NIMBLE User Manual

NIMBLE Development Team

Version 0.4

Contents

1 Welcome to NIMBLE **5**
- 1.1 Why something new? . . . 5
- 1.2 What does NIMBLE do? . . . 6
- 1.3 How to use this manual . . . 6

2 Lightning introduction **7**
- 2.1 A brief example . . . 7
- 2.2 Creating a model . . . 7
- 2.3 Compiling the model . . . 12
- 2.4 Creating, compiling and running a basic MCMC specification . . . 12
- 2.5 Customizing the MCMC . . . 14
- 2.6 Running MCEM . . . 16
- 2.7 Creating your own functions . . . 16

3 More Introduction **20**
- 3.1 NIMBLE adopts and extends the BUGS language for specifying models . . . 20
- 3.2 The NIMBLE language for writing algorithms . . . 21
- 3.3 The NIMBLE algorithm library . . . 22

4 Getting started **23**
- 4.1 Requirements to run NIMBLE . . . 23
- 4.2 Installation . . . 23
  - 4.2.1 Using your own copy of Eigen . . . 23
  - 4.2.2 Using libnimble . . . 24
  - 4.2.3 LAPACK and BLAS . . . 24
  - 4.2.4 Problems with Installation . . . 24
- 4.3 Installing a C++ compiler for R to use . . . 24
  - 4.3.1 OS X . . . 25
  - 4.3.2 Linux . . . 25
  - 4.3.3 Windows . . . 25
- 4.4 Customizing Compilation of the NIMBLE-generated Code . . . 25

5 Building models **27**
- 5.1 Overview of supported features and extensions of BUGS and JAGS . . . 27
  - 5.1.1 Supported features of BUGS and JAGS . . . 27

5.1.2 NIMBLE's Extensions to BUGS and JAGS . . . 27
5.1.3 Not-yet-supported features of BUGS and JAGS . . . 28
5.2 Writing models . . . 28
5.2.1 Declaring stochastic and deterministic nodes . . . 28
5.2.2 Available distributions and functions . . . 30
5.2.3 Available BUGS language functions . . . 33
5.2.4 Available link functions . . . 35
5.2.5 Adding user-defined distributions and functions . . . 36
5.2.6 Data and constants . . . 40
5.2.7 Defining alternative models with the same code . . . 42
5.2.8 Truncation, censoring, and constraints . . . 43
5.2.9 Understanding lifted nodes . . . 45
5.3 Creating model objects . . . 46
5.3.1 Using `nimbleModel()` to specify a model . . . 46
5.3.2 Specifying a model from standard `R2WinBUGS` and `rjags` input files . . . 46

6 Using NIMBLE models from R **49**
6.1 Some basic concepts and terminology . . . 49
6.2 Accessing variables . . . 50
6.2.1 Accessing log probabilities via logProb variables . . . 50
6.3 Accessing nodes . . . 51
6.3.1 How nodes are named . . . 52
6.3.2 Why use node names? . . . 52
6.4 `calculate()`, `simulate()`, and `getLogProb()` . . . 53
6.4.1 For arbitrary collections of nodes . . . 53
6.4.2 Direct access to each node's functions . . . 54
6.5 Querying model structure . . . 55
6.5.1 `getNodeNames()` and `getVarNames()` . . . 55
6.5.2 `getDependencies()` . . . 56
6.5.3 `isData()` . . . 57
6.6 The *modelValues* data structure . . . 58
6.6.1 Accessing contents of modelValues . . . 59
6.7 NIMBLE passes objects by reference . . . 62

7 MCMC **63**
7.1 The MCMC configuration . . . 63
7.1.1 Default MCMC configuration . . . 64
7.1.2 Customizing the MCMC configuration . . . 65
7.2 Building and compiling the MCMC algorithm . . . 68
7.3 Executing the MCMC algorithm . . . 69
7.4 Extracting MCMC samples . . . 69
7.5 Sampler Algorithms provided with NIMBLE . . . 70
7.5.1 Terminal node *end* Sampler . . . 70
7.5.2 Scalar Metropolis-Hastings random walk `RW` sampler . . . 71
7.5.3 Multivariate Metropolis-Hastings `RW_block` sampler . . . 71

7.5.4 Slice sampler . . . 72
7.5.5 Hierarchical `crossLevel` sampler . . . 73
7.5.6 Using customized log likelihood evaluations via the `RW_llFunction` sampler . . . 73
7.5.7 Conjugate (Gibbs) samplers . . . 74
7.6 Detailed MCMC example: `litters` . . . 74
7.7 Comparing different MCMC engines with `MCMCsuite` . . