = Y, x1 = x1, x2 = x2),
  inits = list(beta0 = 0, beta1 = 0, beta2 = 0, sigma = sd(Y),
               z1 = 1, z2 = 1, psi = 0.5))

indicatorModelConf <- configureMCMC(rIndicatorModel)

## Add reversible jump
configureRJ(conf = indicatorModelConf,  ## model configuration
  targetNodes = c("beta1", "beta2"),  ## coefficients for selection
  indicatorNodes = c("z1", "z2"),  ## indicators paired with coefficients
  control = list(mean = 0, scale = 2))

indicatorModelConf$addMonitors("beta1", "beta2", "z1", "z2")

rIndicatorMCMC <- buildMCMC(indicatorModelConf)
cIndicatorModel <- compileNimble(rIndicatorModel)
cIndicatorMCMC <- compileNimble(rIndicatorMCMC, project = rIndicatorModel)

set.seed(1)
samples <- runMCMC(cIndicatorMCMC, 10000, nburnin = 6000)

## posterior probability to be included in the mode
mean(samples[, "z1"])
mean(samples[, "z2"])
## posterior means when in the model

```r
mean(samples[, "beta1"] [samples[, "z1"] != 0])
mean(samples[, "beta2"] [samples[, "z2"] != 0])
```

## Linear regression with intercept and two covariates, without indicator variables

```r
code <- nimbleCode(
  beta0 ~ dnorm(0, sd = 100)
beta1 ~ dnorm(0, sd = 100)
beta2 ~ dnorm(0, sd = 100)
sigma ~ dunif(0, 100)
for(i in 1:N) {
  Ypred[i] <- beta0 + beta1 * x1[i] + beta2 * x2[i]
  Y[i] ~ dnorm(Ypred[i], sd = sigma)
}
)
```

```r
rNoIndicatorModel <- nimbleModel(code, constants = list(N = N),
data = list(Y = Y, x1 = x1, x2 = x2),
inits = list(beta0 = 0, beta1 = 0, beta2 = 0, sigma = sd(Y)))
noIndicatorModelConf <- configureMCMC(rNoIndicatorModel)
## Add reversible jump
configureRJ(conf = noIndicatorModelConf, ## model configuration
targetNodes = c("beta1", "beta2"), ## coefficients for selection
priorProb = 0.5, ## prior probability of inclusion
control = list(mean = 0, scale = 2))
## add monitors
noIndicatorModelConf$addMonitors("beta1", "beta2")
rNoIndicatorMCMC <- buildMCMC(noIndicatorModelConf)
cNoIndicatorModel <- compileNimble(rNoIndicatorModel)
cNoIndicatorMCMC <- compileNimble(rNoIndicatorMCMC, project = rNoIndicatorModel)
set.seed(1)
samples <- runMCMC(cNoIndicatorMCMC, 10000, nburnin = 6000)
## posterior probability to be included in the mode
mean(samples[, "beta1"] != 0)
mean(samples[, "beta2"] != 0)
## posterior means when in the model
mean(samples[, "beta1"] [samples[, "beta1"] != 0])
mean(samples[, "beta2"] [samples[, "beta2"] != 0])
```

## End(Not run)
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, target, 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)`.
- **target**: A character vector providing the target node.
- **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-transition

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
Arguments

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

When sizes are not set, they can be set by a call to \texttt{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

\begin{verbatim}
    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.
\end{verbatim}

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, userEnv = parent.frame())
Arguments

distributionsNames
   a character vector giving the names of the distributions to be deregistered.
userEnv
   environment in which to look for the nimbleFunctions that provide the distribution; this will generally not need to be set by the user as it will default to the environment from which this function was called.

Author(s)

Christian 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

distributionInfo

See Also

Distributions for other standard distributions

Examples

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

distributionInfo  Get information about a distribution

Description

Give information about each BUGS distribution

Usage

getDistributionInfo(dist)

isUserDefined(dist)
pqDefined(dist)

getType(
  dist,
  params = NULL,
  valueOnly = is.null(params) && !includeParams,
  includeParams = !is.null(params)
)

getParamNames(dist, includeValue = TRUE)

Arguments

dist  a character vector of length one, giving the name of the distribution (as used in
      BUGS code), e.g. ‘dnorm’

params  an optional character vector of names of parameters for which dimensions are
         desired (possibly including ‘value’ and alternate parameters)

valueOnly  a logical indicating whether to only return the dimension of the value of the node

includeParams  a logical indicating whether to return dimensions of parameters. If TRUE and
                ‘params’ is NULL then dimensions of all parameters, including the dimension
                of the value of the node, are returned

includeValue  a logical indicating whether to return the string ‘value’, which is the name of the
               node value
Details

NIMBLE provides various functions to give information about a BUGS distribution. In some cases, functions of the same name and similar functionality operate on the node(s) of a model as well (see help(modelBaseClass)).

getDistributionInfo returns an internal data structure (a reference class object) providing various information about the distribution. The output is not very user-friendly, but does contain all of the information that NIMBLE has about the distribution.

isDiscrete tests if a BUGS distribution is a discrete distribution.

isUserDefined tests if a BUGS distribution is a user-defined distribution.

pqAvail tests if a BUGS distribution provides distribution ('p') and quantile ('q') functions.

getDimension provides the dimension of the value and/or parameters of a BUGS distribution. The return value is a numeric vector with an element for each parameter/value requested.

getType provides the type (numeric, logical, integer) of the value and/or parameters of a BUGS distribution. The return value is a character vector with an element for each parameter/value requested. At present, all quantities are stored as numeric (double) values, so this function is of little practical use but could be exploited in the future.

getParamNames provides the value and/or parameter names of a BUGS distribution.

Author(s)

Christopher Paciorek

Examples

distInfo <- getDistributionInfo('dnorm')
distInfo
distInfo$range

isDiscrete('dbin')

isUserDefined('dbin')

pqDefined('dgamma')
pqDefined('dmnorm')

getDimension('dnorm')
getDimension('dnorm', includeParams = TRUE)
getDimension('dnorm', c('var', 'sd'))
getDimension('dcat', includeParams = TRUE)
getDimension('dwish', includeParams = TRUE)

getype('dnorm')
getype('dnorm', includeParams = TRUE)
getype('dnorm', c('var', 'sd'))
getype('dcat', includeParams = TRUE)
getype('dwish', includeParams = TRUE)

getParamNames('dnorm', includeValue = FALSE)
Double-Exponential

The Double Exponential (Laplace) Distribution

Description
Density, distribution function, quantile function and random generation for the double exponential distribution, allowing non-zero location, mu, and non-unit scale, sigma, or non-unit rate, tau

Usage

dexp(x, location = 0, scale = 1, rate = 1/scale, log = FALSE)

rexp(n, location = 0, scale = 1, rate = 1/scale)

pexp(q,
  location = 0,
  scale = 1,
  rate = 1/scale,
  lower.tail = TRUE,
  log.p = FALSE
)

qexp(p,
  location = 0,
  scale = 1,
  rate = 1/scale,
  lower.tail = TRUE,
  log.p = FALSE
)

Arguments

x vector of values.
location vector of location values.
scale vector of scale values.
rate vector of inverse 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

ddexp gives the density, pdexp gives the distribution function, qdexp gives the quantile function, and rdexp generates random deviates.

Author(s)

Christopher Paciorek

References


See Also

Distributions for other standard distributions

Examples

```r
x <- rdexp(50, location = 2, scale = 1)
ddexp(x, 2, 1)
```

---

Description

nimbleList definition for the type of nimbleList returned by `nimEigen`.

Usage

eigenNimbleList

Format

An object of class `list` of length 1.

Author(s)

NIMBLE development team

See Also

`nimEigen`
The Exponential Distribution

Description

Density, distribution function, quantile function and random generation for the exponential distribution with rate (i.e., mean of \(1/\text{rate}\)) 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.
- \(\text{rate}\) vector of rate values.
- \(\text{scale}\) vector of scale values.
- \(\text{log}\) logical; if TRUE, probability density is returned on the log scale.
- \(n\) number of observations.
- \(q\) vector of quantiles.
- \(\text{lower.tail}\) logical; if TRUE (default) probabilities are \(P[X \leq x]\); otherwise, \(P[X > x]\).
- \(\text{log.p}\) 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


See Also

*Distributions* for other standard distributions

Examples

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

---

**extractControlElement**  
*Extract named elements from MCMC sampler control list*

Description

Extract named elements from MCMC sampler control list

Usage

```r
extractControlElement(controlList, elementName, defaultValue, error)
```

Arguments

- `controlList`  
  control list object, which is passed as an argument to all MCMC sampler setup functions.
- `elementName`  
  character string, giving the name of the element to be extracted from the control list.
- `defaultValue`  
  default value of the control list element, giving the value to be used when the `elementName` does not exactly match the name of an element in the `controlList`.
- `error`  
  character string, giving the error message to be printed if no `defaultValue` is provided and `elementName` does not match the name of an element in the `controlList`.

Value

The element of `controlList` whose name matches `elementName`. If no `controlList` name matches `elementName`, then `defaultValue` is returned.

Author(s)

Daniel Turek
The Improper Uniform Distribution

Description

Improper flat distribution for use as a prior distribution in BUGS models

Usage

dflat(x, log = FALSE)
rflat(n = 1)
dhalfflat(x, log = FALSE)
rhalfflat(n = 1)

Arguments

x vector of values.
log logical; if TRUE, probability density is returned on the log scale.
n number of observations.

Value

dflat gives the pseudo-density value of 1, while rflat and rhalfflat return NaN, since one cannot simulate from an improper distribution. Similarly, dhalfflat gives a pseudo-density value of 1 when x is non-negative.

Author(s)

Christopher Paciorek

See Also

Distributions for other standard distributions

Examples

dflat(1)
getBound

Get value of bound of a stochastic node in a model

Description
Get the value of the lower or upper bound for a single stochastic node in a model.

Usage
getBound(model, node, bound, nodeFunctionIndex)

Arguments
- **model**: A NIMBLE model object
- **node**: The name of a stochastic node in the model
- **bound**: Either 'lower' or 'upper' indicating the desired bound for the node
- **nodeFunctionIndex**: For internal NIMBLE use only

Details
Standard usage is as a method of a model, in the form model$getBound(node, bound), but the usage as a simple function with the model as the first argument as above is also allowed.

For nodes that do not involve truncation of the distribution this will return the lower or upper bound of the distribution, which may be a constant or for a limited number of distributions a parameter or functional of a parameter (at the moment in NIMBLE, the only case where a bound is a parameter is for the uniform distribution. For nodes that are truncated, this will return the desired bound, which may be a functional of other quantities in the model or may be a constant.

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.
getConditionallyIndependentSets

Value

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

Author(s)

Christopher Paciorek

See Also

readBUGSmodel for usage in creating a model from a classic BUGS example

---

**getConditionallyIndependentSets**

*Get a list of conditionally independent sets of nodes in a nimble model*

Description

A conditionally independent set of nodes is such that the joint probability (density) of nodes in the set will not change even if any non-given node outside the set is changed. Default given nodes are data nodes and parameter nodes (aka "top-level" nodes, i.e. nodes with no parent nodes), but this can be controlled.

Usage

```r
getConditionallyIndependentSets(
  model,
  nodes,
  givenNodes,
  omit = integer(),
  explore = c("both", "down", "up"),
  unknownAsGiven = TRUE,
  returnType = "names",
  returnScalarComponents = FALSE,
  endAsGiven = FALSE
)
```

Arguments

- **model**
  A nimble model object (uncompiled or compiled), such as returned by `nimbleModel`.

- **nodes**
  A vector of stochastic node names (or their graph IDs) to split into conditionally independent sets, conditioned on the `givenNodes`. If `unknownAsGiven=FALSE`, the nodes are the starting nodes from which conditionally independent sets of nodes should be found, possibly including additional nodes not included in the `nodes` argument. If `nodes` is omitted, the default will be all latent nodes (defined as stochastic nodes that are not data and have at least one stochastic parent node, possibly with deterministic nodes in-between) that are a parent of a `givenNode` (either provided or default). Note that this will omit latent states that have no
hyperparameters. An example is the first latent state in some state-space (time-series) models, which is sometimes declared with known prior.

givenNodes A vector of node names or their graph IDs that should be considered as fixed (given) and hence can be conditioned on. If omitted, the default will be all data nodes and all parameter nodes, the latter defined as nodes with no stochastic parent nodes (skipping over deterministic parent nodes). See endAsGiven for a variant on defaults.

omit A vector of node names or their graph IDs that should be omitted and should block further graph exploration.

explore The method of graph exploration, which may correspond to what the nodes argument represents. For "down", graph exploration starts only down (towards descendants) from nodes. For "up", graph exploration starts only up (towards ancestors) from nodes. For "both" (the default and normal setting), both directions are explored.

unknownAsGiven Logical for whether a model node not in nodes or givenNodes should be treated as given (default = TRUE). Otherwise (and by default) such a node may be grouped into a conditionally independent set, resulting in more output nodes than input nodes.

returnType Either "names" for returned nodes to be node names or "ids" for returned nodes to be graph IDs.

returnScalarComponents If FALSE (default), multivariate nodes are returned as full names (e.g. x[1:3]). If TRUE, they are returned as scalar elements (e.g. x[1], x[2], x[3]).

endAsGiven If TRUE, end nodes (defined as nodes with stochastic parents but no stochastic children, skipping through deterministic nodes) are included in the default for givenNodes.

Details

This function returns sets of conditionally independent nodes. Multiple input nodes might be in the same set or different sets.

The nodes input and the returned sets include only stochastic nodes because conditional independence is a property of random variables. Deterministic nodes are considered in determining the sets. givenNodes may contain stochastic or deterministic nodes.

Value

List of nodes that are in conditionally independent sets. With each set, nodes are returned in topologically sorted order. The sets themselves are returned in topologically sorted order of their first nodes.

Other nodes (not in nodes) may be included in the output if unknownAsGiven=FALSE.

Author(s)

Perry de Valpine
**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**

```r
getDefinition(nf)
```

**Arguments**

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

**Author(s)**

Daniel Turek

**getMacroParameters**

*EXPERIMENTAL: Get list of parameter names generated by model macros*

**Description**

Get a list of all parameter names (or certain categories of parameters) generated by model macros in the model code.

**Usage**

```r
gETCHMARMAMARARTGET
model, includeLHS = TRUE, includeRHS = TRUE, includeDeterm = TRUE, includeStoch = TRUE, includeIndexPars = FALSE
```

)
getMacroParameters

Arguments

model A NIMBLE model object
includeLHS Include generated parameters on the left-hand side (LHS) of assignments (\(<-\) or ~) in the output
includeRHS Include generated parameters on the left-hand side (RHS) of assignments (\(<-\) or ~) in the output
includeDeterm Include deterministic generated parameters in the output
includeStoch Include stochastic generated parameters in the output
includeIndexPars Include index parameters generated for use in for loops in the output

Details

Some model macros will generate new parameters to be included in the output code. NIMBLE automatically detects these new parameters and records them in the model object. This function allows easy access to this stored list, or subsets of it (see arguments).

Value

A named list of generated parameters, with the element names corresponding to the original source macro.

Examples

```r
nimbleOptions(enableModelMacros = TRUE)
nimbleOptions(enableMacroComments = FALSE)
nimbleOptions(verbos e = FALSE)

testMacro <- list(process = function(code, modelInfo, .env){
code <- quote{
  for (i_ in 1:n){
    mu[i_] <- alpha + beta
    y[i_] ~ dnorm(0, sigma)
  }
  alpha ~ dnorm(0, 1)
}

list(code = code, modelInfo=modelInfo)
})
class(testMacro) <- "model_macro"

code <- nimbleCode{
  y[1:n] ~ testMacro()
}

const <- list(y = rnorm(10), n = 10)

mod <- nimbleModel(code, constants=const)

mod$getMacroParameters()
```
getNimbleOption

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

Get the value of a parameter for any single stochastic node in a model.

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

Standard usage is as a method of a model, in the form `model$getParam(node, param)`, but the usage as a simple function with the model as the first argument as above is also allowed.

For example, suppose node 'x[1:5]' follows a multivariate normal distribution (dmnorm) in a model declared by BUGS code. `model$getParam('x[1:5]', 'mean')` would return the current value of the mean parameter (which may be determined from other nodes). 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.

getSamplesDPmeasure

Get posterior samples for a Dirichlet process measure

Description

This function obtains posterior samples from a Dirichlet process distributed random measure of a model specified using the dCRP distribution.

Usage

```r
getSamplesDPmeasure(
  MCMC,
  epsilon = 1e-04,
  setSeed = FALSE,
  progressBar = getNimbleOption("MCMCprogressBar")
)
```

Arguments

- MCMC: an MCMC class object, either compiled or uncompiled.
- epsilon: used for determining the truncation level of the representation of the random measure.
- setSeed: Logical or numeric argument. If a single numeric value is provided, R's random number seed will be set to this value. In the case of a logical value, if TRUE, then R's random number seed will be set to 1. Note that specifying the argument `setSeed = 0` does not prevent setting the RNG seed, but rather sets the random number generation seed to 0. Default value is FALSE.
getSamplesDPmeasure

progressBar Logical specifying whether to display a progress bar during execution (default = TRUE). The progress bar can be permanently disabled by setting the system option nimbleOptions(MCMCprogressBar = FALSE)

Details

This function provides samples from a random measure having a Dirichlet process prior. Realizations are almost surely discrete and represented by a (finite) stick-breaking representation (Sethuraman, 1994), whose atoms (or point masses) are independent and identically distributed. This sampler can only be used with models containing a dCRP distribution.

The MCMC argument is an object of class MCMC provided by buildMCMC, or its compiled version. The MCMC should already have been run, as getSamplesDPmeasure uses the posterior samples to generate samples of the random measure. Note that the monitors associated with that MCMC must include the cluster membership variable (which has the dCRP distribution), the cluster parameter variables, all variables directly determining the dCRP concentration parameter, and any stochastic parent variables of the cluster parameter variables. See help(configureMCMC) or help(addMonitors) for information on specifying monitors for an MCMC.

The epsilon argument is optional and used to determine the truncation level of the random measure. epsilon is the tail probability of the random measure, which together with posterior samples of the concentration parameter, determines the truncation level. The default value is 1e-4.

The output is a list of matrices. Each matrix represents a sample from the random measure. In order to reduce the output's dimensionality, the weights of identical atoms are added up. The stick-breaking weights are named weights and the atoms are named based on the cluster variables in the model.

For more details about sampling the random measure and determining its truncation level, see Section 3 in Gelfand, A.E. and Kottas, A. 2002.

Author(s)

Claudia Wehrhahn and Christopher Paciorek

References


See Also

buildMCMC, configureMCMC

Examples

## Not run:
conf <- configureMCMC(model)
mcmc <- buildMCMC(conf)
cmodel <- compileNimble(model)
cmcmc <- compileNimble(mcmc, project = model)
runMCMC(cmcmc, niter = 1000)
outputG <- getSamplesDPmeasure(cmcmc)

## End(Not run)

getszie

Returns number of rows of modelValues

Description

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

Usage

getszie(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)
getszie(mv)
Create an Identity matrix (Deprecated)

Description

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

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 NIMBLE run code. It is deprecated because now one can use diag(d) instead.

Value

A d-by-d identity matrix

Author(s)

Daniel Turek

Examples

Id <- identityMatrix(d = 3)

Performs initialization of nimble model node values and log probabilities

initializeModel(model, silent = FALSE)

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 specialized 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 calculations

Description

Calculations to handle censoring

Usage

dinterval(x, t, c, log = FALSE)
rinterval(n = 1, t, c)
Inverse-Gamma

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

```r
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)
```

---

**Inverse-Gamma**  
*The Inverse Gamma Distribution*

Description

Density, distribution function, quantile function and random generation for the inverse gamma distribution with rate or scale (mean = scale / (shape - 1)) parameterizations.
Inverse-Gamma

Usage

dinvgamma(x, shape, scale = 1, rate = 1/scale, log = FALSE)

rinvgamma(n = 1, shape, scale = 1, rate = 1/scale)

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

qinvgamma(
  p,
  shape,
  scale = 1,
  rate = 1/scale,
  lower.tail = TRUE,
  log.p = FALSE
)

Arguments

x vector of values.
shape vector of shape values, must be positive.
scale vector of scale values, must be positive.
rate vector of rate values, must be positive.
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

The inverse gamma distribution with parameters \( \text{shape} = \alpha \) and \( \text{scale} = \sigma \) has density

\[
f(x) = \frac{s^{\alpha}}{\Gamma(\alpha)} x^{-(\alpha+1)} e^{-\sigma/x}
\]

for \( x \geq 0, \alpha > 0 \) and \( \sigma > 0 \). (Here \( \Gamma(\alpha) \) is the function implemented by \( \text{R}'s \) \text{gamma()} \) and defined in its help.

The mean and variance are \( E(X) = \frac{\sigma}{\alpha} - 1 \) and \( Var(X) = \frac{\sigma^2}{(\alpha-1)^2(\alpha-2)} \), with the mean defined only for \( \alpha > 1 \) and the variance only for \( \alpha > 2 \).

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

**Value**

dinvgamma gives the density, pinvgamma gives the distribution function, qinvgamma gives the quantile function, and rinvgamma generates random deviates.

**Author(s)**

Christopher Paciorek

**References**


**See Also**

*Distributions* for other standard distributions

**Examples**

```r
x <- rinvgamma(50, shape = 1, scale = 3)
dinvgamma(x, shape = 1, scale = 3)
```

---

In *Inverse-Wishart: The Inverse Wishart Distribution*

**Description**

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

**Usage**

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

rinvwish_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 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

dinvwish_chol gives the density and rinvwish_chol generates random deviates.

Author(s)

Christopher Paciorek

References


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)

---

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, inputIsName = FALSE, where = -1)

Arguments

f

object to be tested

inputIsName

logical indicating whether the function is provided as the character name of the function or the function object itself

where

Optional argument needed due to R package namespace issues but which should not need to be provided by a user.
See Also

nimbleFunction for how to create a nimbleFunction

---

### is.nl

**check if a nimbleList**

**Description**

Checks an object to determine if it is a nimbleList (i.e., a list created by nlDef$new()).

**Usage**

```r
is.nl(l)
```

**Arguments**

- `l` object to be tested

See Also

nimbleList for how to create a nimbleList

---

### LKJ

**The LKJ Distribution for the Cholesky Factor of a Correlation Matrix**

**Description**

Density and random generation for the LKJ distribution for the Cholesky factor of a correlation matrix.

**Usage**

```r
dlkj_corr_cholesky(x, eta, p, log = FALSE)
rlkj_corr_cholesky(n = 1, eta, p)
```

**Arguments**

- `x` upper-triangular Cholesky factor of a correlation matrix.
- `eta` shape parameter.
- `p` size of the correlation matrix (number of rows and columns); required because random generation function has no information about dimension of matrix to generate without this argument.
- `log` logical; if TRUE, probability density is returned on the log scale.
- `n` number of observations (only n=1 is handled currently).
Details
See Stan Development Team for mathematical details.

Value
dlkj_corr_cholesky gives the density and rlkj_corr_cholesky generates random deviates.

Author(s)
Christopher Paciorek

References
Stan Development Team. Stan Reference Functions, version 2.27.

See Also
Distributions for other standard distributions

Examples
```r
etta <- 3
x <- rlkj_corr_cholesky(1, etta, 5)
dlkj_corr_cholesky(x, etta, 5)
```

makeBoundInfo
Make an object of information about a model-bound pairing for getBound. Used internally

Description
Creates a simple getBound_info object, which has a list with a boundID and a type. Unlike makeParamInfo this is more bare-bones, but keeping it for parallelism with getParam.

Usage
```r
makeBoundInfo(model, nodes, bound)
```

Arguments
```r
model A model such as returned by nimbleModel.
nodes A character string naming a stochastic nodes, such as 'mu'.
bound A character string naming a bound of the distribution, either 'lower' or 'upper'.
```

Details
This is used internally by getBound. It is not intended for direct use by a user or even a nimble-Function programmer.
Description

Inspect structure of a nimble model to determine nodes needed as "update" and/or "constant" entries in usage of nimDerivs. This will typically be used in the setup code of a nimbleFunction.

Usage

```r
makeModelDerivsInfo(model, wrtNodes, calcNodes, dataAsConstantNodes = TRUE)
```

Arguments

- **model**: a nimble model object, such as returned from `nimbleModel`.
- **wrtNodes**: a character vector of node names in the model with respect to which derivatives will be taken through a call to `nimDerivs` (same as `derivs`).
- **calcNodes**: a character vector of node names in the model that will be used in `model$calculate(calcNodes)` while derivatives are being recorded.
- **dataAsConstantNodes**: logical indicating whether data nodes in the model should automatically be treated as "constant" entries (TRUE) or "update" entries (FALSE). Defaults to TRUE.

Details

In the compilable parts of a nimbleFunction (i.e. run or other method code, not setup code), a call like `nimDerivs(foo(x), ...)` records derivatives of `foo(x)`. If `foo` contains any calls to `model$calculate(calcNodes)`, it may be necessary to provide auxiliary information about the model in further arguments to `nimDerivs`, specifically the `model`, `updateNodes` and `constantNodes` arguments. `makeModelDerivsInfo` is a utility to set up that information for typical use cases. It returns a list with elements `updateNodes` and `constantNodes` to be passed as arguments of the same name to `nimDerivs` (along with passing the `model` as the `model` argument).

The reason auxiliary information is needed is that recording of derivatives uses a different model than for regular calculations. Together, `updateNodes` and `constantNodes` should contain all nodes whose values are needed for the model calculations being recorded and that are not part of `wrtNodes`. These may include parents of nodes that are in `calcNodes` but are not themselves in `calcNodes`, as well as the values of stochastic nodes in `calcNodes`, which are needed to calculate the corresponding log probabilities. `updateNodes` will have their values updated from the regular model every time that recorded derivative calculations are used. `constantNodes` will not be updated every time, which means their values will be permanently fixed either the first time the call to `nimDerivs` is invoked or on any subsequent call that has `reset=TRUE`. Use of `constantNodes` can be slightly more efficient, but one must be careful to be aware that values will not be updated unless `reset=TRUE`. See the automatic differentiation section of the User Manual for more information.

In the above explanation, care must be taken to understand what should be included in `wrtNodes`. In a typical use case, some arguments to `foo` are put into the model using `values(model, nodes)`. 
<-some_foo_arguments. Next there is typically a call to model$calculate(calcNodes). Here the nodes are considered "with-respect-to" nodes because derivative tracking will follow the arguments of foo, including when they are put into a model and hence used in model$calculate. Therefore these nodes should be the wrtNodes for makeModelDerivsInfo.

**Value**

A list with elements updateNodes and constantNodes. These should be provided as the same-named arguments to nimDerivs (same as derivs).

When using double-taping of derivatives (i.e. foo contains another call to nimDerivs), both calls to nimDerivs should include the model, updateNodes, and constantNodes arguments.

---

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

**Description**

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

**Usage**

```r
makeParamInfo(model, nodes, param, vector = FALSE)
```

**Arguments**

- `model` A model such as returned by `nimbleModel`.
- `nodes` A character string naming 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.
- `vector` A logical indicating whether nodes should definitely be treated as a vector in compiled code, even if it has length = 1. For type consistency, the compiler needs this option. If nodes has length > 1, this argument is ignored.

**Details**

This is used internally by `getParam`. It is not intended for direct use by a user or even a nimble-Function programmer.
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

```
addDefaultSampler( nodes = character(), control = list(), useConjugacy = getNimbleOption("MCMCuseConjugacy"), onlyRW = FALSE, onlySlice = FALSE, multivariateNodesAsScalars = getNimbleOption("MCMCmultivariateNodesAsScalars"), print = TRUE, ... )
```

For internal use. Adds default MCMC samplers to the specified nodes.

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

Arguments:

...: One or more character vectors 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 (default TRUE).

Details:

See the `initialize()` function

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

Arguments:

...: One or more character vectors 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 (default TRUE).

Details:

See the `initialize()` function

```
addOneSampler( thisSamplerName, samplerFunction, targetOne, thisControlList, allowData, print )
```

For internal use only

```
addSampler( target = character(), type = "RW", control = list(), print = NULL, name, targetByNode = FALSE, ... )
```

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. For univariate samplers, only a single target node should be provided (unless 'targetByNode' is TRUE). For multivariate samplers, one instance of the multivariate sampler will be assigned to all nodes specified. Nodes are specified in combination with the 'targetByNode' and 'multivariateNodesAsScalars' arguments.

type: When 'default' is FALSE, specifies 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 (unless 'targetByNode' is TRUE). This argument is not used when the 'default' argument is TRUE.

control: An optional list of control arguments to sampler functions. These will override those specified in the control list argument to configureMCMC. 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'. The default values for control list arguments for samplers (if not otherwise provided as an argument to configureMCMC or addSampler) are contained in the setup code of each sampling algorithm.

print: Logical argument, specifying whether to print the details of newly added sampler(s).

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.

targetByNode: Logical argument, with default FALSE. This arguments controls whether separate instances of the specified sampler 'type' should be assigned to each node contained in 'target'. When FALSE, a single instance of sampler 'type' is assigned to operate on 'target'. When TRUE, potentially multiple instances of sampler 'type' will be added to the MCMC configuration, operating on the distinct nodes which compose 'target'. For example, if 'target' is a vector of distinct node names, then a separate sampler will be assigned to each node in this vector. If 'target' is a model variable which itself is comprised of multiple distinct nodes, then a separate sampler is assigned to each node composing the 'target' variable. Additional control of the handling of multivariate nodes is provided using the 'multivariateNodesAsScalars' argument.

multivariateNodesAsScalars: Logical argument, with default value FALSE. This argument is used in two ways. Functionally, both uses result in separate instances of samplers being added to the scalar components which compose multivariate nodes. See details below.

silent: Logical argument, specifying whether to print warning messages when assigning samplers.

default: Logical argument, with default value FALSE. When FALSE, the 'type' argument dictates what sampling algorithm is assigned to the specified nodes. When TRUE, default samplers will be assigned to the specified nodes following the same logic as the configureMCMC method, and also using the 'useConjugacy', 'onlyRW', 'onlySlice' and 'multivariateNodesAsScalars' arguments.

useConjugacy: 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. This argument is only used when the 'default' argument is TRUE.

onlyRW: 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. Discrete-valued nodes are assigned a slice sampler, and terminal nodes are assigned a posterior_predictive sampler. This argument is only used when the 'default' argument is TRUE.

onlySlice: 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. This argument is only used when the 'default' argument is TRUE.

allowData: Logical argument, with default value FALSE. When FALSE, samplers will not be assigned to operate on data nodes, even if data nodes are included in 'target'. When TRUE,
samplers will be assigned to 'target' without regard to whether nodes are designated as data.

...: Additional named arguments passed through ... will be used as additional control list elements.

Details:
Samplers are added to the end of the list of samplers for this MCMCconf object, and do not replace any existing samplers. Samplers are removed using the removeSamplers method.
Invisibly returns a list of the current sampler configurations, which are samplerConf reference class objects.
'multivariateNodesAsScalars' has two usages. The first usage occurs when 'targetByNode' is TRUE and therefore separate instances of sampler 'type' are assigned to each node which compose 'target'. In this first usage, this argument controls how multivariate nodes (those included in the 'target') are handled. If FALSE, any multivariate nodes in 'target' have a single instance of sampler 'type' assigned. If TRUE, any multivariate nodes appearing in 'target' are themselves decomposed into their scalar elements, and a separate instance of sampler 'type' is assigned to operate on each scalar element.
The second usage of 'multivariateNodesAsScalars' occurs when 'default' is TRUE, and therefore samplers are assigned according to the default logic of configureMCMC, which is further controlled by the arguments 'useConjugacy', 'onlyRW', 'onlySlice' and 'multivariateNodesAsScalars'. In this second usage, if 'multivariateNodesAsScalars' is TRUE, then multivariate nodes will be decomposed into their scalar components, and separate samplers assigned to each scalar element. Note, however, that multivariate nodes appearing in conjugate relationships will still be assigned the corresponding conjugate sampler (provided 'useConjugacy' is TRUE), regardless of the value of this argument. If 'multivariateNodesAsScalars' is FALSE, then a single multivariate sampler will be assigned to update each multivariate node. The default value of this argument can be controlled using the nimble option 'MCMCmultivariateNodesAsScalars'.

getMonitors() Returns a character vector of the current monitors
Details:
See the initialize() function
getMonitors2() Returns a character vector of the current monitors2
Details:
See the initialize() function
getSamplerDefinition(ind, print = FALSE) 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.
getSamplerExecutionOrder() Returns a numeric vector, specifying the ordering of sampler function execution.
The indices of execution specified in this numeric vector correspond to the enumeration of samplers printed by printSamplers(), or returned by getSamplers().
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'.

initialize(model, nodes, control = list(), monitors, thin = 1, monitors2 = character(), thin2 = 1, useConjugacy = ... enableWAIC = getNimbleOption("MCMCenableWAIC"), controlWAIC = list(), print = TRUE, ...)

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

Arguments:
model: A NIMBLE model object, created from nimbleModel(...) nodes: An optional character vector, specifying the nodes for which samplers should be cre-
ated. 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'. The default values for control list
arguments for samplers (if not otherwise provided as an argument to configureMCMC() or
addSampler()) are contained in the setup code of each sampling algorithm.
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 poste-
rior_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.

enableWAIC: A logical argument, specifying whether to enable WAIC calculations for the resulting MCMC algorithm. Defaults to the value of nimbleOptions('MCMCenableWAIC'), which in turn defaults to FALSE. Setting nimbleOptions('MCMCenableWAIC' = TRUE) will ensure that WAIC is enabled for all calls to `configureMCMC` and `buildMCMC`.

controlWAIC A named list of inputs that control the behavior of the WAIC calculation, passed as the 'control' input to `buildWAIC`. See `help(waic)`.

print: A logical argument specifying whether to print the monitors and samplers. Default is TRUE.

...: Additional named control list elements for default samplers, or additional arguments to be passed to the autoBlock function when autoBlock = TRUE.

printMonitors() Prints all current monitors and monitors2

Details:
See the initialize() function

printSamplers(..., ind, type, displayControlDefaults = FALSE, displayNonScalars = FALSE, displayConjugateDependencies = FALSE, executionOrder = FALSE, byType = FALSE)
Prints details of the MCMC samplers.
Arguments:
...: Character node or variable names, or numeric indices. Numeric indices may be used to specify the indices of the samplers to print, or character strings 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.

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.

type: a character vector containing sampler type names. Only samplers with one of these specified types, as printed by this printSamplers method, will be displayed. Standard regular expression matching using is also applied.

displayConjugateDependencies: A logical argument, specifying whether to display the dependency lists of conjugate samplers (default FALSE).

displayNonScalars: A logical argument, specifying whether to display the values of non-scalar control list elements (default FALSE).

executionOrder: A logical argument, specifying whether to print the sampler functions in the (possibly modified) order of execution (default FALSE).

byType: A logical argument, specifying whether the nodes being sampled should be printed, sorted and organized according to the type of sampler (the sampling algorithm) which is acting on the nodes (default FALSE).

removeSampler(...) Alias for removeSamplers method

removeSamplers(..., ind, print = FALSE) Removes one or more samplers from an MCMC-conf object.
This function also has the side effect of resetting the sampler execution ordering so as to iterate over the remaining set of samplers, sequentially, executing each sampler once.

Arguments:

...: Character node names or numeric indices. Character node names specify the node names for samplers to remove, or numeric indices can provide the indices of samplers to remove.

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 specifying whether to print the current list of samplers once the removal has been done (default FALSE).

replaceSampler(...) Alias for replaceSamplers method

replaceSamplers(...) Replaces one or more samplers from an MCMCconf object with newly specified sampler(s). Operation and arguments are identical to the 'addSampler' method, with the additional side effect of first removing any existing samplers which operate on the specified node(s).

This function also has the side effect of resetting the sampler execution ordering so as to iterate over the remaining set of samplers, sequentially, executing each sampler once.

See 'addSamplers' for a description of the arguments.

This function also has the side effect of resetting the sampler execution ordering so as to iterate over the newly specified set of samplers, sequentially, executing each sampler once.

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

Details:

See the initialize() function

setMonitors(..., ind = 1, print = TRUE) Sets new variables to the list of monitors.

Arguments:

...: One or more character vectors of indexed nodes, or variables, which are to be monitored. These replace the current monitors list.

print: A logical argument specifying whether to print all current monitors (default TRUE).

Details:

See the initialize() function

setMonitors2(..., print = TRUE) Sets new variables to the list of monitors2.

Arguments:

...: One or more character vectors of indexed nodes, or variables, which are to be monitored. These replace the current monitors2 list.

print: A logical argument specifying whether to print all current monitors (default TRUE).

Details:

See the initialize() function

setSampler(...) Alias for setSamplers method

setSamplerExecutionOrder(order, print = FALSE) Sets the ordering in which sampler functions will execute.

This allows some samplers to be "turned off", or others to execute multiple times in a single MCMC iteration. The ordering in which samplers execute can also be interleaved.
Arguments:

order: A numeric vector, specifying the ordering in which the sampler functions will execute. The indices of execution specified in this numeric vector correspond to the enumeration of samplers printed by printSamplers(), or returned by getSamplers(). If this argument is omitted, the sampler execution ordering is reset so as to sequentially execute each sampler once.

print: A logical argument specifying whether to print the current list of samplers in the modified order of execution (default FALSE).

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

This function also has the side effect of resetting the sampler execution ordering so as to iterate over the specified set of samplers, sequentially, executing each sampler once.

Arguments:

.... Character strings or numeric indices. Character names may be used to specify the node names for samplers to retain. Numeric indices may be used to specify the indices for the new list of MCMC samplers, in terms of the current ordered list of samplers.

ind: A numeric vector or character vector. A numeric vector may be used to specify the indices 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 specifying whether to print the new list of samplers (default FALSE).

setThin(thin, print = TRUE, ind = 1) 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 (default TRUE).

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 (default TRUE).

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$printMonitors()
conf$printSamplers()

modelBaseClass-class

Class modelBaseClass

Description

This class underlies all NIMBLE model objects: both R model objects created from the return value of \( \text{nimbleModel}() \), 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 \( \text{nimbleFunction}() \), to aid in preparation of node vectors, \( \text{nimbleFunctionLists} \), and other runtime inputs. See documentation for \( \text{nimbleModel} \) for details of creating an R model object.

Methods

calculate(nodes) See `help(calculate)`
calculateDiff(nodes) See `help(calculateDiff)`
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, restrictLink = NULL) 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(nodes, env = parent.frame(), returnScalarComponents = FALSE, returnType = "names", sort = FALSE, unique = TRUE)

Takes a vector of names of nodes or variables and returns the unique and expanded names in
the model, i.e. 'x' expands to 'x[1]', 'x[2]', ...

Arguments:

nodes: a vector of names of nodes (or variables) 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. dmmnorm 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?

unique: should names be the unique names or should original ordering of nodes (after expan-
sion of any variable names into node names) be preserved

getBound(node, bound)
See ‘help(getBound)’

code() Return the code for a model after processing if-then-else statements, expanding macros,
and replacing some keywords (e.g. nimStep for step) to avoid R ambiguity.

codeConditionallyIndependentSets(nodes, givenNodes, omit = integer(), explore = c("both", "down", "up"),

see "help(getConditionallyIndependentSets)", which this calls with the model as the first ar-
gument.

getConstants()
Return model constants, including any changes to the constants made by macros.

codeDependencies(nodes, omit = character(), self = TRUE, determOnly = FALSE, stochOnly = FALSE, includeData = TRUE)

Returns a character vector of the nodes dependent upon the input argument nodes, sorted topo-
logically according to the model graph. In the genealogical metaphor for a graphical model,
this function returns the "children" of the input nodes. In the river network metaphor, it re-
dults downstream nodes. By default, the returned nodes include the input nodes, include both
deterministic and stochastic nodes, and stop at stochastic nodes. Additional 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 structure,
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. If both determOnly and stochOnly are TRUE, no nodes will be returned.

includeData: Logical argument specifying whether to include 'data' nodes (set via nimble-
Model or the setData method). Default is TRUE.
dataOnly: Logical argument specifying whether to return only 'data' nodes. Default is FALSE.

includePredictive: Logical argument specifying whether to include predictive nodes. Predictive nodes are stochastic nodes that are not data and have no downstream stochastic dependents that are data. In Bayesian settings, these are "posterior predictive" nodes. Used primarily to exclude predictive node calculations when setting up MCMC samplers on model parameters. Default value is controlled by 'nimbleOptions("getDependenciesIncludesPredictiveNodes")', which has a default value of 'TRUE'.

predictiveOnly: Logical argument specifying whether to return only predictive nodes (see "includePredictive"). 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 hierarchical model 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 specifying type of object returned. Options are 'names' (returns character vector) and 'ids' (returns numeric graph IDs for model).

returnScalarComponents: Logical argument specifying whether multivariate nodes should be returned as full node names (i.e. 'x[1:2]') or as scalar components (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. Use getDependenciesList for a list of one-step dependent nodes of each node in the model.

getDependenciesList(returnNames = TRUE, sort = TRUE) Returns a list of all dependent 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.

getDimension(node, params = NULL, valueOnly = is.null(params) && !includeParams, includeParams = !is.null(params)) Determines the dimension of the value and/or parameters of the node

Arguments:
node: A character vector specifying a single node
params: an optional character vector of names of parameters for which dimensions are desired (possibly including 'value' and alternate parameters)

valueOnly: a logical indicating whether to only return the dimension of the value of the node
includeParams: a logical indicating whether to return dimensions of parameters. If TRUE and 'params' is NULL then dimensions of all parameters, including the dimension of the value of the node, are returned

Details: The return value is a numeric vector with an element for each parameter/value requested.
getDistribution(nodes) Returns the names of the distributions for the requested node or nodes

Arguments:
- nodes: A character vector specifying one or more node or variable names.

Details: The return value is a character vector with an element for each node indicated in the input. Note that variable names are expanded to their constituent node names, so the length of the output may be longer than that of the input.

defDownstream(...) Identical to getDependencies(..., downstream = TRUE)

Details: See documentation for member method getDependencies.

defLogProb(nodes) See 'help(getLogProb)'

defMacroInits() EXPERIMENTAL: Return initial values generated by macros.

defMacroParameters(includeLHS = TRUE, includeRHS = TRUE, includeDetermin = TRUE, includeStoch = TRUE, includeScalar = TRUE, includeStoch = TRUE, ...)

See 'help(defMacroParameters)'

defNodeNames(determOnly = FALSE, stochOnly = FALSE, includeData = TRUE, dataOnly = FALSE, includeRHSonly = FALSE, ... endOnly = FALSE, includePredictive = TRUE, predictiveOnly = FALSE, returnType = "names", returnScalarComponents = FALSE)

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.
- includePredictive: Logical argument specifying whether to include predictive nodes (stochastic nodes, which themselves are not data and have no downstream stochastic dependents which are data) from the hierarchical model structure.
- predictiveOnly: Logical argument specifying whether to return only predictive nodes (stochastic nodes, which themselves are not data and have no downstream stochastic dependents which are data) from the hierarchical model structure.
- returnType: Character argument specifying type of object returned. Options are 'names' (returns character vector) and 'ids' (returns numeric graph IDs for model)
- returnScalarComponents: Logical argument specifying whether multivariate nodes should return full node name (i.e. 'x[1:2]') or should break down into scalar components (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'.
getParam(node, param) See 'help(getParam)'

getParents( nodes, omit = character(), self = FALSE, determOnly = FALSE, stochOnly = FALSE, includeData = TRUE, dataOnly = FALSE, includeRHSonly = FALSE, upstream = FALSE, immediateOnly = FALSE, returnType = "names", returnScalarComponents = FALSE )

Returns a character vector of the nodes on which the input nodes depend, sorted topologically according to the model graph, by default recursing and stopping at stochastic parent nodes. In the genealogical metaphor for a graphical model, this function returns the "parents" of the input nodes. In the river network metaphor, it returns upstream nodes. By default, the returned nodes omit the input nodes. Additional input arguments provide flexibility in the values returned.

Arguments:

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

omit: Character vector of node names, which will be omitted from the nodes returned. In addition, parent nodes beyond these omitted nodes will not be returned. The omitted nodes argument serves to stop the upward search through the hierarchical model structure, 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 FALSE.

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. If both determOnly and stochOnly are TRUE, no nodes will be returned.

includeData: Logical argument specifying whether to include 'data' nodes (set via nimbleModel or the setData method). 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.

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

immediateOnly: Logical argument specifying whether only the immediate parent nodes should be returned, even if they are deterministic. If FALSE, getParents recurses and stops at stochastic nodes. Default is FALSE.

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

returnScalarComponents: Logical argument specifying whether multivariate nodes should be returned as full node names (i.e. 'x[1:2]') or as scalar components (i.e. 'x[1]' and 'x[2]').

Details: The upward search for dependent nodes propagates through deterministic nodes, but by default will halt at the first level of stochastic nodes encountered. Use getParentsList for a list of one-step parent nodes of each node in the model.

getParentsList(returnNames = TRUE, sort = TRUE) Returns a list of all parent neighbor relationships. Each list element gives the one-step parents 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 returned 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.

getVarNames(includeLogProb = FALSE, nodes) 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

initializeInfo(stochasticLogProbs = FALSE) Provides more detailed information on which model nodes are not initialized.

Arguments:
stochasticLogProbs: Boolean argument. If TRUE, the log-density value associated with each stochastic model variable is calculated and printed.

isBinary(nodes) Determines whether one or more nodes represent binary random variables

Arguments:

Details: The return value is a character vector with an element for each node indicated in the input. Note that variable names are expanded to their constituent node names, so the length of the output may be longer than that of the input.

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

Arguments:

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'. Multivariate nodes for which any elements are flagged as containing 'data' will be assigned a value of TRUE.

isDeterm(nodes, includeRHSonly = FALSE, nodesAlreadyExpanded = FALSE) Determines whether one or more nodes are deterministic

Arguments:

Details: The return value is a character vector with an element for each node indicated in the input. Note that variable names are expanded to their constituent node names, so the length of the output may be longer than that of the input.

isDiscrete(nodes) Determines whether one or more nodes represent discrete random variables

Arguments:
nodes: A character vector specifying one or more node or variable names.
Details: The return value is a character vector with an element for each node indicated in the
input. Note that variable names are expanded to their constituent node names, so the length of
the output may be longer than that of the input.

isEndNode(nodes) Determines whether one or more nodes are end nodes (nodes with no stochastic
dependences)
Arguments:
    nodes: A character vector specifying one or more node or variable names.
Details: The return value is logical vector with an element for each node indicated in the input.
Note that variable names are expanded to their constituent node names, so the length of
the output may be longer than that of the input.

isMultivariate(nodes) Determines whether one or more nodes represent multivariate nodes
Arguments:
    nodes: A character vector specifying one or more node or variable names.
Details: The return value is a logical vector with an element for each node indicated in the
input. Note that variable names are expanded to their constituent node names, so the length of
the output may be longer than that of the input.

isStoch(nodes, nodesAlreadyExpanded = FALSE) Determines whether one or more nodes are stochastic
Arguments:
    nodes: A character vector specifying one or more node or variable names.
    nodesAlreadyExpanded: Boolean argument indicating whether 'nodes' should be expanded. Generally intended for internal use. Default is 'FALSE'.
Details: The return value is a character vector with an element for each node indicated in the
input. Note that variable names are expanded to their constituent node names, so the length of
the output may be longer than that of the input.

isTruncated(nodes) Determines whether one or more nodes are truncated
Arguments:
    nodes: A character vector specifying one or more node or variable names.
Details: The return value is a character vector with an element for each node indicated in the
input. Note that variable names are expanded to their constituent nodes names, so the length
of the output may be longer than that of the input.

isUnivariate(nodes) Determines whether one or more nodes represent univariate random variables
Arguments:
    nodes: A character vector specifying one or more node or variable names.
Details: The return value is a character vector with an element for each node indicated in the
input. Note that variable names are expanded to their constituent nodes names, so the length
of the output may be longer than that of the input

newModel(data = NULL, inits = NULL, modelName = character(), replicate = FALSE, check = getNimbleOption("checkModel"), calculate = TRUE)
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.

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.

replicate: Logical specifying whether to replicate 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.

calculate: A logical indicating whether to run 'calculate' on the model; 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: 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 stochastic 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. This also allows one to set the 'data' flag for nodes appearing only on the right-hand side of model declarations, thereby preventing their values from being overwritten via 'inits'.

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 be ignored. Designating part of a multivariate node as 'data' and part as non-data (NA) is allowed, but 'isData()' will report such a node as being 'data', calculations with the node will generally return NA, and MCMC samplers will not be assigned to such nodes.

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.

simulate(nodes, includeData = FALSE) See help(simulate)

topologicallySortNodes(nodes, returnType = "names") Sorts the input list of node names according to the topological dependence ordering of the model structure.
Arguments:
  nodes: A character vector of node or variable 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 important prior to calls such as model$simulate(nodes) or model$calculate(nodes), to enforce that the operation is performed in topological order.

Author(s)
Daniel Turek

See Also
  initializeModel

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)
**modelInitialization**  
*Information on initial values in a NIMBLE model*

**Description**

Having uninitialized nodes in a NIMBLE model can potentially cause some algorithms to fail and can lead to poor performance in others. Here are some general guidelines on how non-initialized variables can affect performance:

- MCMC will auto-initialize but will do so from the prior distribution. This can cause slow convergence, especially in the case of diffuse priors.
- Likewise, particle filtering methods will initialize top-level parameters from their prior distributions, which can lead to errors or poor performance in these methods.


**modelValues**  
*Create a NIMBLE modelValues Object*

**Description**

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

**Usage**

```r
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

```r
# 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'][] <- matrix(1, nrow = 2, ncol = 2)
```

---

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 the User Manual for more details.

Examples

```r
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',]
```

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
model_macro_builder

**Examples**

```r
#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',]
```

---

**Description**

A model macro expands one line of code in a nimbleModel into one or more new lines. This supports compact programming by defining re-usable modules. `model_macro_builder` takes as input a function that constructs new lines of model code from the original line of code. It returns a function suitable for internal use by `nimbleModel` that arranges arguments for input function. Macros are an experimental feature and are available only after setting `nimbleOptions(enableModelMacros = TRUE)`.

**Usage**

```r
model_macro_builder(fun, use3pieces = TRUE, unpackArgs = TRUE)
```

**Arguments**

- **fun**
  - A function written to construct new lines of model code (see below).
- **use3pieces**
  - logical indicating whether the arguments from the input line be split into pieces for the LHS (left-hand side), RHS (right-hand side, possibly further split depending on unpackArgs), and stoch (TRUE if the line uses a `~` and FALSE otherwise). (default = TRUE)
- **unpackArgs**
  - logical indicating whether arguments be passed as a list (FALSE) or as separate arguments (TRUE). (default = TRUE)

**Details**

The arguments `use3pieces` and `unpackArgs` indicate how `fun` expects to have arguments arranged from an input line of code (processed by `nimbleModel`).

Consider the defaults `use3pieces = TRUE` and `unpackArgs = TRUE`, for a macro called `macro1`. In this case, the line of model code `x ~ macro1(arg1 = z[1:10], arg2 = "hello")` will be passed to `fun` as `fun(stoch = TRUE, LHS = x, arg1 = z[1:10], arg2 = "hello")`.

If `use3pieces = TRUE` but `unpackArgs = FALSE`, then the RHS will be passed as is, without unpacking its arguments into separate arguments to `fun`. In this case, `x ~ macro1(arg1 = z[1:10], arg2 = "hello")` will be passed to `fun` as `fun(stoch = TRUE, LHS = x, RHS = macro1(arg1 = z[1:10], arg2 = "hello")).`
If `use3pieces = FALSE` and `unpackArgs = FALSE`, the entire line of code is passed as a single object. In this case, \( x \sim \text{macro1}(\text{arg1} = z[1:10], \text{arg2} = \text{"hello"}) \) will be passed to `fun` as `fun(x \sim \text{macro1}(\text{arg1} = z[1:10], \text{arg2} = \text{"hello"}) )`. It is also possible in this case to pass a macro without using `~` or `<-`. For example, the line `macro1(\text{arg1} = z[1:10], \text{arg2} = \text{"hello"})` will be passed to `fun` as `fun(macro1(\text{arg1} = z[1:10], \text{arg2} = \text{"hello"}) )`.

If `use3pieces = FALSE` and `unpackArgs = TRUE`, it won’t make sense to anticipate a declaration using `~` or `<-`. Instead, arguments from an arbitrary call will be passed as separate arguments. For example, the line `macro1(\text{arg1} = z[1:10], \text{arg2} = \text{"hello"})` will be passed to `fun` as `fun(\text{arg1} = z[1:10], \text{arg2} = \text{"hello"})`.

In addition, the final two arguments of `fun` must be called `modelInfo` and `.env` respectively.

During macro processing, `nimbleModel` passes a named list to the `modelInfo` argument of `fun` containing, among other things, elements called `constants` and `dimensions`. Macro developers can modify these two elements (for example, to add a new constant needed for a macro) and these changes will be reflected in the final model object. Note that currently it is not possible for a macro to modify the data. Furthermore, if your macro add a new element to the constants that `nimbleModel` then moves to the data, this new data will not be retained in the final model object and thus will not be usable.

`nimbleModel` passes the R environment from which `nimbleModel` was called to the `.env` argument.

The `fun` function must return a named list with two elements: `code`, the replacement code, and `modelInfo`, the `modelInfo` list described above. `modelInfo` must be in the output even if the macro does not modify it.

It is extremely useful to be familiar with processing R code as an object to write `fun` correctly. Functions such as `substitute` and `as.name` (e.g. `as.name('~')`), `quote`, `parse` and `deparse` are particularly handy.

Multiple lines of new code should be contained in `{}`. Extra curly braces are not a problem. See example 2.

Macro expansion is done recursively: One macro can return code that invokes another macro.

**Value**

A list of class `model_macro` with one element called `process`, which contains the macro function suitable for use by `nimbleModel`.

**Examples**

```r
nimbleOptions(enableModelMacros = TRUE)
nimbleOptions(enableMacroComments = FALSE)
nimbleOptions(quiet = FALSE)

## Example 1: Say one is tired of writing "for" loops.
## This macro will generate a "for" loop with dnorm declarations
all_dnorm <- model_macro_builder(
  function(stoch, LHS, RHSVar, start, end, sd = 1, modelInfo, .env) {
    newCode <- substitute(
    for(i in START:END) {
      LHS[i] ~ dnorm(RHSVar[i], SD)
    })
  })
```

model_macro_builder

list(START = start,
    END = end,
    LHS = LHS,
    RHSvar = RHSvar,
    SD = sd))
list(code = newCode)
}, use3pieces = TRUE,
unpackArgs = TRUE
)

model1 <- nimbleModel(
nimbleCode(
    {
        ## Create a "for" loop of dnorm declarations by invoking the macro
        x ~ all_dnorm(mu, start = 1, end = 10)
    }
))
## show code from expansion of macro
model1$getCode()
## The result should be:
## {
##   for (i in 1:10) {
##       x[i] ~ dnorm(mu[i], 1)
##   }
##}
## Example 2: Say one is tired of writing priors.
## This macro will generate a set of priors in one statement
flat_normal_priors <- model_macro_builder(
    function(..., modelInfo, .env) {
        allVars <- list(...) 
        priorDeclarations <- lapply(allVars,
            function(x)
                substitute(VAR ~ dnorm(0, sd = 1000),
                list(VAR = x))
        )
        newCode <- quote({})
        newCode[2:(length(allVars)+1)] <- priorDeclarations
        list(code = newCode)
    }, use3pieces = FALSE,
    unpackArgs = TRUE
)

model2 <- nimbleModel(
nimbleCode(
    {
        flat_normal_priors(mu, beta, gamma)
    }
))
## show code from expansion of macro
model2$getCode()
## The result should be:
## {
## mu ~ dnorm(0, sd = 1000)
## beta ~ dnorm(0, sd = 1000)
## gamma ~ dnorm(0, sd = 1000)
## }

## Example 3: Macro that modifies constants
new_constant <- model_macro_builder(
    function(stoch, LHS, RHS, modelInfo, .env) {
        # number of elements
        n <- as.numeric(length(modelInfo$constants[[deparse(LHS)]]))
        code <- substitute({
            for (i in 1:N)
                L[i] ~ dnorm(mu[i], 1)
        }, list(L = LHS, N = n))
        # Add a new constant mu
        modelInfo$constants$mu <- rnorm(n, 0, 1)
        list(code = code, modelInfo = modelInfo)
    },
    use3pieces = TRUE,
    unpackArgs = TRUE
)

const <- list(y = rnorm(10))
code <- nimbleCode({
    y ~ new_constant()
})

mod <- nimbleModel(code = code, constants=const)
mod$getCode()
mod$getConstants() # new constant is here

---

**Multinomial**

The Multinomial Distribution

**Description**

Density and random generation for the multinomial distribution

**Usage**

```r
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, internally normalized to sum 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**


**See Also**

Distributions for other standard distributions

**Examples**

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

---

**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**

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

rmvtnorm(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, $\Sigma^{-1}$, given in Gelman et al.

Value

dmv_t_chol gives the density and rmv_t_chol generates random deviates.

Author(s)

Peter Sujan

References


See Also

- **Distributions** for other standard distributions

Examples

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

```r
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


See Also

`Distributions` for other standard distributions
Examples

```r
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

```r
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

```r
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')
    }
  },
  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()
```

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  \hspace{2cm} Mathematical functions for BUGS and nimbleFunction programming

Description

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

Author(s)

NIMBLE Development Team
nimble-R-functions  

NIMBLE language functions for R-like vector construction

Description

The functions `c`, `rep`, `seq`, `which`, `diag`, `length`, `seq_along`, `is.na`, `is.nan`, `any`, and `all` can be used in `nimbleFunctions` and compiled using `compileNimble`.

Usage

nimC(...)

nimRep(x, ...)

nimSeq(from, to, by, length.out)

Arguments

... values to be concatenated.

x vector of values to be replicated (rep), or logical array or vector (which), or object whose length is wanted (length), or input value (diag), or vector of values to be tested/checked (is.na, is.nan, any, all).

from starting value of sequence.

to end value of sequence.

by increment of the sequence.

length.out desired length of the sequence.

Details

For `c`, `rep`, `seq`, these functions are NIMBLE's version of similar R functions, e.g., `nimRep` for `rep`. In a `nimbleFunction`, either the R name (e.g., `rep`) or the NIMBLE name (e.g., `nimRep`) can be used. If the R name is used, it will be converted to the NIMBLE name. For `which`, `length`, `diag`, `seq_along`, `is.na`, `is.nan`, `any`, all simply use the standard name without "nim". These functions largely mimic (see exceptions below) the behavior of their R counterparts, but they can be compiled in a `nimbleFunction` using `compileNimble`.

`nimC` is NIMBLE's version of `c` and behaves identically.

`nimRep` is NIMBLE's version of `rep`. It should behave identically to `rep`. There are no NIMBLE versions of `rep.int` or `rep_len`.

`nimSeq` is NIMBLE's version of `seq`. It behaves like `seq` with support for `from`, `to`, `by` and `length.out` arguments. The `along.with` argument is not supported. There are no NIMBLE versions of `seq.int`, `seq_along` or `seq_len`, with the exception that `seq_along` can take a `nimbleFunctionList` as an argument to provide the index range of a for-loop (User Manual Ch. 13).

which behaves like the R version but without support for `arr.ind` or `useNames` arguments.

diag behaves like the R version but without support for the `nrow` and `ncol` arguments.
length behaves like the R version.
seq_along behaves like the R version.
is.na behaves like the R version but does not correctly handle NA values from R that are type 'logical', so convert these using as.numeric() before passing from R to NIMBLE.
is.nan behaves like the R version, but treats NA of type 'double' as being NaN and NA of type 'logical' as not being NaN.
any behaves like the R version but takes only one argument and treats NAs as FALSE.
all behaves like the R version but takes only one argument and treats NAs as FALSE.

## 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

```r
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

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

Create a nimbleFunction that wraps a call to external compiled code

Description

Given C header information, a function that takes scalars or pointers can be called from a compiled nimbleFunction. If non-scalar return values are needed, an argument can be selected to behave as the return value in nimble.

Usage

nimbleExternalCall(
  prototype,
  returnType,
  Cfun,
  headerFile,
  oFile,
  where = getNimbleFunctionEnvironment()
)

Arguments

- **prototype**: Argument type information. This can be provided as an R function using nimbleFunction type declarations or as a list of nimbleType objects.
- **returnType**: Return object type information. This can be provided similarly to prototype as either a nimbleFunction type declaration or as a nimbleType object. In the latter case, the name will be ignored. If there is no return value, this should be void().
- **Cfun**: Name of the external function (character).
- **headerFile**: Name (possibly including file path) of the header file where Cfun is declared.
- **oFile**: Name (possibly including path) of the .o file where Cfun has been compiled. Spaces in the path may cause problems.
- **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

The only argument types allowed in Cfun are double, int, and bool, corresponding to nimbleFunction types double, integer, and logical, respectively.

If the dimensionality is greater than zero, the arguments in Cfun should be pointers. This means it will typically be necessary to pass additional integer arguments telling Cfun the size(s) of non-scalar arguments.
The return argument can only be a scalar or void. Since non-scalar arguments are passed by pointer, you can use an argument to return results from Cfun. If you wish to have a nimbleFunction that uses one argument of Cfun as a return object, you can wrap the result of nimbleExternalCall in another nimbleFunction that allocates the return object. This is useful for using Cfun in a nimbleModel. See example below.

Note that a nimbleExternalCall can only be executed in a compiled nimbleFunction, not an uncompiled one.

If you have problems with spaces in file paths (e.g. for oFile), try compiling everything locally by including dirName = "." as an argument to compileNimble.

Value

A nimbleFunction that takes the indicated input arguments, calls Cfun, and returns the result.

Author(s)

Perry de Valpine

See Also

nimbleRcall for calling arbitrary R code from compiled nimbleFunctions.

Examples

```r
## Not run:
sink("add1.h")
cat(""
   extern "C" {
      void my_internal_function(double *p, double*ans, int n);
   }
"
)sink()
sink("add1.cpp")
cat(""
#include <cstdio>
#include "add1.h"
void my_internal_function(double *p, double *ans, int n) {
   printf("In my_internal_function\n");
   /* cat reduces the double slash to single slash */
   for(int i = 0; i < n; i++)
      ans[i] = p[i] + 1.0;
}
"
)sink()
system("g++ add1.cpp -c -o add1.o")
Radd1 <- nimbleExternalCall(function(x = double(1), ans = double(1), n = integer()){}, Cfun = 'my_internal_function',
headerFile = file.path(getwd(), 'add1.h'), returnType = void(),
oFile = file.path(getwd(), 'add1.o'))
## If you need to use a function with non-scalar return object in model code,
## you can wrap it in another nimbleFunction like this:
```

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,
  buildDerivs = list(),
  name = NA,
  check = getNimbleOption("checkNimbleFunction"),
  where = getNimbleFunctionEnvironment()
)

Arguments

  setup An optional R function definition for setup processing.
  run An optional NIMBLE function definition that 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.

buildDerivs  A list of names of function methods for which to build derivatives capabilities.

name  An optional name used internally, for example in generated C++ code. Usually this is left blank and NIMBLE provides a name.

check  Boolean indicating whether to check the run code for function calls that NIMBLE cannot compile. Checking can be turned off for all calls to `nimbleFunction` using `nimbleOptions(checkNimbleFunction = FALSE)`.

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,
  methodControl = list()
)

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 returnType.
name An optional name used internally by the NIMBLE compiler. This is usually omitted and NIMBLE provides one.
methodControl An optional list that allows specification of methods with defaults.
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

---

nimbleList create a nimbleList

Description

create a nimbleList from a nimbleList definition

Usage

nimbleList(
  ..., 
  name = as.character(NA), 
  predefined = FALSE, 
  where = getNimbleFunctionEnvironment()
)

Arguments

... arbitrary set of names and types for the elements of the list or a single R list of type nimbleType.

name optional character providing a name used internally, for example in generated C++ code. Usually this is left blank and NIMBLE provides a name.

predefined logical for internal use only.

where optional 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 function creates a definition for a nimbleList. The types argument defines the names, types, and dimensions of the elements of the nimbleList. Elements of nimbleLists can be either basic types (e.g., integer, double) or other nimbleList definitions. The types argument can be either a series of expressions of the form name = type(dim), or a list of nimbleType objects.

nimbleList returns a definition, which can be used to create instances of this type of nimbleList via the new() member function.

Definitions can be created in R’s general environment or in nimbleFunction setup code. Instances can be created using the new() function in R’s global environment, in nimbleFunction setup code, or in nimbleFunction run code.

Instances of nimbleList definitions can be used as arguments to run code of nimbleFunctions, and as the return type of nimbleFunctions.

Author(s)

NIMBLE development team

Examples

exampleNimListDef <- nimbleList(x = integer(0), Y = double(2))

nimbleListTypes <- list(nimbleType(name = 'x', type = 'integer', dim = 0),
                        nimbleType(name = 'Y', type = 'double', dim = 2))

## this nimbleList definition is identical to the one created above
exampleNimListDef <- nimbleList(nimbleListTypes)
dimensions = list(),
model,
monitors,
thin = 1,
niter = 10000,
nburnin = 0,
nchains = 1,
check = TRUE,
setSeed = FALSE,
progressBar = getNimbleOption("MCMCprogressBar"),
samples = TRUE,
samplesAsCodaMCMC = FALSE,
summary = FALSE,
WAIC = FALSE
)

Arguments

code The quoted code expression representing the model, such as the return value from a call to nimbleCode). Not required if model is provided.

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. Values that are NA will not be flagged as data.

inits Argument to specify initial values for each MCMC chain. See details.

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.

model A compiled or uncompiled NIMBLE model object. When provided, this model will be used to configure the MCMC algorithm to be executed, rather than using the code, constants, data and inits arguments to create a new model object. However, if also provided, the inits argument will still be used to initialize this model prior to running each MCMC chain.

monitors A character vector giving the node names or variable names to monitor. The samples corresponding to these nodes will returned, and/or will have summary statistics calculated. Default value is all top-level stochastic nodes of the model.

thin Thinning interval for collecting MCMC samples. Thinning occurs after the initial nburnin samples are discarded. Default value is 1.

niter Number of MCMC iterations to run. Default value is 10000.

nburnin Number of initial, pre-thinning, MCMC iterations to discard. Default value is 0.

nchains Number of MCMC chains to run. Default value is 1.

check Logical argument, specifying whether to check the model object for missing or invalid values. Default value is TRUE.
**setSeed** Logical or numeric argument. If a single numeric value is provided, R’s random number seed will be set to this value at the onset of each MCMC chain. If a numeric vector of length `nchains` is provided, then each element of this vector is provided as R’s random number seed at the onset of the corresponding MCMC chain. Otherwise, in the case of a logical value, if `TRUE`, then R’s random number seed for the ith chain is set to be i, at the onset of each MCMC chain. Note that specifying the argument `setSeed = 0` does not prevent setting the RNG seed, but rather sets the random number generation seed to 0 at the beginning of each MCMC chain. Default value is `FALSE`.

**progressBar** Logical argument. If `TRUE`, an MCMC progress bar is displayed during execution of each MCMC chain. Default value is defined by the nimble package option `MCMCprogressBar`.

**samples** Logical argument. If `TRUE`, then posterior samples are returned from each MCMC chain. These samples are optionally returned as `coda mcmc` objects, depending on the `samplesAsCodaMCMC` argument. Default value is `TRUE`. See details.

**samplesAsCodaMCMC** 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. This argument is only used when `samples` is `TRUE`. Default value is `FALSE`. See details.

**summary** Logical argument. When `TRUE`, summary statistics for the posterior samples of each parameter are also returned, for each MCMC chain. This may be returned in addition to the posterior samples themselves. Default value is `FALSE`. See details.

**WAIC** Logical argument. When `TRUE`, the WAIC (Watanabe, 2010) of the model is calculated and returned. If multiple chains are run, then a single WAIC value is calculated using the posterior samples from all chains. Default value is `FALSE`. Note that the version of WAIC used is the default WAIC conditional on random effects/latent states and without any grouping of data nodes. See `help(waic)` for more details. If a different version of WAIC is desired, do not use `nimbleMCMC`. Instead, specify the `controlWAIC` argument to `configureMCMC` or `buildMCMC`, and then use `runMCMC`.

**Details**

The entry point for this function is providing the code, constants, data and `inits` arguments, to create a new NIMBLE model object, or alternatively providing an existing NIMBLE model object as the model argument.

At least one of `samples`, `summary` or `WAIC` must be `TRUE`, since otherwise, nothing will be returned. Any combination of these may be `TRUE`, including possibly all three, in which case posterior samples, summary statistics, and WAIC values are returned for each MCMC chain.

When `samples = TRUE`, the form of the posterior samples is determined by the `samplesAsCodaMCMC` argument, as either matrices of posterior samples, or `coda mcmc` and `mcmc.list` objects.

Posterior summary statistics are returned individually for each chain, and also as calculated from all chains combined (when `nchains > 1`).

The `inits` argument can be one of three things:
(1) a function to generate initial values, which will be executed once to initialize the model object,
and once to generate initial values at the beginning of each MCMC chain, or (2) a single named list
of initial values which, will be used to initialize the model object and for each MCMC chain, or (3)
a list of length nchains, each element being a named list of initial values. The first element will be
used to initialize the model object, and once element of the list will be used for each MCMC chain.
The inits argument may also be omitted, in which case the model will not be provided with initial
values. This is not recommended.

The niter argument specifies the number of pre-thinning MCMC iterations, and the nburnin ar-
gument specifies the number of pre-thinning MCMC samples to discard. After discarding these
burn-in samples, thinning of the remaining samples will take place. The total number of posterior
samples returned will be floor((niter-nburnin)/thin).

Value

A list is returned with named elements depending on the arguments passed to nimbleMCMC, unless
only one among samples, summary, and WAIC are requested, in which case only that element is
returned. These elements may include samples, summary, and WAIC. When nchains = 1, posterior
samples are returned as a single matrix, and summary statistics as a single matrix. When nchains
> 1, posterior samples are returned as a list of matrices, one matrix for each chain, and summary
statistics are returned as a list containing nchains+1 matrices: one matrix corresponding to each
chain, and the final element providing a summary of all chains, combined. If samplesAsCodaMCMC is
TRUE, then posterior samples are provided as coda mcmc and mcmc.list objects. When WAIC is
TRUE, a WAIC summary object is returned.

Author(s)

Daniel Turek

See Also

configureMCMC buildMCMC runMCMC

Examples

```r
## Not run:

code <- nimbleCode(
  mu ~ dnorm(0, sd = 1000)
  sigma ~ dunif(0, 1000)
  for(i in 1:10) {
    x[i] ~ dnorm(mu, sd = sigma)
  }
)

data <- list(x = c(2, 5, 3, 4, 1, 0, 1, 3, 5, 3))
inits <- function() list(mu = rnorm(1,0,1), sigma = runif(1,0,10))
mcmc.output <- nimbleMCMC(code, data = data, inits = inits,
  monitors = c("mu", "sigma"), thin = 10,
  niter = 20000, nburnin = 1000, nchains = 3,
  summary = TRUE, WAIC = TRUE)

## End(Not run)
```
nimbleModel

Create a NIMBLE model from BUGS code

Description

Processes BUGS model code and optional constants, data, and initial values. Returns a NIMBLE model (see modelBaseClass) 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 = NULL,
  buildDerivs = getNimbleOption("buildModelDerivs"),
  userEnv = parent.frame()
)

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).
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 `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.

buildDerivs

logical indicating whether to build derivative capabilities for the model.

userEnv

environment in which if-then-else statements in BUGS code will be evaluated if needed information not found in `constants`; intended primarily for internal use only

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

See Also

`readBUGSmodel` for creating models from BUGS-format model files
Examples

```r
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. Call `nimbleOptions()` with no arguments to see a list of available options.

**Usage**

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

**Arguments**

`...` any options to be defined as one or more `name = value` pairs or as a single list of `name=value` pairs.

**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, returns a list with the current values of all options. When invoked with one or more arguments, returns a list of the the updated options with their updated values.

**Author(s)**

Christopher Paciorek
Examples

# Set one option:
nimbleOptions(verifyConjugatePosteriors = FALSE)

# Compactly print all options:
str(nimbleOptions(), max.level = 1)

# Save-and-restore options:
old <- nimbleOptions() # Saves old options.
nimbleOptions(showCompilerOutput = TRUE,
               verboseErrors = TRUE) # Sets temporary options.
# ...do stuff...
nimbleOptions(old) # Restores old options.

nimbleRcall Make an R function callable from compiled nimbleFunctions (including nimbleModels).

Description

Normally compiled nimbleFunctions call other compiled nimbleFunctions. nimbleRcall enables any R function (with viable argument types and return values) to be called (and evaluated in R) from compiled nimbleFunctions.

Usage

nimbleRcall(
  prototype,
  returnType,
  Rfun,
  where = getNimbleFunctionEnvironment()
)

Arguments

prototype Argument type information for Rfun. This can be provided as an R function using nimbleFunction type declarations or as a list of nimbleType objects.
returnType Return object type information. This can be provided similarly to prototype as either a nimbleFunction type declaration or as a nimbleType object. In the latter case, the name will be ignored. If there is no return value this should be void().
Rfun The name of an R function to be called from compiled nimbleFunctions.
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

The `nimbleFunction` returned by `nimbleRcall` can be used in other `nimbleFunction`s. When called from a compiled `nimbleFunction` (including from a model), arguments will be copied according to the declared types, the function named by `Rfun` will be called, and the returned object will be copied if necessary. The example below shows use of an R function in a compiled `nimbleModel`.

A `nimbleFunction` returned by `nimbleRcall` can only be used in a compiled `nimbleFunction`. `Rfun` itself should work in an uncompiled `nimbleFunction`.

Value

A `nimbleFunction` that wraps a call to `Rfun` with type-declared arguments and return object.

Author(s)

Perry de Valpine

See Also

`nimbleExternalCall` for calling externally provided C (or other) compiled code.

Examples

```r
## Not run:
## Say we want an R function that adds 2 to every value in a vector
add2 <- function(x) {
  x + 2
}
Radd2 <- nimbleRcall(function(x = double(1)){}, Rfun = 'add2',
  returnType = double(1))
demoCode <- nimbleCode({
  for(i in 1:4) {x[i] ~ dnorm(0,1)}
  z[1:4] <- Radd2(x[1:4])
})
demoModel <- nimbleModel(demoCode, inits = list(x = rnorm(4)),
  check = FALSE, calculate = FALSE)
CdemoModel <- compileNimble(demoModel)

## End(Not run)
```

Description

Create a `nimbleType` object, with information on the name, type, and dimension of an object to be placed in a `nimbleList`.
Arguments

name  The name of the object, given as a character string.

type  The type of the object, given as a character string.

dim   The dimension of the object, given as an integer. This can be left blank if the object is a nimbleList.

Details

This function creates `nimbleType` objects, which can be used to define the elements of a `nimbleList`.

The `type` argument can be chosen from among character, double, integer, and logical, or can be the name of a previously created `nimbleList` definition.

See the NIMBLE User Manual for additional examples.

Author(s)

NIMBLE development team

Examples

```r
nimbleTypeList <- list()
nimbleTypeList[[1]] <- nimbleType(name = 'x', type = 'integer', dim = 0)
nimbleTypeList[[2]] <- nimbleType(name = 'Y', type = 'double', dim = 2)
```

Description

cat function for use in nimbleFunctions

Usage

```r
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.

For numeric values, the number of digits printed is controlled by the system option `nimbleOptions('digits')`.

See Also

`print`

Examples

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

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

Arguments

from Either a NIMBLE model or modelValues object
to Either a NIMBLE model or modelValues object
nodes Vector of one or more node names of object from that will be copied from
nodesTo Vector of one or more node names of object to that will be copied to. If nodesTo is NULL, will automatically be set to nodes
row If from is a modelValues, the row that will be copied from
rowTo If to is a modelValues, the row which will be copied to. If rowTo is 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)
logProbOnly A logical value indicating whether only the log probabilities of the given nodes should be copied (i.e. if nodes = 'x' and logProbOnly = TRUE, then only logProb_x will be copied)

details

This function copies values from one or more nodes (possibly including log probabilities for nodes) between models and modelValues objects. For modelValues objects, the row must be specified.
This function allows one to conveniently copy multiple nodes, avoiding having to write a loop.

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)

nimDerivs  

Nimble Derivatives

Description
Computes the value, 1st order (Jacobian), and 2nd order (Hessian) derivatives of a given nimbleFunction method and/or model log probabilities

Usage
nimDerivs(call = NA, wrt = NULL, order = nimC(0, 1, 2), model = NA, ...)

Arguments
call  
a call to a nimbleFunction method with arguments included. Can also be a call to model$calculate(nodes), or to calculate(model, nodes).

wrt  
a character vector of either: names of function arguments (if taking derivatives of a nimbleFunction method), or node names (if taking derivatives of model$calculate(nodes)) to take derivatives with respect to. If left empty, derivatives will be taken with respect to all arguments to nimFxn.

order  
an integer vector with values within the set 0, 1, 2, corresponding to whether the function value, Jacobian, and Hessian should be returned respectively. Defaults to c(0, 1, 2).

model  
(optional) for derivatives of a nimbleFunction that involves model. calculations, the uncompiled model that is used. This is needed in order to be able to correctly restore values into the model when order does not include 0 (or in all cases when double-taping).

...  
additional arguments intended for internal use only.

Details
Derivatives for uncompiled nimbleFunctions are calculated using the numDeriv package. If this package is not installed, an error will be issued. Derivatives for matrix valued arguments will be returned in column-major order.
Value

a nimbleList with elements value, jacobian, and hessian.

Examples

```r
## Not run:
model <- nimbleModel(code = ...)
calcDerivs <- nimDerivs(model$calculate(model$getDependencies('x')),
  wrt = 'x')

## End(Not run)
```

### nimDim

- **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)`
nimEigen  

Spectral Decomposition of a Matrix

Description
Computes eigenvalues and eigenvectors of a numeric matrix.

Usage
nimEigen(x, symmetric = FALSE, only.values = FALSE)

Arguments
x a numeric matrix (double or integer) whose spectral decomposition is to be computed.
symmetric if TRUE, the matrix is guaranteed to be symmetric, and only its lower triangle (diagonal included) is used. Otherwise, the matrix is checked for symmetry. Default is FALSE.
only.values if TRUE, only the eigenvalues are computed, otherwise both eigenvalues and eigenvectors are computed. Setting only.values = TRUE can speed up eigendecompositions, especially for large matrices. Default is FALSE.

Details
Computes the spectral decomposition of a numeric matrix using the Eigen C++ template library. In a nimbleFunction, eigen is identical to nimEigen. If the matrix is symmetric, a faster and more accurate algorithm will be used to compute the eigendecomposition. Note that non-symmetric matrices can have complex eigenvalues, which are not supported by NIMBLE. If a complex eigenvalue or a complex element of an eigenvector is detected, a warning will be issued and that element will be returned as NaN.

Additionally, returnType(eigenNimbleList()) can be used within a link{nimbleFunction} to specify that the function will return a nimbleList generated by the nimEigen function. eigenNimbleList() can also be used to define a nested nimbleList element. See the User Manual for usage examples.

Value
The spectral decomposition of x is returned as a nimbleList with elements:

- values vector containing the eigenvalues of x, sorted in decreasing order. Since x is required to be symmetric, all eigenvalues will be real numbers.
- vectors. matrix with columns containing the eigenvectors of x, or an empty matrix if only.values is TRUE.

Author(s)
NIMBLE development team
nimIntegrate

Integration of One-Dimensional Functions

Description

NIMBLE wrapper around R’s builtin integrate. Adaptive quadrature of functions of one variable over a finite or infinite interval.

Usage

nimIntegrate(
  f,
  lower,
  upper,
  param,
  subdivisions = 100L,
  rel.tol = .Machine$double.eps^0.25,
  abs.tol = .Machine$double.eps^0.25,
  stop.on.error = TRUE
)

Arguments

  f          nimbleFunction of one input for which the integral is desired. See below for details on requirements for how f must be defined.
  lower     an optional scalar lower bound for the input of the function.
  upper     an optional scalar upper bound for the input of the function.
  param     additional parameter(s) to the function that are fixed with respect to the integration. If f takes no additional arguments (beyond the variable of integration), this must be provided but need not be used in f. Can be of length one or more.

See Also

nimSvd for singular value decompositions in NIMBLE.

Examples

eigenvaluesDemoFunction <- nimbleFunction(
  setup = function(){
    demoMatrix <- diag(4) + 2
  },
  run = function(){
    eigenvalues <- eigen(demoMatrix, symmetric = TRUE)$values
    returnType(double(1))
    return(eigenvalues)
  })
subdivisions  the maximum number of subintervals.
rel.tol     relative accuracy requested.
abs.tol     absolute accuracy requested.
stop.on.error  logical. If TRUE (the default) an error stops the function. Otherwise some errors will give a result with the error code given in the third element of the result vector.

Details

The function \( f \) should take two arguments, the first of type double(1), i.e., vector. \( f \) should be vectorized in that it should also return a double(1) object, containing the result of applying the function to each element of the first argument. (The result can be calculated using vectorized NIMBLE code or using a loop.) The second argument is required to also be of type double(1), containing any additional parameter(s) to the function that are not being integrated over. This argument can be unused in the function if the function does not need additional parameters. Note that this must be of type double(1) even if \( \text{param} \) contains a single element (NIMBLE will manage the lengths behind the scenes).

Note that unlike with R’s integrate, additional parameters must be passed as part of a vector, specified via \( \text{param} \), and cannot be passed as individual named arguments.

Value

A vector with three values, the first the estimate of the integral, the second an estimate of the modulus of the absolute error, and the third a result code corresponding to the message returned by integrate. The numerical result code can be interpreted as follows:

- 0: "OK"
- 1: "maximum number of subdivisions reached"
- 2: "roundoff error was detected"
- 3: "extremely bad integrand behaviour"
- 4: "roundoff error is detected in the extrapolation table"
- 5: "the integral is probably divergent"
- 6: "the input is invalid"

Author(s)

Christopher Paciorek, Paul van Dam-Bates, Perry de Valpine

See Also

integrate
Examples

```r
integrand <- nimbleFunction(
  run = function(x = double(1), theta = double(1)) {
    return(x*theta[1])
    returnType(double(1))
  }
)

fun <- nimbleFunction(
  run = function(theta = double(0), lower = double(0), upper = double(0)) {
    param = c(theta, 0)  # cannot be scalar, so pad with zero.
    output = integrate(integrand, lower, upper, param)
    returnType(double(1))
    return(output)
  }
)

fun(3.1415927, 0, 1)
```

## Not run:
```r
cfun <- compileNimble(fun)
cfun(3.1415927, 0, 1)
```
## End(Not run)

---

**nimMatrix**

*Creates matrix or array objects for use in nimbleFunctions*

**Description**

In a `nimbleFunction`, `matrix` and `array` are identical to `nimMatrix` and `nimArray`, respectively.

**Usage**

```r
nimMatrix(
  value = 0,
  nrow = NA,
  ncol = NA,
  init = TRUE,
  fillZeros = TRUE,
  recycle = TRUE,
  type = "double"
)
```

```r
nimArray(
  value = 0,
  dim = c(1, 1),
  init = TRUE,
  fillZeros = TRUE,
```
nimMatrix

recycle = TRUE,

) 

Arguments

value value(s) for initialization (default = 0). This can be a vector, matrix or array, but it will be used as a vector.
nrow the number of rows in a matrix (default = 1)
ncol the number of columns in a matrix (default = 1)
init logical, whether to initialize values (default = TRUE)
fillZeros logical, whether to initialize any elements not filled by (possibly recycled) value with 0 (or FALSE for nimLogical) (default = TRUE)
recycle logical, whether value should be recycled to fill the entire contents of the new object (default = TRUE)
type character representing the data type, i.e. 'double', 'integer', or 'logical' (default = 'double')
dim vector of dimension sizes in an array (default = c(1, 1))
nDim number of dimensions in an array. This is only necessary for compileNimble if the length of dim cannot be determined during compilation.

Details

These functions are similar to R’s matrix and array functions, but they can be used in a nimbleFunction and compiled using compileNimble. Largely for compilation purposes, finer control is provided over initialization behavior, similarly to nimNumeric, nimInteger, and nimLogical. If init = FALSE, no initialization will be done, and value, fillZeros and recycle will be ignored. If init=TRUE and recycle=TRUE, then fillZeros will be ignored, and value will be repeated (according to R’s recycling rule) as much as necessary to fill the object. If init=TRUE and recycle=FALSE, then if fillZeros=TRUE, values of 0 (or FALSE for nimLogical) will be filled in after value. Compiled code will be more efficient if unnecessary initialization is not done, but this may or may not be noticeable depending on the situation.

When used in a nimbleFunction (in run or other member function), matrix and array are immediately converted to nimMatrix and nimArray, respectively.

The nDim argument is only necessary for a use like dim <- c(2, 3, 4); A <- nimArray(0, dim = dim, nDim = 3). It is necessary because the NIMBLE compiler must determine during compilation that A will be a 3-dimensional numeric array. However, the compiler doesn’t know for sure what the length of dim will be at run time, only that it is a vector. On the other hand, A <- nimArray(0, dim = c(2, 3, 4)) is allowed because the compiler can directly determine that a vector of length three is constructed inline for the dim argument.

Author(s)

Daniel Turek and Perry de Valpine
nimNumeric

See Also

nimNumeric, nimInteger, nimLogical

Description

In a nimbleFunction, numeric, integer and logical are identical to nimNumeric, nimInteger and nimLogical, respectively.

Usage

nimNumeric(
  length = 0,
  value = 0,
  init = TRUE,
  fillZeros = TRUE,
  recycle = TRUE
)

nimInteger(
  length = 0,
  value = 0,
  init = TRUE,
  fillZeros = TRUE,
  recycle = TRUE
)

nimLogical(
  length = 0,
  value = 0,
  init = TRUE,
  fillZeros = TRUE,
  recycle = TRUE
)

Arguments

- length: the length of the vector (default = 0)
- value: value(s) for initializing the vector (default = 0). This may be a vector, matrix or array but will be used as a vector.
- init: logical, whether to initialize elements of the vector (default = TRUE)
- fillZeros: logical, whether to initialize any elements not filled by (possibly recycled) value with 0 (or FALSE for nimLogical) (default = TRUE)
- recycle: logical, whether value should be recycled to fill the entire length of the new vector (default = TRUE)
Details

These functions are similar to R’s numeric, integer, logical functions, but they can be used in a nimbleFunction and then compiled using compileNimble. Largely for compilation purposes, finer control is provided over initialization behavior. If init = FALSE, no initialization will be done, and value, fillZeros and recycle will be ignored. If init = TRUE and recycle = TRUE, then fillZeros will be ignored, and value will be repeated (according to R’s recycling rule) as much as necessary to fill a vector of length length. If init = TRUE and recycle = FALSE, then if fillZeros = TRUE, values of 0 (or FALSE for nimLogical) will be filled in after value up to length length. Compiled code will be more efficient if unnecessary initialization is not done, but this may or may not be noticeable depending on the situation.

When used in a nimbleFunction (in run or other member function), numeric, integer and logical are immediately converted to nimNumeric, nimInteger and nimLogical, respectively.

Author(s)

Daniel Turek, Christopher Paciorek, Perry de Valpine

See Also

nimMatrix, nimArray

Description

NIMBLE wrapper around R’s built-in optim, with flexibility for additional methods.

Usage

nimOptim(par,
    fn,
    gr = "NULL",
    he = "NULL",
    ...,
    method = "Nelder-Mead",
    lower = -Inf,
    upper = Inf,
    control = nimOptimDefaultControl(),
    hessian = FALSE
)
Arguments

par
Initial values for the parameters to be optimized over.

fn
A function to be minimized (or maximized), with first argument the vector of parameters over which minimization is to take place. It should return a scalar result.

gr
A function to return the gradient for the "BFGS", "CG" and "L-BFGS-B" methods. If not provided, a finite-difference approximation to derivatives will be used.

he
A function to return the Hessian matrix of second derivatives. Used (but not required) in "nlminb" or (optionally) user-provided methods.

... 
IGNORED

method
The method to be used. See ‘Details’ section of optim. One of: "Nelder-Mead", "BFGS", "CG", "L-BFGS-B", or "nlminb". Note that the R methods "SANN", "Brent" are not supported. It is also possible to provide a new method; see details.

lower
Vector or scalar of lower bounds for parameters.

upper
Vector or scalar of upper bounds for parameters.

control
A list of control parameters. See Details section of optim. For code in a nimbleFunction to be compiled, this must be an optimControlNimbleList, which has fields for most elements in the control list for R’s optim.

hessian
Logical. Should a Hessian matrix be returned?

Details

This function for use in nimbleFunctions for compilation by compileNimble provides capabilities similar to R’s optim and nlminb. For the supported methods provided by optim, a compiled nimbleFunction will directly call the C code used by R for these methods.

If optim appears in a nimbleFunction, it will be converted to nimOptim.

Note that if a gradient function (gr) is not provided, optim provides a finite difference approximation for use by optimization methods that need gradients. nimble’s compiled version of nimOptim does the same thing, although results might not be completely identical.

For method="nlminb", a compiled nimbleFunction will run R’s nlminb directly in R, with fn, gr (if provided) and he (if provided) that call back into compiled code.

An experimental feature is the capability to provide one’s own optimization method in R and register it for use by nimOptim. One must write a function that takes arguments par, fn, gr, he, lower, upper, control, and hessian. The function must return a list with elements par, value, convergence, counts, evaluations, message, and hessian (which may be NULL). If hessian=TRUE but the function does not return a matrix in the hessian element of its return list, nimOptim will fill in that element using finite differences of the gradient.

The control list passed from a nimbleFunction to the optimization function will include a minimum of options, including abstol, reltol, maxit, and trace. Other options for a specific method may be set within the custom optimization function but cannot be passed from nimOptim.
The elements parscale and fnscale in control are used in a special way. They are implemented by nimOptim such that for *any* the method is expected to do minimization and nimOptim will arrange for it to minimize \( fn(par)/fnscale \) in the parameter space \( par/parscale \).

An optimizer fun may be registered by `nimOptimMethod("method_name", fun)`, and then "method_name" can be used as the method argument to `nimOptim` to use fun. An optimizer may be found by `nimOptimMethod("method_name")` and may be removed by `nimOptimMethod("method_name", NULL)`.

Support for method="nlminb" is provided in this way, and can be studied as an example via `nimOptimMethod("nlminb")`.

The system for providing one’s own optimizer is not considered stable and is subject to change in future versions.

Value

`optimResultNimbleList`

See Also

`optim`

Examples

```r
## Not run:
objectiveFunction <- nimbleFunction(
    run = function(par = double(1)) {
        return(sum(par) * exp(-sum(par ^ 2) / 2))
        returnType(double(0))
    }
)
optimizer <- nimbleFunction(
    run = function(method = character(0), fnscale = double(0)) {
        control <- optimDefaultControl()
        control$fnscale <- fnscale
        par <- c(0.1, -0.1)
        return(optim(par, objectiveFunction, method = method, control = control))
        returnType(optimResultNimbleList())
    }
)
cOptimizer <- compileNimble(optimizer)
cOptimizer(method = 'BFGS', fnscale = -1)
## End(Not run)
```
**nimOptimMethod**

**Description**

Creates a default control argument for **nimOptim**.

**Usage**

`nimOptimDefaultControl()`

**Value**

`optimControlNimbleList`

**See Also**

`nimOptim`, `optim`

---

**nimOptimMethod**  
*Set or get an optimization function to be used by nimOptim*

**Description**

Add, check, or remove an R optimization function to/from NIMBLE’s set of registered optimization functions that can be called from **nimOptim**.

**Usage**

`nimOptimMethod(name, value)`

**Arguments**

- `name`  
  character string, giving the name of optimization method that can be referred to by the method argument of **nimOptim** (aka **optim** in a nimbleFunction).

- `value`  
  An optimization function with specifications described below and in **nimOptim**. If value is NULL, then name will be found in NIMBLE’s set of registered optimizer names. If value is missing, the registered optimizer for name will be returned.

**Details**

When programming in nimbleFunctions, **optim**, which is converted automatically to **nimOptim**, provides a generalization of R’s **optim** methods for optimization. If one of the supported original **optim** methods is not chosen with the method argument to **nimOptim**, an arbitrary method name can be given. If that name has been registered as a name by a call to **nimOptimMethod**, then the corresponding function (value) will be called for optimization.

The function value must perform minimization. If the call to **nimOptim** includes a control list with either `fnscale` (which, if negative, turns the problem into a maximization) or `parscale`, these will be managed by **nimOptim** outside of the optimizer such that the optimization should be minimization.
The function `value` must take named arguments `par` (initial parameter vector), `fn` (objective function), `gr` (optional gradient function), `he` (optional Hessian function), `lower` (vector of lower bounds), `upper` (vector of upper bounds), `control` (arbitrary control list), and `hessian` (logical indicating whether a Hessian at the optimum is requested). It must return a list with elements elements `par` (parameter values of the optimum, i.e., "arg min"), `value` (function value at the minimum), `convergence` (should be 0 if convergence occurred), `counts` (optional vector of counts of calls to `fn`, `gr`, and `he`), `evaluations` (optional total function evaluations), `message` (optional character message), and `hessian` (optional Hessian matrix, which may be NULL).

If the call to `nimOptim` has `hessian=TRUE`, that will be passed as `hessian=TRUE` to the optimizer. However, if the optimizer returns a NULL in the `hessian` element of the return list, then `nimOptim` will calculate the Hessian by finite element differences. Hence, an optimizer need not provide a Hessian capability.

The control list passed from `nimOptim` to the optimizer will have only a limited set of the `optim` control list options. These will include `abstol`, `reltol`, `maxit`, and `trace`. The optimizer may use these as it wishes. Other control options for a particular optimizer must be managed in some other way.

Note that it is possible to use `browser()` inside of `value`, or to set `debug(value)`, to enter a browser when the optimizer (`value`) is called and then inspect its arguments to make sense of the situation.

This whole feature is particularly helpful when the nimbleFunction using `nimOptim` has been compiled by `compileNimble`. Many optimizers are available through R, so `nimOptim` arranges to call a named (registered) optimizer in R, while providing `fn` and optionally `gr` or `he` as functions that will call the compiled (by nimble) versions of the corresponding functions provided in the call to `nimOptim`.

R’s optimizer `nlminb` is automatically registered under the name "nlminb".

---

### Description

`print function for use in nimbleFunctions`

### Usage

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

### Arguments

... an arbitrary 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.

For numeric values, the number of digits printed is controlled by the system option `nimbleOptions('digits')`.  

---

#### nimPrint

`nimPrint` 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.

For numeric values, the number of digits printed is controlled by the system option `nimbleOptions('digits')`.  

---
nimStop

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 function. 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
nimSvd

Singular Value Decomposition of a Matrix

Description
Computes singular values and, optionally, left and right singular vectors of a numeric matrix.

Usage
nimSvd(x, vectors = "full")

Arguments
- **x**: a symmetric numeric matrix (double or integer) whose spectral decomposition is to be computed.
- **vectors**: character that determines whether to calculate left and right singular vectors. Can take values 'none', 'thin' or 'full'. Defaults to 'full'. See ‘Details’.

Details
Computes the singular value decomposition of a numeric matrix using the Eigen C++ template library.

The vectors character argument determines whether to compute no left and right singular vectors ('none'), thinned left and right singular vectors ('thin'), or full left and right singular vectors ('full'). For a matrix x with dimensions n and p, setting vectors = 'thin' will do the following (quoted from eigen website): In case of a rectangular n-by-p matrix, letting m be the smaller value among n and p, there are only m singular vectors; the remaining columns of U and V do not correspond to actual singular vectors. Asking for thin U or V means asking for only their m first columns to be formed. So U is then a n-by-m matrix, and V is then a p-by-m matrix. Notice that thin U and V are all you need for (least squares) solving.

Setting vectors = 'full' will compute full matrices for U and V, so that U will be of size n-by-n, and V will be of size p-by-p.

In a nimbleFunction, svd is identical to nimSvd.

**returnType(svdNimbleList())** can be used within a link(nimbleFunction) to specify that the function will return a nimbleList generated by the nimSvd function. svdNimbleList() can also be used to define a nested nimbleList element. See the User Manual for usage examples.

Value
The singular value decomposition of x is returned as a nimbleList with elements:
- **d**: length m vector containing the singular values of x, sorted in decreasing order.
- **v**: matrix with columns containing the left singular vectors of x, or an empty matrix if vectors = 'none'.
- **u**: matrix with columns containing the right singular vectors of x, or an empty matrix if vectors = 'none'.


Author(s)

NIMBLE development team

See Also

nimEigen for spectral decompositions.

Examples

```r
singularValuesDemoFunction <- nimbleFunction(
  setup = function(){
    demoMatrix <- diag(4) + 2
  },
  run = function(){
    singularValues <- svd(demoMatrix)$d
    returnType(double(1))
    return(singularValues)
  })
```

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

```r
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

Standard usage is as a method of a model, in the form `model$calculate(nodes)`, but the usage as a simple function with the model as the first argument as above is also allowed.

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 collects and returns the sum of the log probabilities of nodes, using the log probability values currently stored in the model (as generated from the most recent call to calculate on each node)

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
optimControlNimbleList

*Data type for the control parameter of* nimOptim

**Description**

nimbleList definition for the type of nimbleList input as the control parameter to nimOptim. See optim for details.

**Usage**

optimControlNimbleList

**Format**

An object of class list of length 1.

**See Also**

optim, nimOptim

optimDefaultControl

*Creates a deafult control argument for optim (just an empty list).*

**Description**

Creates a deafult control argument for optim (just an empty list).

**Usage**

optimDefaultControl()

**Value**

an empty list.

**See Also**

nimOptim, optim
optimResultNimbleList  *Data type for the return value of* *nimOptim*

**Description**

*nimbleList* definition for the type of *nimbleList* returned by *nimOptim*.

**Usage**

optimResultNimbleList

**Format**

An object of class *list* of length 1.

**Fields**

- **par**  The best set of parameters found.
- **value**  The value of fn corresponding to par.
- **counts**  A two-element integer vector giving the number of calls to fn and gr respectively.
- **convergence**  An integer code. 0 indicates successful completion. Possible error codes are 1 indicates that the iteration limit maxit had been reached. 10 indicates degeneracy of the Nelder-Mead simplex. 51 indicates a warning from the "L-BFGS-B" method; see component message for further details. 52 indicates an error from the "L-BFGS-B" method; see component message for further details.
- **message**  A character string giving any additional information returned by the optimizer, or NULL.
- **hessian**  Only if argument hessian is true. A symmetric matrix giving an estimate of the Hessian at the solution found.

**See Also**

optim, nimOptim

---

**parameterTransform**  *Automated transformations of model nodes to unconstrained scales*

**Description**

Provide general transformations of constrained continuous-valued model nodes (parameters) to unconstrained scales. It handles the cases of interval-bounded parameters (e.g. uniform or beta distributions), semi-interval-bounded parameters (e.g. exponential or gamma distributions), and the multivariate Wishart, inverse Wishart, Dirichlet, and LKJ distributions. Utilities are provided to transform parameters to an unconstrained scale, back-transform from the unconstrained scale to the original scale of the constrained parameterization, and to calculate the natural logarithm of the determinant of the Jacobian matrix of the inverse transformation, calculated at any location in the transformed (unconstrained) space.
**Usage**

```r
callParameterTransform(model, nodes = character(0), control = list())
```

**Arguments**

- `model` A nimble model object. See details.
- `nodes` A character vector specifying model node names to undergo transformation. See details.
- `control` An optional list allowing for additional control of the transformation. This currently supports a single element `allowDeterm`.

**Details**

The `parameterTransform` nimbleFunction is an unspecialized function. Calling `parameterTransform(model, nodes)` will generate and return a specialized nimbleFunction, which provides transformation functionality for the specified hierarchical model and set of model nodes. The nodes argument can represent multiple model nodes arising from distinct prior distributions, which will be simultaneously transformed according to their respective distributions and constraints.

If the `nodes` argument is missing or has length zero, then no nodes will be transformed. A specialized nimbleFunction is created, but will not transform or operate on any model nodes.

The `control` argument is a list that supports one additional setting. If `control$allowDeterm=FALSE` (the default), deterministic nodes are not allowed in the `nodes` argument. If `control$allowDeterm=TRUE`, deterministic nodes are allowed and assumed to have no constraints on valid values.

This specialized nimbleFunction has the following methods:

- `transform`: Transforms a numeric vector of values from the original constrained model scale to a vector of values on the unconstrained scale.
- `inverseTransform`: Transforms a numeric vector of values from the unconstrained scale to the original constrained parameterization scale.

The unconstrained scale may have different dimensionality from the original constrained scale of the model parameters. For example, a d-dimensional dirichlet distribution is constrained to reside on a simplex in d-dimensional space. In contrast, the corresponding unconstrained parameterization is unrestrained in (d-1) dimensional space. The specialized `parameterTransform` nimbleFunction also provides utilities to return the dimensionality of the original (constrained) parameterization, and the transformed (unconstrained) parameterization:

- `getOriginalLength`: Returns the dimensionality (number of scalar elements) of the original constrained parameterization.
- `getTransformedLength`: Returns the dimensionality (number of scalar elements) comprising the transformed unconstrained parameterization.

The specialized `parameterTransform` nimbleFunction also provides a method for calculating the natural logarithm of the jacobian of the inverse transformation, calculated at any point in the transformed (unconstrained) space:

- `logDetJacobian`

The `parameterTransform` function has no facility for handling discrete-valued parameters.
Author(s)
Daniel Turek

See Also
buildLaplace

Examples
```r
## Not run:
code <- nimbleCode({
a ~ dnorm(0, 1)
b ~ dgamma(1, 1)
c ~ dunif(2, 10)
d[1:3] ~ dmnorm(mu[1:3], cov = C[1:3,1:3])
e[1:3,1:3] ~ dwish(R = C[1:3,1:3], df = 5)
})

constants <- list(mu=rep(0,3), C=diag(3))

Rmodel <- nimbleModel(code, constants)

## create a specialized parameterTransform function:

nodes <- c('a', 'b', 'c', 'd', 'e')

pt <- parameterTransform(Rmodel, nodes)

vals <- c(1, 10, 5, 1, 2, 3, as.numeric(diag(3)))

## transform values to unconstrained scale:
transformedVals <- pt$transform(vals)

## back-transform to original constrained scale of parameterization
pt$inverseTransform(transformedVals) ## return is same as original vals

## dimensionality of original constrained scale = 1 + 1 + 1 + 3 + 9
pt$getOriginalLength()  ## 15

## dimensionality of transformed (unconstrained) scale = 1 + 1 + 1 + 3 + 6
pt$getTransformedLength() ## 12

## log of the jacobian of the inverse transformation matrix:
pt$logDetJacobian(transformedVals)

## End(Not run)
```

pow_int | Power function for integer-valued exponent
printErrors

Description

pow function with exponent required to be integer

Usage

pow_int(a, b)

Arguments

<table>
<thead>
<tr>
<th>a</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Exponent</td>
</tr>
</tbody>
</table>

Details

This is required in nimble models and nimbleFunctions if derivatives will be tracked but tracked only with respect to a, such that b might be any (positive, 0, or negative) integer. This contrasts with pow(a, b) (equivalent to \(a^b\)), which requires \(b > 0\) if derivatives will be tracked, even if they will only be requested with respect to a.

Value

\(a^b\)

printErrors

Print error messages after failed compilation

Description

Retrieves the error file from R’s tempdir and prints to the screen.

Usage

printErrors(excludeWarnings = TRUE)

Arguments

<table>
<thead>
<tr>
<th>excludeWarnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical indicating whether compiler warnings should be printed: generally such warnings can be ignored.</td>
</tr>
</tbody>
</table>

Author(s)

Christopher Paciorek
Generates a weighted sample (with replacement) of ranks

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 2 3 3
rankSample(weights = c(1, 1, 2), size = 10000, sampInts)
table(sampInts)
sampInts
# 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()
rSamp$run(1:4, 5)
# [1] 3 3 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**

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

**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 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.

buildDerivs logical indicating whether to build derivative capabilities for the model.

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 NIMBLE in the classic-bugs directory of the installed NIMBLE package or JAGS (https://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
See Also
nimbleModel

Examples

```r
## Reading a model defined in the R session

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')

## Reading a classic BUGS model

pumpModel <- readBUGSmodel('pump.bug', dir = getBUGSexampleDir('pump'))
pumpModel$getVarNames()
pumpModel$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

```r
registerDistributions(
  distributionsInput,
  userEnv = parent.frame(),
  verbose = nimbleOptions("verbose")
)
```

Arguments

distributionsInput
  either a list or character vector specifying the user-supplied distributions. If a list, it should be a named list of lists in the form of that shown in nimble::distributionsInputList with each list having required field BUGSdist and optional fields Rdist, altParams, discrete, pqAvail, types, and with the name of the list the same as that of the density function. Alternatively, simply a character vector providing the names of the density functions for the user-supplied distributions.
userEnv: environment in which to look for the nimbleFunctions that provide the distribution; this will generally not need to be set by the user as it will default to the environment from which this function was called.

verbose: logical indicating whether to print additional logging information

Details

When distributionsInput is a list of lists, see below for more information on the structure of the list. When distributionsInput is a character vector, the distribution is assumed to be of standard form, with parameters assumed to be the arguments provided in the density nimbleFunction, no alternative parameterizations, and the distribution assumed to be continuous with range from minus infinity to infinity. The availability of distribution and quantile functions is inferred from whether appropriately-named functions exist in the global environment.

One usually does not need to explicitly call registerDistributions as it will be called automatically when the user-supplied distribution is used for the first time in BUGS code. However, if one wishes to provide alternative parameterizations, to provide a range, or to indicate a distribution is discrete, then one still must explicitly register the distribution using registerDistributions with the argument in the list format.

Format of the component lists when distributionsInput is a list of lists:

- 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 (which must have no 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: An 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 = integer(0)) {
    returnType(double(0))
    logProb <- log(rate) - x*rate
    if(log) {
      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))
m$calculate('y')
m$r <- 2
m$calculate('y')
m$resetData()
m$simulate('y')
m$y

# alternatively, simply specify a character vector with the
# name of one or more 'd' functions
deregisterDistributions('dmyexp')
registerDistributions('dmyexp')

# or simply use in BUGS code without registration
deregisterDistributions('dmyexp')
m <- nimbleModel(code, inits = list(r = 1), data = list(y = 2))

# example of Dirichlet-multinomial registration to illustrate
# use of 'types' (note that registration is not actually needed
# in this case)
ddirchmulti <- nimbleFunction(
  run = function(x = double(1), alpha = double(1), size = double(0),
            log = integer(0, default = 0)) {
    returnType(double(0))
    logProb <- lgamma(size) - sum(lgamma(x)) + lgamma(sum(alpha)) -
                sum(lgamma(alpha)) + sum(lgamma(alpha + x)) - lgamma(sum(alpha) +
                size)
    if(log) return(logProb)
    else return(exp(logProb))
  })

rdirchmulti <- nimbleFunction(
  run = function(n = integer(0), alpha = double(1), size = double(0)) {
    returnType(double(0))
    if(n != 1) print("rdirchmulti only allows n = 1; using n = 1.")
    p <- rdirch(1, alpha)
    return(rmulti(1, size = size, prob = p))
  })

registerDistributions(list(
  ddirchmulti = list(BUGSdist = "ddirchmulti(alpha, size)",
    types = c('value = double(1)', 'alpha = double(1)'))
))
**Rmatrix2mvOneVar**

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

```r
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 information about modelValues objects

Author(s)

Clifford Anderson-Bergman

Examples

```r
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

```r
Rmatrix2mvOneVar(mat, mv, varName, k)
```
run.time

Arguments

<table>
<thead>
<tr>
<th>mat</th>
<th>Input matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>mv</td>
<td>modelValues object to be modified.</td>
</tr>
<tr>
<td>varName</td>
<td>Character string giving the name of the variable on mv to be set</td>
</tr>
<tr>
<td>k</td>
<td>Number of rows to use</td>
</tr>
</tbody>
</table>

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
runCrossValidate

Perform k-fold cross-validation on a NIMBLE model fit by MCMC

Description

Takes a NIMBLE model MCMC configuration and conducts k-fold cross-validation of the MCMC fit, returning a measure of the model’s predictive performance.

Usage

runCrossValidate(
  MCMCconfiguration,
  k,
  foldFunction = "random",
  lossFunction = "MSE",
  MCMCcontrol = list(),
  returnSamples = FALSE,
  nCores = 1,
  nBootReps = 200,
  silent = FALSE
)

Arguments

MCMCconfiguration

a NIMBLE MCMC configuration object, returned by a call to configureMCMC.

k

number of folds that should be used for cross-validation.

foldFunction

one of (1) an R function taking a single integer argument i, and returning a character vector with the names of the data nodes to leave out of the model for fold i, or (2) the character string "random", indicating that data nodes will be randomly partitioned into k folds. Note that choosing "random" and setting k equal to the total number of data nodes in the model will perform leave-one-out cross-validation. Defaults to "random". See ‘Details’.

lossFunction

one of (1) an R function taking a set of simulated data and a set of observed data, and calculating the loss from those, or (2) a character string naming one of NIMBLE’s built-in loss functions. If a character string, must be one of "predictive" to use the log predictive density as a loss function or "MSE" to use the mean squared error as a loss function. Defaults to "MSE". See ‘Details’ for information on creating a user-defined loss function.

MCMCcontrol

(optional) an R list with parameters governing the MCMC algorithm, See ‘Details’ for specific parameters.

returnSamples

logical indicating whether to return all posterior samples from all MCMC runs. This can result in a very large returned object (there will be k sets of posterior samples returned). Defaults to FALSE.

nCores

number of cpu cores to use in parallelizing the CV calculation. Only MacOS and Linux operating systems support multiple cores at this time. Defaults to 1.
nBootReps number of bootstrap samples to use when estimating the Monte Carlo error of the cross-validation metric. Defaults to 200. If no Monte Carlo error estimate is desired, nBootReps can be set to NA, which can potentially save significant computation time.
silent Boolean specifying whether to show output from the algorithm as it’s running (default = FALSE).

Details
k-fold CV in NIMBLE proceeds by separating the data in a nimbleModel into k folds, as determined by the foldFunction argument. For each fold, the corresponding data are held out of the model, and MCMC is run to estimate the posterior distribution and simultaneously impute posterior predictive values for the held-out data. Then, the posterior predictive values are compared to the known, held-out data values via the specified lossFunction. The loss values are averaged over the posterior samples for each fold, and these averaged values for each fold are then averaged over all folds to produce a single out-of-sample loss estimate. Additionally, estimates of the Monte Carlo error for each fold are returned.

Value
an R list with four elements:
  • CVvalue. The CV value, measuring the model’s ability to predict new data. Smaller relative values indicate better model performance.
  • CVstandardError. The standard error of the CV value, giving an indication of the total Monte Carlo error in the CV estimate.
  • foldCVInfo. An list of fold CV values and standard errors for each fold.
  • samples. An R list, only returned when returnSamples = TRUE. The i'th element of this list will be a matrix of posterior samples from the model with the i'th fold of data left out. There will be k sets of samples.

The foldFunction Argument
If the default 'random' method is not used, the foldFunction argument must be an R function that takes a single integer-valued argument i. i is guaranteed to be within the range [1, k]. For each integer value i, the function should return a character vector of node names corresponding to the data nodes that will be left out of the model for that fold. The returned node names can be expanded, but don’t need to be. For example, if fold i is intended to leave out the model nodes x[1], x[2] and x[3] then the function could return either c('x[1]', 'x[2]', 'x[3]') or 'x[1:3]'.

The lossFunction Argument
If you don’t wish to use NIMBLE’s built-in "MSE" or "predictive" loss functions, you may provide your own R function as the lossFunction argument to runCrossValidate. A user-supplied lossFunction must be an R function that takes two arguments: the first, named simulatedDataValues, will be a vector of simulated data values. The second, named actualDataValues, will be a vector of observed data values corresponding to the simulated data values in simulatedDataValues. The loss function should return a single scalar number. See "Examples" for an example of a user-defined loss function.
The MCMCcontrol Argument

The MCMCcontrol argument is a list with the following elements:

- `niter`. An integer argument determining how many MCMC iterations should be run for each loss value calculation. Defaults to 10000, but should probably be manually set.
- `nburnin`. The number of samples from the start of the MCMC chain to discard as burn-in for each loss value calculation. Must be between 0 and `niter`. Defaults to 10

Author(s)

Nicholas Michaud and Lauren Ponisio

Examples

```r
## Not run:
## We conduct CV on the classic "dyes" BUGS model.

dyesModel <- nimbleModel('
  # Define the fold function.
  # This function defines the data to leave out for the i'th fold
  # as the i'th row of the data matrix y. This implies we will have

  # niter = 10000
  # nburnin = 10
  # MCMCcontrol = list(niter = 10000, nburnin = 10)

dyesModel <- nimbleModel(code = dyesCode,
                          constants = dyesConsts,
                          data = dyesData,
                          inits = dyesInits)

# Define the fold function.
# This function defines the data to leave out for the i'th fold
# as the i'th row of the data matrix y. This implies we will have
```
# 6 folds.

dyesFoldFunction <- function(i){
  foldNodes_i <- paste0('y[', i, ', ]') # will return 'y[1,]' for i = 1 e.g.
  return(foldNodes_i)
}

# We define our own loss function as well.
# The function below will compute the root mean squared error.

RMSElossFunction <- function(simulatedDataValues, actualDataValues){
  dataLength <- length(simulatedDataValues) # simulatedDataValues is a vector
  SSE <- 0
  for(i in 1:dataLength){
    SSE <- SSE + (simulatedDataValues[i] - actualDataValues[i])^2
  }
  MSE <- SSE / dataLength
  RMSE <- sqrt(MSE)
  return(RMSE)
}

dyesMCMCconfiguration <- configureMCMC(dyesModel)

crossValOutput <- runCrossValidate(MCMCconfiguration = dyesMCMCconfiguration,
  k = 6,
  foldFunction = dyesFoldFunction,
  lossFunction = RMSElossFunction,
  MCMCcontrol = list(niter = 5000,
    nburnin = 500))

## End(Not run)

---

**runLaplace**

Combine steps of running Laplace or adaptive Gauss-Hermite quadrature approximation

---

**Description**

Use an approximation (compiled or uncompiled) returned from ‘buildLaplace’ or ‘buildAGHQ’ to find the maximum likelihood estimate and return it with random effects estimates and/or standard errors.

**Usage**

```r
runLaplace(
  laplace,  
pStart,    
  method = "BFGS",  
  originalScale = TRUE,
)```

```
runLaplace

```
randomEffectsStdError = TRUE,
jointCovariance = FALSE
```

runAGHQ(
    AGHQ,
    pStart,
    method = "BFGS",
    originalScale = TRUE,
    randomEffectsStdError = TRUE,
    jointCovariance = FALSE
)

Arguments

**laplace**  
A (compiled or uncompiled) nimble laplace approximation object returned from ‘buildLaplace’ or ‘buildAGHQ’. These return the same type of approximation algorithm object. ‘buildLaplace’ is simply ‘buildAGHQ’ with ‘nQuad=1’.

**pStart**  
Initial values for parameters to begin optimization search for the maximum likelihood estimates. If omitted, the values currently in the (compiled or uncompiled) model object will be used.

**method**  
Optimization method for outer optimization. See method argument to findMLE method in buildLaplace.

**originalScale**  
If TRUE, return all results on the original scale of the parameters and/or random effects as written in the model. Otherwise, return all results on potentially unconstrained transformed scales that are used in the actual computations. Transformed scales (parameterizations) are used if any parameter or random effect has contraint(s) on its support (range of allowed values). Default = TRUE.

**randomEffectsStdError**  
If TRUE, include standard errors for the random effects estimates. Default = TRUE.

**jointCovariance**  
If TRUE, return the full joint covariance matrix (inverse of the Hessian) of parameters and random effects. Default = FALSE.

**AGHQ**  
Same as laplace.

Details

Adaptive Gauss-Hermite quadrature is a generalization of Laplace approximation. runLaplace simply calles runAGHQ and provides a convenient name.

These functions manage the steps of calling the ‘findMLE‘ method to obtain the maximum likelihood estimate of the parameters and then the ‘summaryLaplace‘ function to obtain standard errors, (optionally) random effects estimates (conditional modes), their standard errors, and the full parameter-random effects covariance matrix.

Note that for ‘nQuad > 1‘ (see buildAGHQ), i.e., AGHQ with higher order than Laplace approximation, maximum likelihood estimation is available only if all random effects integrations are univariate. With multivariate random effects integrations, one can use ‘nQuad > 1‘ only to calculate
marginal log likelihoods at given parameter values. This is useful for checking the accuracy of the log likelihood at the MLE obtained for Laplace approximation (`nQuad == 1`). `nQuad` can be changed using the `updateSettings` method of the approximation object.

See `summaryLaplace`, which is called for the summary components.

**Value**

A list with elements `MLE` and `summary`.

`MLE` is the result of the `findMLE` method, which contains the parameter estimates and Hessian matrix. This is considered raw output, and one should normally use instead the contents of `summary`. (For example not that the Hessian matrix in `MLE` may not correspond to the same scale as the parameter estimates if a transformation was used to operate in an unconstrained parameter space.)

`summary` is the result of `summaryLaplace` (or equivalently `summaryAGHQ`), which contains parameter estimates and standard errors, and optionally other requested components. All results in this object will be on the same scale (parameterization), either original or transformed, as requested.

---

**runMCMC**

*Run one or more chains of an MCMC algorithm and return samples, summary and/or WAIC*

**Description**

Takes as input an MCMC algorithm (ideally a compiled one for speed) and runs the MCMC with one or more chains, any returns any combination of posterior samples, posterior summary statistics, and a WAIC value.

**Usage**

```r
runMCMC(
mcmc,
niter = 10000,
nburnin = 0,
thin,
thin2,
nchains = 1,
inits,
setSeed = FALSE,
progressBar = getNimbleOption("MCMCprogressBar"),
samples = TRUE,
samplesAsCodaMCMC = FALSE,
summary = FALSE,
WAIC = FALSE,
perChainWAIC = FALSE
)
```
Arguments

mcmc
A NIMBLE MCMC algorithm. See details.
niter
Number of iterations to run each MCMC chain. Default value is 10000.
nburnin
Number of initial, pre-thinning, MCMC iterations to discard. Default value is 0.
thin
Thinning interval for collecting MCMC samples, corresponding to monitors. Thinning occurs after the initial nburnin samples are discarded. Default value is 1.
thin2
Thinning interval for collecting MCMC samples, corresponding to the second, optional set of monitors2. Thinning occurs after the initial nburnin samples are discarded. Default value is 1.
nchains
Number of MCMC chains to run. Default value is 1.
init
Optional argument to specify initial values for each chain. See details.
setSeed
Logical or numeric argument. If a single numeric value is provided, R’s random number seed will be set to this value at the onset of each MCMC chain. If a numeric vector of length nchains is provided, then each element of this vector is provided as R’s random number seed at the onset of the corresponding MCMC chain. Otherwise, in the case of a logical value, if TRUE, then R’s random number seed for the ith chain is set to be i, at the onset of each MCMC chain. Note that specifying the argument setSeed = 0 does not prevent setting the RNG seed, but rather sets the random number generation seed to 0 at the beginning of each MCMC chain. Default value is FALSE.
progressBar
Logical argument. If TRUE, an MCMC progress bar is displayed during execution of each MCMC chain. Default value is defined by the nimble package option MCMCprogressBar.
samples
Logical argument. If TRUE, then posterior samples are returned from each MCMC chain. These samples are optionally returned as coda mcmc objects, depending on the samplesAsCodaMCMC argument. Default value is TRUE. See details.
samplesAsCodaMCMC
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. This argument is only used when samples is TRUE. Default value is FALSE. See details.
summary
Logical argument. When TRUE, summary statistics for the posterior samples of each parameter are also returned, for each MCMC chain. This may be returned in addition to the posterior samples themselves. Default value is FALSE. See details.
WAIC
Logical argument. When TRUE, the WAIC (Watanabe, 2010) of the model is calculated and returned. Note that in order for the WAIC to be calculated, the mcmc object must have also been created with the argument ‘enableWAIC = TRUE’. If multiple chains are run, then a single WAIC value is calculated using the posterior samples from all chains. Default value is FALSE. See help(waic).
perChainWAIC
Logical argument. When TRUE and multiple chains are run, the WAIC for each chain is returned as a means of helping assess the stability of the WAIC estimate. Default value is FALSE.
Details

At least one of samples, summary or WAIC must be TRUE, since otherwise, nothing will be returned. Any combination of these may be TRUE, including possibly all three, in which case posterior samples and summary statistics are returned for each MCMC chain, and an overall WAIC value is calculated and returned.

When samples = TRUE, the form of the posterior samples is determined by the samplesAsCodaMCMC argument, as either matrices of posterior samples, or coda mcmc and mcmc.list objects.

Posterior summary statistics are returned individually for each chain, and also as calculated from all chains combined (when nchains > 1).

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, or (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 the specific sampler assignments, must be specified in advance using the MCMC configuration object (created using configureMCMC), which is then used to build an MCMC algorithm (using buildMCMC) argument.

The niter argument specifies the number of pre-thinning MCMC iterations, and the nburnin argument specifies the number of pre-thinning MCMC samples to discard. After discarding these burn-in samples, thinning of the remaining samples will take place. The total number of posterior samples returned will be floor((niter-nburnin)/thin).

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

Value

A list is returned with named elements depending on the arguments passed to nimbleMCMC, unless this list contains only a single element, in which case only that element is returned. These elements may include samples, summary, and WAIC, and when the MCMC is monitoring a second set of nodes using monitors2, also samples2. When nchains = 1, posterior samples are returned as a single matrix, and summary statistics as a single matrix. When nchains > 1, posterior samples are returned as a list of matrices, one matrix for each chain, and summary statistics are returned as a list containing nchains+1 matrices: one matrix corresponding to each chain, and the final element providing a summary of all chains, combined. If samplesAsCodaMCMC is TRUE, then posterior samples are provided as coda mcmc and mcmc.list objects. When WAIC is TRUE, a WAIC summary object is returned.

Author(s)

Daniel Turek

See Also

configureMCMC buildMCMC nimbleMCMC
Examples

```r
## 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; HMC samplers are in the nimbleHMC package and particle filter samplers are in the nimbleSMC package.

**Usage**

- `sampler_BASE()`
- `sampler_prior_samples(model, mvSaved, target, control)`
- `sampler_posterior_predictive(model, mvSaved, target, control)`
- `sampler_binary(model, mvSaved, target, control)`
- `sampler_categorical(model, mvSaved, target, control)`
- `sampler_noncentered(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_AF_slice(model, mvSaved, target, control)
sampler_crossLevel(model, mvSaved, target, control)
sampler_RW_1lFunction_block(model, mvSaved, target, control)
sampler_RW_dirichlet(model, mvSaved, target, control)
sampler_RW_wishart(model, mvSaved, target, control)
sampler_RW_lkj_corr_cholesky(model, mvSaved, target, control)
sampler_RW_block_lkj_corr_cholesky(model, mvSaved, target, control)
sampler_CAR_normal(model, mvSaved, target, control)
sampler_CAR_proper(model, mvSaved, target, control)
sampler_polyagamma(model, mvSaved, target, control)
sampler_RJ_fixed_prior(model, mvSaved, target, control)
sampler_RJ_indicator(model, mvSaved, target, control)
sampler_RJ_toggled(model, mvSaved, target, control)
sampler_CRP_concentration(model, mvSaved, target, control)
sampler_CRP(model, mvSaved, target, control)
sampler_slice_CRP_base_param(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 sampler type. The default values for control list are specified in the setup code of each sampling algorithm. Descriptions of each sampling algorithm, and the possible customizations for each sampler (using the control argument) appear below.
Hamiltonian Monte Carlo samplers

Hamiltonian Monte Carlo (HMC) samplers are provided separately in the `nimbleHMC` R package. After loading `nimbleHMC`, see `help(HMC)` for details.

Particle filter samplers

As of Version 0.10.0 of NIMBLE, the `RW_PF` and `RW_PF_block` samplers are provided separately in the `nimbleSMC` package. After loading `nimbleSMC`, see `help(samplers)` for details.

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.

categorical sampler

The categorical sampler performs Gibbs sampling for a single node, which generally would follow a categorical (`dcat`) distribution. The categorical sampler can be assigned to other distributions as well, in which case the number of possible outcomes (1, 2, 3, ..., k) of the distribution must be specified using the 'length' control argument.

The categorical sampler accepts the following control list elements:

- length. A character string or a numeric argument. When a character string, this should be the name of a parameter of the distribution of the target node being sampled. The length of this distribution parameter (considered as a 1-dimensional vector) will be used to determine the number of possible outcomes of the target node’s distribution. When a numeric value, this value will be used as the number of possible outcomes of the target node’s distribution. (default = "prob")
- check. A logical argument. When FALSE, no check for a 'dcat' prior distribution for the target node takes place. (default = TRUE)

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)

• **adaptFactorExponent.** Exponent controlling the rate of decay of the scale adaptation factor. See Shaby and Wells, 2011, for details. (default = 0.8)

• **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.

After an MCMC algorithm has been configuration and built, the value of the proposal standard deviation of a RW sampler can be modified using the setScale method of the sampler object. This uses the scalar argument to modify the current value of the proposal standard deviation, as well as modifying the initial (pre-adaptation) value to which the proposal standard deviation is reset, at the onset of a new MCMC chain.

**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 adaptation should be done only for scale (TRUE) or also for propCov (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 = TRUE, 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)

• **adaptFactorExponent.** Exponent controlling the rate of decay of the scale adaptation factor. See Shaby and Wells, 2011, for details. (default = 0.8)
• 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')

• tries. The number of times this sampler will repeatedly operate on each MCMC iteration. Each try consists of a new proposed transition and an accept/reject decision of this proposal. Specifying tries > 1 can help increase the overall sampler acceptance rate and therefore chain mixing. (default = 1)

After an MCMC algorithm has been configuration and built, the value of the proposal standard deviation of a RW_block sampler can be modified using the setScale method of the sampler object. This use the scalar argument to will modify the current value of the proposal standard deviation, as well as modifying the initial (pre-adaptation) value which the proposal standard deviation is reset to, at the onset of a new MCMC chain.

Operating analogous to the setScale method, the RW_block sampler also has a setPropCov method. This method accepts a single matrix-valued argument, which will modify both the current and initial (used at the onset of a new MCMC chain) values of the multivariate normal proposal covariance.

Note that modifying elements of the control list may greatly affect the performance of this sampler. In particular, the sampler can take a long time to find a good proposal covariance when the elements being sampled are not on the same scale. We recommend providing an informed value for propCov in this case (possibly simply a diagonal matrix that approximates the relative scales), as well as possibly providing a value of scale that errs on the side of being too small. You may also consider decreasing adaptFactorExponent and/or adaptInterval, as doing so has greatly improved performance in some cases.

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 = 1)
- maxContractions. The maximum number of contractions of the interval that may occur during sampling (this prevents infinite looping in unusual situations). (default = 100)
- maxContractionsWarning. A logical argument specifying whether to warn when the maximum number of contractions is reached. (default = TRUE)

ess sampler

The ess sampler performs elliptical slice sampling of a single node, which must follow either a univariate or multivariate normal distribution (Murray, 2010). The algorithm is an extension of slice sampling (Neal, 2003), generalized to context of the Gaussian distribution. An auxiliary 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 Gaussian distributions.

The ess sampler accepts the following control list arguments.

- maxContractions. The maximum number of contractions of the interval that may occur during sampling (this prevents infinite looping in unusual situations). (default = 100)
- maxContractionsWarning. A logical argument specifying whether to warn when the maximum number of contractions is reached. (default = TRUE)
AF_slice sampler

The automated factor slice sampler conducts a slice sampling algorithm on one or more model nodes. The sampler uses the eigenvectors of the posterior covariance between these nodes as an orthogonal basis on which to perform its 'stepping Out' procedure. The sampler is adaptive in updating both the width of the slices and the values of the eigenvectors. The sampler can be applied to any set of continuous or discrete-valued model nodes, to any single continuous or discrete-valued multivariate model node, or to any combination thereof. The automated factor slice sampler accepts the following control list elements:

- sliceWidths. A numeric vector of initial slice widths. The length of the vector must be equal to the sum of the lengths of all nodes being used by the automated factor slice sampler. Defaults to a vector of 1’s.
- sliceAdaptFactorMaxIter. The number of iterations for which the factors (eigenvectors) will continue to adapt to the posterior correlation. (default = 15000)
- sliceAdaptFactorInterval. The interval on which to perform factor adaptation. (default = 200)
- sliceAdaptWidthMaxIter. The maximum number of iterations for which to adapt the widths for a given set of factors. (default = 512)
- sliceAdaptWidthTolerance. The tolerance for when widths no longer need to adapt, between 0 and 0.5. (default = 0.1)
- sliceMaxSteps. The maximum number of expansions which may occur during the 'stepping out' procedure. (default = 100)
- maxContractions. The maximum number of contractions of the interval that may occur during sampling (this prevents infinite looping in unusual situations). (default = 100)
- maxContractionsWarning. A logical argument specifying whether to warn when the maximum number of contractions is reached. (default = TRUE)

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. This sampler is a conjugate version of Scheme 3 in Knorr-Held and Rue (2002). It can also be seen as a Metropolis-based version of collapsed Gibbs sampling (in particular Sampler 3 of van Dyk and Park (2008)).

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_llFunction_block** 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_llFunction_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 is TRUE)

• adaptScaleOnly. A logical argument, specifying whether adaption should be done only for scale (TRUE) or also for propCov (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 = TRUE, only the proposal scale is adapted. (default = FALSE)

• adaptInterval. The interval on which to perform adaptation. (default = 200)

• adaptFactorExponent. Exponent controlling the rate of decay of the scale adaptation factor. See Shaby and Wells, 2011, for details. (default = 0.8)

• 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_dirichlet sampler**

This sampler is designed for sampling non-conjugate Dirichlet distributions. The sampler performs a series of Metropolis-Hastings updates (on the log scale) to each component of a gamma-reparameterization of the target Dirichlet distribution. The acceptance or rejection of these proposals follows a standard Metropolis-Hastings procedure.

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

- **adaptive**: A logical argument, specifying whether the sampler should independently adapt the scale (proposal standard deviation, on the log scale) for each componentwise Metropolis-Hastings update, to achieve a theoretically desirable acceptance rate for each. (default = TRUE)
- **adaptInterval**: The interval on which to perform adaptation. Every adaptInterval MCMC iterations (prior to thinning), the sampler will perform its adaptation procedure. (default = 200)
- **adaptFactorExponent**: Exponent controlling the rate of decay of the scale adaptation factor. See Shaby and Wells, 2011, for details. (default = 0.8)
- **scale**: The initial value of the proposal standard deviation (on the log scale) for each component of the reparameterized Dirichlet distribution. If adaptive = FALSE, the proposal standard deviations will never change. (default = 1)

**RW_wishart sampler**

This sampler is designed for sampling non-conjugate Wishart and inverse-Wishart distributions. More generally, it can update any symmetric positive-definite matrix (for example, scaled covariance or precision matrices). The sampler performs block Metropolis-Hastings updates following a transformation to an unconstrained scale (Cholesky factorization of the original matrix, then taking the log of the main diagonal elements).

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

- **adaptive**: A logical argument, specifying whether the sampler should adapt the scale and proposal covariance for the multivariate normal Metropolis-Hastings proposals, to achieve a theoretically desirable acceptance rate for each. (default = TRUE)
- **adaptInterval**: The interval on which to perform adaptation. Every adaptInterval MCMC iterations (prior to thinning), the sampler will perform its adaptation procedure. (default = 200)
- **adaptFactorExponent**: Exponent controlling the rate of decay of the scale adaptation factor. See Shaby and Wells, 2011, for details. (default = 0.8)
- **scale**: The initial value of the scalar multiplier for the multivariate normal Metropolis-Hastings proposal covariance. If adaptive = FALSE, scale will never change. (default = 1)

**RW_block_lkj_corr_cholesky sampler**

This sampler is designed for sampling non-conjugate LKJ correlation Cholesky factor distributions. The sampler performs a blocked Metropolis-Hastings update following a transformation to an unconstrained scale (using the signed stickbreaking approach documented in Section 10.12 of the Stan Language Reference Manual, version 2.27).

The `RW_block_lkj_corr_cholesky` 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 adaptation should be done only for scale (TRUE) or also for propCov (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 = TRUE, 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)
• adaptFactorExponent. Exponent controlling the rate of decay of the scale adaptation factor. See Shaby and Wells, 2011, for details. (default = 0.8)
• 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')

This is the default sampler for the LKJ distribution. However, blocked samplers may perform poorly if the adaptation configuration is poorly chosen. See the comments in the RW_block section of this documentation.

**RW_lkj_corr_cholesky sampler**

This sampler is designed for sampling non-conjugate LKJ correlation Cholesky factor distributions. The sampler performs individual Metropolis-Hastings updates following a transformation to an unconstrained scale (using the signed stickbreaking approach documented in Section 10.12 of the Stan Language Reference Manual, version 2.27).

The RW_lkj_corr_cholesky sampler accepts the following control list elements:

• adaptive. A logical argument, specifying whether the sampler should adapt the scales of the univariate normal Metropolis-Hasting proposals, to achieve a theoretically desirable acceptance rate for each. (default = TRUE)
• adaptInterval. The interval on which to perform adaptation. Every adaptInterval MCMC iterations (prior to thinning), the sampler will perform its adaptation procedure. (default = 200)
• adaptFactorExponent. Exponent controlling the rate of decay of the scale adaptation factor. See Shaby and Wells, 2011, for details. (default = 0.8)
• scale. The initial value of the scalar multiplier for the multivariate normal Metropolis-Hastings proposal covariance. If adaptive = FALSE, scale will never change. (default = 1)

Note that this sampler is likely run much more slowly than the blocked sampler for the LKJ distribution, as updating each single element will generally incur the full cost of updating all dependencies of the entire matrix.
polyagamma sampler

The polyagamma sampler uses Pólya-gamma data augmentation to do conjugate sampling for the parameters in the linear predictor of a logistic regression model (Polson et al., 2013), analogous to the Albert-Chib data augmentation scheme for probit regression. This sampler is not assigned as a default sampler by configureMCMC and so can only be used if manually added to an MCMC configuration.

As an example, consider model code containing:

```r
for(i in 1:n) {
  logit(prob[i]) <- beta0 + beta1*x1[i] + beta2*x2[i] + u[group[i]]
  y[i] ~ dbinom(prob = prob[i], size = size[i])
}
for(j in 1:num_groups)
  u[j] ~ dnorm(0, sd = sigma_group)
```

where beta0, beta1, and beta2 are fixed effects with normal priors, u is a random effect associated with groups of data, and group[i] gives the group index of the i-th observation, y[i]. In this model, the parameters beta0, beta1, beta2, and all the u[j] can be jointly sampled by the polyagamma sampler, i.e., they will be its target nodes.

After building a model (calling nimbleModel) containing the above code, one would configure the sampler as follows:

```r
MCMCconf <- configureMCMC(model)
logistic_nodes <- c("beta0", "beta1", "beta2", "u")
# Optionally, remove default samplers.
MCMCconf$removeSamplers(logistic_nodes)
MCMCconf$addSampler(target = logistic_nodes, type = "polyagamma",
                     control = list(fixedDesignColumns=TRUE))
```

As shown here, the stochastic dependencies (y[i] here) of the target nodes must follow dbin or dbern distributions. The logit transformation of their probability parameter must be a linear function (technically an affine function) of the target nodes, which themselves must have dnorm or dmnorm priors. Zero inflation to account for structural zeroes is also supported, allowed as discussed below. The stochastic dependencies will often but not always be the observations in the logistic regression and will be referred to as 'responses' henceforth. Internally, the sampler draws latent values from the Pólya-gamma distribution, one per response. These latent values are then used to draw from the multivariate normal conditional distribution of the target nodes.

Importantly, note that because the Pólya-gamma draws are not retained when an iteration of the sampler finishes, one generally wants to apply the sampler to all parameter nodes involved in the linear predictor of the logistic regression, to avoid duplicative Pólya-gamma draws of the latent values. If there are stochastic indices (e.g., if group[i] above is stochastic), the Pólya-gamma sampler can still be used, but the stochastic nodes cannot be sampled by it and must have separate sampler(s). It is also possible in some models that regression parameters can be split into (conditionally independent) groups that can be sampled independently, e.g., if one has distinct logistic regression specifications for different sets of responses in the model.

Sampling involves use of the design matrix. The design matrix includes one column corresponding to each regression covariate as well as one columnar block corresponding to each random effect. In
the example above, the columns would include a vector of ones (to multiply beta0), the vectors x1 and x2, and a vector of indicators for each u[j] (with a 1 in row i if group[i] is j), resulting in 3+num_groups columns. Note that the polyagamma sampler can determine the design matrix from the model, even when written as above such that the design matrix is not explicitly in the model code. It is also possible to write model code for the linear prediction using matrix multiplication and an explicit design matrix, but that is not necessary.

Often the design matrix is fixed in advance, but in some cases elements of the matrix may be stochastic and sampled during the MCMC. That would be the case if there are missing values (e.g., missing covariate values (declared as stochastic nodes)) or stochastic indexing (e.g., unknown assignment of responses to clusters, as mentioned above). Note that changes in the values of any of the beta or u[j] target nodes in the example above do not change the design matrix.

Recalculating the elements of the design matrix at every iteration is costly and will likely greatly slow the sampler. To alleviate this, users can specify which columns of the design matrix are fixed (non-stochastic) using the fixedDesignColumns control argument. If all columns are fixed, which will often be the case, this argument can be specified simply as TRUE. Note that the sampler does not determine if any or all columns are fixed, so users wishing to take advantage of the large speed gains from having fixed columns should provide this control argument. Columns indicated as fixed will be determined (if necessary) when the sampler is first run and retained for subsequent iterations.

By default, NIMBLE will determine the design matrix (and as discussed above will do so repeatedly at each iteration for any columns not indicated as being fixed). If the matrix has no stochastic elements, users may choose to provide the matrix directly to the sampler via the designMatrix control argument. This will save time in computing the matrix initially but likely will have limited benefit relative to the cost of running many iterations of MCMC and therefore can be omitted in most cases.

The sampler allows for binomial responses with stochastic sizes. This would be the case in the above example if the size[i] values are themselves declared as unobserved stochastic nodes and thus are sampled by MCMC.

The sampler allows for zero inflation of the response probability in that the probability determined by the inverse logit transformation of the linear predictor can be multiplied by one or more binary scalar nodes to produce the response probability. These binary nodes must be specified via the dbern distribution or the dbin distribution with size equal to one. This functionality is intended for use in cases where another part of the model introduces structural zeroes, such as in determining occupancy in ecological occupancy models. An example would be if the above were modified by y[i] ~ dbinom(prob = z[i] * p[i], size = size[i]), where each z[i] is either 0 or 1.

The polyagamma sampler accepts the following control list elements:

- **fixedDesignColumns.** Either a single logical value indicating if the design matrix is fixed (non-stochastic) or a logical vector indicating which columns are fixed. In the latter case, the columns must be ordered exactly as the ordering of target node elements given by model$expandNodeNames(target, returnScalarComponents = TRUE), where target is the same as the target argument to configureMCMC$addSampler above. (default = FALSE)

- **designMatrix.** The full design matrix with rows corresponding to the ordering of the responses and columns ordered exactly as the ordering of target node elements given by model$expandNodeNames(target, returnScalarComponents = TRUE), where target is the same as the target argument to configureMCMC$addSampler above. If provided, all columns are assumed to be fixed, ignoring the fixedDesignColumns control element.
• nonTargetNodes. Additional stochastic nodes involved in the linear predictor that are not to be sampled as part of the sampler. This must include any nodes specifying stochastic indexes (e.g., "group" if the group[i] values are stochastic) and any parameters considered known or that for any reason one does not want to sample. Providing nonTargetNodes is required in order to allow NIMBLE to check for the presence of zero inflation.

• check. A logical value indicating whether NIMBLE should check various conditions required for validity of the sampler. This is provided for rare cases where the checking may be overly conservative and a user is sure that the sampler is valid and wants to override the checking. (default = TRUE)

noncentered sampler

The noncentered sampler is designed to sample the mean or standard deviation of a set of centered random effects while also moving the random effects values to possibly allow better mixing. The noncentered sampler deterministically shifts or scales the dependent node values to be consistent with the proposed value of the target (the mean or the standard deviation) such that the effect is to sample in a noncentered parameterization (Yu and Meng 2011), via an on-the-fly reparameterization. This can improve mixing by updating the target node based on information in the model nodes whose parent nodes are the dependent nodes of the target (i.e., the "grandchild" nodes of the target; these will often be data nodes). This comes at the extra computational cost of calculating the logProbability of the "grandchild" nodes.

It is still necessary to have other samplers on the random effects values.

Mathematically, the noncentered sampler operates in one dimension of a transformed parameter space. When sampling a mean, all random effects will be shifted by the same amount as the mean. When sampling a standard deviation, all random effects (relative to their means) will be scaled by the same factor as the standard deviation. Consider a model that includes the following code:

```r
for(i in 1:n)
  y[i] ~ dnorm(beta1*x[i] + u[group[i]], sd = sigma_obs)
for(j in 1:num_groups)
  u[j] ~ dnorm(beta0, sd = sigma_group)
```

where u is a random effect associated with groups of data, and group[i] gives the group index of the i-th observation. This model has a centered random effect, because the u[j] have the intercept beta0 as their mean. In basic univariate sampling, updates to beta0 or to sigma_group do not change u[j], making only small moves possible if num_groups is large. When the noncentered sampler considers a new value for beta0, it will shift all the u[j] so that (in this case) their prior probabilities do not change. If the noncentered sampler considers a new value for sigma_group, it will rescale all the u[j] accordingly.

The effect of such a sampling strategy is to update beta0 and sigma_group as if the model had been written in a different (noncentered) way. For updating beta0, it would be:

```r
for(i in 1:n)
  y[i] ~ dnorm(beta0 + beta1*x[i] + u[group[i]], sd = sigma_obs)
for(j in 1:num_groups)
  u[j] ~ dnorm(0, sd = sigma_group)
```

For updating sigma_group, it would be:
for(i in 1:n)
  y[i] ~ dnorm(beta1*x[i] + u[group[i]] * sigma_group, sd = sigma_obs)
for(j in 1:num_groups)
  u[j] ~ dnorm(beta0, sd = 1)

Whether centered or noncentered parameterizations result in better sampling can depend on the model and the data. Therefore Yu and Meng (2011) recommended an “interweaving” strategy of using both kinds of samplers. Adding the noncentered sampler (on either the mean or standard deviation or both) to an existing MCMC configuration for a model specified using the centered parameterization (and with a sampler already assigned to the target node) produces an overall sampling approach that is a variation on the interweaving strategy of Yu and Meng (2011). This provides the benefits of sampling in both the centered and noncentered parameterizations in a single MCMC.

There is a higher computational cost to the noncentered sampler (or to writing the model directly in one of the equivalent ways shown). The cost is that when updating beta0 or sigma_group, the relevant log probabilities calculations will include (in this case) all the of y[i], i.e. the "grandchild" nodes of beta0 or sigma_group.

The noncentered sampler is not assigned by default by configureMCMC but must be manually added. For example:

```
MCMCconf <- configureMCMC(model)
MCMCconf$addSampler(target = "beta0", type = "noncentered",
                  control = list(param = "location", sampler = "RW"))
MCMCconf$addSampler(target = "sigma_group", type = "noncentered",
                  control = list(param = "scale", sampler = "RW"))
```

While the target node will generally be either the mean (location) or standard deviation (scale) of a set of other nodes (e.g., random effects), it could in theory be used in other contexts and one can choose whether the transformation is a shift or a scale operation. In a shift operation (e.g., when the sampling target is a mean), the dependent nodes are set to their previous values plus the difference between the proposed value and previous value for the target. In a scale operation (e.g., when the sampling target is the standard deviation), the dependent nodes minus their means are multiplied by the ratio of the proposed value to the previous value for the target and the previous value for the target. Whether to shift or scale is determined from the param element of the control list.

The sampling algorithm for the target node can either be adaptive Metropolis random walk (which uses NIMBLE’s RW sampler) or slice sampling (which uses NIMBLE’s slice sampler), determined from the sampler element of the control list. In either case, the underlying sampling accounts for the Jacobian of the deterministic shifting or scaling of the dependent nodes (in the case of shifting, the Jacobian is equal to 1 and has no impact). When the target is the standard deviation of normally-distributed dependent nodes, the Jacobian cancels with the prior distribution for the dependent nodes, and the update is in effect based only on the prior for the target and the distribution of the "grandchild" nodes.

The noncentered sampler accepts the following control list elements:

- **sampler.** A character string, either "RW" or "slice" specifying the type of sampler to be used for the target node. (default = "RW")
- **param.** A character string, either "location" or "scale" specifying whether sampling is done as shifting or scaling the dependent nodes. (default = "location")
CAR_normal sampler

The CAR_normal sampler operates uniquely on improper (intrinsic) Gaussian conditional autoregressive (CAR) nodes, those with a dcar_normal prior distribution. It internally assigns one of three univariate samplers to each dimension of the target node: a posterior predictive, conjugate, or RW sampler; however these component samplers are specialized to operate on dimensions of a dcar_normal distribution.

The CAR_normal sampler accepts the following control list elements:

- carUseConjugacy. A logical argument, specifying whether to assign conjugate samplers for conjugate components of the target node. If FALSE, a RW sampler would be assigned instead. (default = TRUE)
- adaptive. A logical argument, specifying whether any component RW samplers should adapt the scale (proposal standard deviation), to achieve a theoretically desirable acceptance rate. (default = TRUE)
- adaptInterval. The interval on which to perform adaptation for any component RW samplers. Every adaptInterval MCMC iterations (prior to thinning), component RW samplers will perform an 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 for any component RW samplers. If adaptive = FALSE, scale will never change. (default = 1)

CAR_proper sampler

The CAR_proper sampler operates uniquely on proper Gaussian conditional autoregressive (CAR) nodes, those with a dcar_proper prior distribution. It internally assigns one of three univariate samplers to each dimension of the target node: a posterior predictive, conjugate, or RW sampler, however these component samplers are specialized to operate on dimensions of a dcar_proper distribution.

The CAR_proper sampler accepts the following control list elements:

- carUseConjugacy. A logical argument, specifying whether to assign conjugate samplers for conjugate components of the target node. If FALSE, a RW sampler would be assigned instead. (default = TRUE)
- adaptive. A logical argument, specifying whether any component RW samplers should adapt the scale (proposal standard deviation), to achieve a theoretically desirable acceptance rate. (default = TRUE)
- adaptInterval. The interval on which to perform adaptation for any component RW samplers. Every adaptInterval MCMC iterations (prior to thinning), component RW samplers will perform an 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 for any component RW samplers. If adaptive = FALSE, scale will never change. (default = 1)

CRP sampler

The CRP sampler is designed for fitting models involving Dirichlet process mixtures. It is exclusively assigned by NIMBLE’s default MCMC configuration to nodes having the Chinese Restaurant
Process distribution, \textit{dCRP}. It executes sequential sampling of each component of the node (i.e., the cluster membership of each element being clustered). Internally, either of two samplers can be assigned, depending on conjugate or non-conjugate structures within the model. For conjugate and non-conjugate model structures, updates are based on Algorithm 2 and Algorithm 8 in Neal (2000), respectively.

- \textit{checkConjugacy}. A logical argument, specifying whether to assign conjugate samplers if valid. (default = \texttt{TRUE})
- \textit{printTruncation}. A logical argument, specifying whether to print a warning when the MCMC attempts to use more clusters than the maximum number specified in the model. Only relevant where the user has specified the maximum number of clusters to be less than the number of observations. (default = \texttt{TRUE})

\textbf{CRP\_concentration sampler}

The CRP\_concentration sampler is designed for Bayesian nonparametric mixture modeling. It is exclusively assigned to the concentration parameter of the Dirichlet process when the model is specified using the Chinese Restaurant Process distribution, \textit{dCRP}. This sampler is assigned by default by NIMBLE’s default MCMC configuration and can only be used when the prior for the concentration parameter is a gamma distribution. The assigned sampler is an augmented beta-gamma sampler as discussed in Section 6 in Escobar and West (1995).

\textbf{prior\_samples sampler}

The prior\_samples sampler uses a provided set of numeric values (\texttt{samples}) to define the prior distribution of one or more model nodes. One every MCMC iteration, the prior\_samples sampler takes value(s) from the numeric values provided, and stores these value(s) into the target model node(s). This allows one to define the prior distribution of model parameters empirically, using a set of numeric samples, presumably obtained previously using MCMC. The target node may be either a single scalar node (scalar case), or a collection of model nodes.

The prior\_samples sampler provides two options for selection of the value to use on each MCMC iteration. The default behaviour is to take sequential values from the \texttt{samples} vector (scalar case), or in the case of multiple dimensions, sequential rows of the \texttt{samples} matrix are used. The alternative behaviour, by setting the control argument \texttt{randomDraws = TRUE}, will instead use random draws from the \texttt{samples} vector (scalar case), or randomly selected rows of the \texttt{samples} matrix in the multidimensional case.

If the default of sequential selection of values is used, and the number of MCMC iterations exceeds the length of the \texttt{samples} vector (scalar case) or the number of rows of the \texttt{samples} matrix, then \texttt{samples} will be recycled as necessary for the number of MCMC iterations. A message to this effect is also printed at the beginning of the MCMC chain.

Logically, prior\_samples samplers might want to operate first, in advance of other samplers, on every MCMC iteration. By default, at the time of MCMC building, all prior\_samples samplers are re-ordered to appear first in the list of MCMC samplers. This behaviour can be subverted, however, by setting \texttt{nimbleOptions(MCMCorderPriorSamplesSamplersFirst = \texttt{FALSE})}.

The prior\_samples sampler can be assigned to non-stochastic model nodes (nodes which are not assigned a prior distribution in the model). In fact, it is recommended that nodes being assigned a prior\_samples are not provided with a prior distribution in the model, and rather, that these nodes only appear on the right-hand-side of model declaration lines. In such case that a prior\_samples
sampler is assigned to a nodes with a prior distribution, the prior distribution will be overridden by the sample values provided to the sampler; however, the node will still be a stochastic node for other purposes, and will contribute to the model joint-density (using the sample values provided relative to the prior distribution), will have an MCMC sampler assigned to it by default, and also may introduce potential for confusion. In this case, a message is issued at the time of MCMC building.

The prior_samples sampler accepts the following control list elements:

- samples. A numeric vector or matrix. When the target node is a single scalar-valued node, samples should be a numeric vector. When the target node specifies \( d > 2 \) model dimensions, samples should be a matrix containing \( d \) columns. The samples control argument is required.
- randomDraws. A logical argument, specifying whether to use a random draw from samples on each iteration. If samples is a matrix, then a randomly-selected row of the samples matrix is used. When FALSE, sequential values (or sequential matrix rows) are used (default = FALSE).

**posterior_predictive sampler**

The posterior_predictive sampler operates only on posterior predictive stochastic nodes. A posterior predictive node is a node that is not itself data and has no data nodes in its entire downstream (descendant) dependency network. Note that such nodes play no role in inference for model parameters but have often been included in BRUGS models to make predictions, including for posterior predictive checks. As of version 0.13.0, NIMBLE samples model parameters without conditioning on the posterior predictive nodes and samples conditionally from the posterior predictive nodes as the last step of each MCMC iteration.

(Also note that 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 (posterior predictive) nodes have been included in BRUGS models, they are not needed when using NIMBLE.)

The posterior_predictive sampler functions by simulating new values for all downstream (dependent) nodes using their conditional distributions, as well as updating the associated model probabilities. A posterior_predictive sampler will automatically be assigned to all trailing non-data stochastic nodes in a model, or when possible, to any node at a point in the model after which all downstream (dependent) stochastic nodes are non-data.

The posterior_predictive sampler accepts no control list arguments.

**RJ_fixed_prior sampler**

This sampler proposes addition/removal for variable of interest in the framework of variable selection using reversible jump MCMC, with a specified prior probability of inclusion. A normal proposal distribution is used to generate proposals for the addition of the variable. This is a specialized sampler used by configureRJ function, when the model code is written without using indicator variables. See help(configureRJ) for details. It is not intended for direct assignment.

**RJ_indicator sampler**

This sampler proposes transitions of a binary indicator variable, corresponding to a variable of interest, in the framework of variable selection using reversible jump MCMC. This is a specialized
sampler used by `configureRJ` function, when the model code is written using indicator variables. See `help(configureRJ)` for details. It is not intended for direct assignment.

**RJ_toggled sampler**

This sampler operates in the framework of variable selection using reversible jump MCMC. Specifically, it conditionally performs updates of the target variable of interest using the originally-specified sampling configuration, when variable is "in the model". This is a specialized sampler used by `configureRJ` when adding a reversible jump MCMC. See `help(configureRJ)` for details. It is not intended for direct assignment.

**Author(s)**

Daniel Turek

**References**


setAndCalculate

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

See Also

configureMCMC addSampler buildMCMC runMCMC

Examples

```r
## y[1] ~ dbern() or dbinom():
# mcmcConf$addSampler(target = 'y[1]', type = 'binary')
#
# mcmcConf$addSampler(target = 'a', type = 'RW',
#     control = list(log = TRUE, adaptive = FALSE, scale = 3))
# mcmcConf$addSampler(target = 'b', type = 'RW',
#     control = list(adaptive = TRUE, adaptInterval = 200))
# mcmcConf$addSampler(target = 'p', type = 'RW',
#     control = list(reflective = TRUE))

## a, b, and c all continuous-valued:
# mcmcConf$addSampler(target = c('a', 'b', 'c'), type = 'RW_block')
#
# mcmcConf$addSampler(target = 'p', type = 'RW_llFunction',
#     control = list(llFunction = RllFun, includesTarget = FALSE))

## y[1] is a posterior predictive node:
# mcmcConf$addSampler(target = 'y[1]', type = 'posterior_predictive')

## setAndCalculate

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

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, ..., copy = TRUE, fillZeros = TRUE)

Arguments

numObj

This is the object to be resized

... sizes, provided as scalars, in order, or as a single vector

copy

logical indicating whether values should be preserved (in column-major order)

fillZeros

logical indicating whether newly allocated space should be initialized with zeros (in compiled code)

Details

Note that assigning the result of numeric, integer, logical, matrix, or array is often as good or better than using setSize. For example, ‘x <- matrix(nrow = 5, ncol = 5)’ is equivalent to ‘setSize(x, 5, 5)’ but the former allows more control over initialization.

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.

If preserving values in the resized object and/or initializing new values with 0 is not necessary, then setting these arguments to FALSE will yield slightly more efficient compiled code.

Author(s)

NIMBLE development team
setupMargNodes

Organize model nodes for marginalization

Description

Process model to organize nodes for marginalization (integration over latent nodes or random effects) as by Laplace approximation.

Usage

setupMargNodes(
  model,  
  paramNodes, 
  randomEffectsNodes, 
  calcNodes, 
  calcNodesOther, 
  split = TRUE, 
  check = TRUE, 
  allowDiscreteLatent = FALSE
)

Arguments

model  
A nimble model such as returned by \texttt{nimbleModel}.

paramNodes  
A character vector of names of stochastic nodes that are parameters of nodes to be marginalized over (\texttt{randomEffectsNodes}). See details for default.

randomEffectsNodes  
A character vector of nodes to be marginalized over (or "integrated out"). In the case of calculating the likelihood of a model with continuous random effects, the nodes to be marginalized over are the random effects, hence the name of this argument. However, one can marginalize over any nodes desired as long as they are continuous. See details for default.

calcNodes  
A character vector of nodes to be calculated as the integrand for marginalization. Typically this will include \texttt{randomEffectsNodes} and some data nodes. See details for default.

calcNodesOther  
A character vector of nodes to be calculated as part of the log likelihood that are not connected to the \texttt{randomEffectsNodes} and so are not actually part of the marginalization. These are somewhat extraneous to the purpose of this function, but it is convenient to handle them here because often the purpose of marginalization is to calculate log likelihoods, including from "other" parts of the model.

split  
A logical indicating whether to split \texttt{randomEffectsNodes} into conditionally independent sets that can be marginalized separately (\texttt{TRUE}) or to keep them all in one set for a single marginalization calculation.

check  
A logical indicating whether to try to give reasonable warnings of badly formed inputs that might be missing important nodes or include unnecessary nodes.

allowDiscreteLatent  
A logical indicating whether to allow discrete latent states. (default = \texttt{FALSE})
Details

This function is used by buildLaplace to organize model nodes into roles needed for setting up the (approximate) marginalization done by Laplace approximation. It is also possible to call this function directly and pass the resulting list (possibly modified for your needs) to buildLaplace.

Any of the input node vectors, when provided, will be processed using nodes <- model$expandNodeNames(nodes), where nodes may be paramNodes, randomEffectsNodes, and so on. This step allows any of the inputs to include node-name-like syntax that might contain multiple nodes. For example, paramNodes = 'beta[1:10]' can be provided if there are actually 10 scalar parameters, 'beta[1]' through 'beta[10]'. The actual node names in the model will be determined by the expandNodeNames step.

This function does not do any of the marginalization calculations. It only organizes nodes into roles of parameters, random effects, integrand calculations, and other log likelihood calculations.

The checking done if 'check=TRUE' tries to be reasonable, but it can't cover all cases perfectly. If it gives an unnecessary warning, simply set 'check=FALSE'.

If paramNodes is not provided, its default depends on what other arguments were provided. If neither randomEffectsNodes nor calcNodes were provided, paramNodes defaults to all top-level, stochastic nodes, excluding any posterior predictive nodes (those with no data anywhere downstream). These are determined by model$getNodeNames(topOnly = TRUE, stochOnly = TRUE, includePredictive = FALSE). If randomEffectsNodes was provided, paramNodes defaults to stochastic parents of randomEffectsNodes. In these cases, any provided calcNodes or calcNodesOther are excluded from default paramNodes. If calcNodes but not randomEffectsNodes was provided, then the default for randomEffectsNodes is determined first, and then paramNodes defaults to stochastic parents of randomEffectsNodes. Finally, any stochastic parents of calcNodes (whether provided or default) that are not in calcNodes are added to the default for paramNodes, but only after paramNodes has been used to determine the defaults for randomEffectsNodes, if necessary.

Note that to obtain sensible defaults, some nodes must have been marked as data, either by the data argument in nimbleModel or by model$setData. Otherwise, all nodes will appear to be posterior predictive nodes, and the default paramNodes may be empty.

For purposes of buildLaplace, paramNodes does not need to (but may) include deterministic nodes between the parameters and any calcNodes. Such deterministic nodes will be included in calculations automatically when needed.

If randomEffectsNodes is missing, the default is a bit complicated: it includes all latent nodes that are descendants (or "downstream") of paramNodes (if provided) and are either (i) ancestors (or "upstream") of data nodes (if calcNodes is missing), or (ii) ancestors or elements of calcNodes (if calcNodes are provided but paramNodes is missing). In all cases, discrete nodes (with warning if check=TRUE), posterior predictive nodes and paramNodes are excluded.

randomEffectsNodes should only include stochastic nodes.

If calcNodes is missing, the default is randomEffectsNodes and their descendants to the next stochastic nodes, excluding posterior predictive nodes. These are determined by model$getDependencies(randomEffectsNodes, includePredictive=FALSE).

If calcNodesOther is missing, the default is all stochastic descendants of paramNodes, excluding posterior predictive nodes (from model$getDependencies(paramNodes, stochOnly=TRUE, self=FALSE, includePosterior=FALSE)) that are not part of calcNodes.
For purposes of buildLaplace, neither calcNodes nor calcNodesOther needs to (but may) contain deterministic nodes between paramNodes and calcNodes or calcNodesOther, respectively. These will be included in calculations automatically when needed.

If split is TRUE, model$getConditionallyIndependentSets is used to determine sets of the randomEffectsNodes that can be independently marginalized. The givenNodes are the paramNodes and calcNodes excluding any randomEffectsNodes and their deterministic descendants. The nodes (to be split into sets) are the randomEffectsNodes.

If split is a numeric vector, randomEffectsNodes will be split by split(randomEffectsNodes, control$split). The last option allows arbitrary control over how randomEffectsNodes are blocked.

If check=TRUE, then defaults for each of the four categories of nodes are created even if the corresponding argument was provided. Then warnings are emitted if there are any extra (potentially unnecessary) nodes provided compared to the default or if there are any nodes in the default that were not provided (potentially necessary). These checks are not perfect and may be simply turned off if you are confident in your inputs.

(If randomEffectsNodes was provided but calcNodes was not provided, the default (for purposes of check=TRUE only) for randomEffectsNodes differs from the above description. It uses stochastic descendants of randomEffectsNodes in place of the "data nodes" when determining ancestors of data nodes. And it uses item (ii) instead of (iii) in the list above.)

Value

A list is returned with elements:

- paramNodes: final processed version of paramNodes
- randomEffectsNodes: final processed version of randomEffectsNodes
- calcNodes: final processed version of calcNodes
- calcNodesOther: final processed version of calcNodesOther
- givenNodes: Input to model$getConditionallyIndependentSets, if split=TRUE.
- randomEffectsSets: Output from model$getConditionallyIndependentSets, if split=TRUE. This will be a list of vectors of node names. The node names in one list element can be marginalized independently from those in other list elements. The union of the list elements should be all of randomEffectsNodes. If split=FALSE, randomEffectsSets will be a list with one element, simply containing randomEffectsNodes. If split is a numeric vector, randomEffectsSets will be the result of split(randomEffectsNodes, control$split).

Author(s)

Wei Zhang, Perry de Valpine, Paul van Dam-Bates
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 code is not used in member functions.

Arguments

... An arbitrary set of names

Details

Normally any object created in setup whose name appears in run or another member function is included in the saved 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 compiled. For example, `myCalc <- calcNodes(model, nodes); ans <- myCalc()` is equivalent to `ans <- model$calculate(nodes)`, but one can also do `CmyCalc <- compileNimble(myCalc)` to get a faster version. Note that this will often be much faster than using `calculate` from R with a compiled model, such as `compiled_model$calculate(nodes)` because of overhead in running `calculate` from R.

In nimbleFunctions, one would generally use `model$calculate(nodes)` in the run-time code (and similarly for `simulate` and `getLogProb`).

**Author(s)**

Perry de Valpine

---

**Description**

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

**Usage**

```r
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` (only for `simNodesMV`). Number of simulations requested.
- `saveLP` (only for `calcNodesMV`). 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**

```r
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
result <- as.matrix(Csim$mv)

## End(Not run)
```
Description

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

StickBreakingFunction  The Stick Breaking Function

Description

Computes probabilities based on stick breaking construction.

Usage

stick_breaking(z, log = 0)

Arguments

z  vector argument.
log  logical; if TRUE, weights are returned on the log scale.

Details

The stick breaking function produces a vector of probabilities that add up to one, based on a series of individual probabilities in z, which define the breaking points relative to the remaining stick length. The first element of z determines the first probability based on breaking a proportion z[1] from a stick of length one. The second element of z determines the second probability based on breaking a proportion z[2] from the remaining stick (of length 1-z[1]), and so forth. Each element of z should be in (0, 1). The returned vector has length equal to the length of z plus 1. If z[k] is equal to 1 for any k, then the returned vector has length smaller than z. If one of the components is smaller than 0 or greater than 1, NaNs are returned.

Author(s)

Claudia Wehrhahn

References

Examples

```r
z <- rbeta(5, 1, 1)
stick_breaking(z)

## Not run:
cstick_breaking <- compileNimble(stick_breaking)
cstick_breaking(z)

## End(Not run)
```

summaryLaplace

`summaryLaplace` *Summarize results from Laplace or adaptive Gauss-Hermite quadrature approximation*

Description

Process the results of the `findMLE` method of a nimble Laplace or AGHQ approximation into a more useful format.

Usage

```r
summaryLaplace(
  laplace, 
  MLEoutput, 
  originalScale = TRUE, 
  randomEffectsStdError = FALSE, 
  jointCovariance = FALSE 
)
```

```r
summaryAGHQ(
  AGHQ, 
  MLEoutput, 
  originalScale = TRUE, 
  randomEffectsStdError = FALSE, 
  jointCovariance = FALSE 
)
```

Arguments

- `laplace` The Laplace approximation object, typically the compiled one. This would be the result of compiling an object returned from `buildLaplace`.
- `MLEoutput` The maximum likelihood estimate using Laplace or AGHQ, returned from e.g. `approx$findMLE(...)`, where `approx` is the algorithm object returned by `buildLaplace` or `buildAGHQ`, or (more typically) the result of compiling that object with `compileNimble`. See `help(buildLaplace)` for more information.
originalScale

Should results be returned using the original parameterization in the model code (TRUE) or the potentially transformed parameterization used internally by the Laplace approximation (FALSE). Transformations are used for any parameters and/or random effects that have constrained ranges of valid values, so that in the transformed parameter space there are no constraints.

randomEffectsStdError

If TRUE, calculate the standard error of the estimates of random effects values.

jointCovariance

If TRUE, calculate the joint covariance matrix of the parameters and random effects together. If FALSE, calculate the covariance matrix of the parameters.

AGHQ

Same as laplace. Note that ‘buildLaplace’ and ‘buildAGHQ’ create the same kind of algorithm object that can be used interchangeably. ‘buildLaplace’ simply sets the number of quadrature points (‘nQuad’) to 1 to achieve Laplace approximation as a special case of AGHQ.

Details

The numbers obtained by this function can be obtained more directly by ‘approx$summary(...’)’. The added benefit of ‘summaryLaplace’ is to arrange the results into data frames (for parameters and random effects), with row names for the model nodes, and also adding row and column names to the covariance matrix.

Value

A list with data frames ‘params’ and ‘randomEffects’, each with columns for ‘estimate’ and (possibly) ‘se’ (standard error) and row names for model nodes, a matrix ‘vcov’ with the covariance matrix with row and column names, and ‘originalScale’ with the input value of ‘originalScale’ so it is recorded for later use if wanted.

svdNimbleList

svdNimbleList definition

Description

nimbleList definition for the type of nimbleList returned by nimSvd.

Usage

svdNimbleList

Format

An object of class list of length 1.

Author(s)

NIMBLE development team
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

See Also
Distributions for other standard distributions

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

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,
  expectModelWarning = FALSE,
  debug = FALSE,
  verbose = nimbleOptions("verbose")
)

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.

expectModelWarning boolean indicating whether `nimbleModel` is expected to produce a warning or character string giving part of expected warning.

debug logical indicating whether to put the user in a browser for debugging when `testBUGSmodel` calls `readBUGSmodel`. Intended for developer use.

verbose logical indicating whether to print additional logging information

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

```r
## Not run:
testBUGSmodel('pump')
## End(Not run)
```
values

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 `buildInterfacesForCompiledNestedNimbleFunctions` 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

```r
values(model, nodes, accessorIndex)
values(model, nodes, accessorIndex) <- value
```
Arguments

- **model**: a NIMBLE model object, either compiled or uncompiled
- **nodes**: a vector of node names, allowing index blocks that will be expanded
- **accessorIndex**: For internal NIMBLE use only
- **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

---

Using WAIC

Description

Details of the WAIC measure for comparing models. NIMBLE implements an online WAIC algorithm, computed during the course of the MCMC iterations.

Details

To obtain WAIC, set `WAIC = TRUE` in `nimbleMCMC`. If using a more customized workflow, set `enableWAIC = TRUE` in `configureMCMC` or (if skipping `configureMCMC`) in `buildMCMC`, followed by setting `WAIC = TRUE` in `runMCMC`, if using `runMCMC` to manage sample generation.

By default, NIMBLE calculates WAIC using an online algorithm that updates required summary statistics at each post-burnin iteration of the MCMC.
One can also use `calculateWAIC` to run an offline version of the WAIC algorithm after all MCMC sampling has been done. This allows calculation of WAIC from a matrix (or dataframe) of posterior samples and also retains compatibility with WAIC in versions of NIMBLE before 0.12.0. However, the offline algorithm is less flexible than the online algorithm and only provides conditional WAIC without the ability to group data points. See `help(calculateWAIC)` for details.

**controlWAIC list**

The `controlWAIC` argument is a list that controls the behavior of the WAIC algorithm and is passed to either `configureMCMC` or (if not using `configureMCMC`) `buildMCMC`. One can supply any of the following optional components:

- **online**: Logical value indicating whether to calculate WAIC during the course of the MCMC. Default is `TRUE` and setting to `FALSE` is primarily for backwards compatibility to allow use of the old `calculateWAIC` method that calculates WAIC from monitored values after the MCMC finishes.

- **dataGroups**: Optional list specifying grouping of data nodes, one element per group, with each list element containing the node names for the data nodes in that group. If provided, the predictive density values computed will be the joint density values, one joint density per group. Defaults to one data node per ‘group’. See details.

- **marginalizeNodes**: Optional set of nodes (presumably latent nodes) over which to marginalize to compute marginal WAIC (i.e., WAIC based on a marginal likelihood), rather than the default conditional WAIC (i.e., WAIC conditioning on all parent nodes of the data nodes). See details.

- **niterMarginal**: Number of Monte Carlo iterations to use when marginalizing (default is 1000).

- **convergenceSet**: Optional vector of numbers between 0 and 1 that specify a set of shorter Monte Carlo simulations for marginal WAIC calculation as fractions of the full (`niterMarginal`) Monte Carlo simulation. If not provided, NIMBLE will use 0.25, 0.50, and 0.75. NIMBLE will report the WAIC, lppd, and pWAIC that would have been obtained for these smaller Monte Carlo simulations, allowing assessment of the number of Monte Carlo samples needed for stable calculation of WAIC.

- **thin**: Logical value for specifying whether to do WAIC calculations only on thinned samples (default is `FALSE`). Likely only useful for reducing computation when using marginal WAIC.

- **nburnin_extra**: Additional number of pre-thinning MCMC iterations to discard before calculating online WAIC. This number is discarded in addition to the usual MCMC burnin, `nburnin`. The purpose of this option is to allow a user to retain some samples for inspection without having those samples used for online WAIC calculation (default = 0).

**Extracting WAIC**

The calculated WAIC and related quantities can be obtained in various ways depending on how the MCMC is run. If using `nimbleMCMC` and setting `WAIC = TRUE`, see the WAIC component of the output list. If using `runMCMC` and setting `WAIC = TRUE`, either see the WAIC component of the output list or use the `getWAIC` method of the MCMC object (in the latter case `WAIC = TRUE` is not required). If using the `run` method of the MCMC object, use the `getWAIC` method of the MCMC object.

The output of running WAIC (unless one sets `online = FALSE`) is a list containing the following components:

- **WAIC**: The computed WAIC, on the deviance scale. Smaller values are better when comparing WAIC for two models.

- **lppd**: The log predictive density component of WAIC.
pWAIC: The pWAIC estimate of the effective number of parameters, computed using the $p$WAIC2 method of Gelman et al. (2014).

To get further information, one can use the getWAICdetails method of the MCMC object. The result of running getWAICdetails is a list containing the following components:

- **marginal**: Logical value indicating whether marginal (TRUE) or conditional (FALSE) WAIC was calculated.
- **niterMarginal**: Number of Monte Carlo iterations used in computing marginal likelihoods if using marginal WAIC.
- **thin**: Whether WAIC was calculated based only on thinned samples.
- **online**: Whether WAIC was calculated during MCMC sampling.
- **nburnin_extra**: Number of additional iterations discarded as burnin, in addition to original MCMC burnin.

- **WAIC_partialMC**, **lppd_partialMC**, **pWAIC_partialMC**: The computed marginal WAIC, lppd, and pWAIC based on fewer Monte Carlo simulations, for use in assessing the sensitivity of the WAIC calculation to the number of Monte Carlo iterations.
- **niterMarginal_partialMC**: Number of Monte Carlo iterations used for the values in WAIC_partialMC, lppd_partialMC, and pWAIC_partialMC.
- **WAIC_elements**, **lppd_elements**, **pWAIC_elements**: Vectors of individual WAIC, lppd, and pWAIC values, one element per data node (or group of nodes in the case of specifying dataGroups). Of use in computing the standard error of the difference in WAIC between two models, following Vehtari et al. (2017).

**Online WAIC**

As of version 0.12.0, NIMBLE provides enhanced WAIC functionality, with user control over whether to use conditional or marginal versions of WAIC and whether to group data nodes. In addition, users are no longer required to carefully choose MCMC monitors. WAIC by default is now calculated in an online manner (updating the required summary statistics at each MCMC iteration), using all post-burnin samples. The WAIC (Watanabe, 2010) is calculated from Equations 5, 12, and 13 in Gelman et al. (2014) (i.e., using ‘pWAIC2’).

Note that there is not a unique value of WAIC for a model. By default, WAIC is calculated conditional on the parent nodes of the data nodes, and the density values used are the individual density values of the data nodes. However, by modifying the marginalizeNodes and dataGroups elements of the control list, users can request a marginal WAIC (using a marginal likelihood that integrates over user-specified latent nodes) and/or a WAIC based on grouping observations (e.g., all observations in a cluster) to use joint density values. See the MCMC Chapter of the NIMBLE User Manual for more details.

For more detail on the use of different predictive distributions, see Section 2.5 from Gelman et al. (2014) or Ariyo et al. (2019).

Note that based on a limited set of simulation experiments in Hug and Paciorek (2021) our tentative recommendation is that users only use marginal WAIC if also using grouping.

**Author(s)**

Joshua Hug and Christopher Paciorek
References


See Also

calculateWAIC configureMCMC buildMCMC runMCMC nimbleMCMC

Examples

code <- nimbleCode(
  for(j in 1:J) {
    for(i in 1:n)
      y[j, i] ~ dnorm(mu[j], sd = sigma)
    mu[j] ~ dnorm(mu0, sd = tau)
  }
  sigma ~ dunif(0, 10)
  tau ~ dunif(0, 10)
)
J <- 5
n <- 10
groups <- paste0('y[', 1:J, ', 1:]', n, ']

y <- matrix(rnorm(J*n), J, n)
Rmodel <- nimbleModel(code, constants = list(J = J, n = n), data = list(y = y), inits = list(tau = 1, sigma = 1))

## Various versions of WAIC available via online calculation.
## Conditional WAIC without data grouping:
conf <- configureMCMC(Rmodel, enableWAIC = TRUE)

## Conditional WAIC with data grouping
conf <- configureMCMC(Rmodel, enableWAIC = TRUE, controlWAIC = list(dataGroups = groups))

## Marginal WAIC with data grouping:
conf <- configureMCMC(Rmodel, enableWAIC = TRUE, controlWAIC = list(dataGroups = groups, marginalizeNodes = 'mu'))

## Not run:
Rmcmc <- buildMCMC(conf)
Cmodel <- compileNimble(Rmodel)
Cmcmc <- compileNimble(Rmcmc, project = Rmodel)
output <- runMCMC(Cmcmc, niter = 1000, WAIC = TRUE)
output$WAIC
## Alternatively call via the 'getWAIC' method; this doesn't require setting
```
## `waic=TRUE` in `runMCMC`
Cmcmc$getWAIC()
Cmcmc$getWAICdetails()

## End(Not run)
```

---

**Description**

waicDetailsNimbleList definition for the nimbleList type returned by WAIC computation.

**Usage**

waicDetailsNimbleList

**Format**

An object of class list of length 1.

**Details**

See help(waic) for details on the elements of the list.

**Author(s)**

NIMBLE development team

---

**Description**

waicNimbleList definition for the nimbleList type returned by WAIC computation.

**Usage**

waicNimbleList

**Format**

An object of class list of length 1.

**Details**

See help(waic) for details on the elements of the list.
Wishart

Author(s)

NIMBLE development team

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

See Also

Distributions for other standard distributions

Examples

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

**withNimbleOptions**  
*Temporarily set some NIMBLE options.*

**Description**

Temporarily set some NIMBLE options.

**Usage**

```r
withNimbleOptions(options, expr)
```

**Arguments**

- `options`: a list of options suitable for `nimbleOptions`.
- `expr`: an expression or statement to evaluate.

**Value**

`expr` as evaluated with given options.

**Examples**

```r
## Not run:
if (!(getNimbleOption('showCompilerOutput') == FALSE)) stop()
nf <- nimbleFunction(run = function(){ return(0); returnType(double()) })
cnf <- withNimbleOptions(list(showCompilerOutput = TRUE), {
  if (!(getNimbleOption('showCompilerOutput') == TRUE)) stop()
  compileNimble(nf)
})
if (!(getNimbleOption('showCompilerOutput') == FALSE)) stop()

## End(Not run)
```
Index

* datasets
  ADNimbleList, 7
  eigenNimbleList, 66
  optimControlNimbleList, 155
  optimResultNimbleList, 156
  svdNimbleList, 207
  waicDetailsNimbleList, 216
  waicNimbleList, 216
  [,CmodelValues-method
    (modelValuesBaseClass-class), 106
  [,CmodelValues-method, ANY, ANY
    (modelValuesBaseClass-class), 106
  [,CmodelValues-method, character, missing
    (modelValuesBaseClass-class), 106
  [,CmodelValues-method, character, missing, ANY-method
    (modelValuesBaseClass-class), 106
  [,distributionsClass-method
    (nimble-internal), 117
  [,modelValuesBaseClass-method
    (modelValuesBaseClass-class), 106
  [,numberedModelValuesAccessors-method
    (nimble-internal), 117
  [,numberedObjects-method
    (nimble-internal), 117
  [<-,CmodelValues-method
    (modelValuesBaseClass-class), 106
  [<-,modelValuesBaseClass-method
    (modelValuesBaseClass-class), 106
  [<-,numberedModelValuesAccessors-method
    (nimble-internal), 117
  [<-,numberedObjects-method
    (nimble-internal), 117
  [[[CNumericList-method
    (nimble-internal), 117
  [[[CmodelValues-method
    (modelValuesBaseClass-class), 106
  [[[RNumericList-method
    (nimble-internal), 117
  [[[conjugacyRelationshipsClass-method
    (nimble-internal), 117
  [[[distributionsClass-method
    (nimble-internal), 117
  [[[modelBaseClass-method
    (modelBaseClass-class), 96
  [[<-,CNumericList-method
    (nimble-internal), 117
  [[<-,CmodelValues-method
    (modelValuesBaseClass-class), 106
  [[<-,modelBaseClass-method
    (modelBaseClass-class), 96
  [[<-,nimPointerList-method
    (nimble-internal), 117
  [[<-,nimbleFunctionList-method
    (nimble-internal), 117
  AD (nimDerivs), 138
  ADbreak, 6
  addMonitors (MCMCconf-class), 89
  addMonitors2 (MCMCconf-class), 89
  addSampler, 195
  addSampler (MCMCconf-class), 89
  ADNimbleList, 7
  ADproxyModelClass
    (ADproxyModelClass-class), 7
  ADproxyModelClass-class, 7
  AF_slice (sampler_BASE), 177
AGHQ (buildLaplace), 16
AGHQuad (buildLaplace), 16
AGHQuad_params (nimble-internal), 117
AGHQuad_summary (nimble-internal), 117
all (nimble-R-functions), 118
any (nimble-R-functions), 118
any_na, 8
any_nan (any_na), 8
array, 144
array (nimMatrix), 143
as.carAdjacency, 8
as.carCM, 9
as.list.modelValuesBaseClass-Class
(nimble-internal), 117
as.matrix.modelValuesBaseClass-Class
(nimble-internal), 117
as.name, 109
asCol (asRow), 10
asRow, 10
autoBlock, 11, 54
autoBlockClass-Class (nimble-internal), 117
besselK (nimble-math), 117
binary (sampler_BASE), 177
BUGScontextClass-Class
(nimble-internal), 117
BUGSdeclClass (BUGSdeclClass-class), 12
BUGSdeclClass-class, 12
BUGSsingleContextClass-Class
(nimble-internal), 117
buildAGHQ, 173
buildAGHQ (buildLaplace), 16
buildAGHQGrid, 13
buildAuxiliaryFilter, 14
buildBootstrapFilter, 15
build EnsembleKF, 15
buildIteratedFilter2, 15
buildLaplace, 16, 158, 173
buildLiuWestFilter, 25
buildMCMC, 26
buildMCMC, 32, 37, 53, 54, 77, 89, 129, 176, 195, 215
buildWAIC (waic), 212
c (nimble-R-functions), 118
calc_dcatConjugacyContributions
(nimble-internal), 117
calc_dmnormAltParams (nimble-internal), 117
calc_dmnormConjugacyContributions
(nimble-internal), 117
calc_dwishAltParams (nimble-internal), 117
calcAdaptationFactor (nimble-internal), 117
calcNodes (simNodes), 202
calcNodesMV (simNodesMV), 203
calculate, 131, 162
calculate (nodeFunctions), 153
calculateDiff (nodeFunctions), 153
calculateWAIC, 35, 215
CAR-Normal, 38, 42
CAR-Proper, 40, 40
CAR_calcC (nimble-internal), 117
CAR_calcCmatrix (nimble-internal), 117
CAR_calc EVs2 (nimble-internal), 117
CAR_calc EVs3 (nimble-internal), 117
CAR_calcM (nimble-internal), 117
CAR_calcNumIslands, 45
carBounds, 42, 44, 45
carMaxBound, 43, 43, 45
carMinBound, 43, 44, 44
cat, 136, 151
cat (nimCat), 135
Categorical, 46
categorical (sampler_BASE), 177
cc_getNodesInExpr (nimble-internal), 117
checkConjugacy (modelBaseClass-class), 96
checkInterrupt, 47
ChineseRestaurantProcess, 47
clearCompiled, 48
cloglog (nimble-math), 117
CmodelBaseClass
(CmodelBaseClass-class), 49
CmodelBaseClass-class, 49
Cmulti NimbleFunctionClass-Class
(nimble-internal), 117
Cmulti NimbleFunctionClass-class
(nimble-internal), 117
Cmulti Nimble ObjClass-Class
(nimble-internal), 117
CnimbleFunctionBase
(CnimbleFunctionBase-class), 49
CnimbleFunctionBase-class, 49
findClass (nimble-internal), 117
findMethodsInExprClass (nimble-internal), 117
flat, 69
gamma, 82
getBound, 70, 86
getBUGSexampleDir, 70
class (modelBaseClass-class), 96
getClass (modelBaseClass-class), 117
getConditionallyIndependentSets, 71
code (modelBaseClass-class), 96
codeClass (modelBaseClass-class), 96
definition, 73
definition (modelBaseClass-class), 96
definition (modelBaseClass-class), 117
definitions (modelBaseClass-class), 96
definitionsList (modelBaseClass-class), 96
dimension (modelBaseClass-class), 96
distribution (modelBaseClass-class), 96
distributionInfo (distributionInfo), 63
downstream (modelBaseClass-class), 96
downstreamClasses (modelBaseClass-class), 96
logProb (nodeFunctions), 153
logProbNodes (simNodes), 202
distributionInfo (distributionInfo), 96
distributionInfo (distributionInfo), 117
logProbNodesMV (simNodesMV), 203
macroInits (modelBaseClass-class), 96
macroParameters, 73
Monitors (MCMCconf-class), 89
Monitors2 (MCMCconf-class), 89
nimbleOption, 75
nimbleProject (nimble-internal), 117
nodeFunctionIndexedInfo (nimble-internal), 117
nodeNames (modelBaseClass-class), 96
param, 75, 88
paramNames (distributionInfo), 63
parents (modelBaseClass-class), 96
refClass (modelBaseClass-class), 117
SamplerExecutionOrder (MCMCconf-class), 89
samplers (MCMCconf-class), 89
samplesDPmeasure, 76
size, 78
type (distributionInfo), 63
varNames (modelBaseClass-class), 96
WAIC (waic), 212
WAICdetails (waic), 212
halfflat (flat), 69
icloglog (nimble-math), 117
identityMatrix, 79
ilogit (nimble-math), 117
indexedNodeInfoTableClass (nimble-internal), 117
initializeInfo (modelBaseClass-class), 96
initializeModel, 79, 104
inprod (nimble-math), 117
integer, 146
integer (nimble-internal), 145
integrate, 141, 142
integrate (nimble-internal), 141
interval, 80
inverse (nimble-math), 117
Inverse-Gamma, 81
Inverse-Wishart, 83
inverse-wishart (inverse-Wishart), 83
iprobit (nimble-math), 117
is.Cmodel (nimble-internal), 117
is.Cnf (nimble-internal), 117
is.model (nimble-internal), 117
is.na (nimble-R-functions), 118
is.nan (nimble-R-functions), 118
is.nf, 84
is.nfGenerator (nimble-internal), 117
is.nl, 85
is.Rmodel (nimble-internal), 117
isBinary (modelBaseClass-class), 96
isClass (modelBaseClass-class), 96
isData (modelBaseClass-class), 96
isDetermin (modelBaseClass-class), 96
isDiscrete (modelBaseClass-class), 96
isEndNode (modelBaseClass-class), 96
isMultivariate (modelBaseClass-class), 96
isSealedClass (modelBaseClass-class), 117
isStoch (modelBaseClass-class), 96
isTruncated (modelBaseClass-class), 96
isUnivariate (modelBaseClass-class), 96
isUserDefined (distributionInfo), 63
isVirtualClass (modelBaseClass-class), 117
isXs3Class (modelBaseClass-class), 117
INDEX

keywordInfoClass-Class
  (nimble-internal), 117
Laplace (buildLaplace), 16
laplace (buildLaplace), 16
length (nimble-R-functions), 118
LKJ, 85
lkj (LKJ), 85
lkj_corr (LKJ), 85
lkj_corr_cholesky (LKJ), 85
logdet (nimble-math), 117
logfact (nimble-math), 117
loggam (nimble-math), 117
logical, 146
logical (nimNumeric), 145
logit (nimble-math), 117
makeBoundInfo, 86
MakeCustomModelClass-Class
  (nimble-internal), 117
makeCustomModelValuesClass-Class
  (nimble-internal), 117
makeModelDerivsInfo, 87
makeParamInfo, 88
mapsClass-Class (nimble-internal), 117
matrix, 144
matrix (nimMatrix), 143
MCEM_mcse (nimble-internal), 117
mcmc_createModelObject
  (nimble-internal), 117
MCMCconf, 54
MCMCconf (MCMCconf-class), 89
MCMCconf-class, 89
messageIfVerbose (nimble-internal), 117
model_macro_builder, 108
modelBaseClass, 104, 130, 131
modelBaseClass (modelBaseClass-class), 96
modelBaseClass-class, 96
modelDefClass (modelDefClass-class), 104
modelDefClass-class, 104
modelDefInfoClass-Class
  (nimble-internal), 117
modelInitialization, 105
modelValues, 105
modelValuesBaseClass
  (modelValuesBaseClass-class), 106
modelValuesBaseClass-class, 106
modelValuesConf, 107
Multinomial, 111
multinomial (Multinomial), 111
Multivariate-t, 112
multivariate-t (Multivariate-t), 112
MultivariateNormal, 114
mvInfoClass-Class (nimble-internal), 117
mv (Multivariate-t), 112
newModel (modelBaseClass-class), 96
nf_preProcessMemberDataObject
  (nimble-internal), 117
nfCompilationInfoClass-Class
  (nimble-internal), 117
nfMethod, 115, 116
nfVar, 116
nfVar<- (nfVar), 116
nimArray, 146
nimArray (nimMatrix), 143
nimble-internal, 117
nimble-math, 117
nimble-R-functions, 118
nimbleCode, 119, 130, 161
nimbleExternalCall, 120, 134
nimbleFunction, 85, 122, 125
nimbleFunctionBase
  (nimbleFunctionBase-class), 123
nimbleFunctionBase-class, 123
nimbleFunctionList
  (nimbleFunctionList-class), 124
nimbleFunctionList-class, 124
nimbleFunctionVirtual, 123, 124
nimbleGraphClass-Class
  (nimble-internal), 117
nimbleInternalFunctions
  (nimble-internal), 117
nimbleList, 7, 85, 125, 134, 135, 140, 152, 155, 156
nimbleListDefClass-Class
  (nimble-internal), 117
nimbleMCMC, 34, 37, 54, 126, 176, 215
nimbleModel, 11, 52, 86, 88, 96, 119, 130, 130, 131, 162, 163
nimbleOptions, 131, 132, 162, 211
nimbleProjectClass-Class
  (nimble-internal), 117
nimbleRcall, 121, 133
nimbleType, 126
nimbleType (nimbleType-class), 134
nimType-class, 134
nimUserNamespace (nimble-internal), 117
nimC (nimble-R-functions), 118
nimCat, 135
nimCopy, 136
nimDerivs, 7, 138
nimDerivsInfoClass-Class
  (nimble-internal), 117
nimDim, 139
nimEigen, 66, 140, 153
nimEquals (nimble-math), 117
nimInteger, 144, 145
nimInteger (nimNumeric), 145
nimIntegrate, 141
nimLogical, 144, 145
nimLogical (nimNumeric), 145
nimMatrix, 143, 146
nimNumeric, 144, 145, 145
nimOptim, 19, 20, 146, 148, 149, 155, 156
nimOptimDefaultControl, 148
nimOptimMethod, 149
nimPrint, 150
nimRep (nimble-R-functions), 118
nimRound (nimble-math), 117
nimSeq (nimble-R-functions), 118
nimStep (nimble-math), 117
nimStop, 151
nimSvd, 141, 152, 207, 208
nimSwitch (nimble-math), 117
nlCompilationInfoClass-Class
  (nimble-internal), 117
nodeFunctions, 153
numeric, 146
numeric (nimNumeric), 145
optim, 14, 28, 146–149, 155, 156
optimControlNimbleList, 149, 155
optimDefaultControl, 155
optimResultNimbleList, 148, 156
parameterTransform, 24, 28, 30, 156
parse, 109
pexp_nimble (Exponential), 67
pexp_nimble (Exponential), 67
phi (nimble-math), 117
pinvgamma (Inverse-Gamma), 81
posterior_predictive (sampler_BASE), 177
posteriorClass-Class (nimble-internal), 117
pow (nimble-math), 117
pow_int, 158
pqDefined (distributionInfo), 63
print, 136
print (nimPrint), 150
printErrors, 159
printMonitors (MCMCconf-class), 89
printSamplers (MCMCconf-class), 89
prior_samples (sampler_BASE), 177
probit (nimble-math), 117
promptClass-Class (nimble-internal), 117
pt_nonstandard (t), 208
qexp_nimble (Exponential), 67
qinvgamma (Inverse-Gamma), 81
qt_nonstandard (t), 208
quote, 109, 119, 130
R6Class-Class (nimble-internal), 117
rankSample, 160
rcar_normal (CAR-Normal), 38
rcar_proper (CAR-Proper), 40
rcar_normal (CAR-Normal), 38
rcat (Categorical), 46
RCfunctionCompileClass-Class
  (nimble-internal), 117
RCfunInfoClass-Class (nimble-internal), 117
rconstraint (Constraint), 58
rCRP (ChineseRestaurantProcess), 47
rdexp (Double-Exponential), 65
rdiag (Dirichlet), 62
readBUGSmodel, 71, 119, 131, 161, 162
registerDistributions, 163
removeClass-Class (nimble-internal), 117
removeSamplers (MCMCconf-class), 89
rep (nimble-R-functions), 118
resetClass-Class (nimble-internal), 117
resetData (modelBaseClass-class), 96
resetMonitors (MCMCconf-class), 89
resize, 166
rexp_nimble (Exponential), 67
rflat (flat), 69
rhalfflat (flat), 69
rinterval (Interval), 80
rinvgamma (Inverse-Gamma), 81
rinwish_chol (Inverse-Wishart), 83
nsicr (nimble-internal), 117
nsicr (nimble-internal), 117

INDEX
INDEX

RJ_fixed_prior (sampler_BASE), 177
RJ_indicator (sampler_BASE), 177
RJ_toggled (sampler_BASE), 177
rlkj (LKJ), 85
rlkj_corr_cholesky (LKJ), 85
RMakemCustomModelClass-Class
  (nibble-internal), 117
Rmatrix2mvOneVar, 167
rmnorm_chol (MultivariateNormal), 114
RmodelBaseClass
  (RmodelBaseClass-class), 168
RmodelBaseClass-class, 168
rmulti (Multinomial), 111
rmvt_chol (Multivariate-t), 112
rsqrtinvgamma (nibble-internal), 117
rt_nonstandard (t), 208
run.time, 168
runAGHQ, 17
runAGHQ (runLaplace), 172
runCrossValidate, 169
runLaplace, 172
runMCMC, 34, 37, 54, 129, 174, 195, 215
RW (sampler_BASE), 177
RW_block (sampler_BASE), 177
RW_block_lkj_corr_cholesky
  (sampler_BASE), 177
RW_dirichlet (sampler_BASE), 177
RW_lkj_corr_cholesky (sampler_BASE), 177
RW_llfFunction (sampler_BASE), 177
RW_llfFunction_block (sampler_BASE), 177
RW_PF (sampler_BASE), 177
RW_PF_block (sampler_BASE), 177
RW_wishart (sampler_BASE), 177
rwish_chol (Wishart), 217
S3Class-Class (nibble-internal), 117
sampler (sampler_BASE), 177
sampler_AF_slice (sampler_BASE), 177
sampler_BASE, 177
sampler_binary (sampler_BASE), 177
sampler_CAR_normal (sampler_BASE), 177
sampler_CAR_proper (sampler_BASE), 177
sampler_categorical (sampler_BASE), 177
sampler_crossLevel (sampler_BASE), 177
sampler_CRP (sampler_BASE), 177
sampler_CRP_concentration
  (sampler_BASE), 177
sampler_ess (sampler_BASE), 177
sampler_noncentered (sampler_BASE), 177
sampler_polyagamma (sampler_BASE), 177
sampler_posterior_predictive, 53
sampler_posterior_predictive
  (sampler_BASE), 177
sampler_prior_samples (sampler_BASE), 177
sampler_RJ_fixed_prior (sampler_BASE), 177
sampler_RJ_indicator (sampler_BASE), 177
sampler_RJ_toggled (sampler_BASE), 177
sampler_RW, 52, 53
sampler_RW (sampler_BASE), 177
sampler_RW_block (sampler_BASE), 177
sampler_RW_block_lkj_corr_cholesky
  (sampler_BASE), 177
sampler_RW_dirichlet (sampler_BASE), 177
sampler_RW_lkj_corr_cholesky
  (sampler_BASE), 177
sampler_RW_llfFunction (sampler_BASE), 177
sampler_RW_llfFunction_block
  (sampler_BASE), 177
sampler_RW_multinomial (sampler_BASE), 177
sampler_RW_wishart (sampler_BASE), 177
sampler_slice, 53
sampler_slice (sampler_BASE), 177
sampler_slice_CRP_base_param
  (sampler_BASE), 177
samplers, 55
samplers (sampler_BASE), 177
samplesSummary (nibble-internal), 117
sealClass-Class (nibble-internal), 117
seq (nibble-R-functions), 118
seq_along (nibble-R-functions), 118
setAndCalculate, 195
setAndCalculateDiff (setAndCalculate), 195
setAndCalculateOne, 197
setClass-Class (nibble-internal), 117
setData (modelBaseClass-class), 96
setInits (modelBaseClass-class), 96
setMonitors (MCMCconf-class), 89
setMonitors2 (MCMCconf-class), 89
setOldClass-Class (nibble-internal), 117
setRefClass, 131
setRefClass-Class (nibble-internal), 117
setSamplerExecutionOrder
setSamplers (MCMCconf-class), 89
setSize, 198
setThin (MCMCconf-class), 89
setThin2 (MCMCconf-class), 89
setupCodeTemplateClass-Class
(nimble-internal), 117
setupMargNodes, 16, 18, 26, 28, 30, 199
setupOutputs, 202
showClass-Class (nimble-internal), 117
simNodes, 202
simNodesMV, 203
simulate, 154
simulate (nodeFunctions), 153
singleModelValuesAccess
(nimble-internal), 117
singleModelValuesAccessClass-Class
(nimble-internal), 117
singleVarAccessClass
(singleVarAccessClass-class), 205
singleVarAccessClass-class, 205
slice (sampler_BASE), 177
stick_breaking (StickBreakingFunction), 205
stickbreaking (StickBreakingFunction), 205
StickBreakingFunction, 205
stop (nimStop), 151
substitute, 109
summaryAGHQ (summaryLaplace), 206
summaryLaplace, 18, 174, 206
svd (nimSvd), 152
svdNimbleList, 207
t, 208
testBUGSmodel, 209
topologicallySortNodes
(modelBaseClass-class), 96
valueInCompiledNimbleFunction, 210
values, 211
values<- (values), 211
varInfoClass-Class (nimble-internal), 117
WAIC (waic), 212
waic, 37, 212
waicDetailsNimbleList, 216
waicNimbleList, 216
which (nimble-R-functions), 118
Wishart, 217
wishart (Wishart), 217
withNimbleOptions, 218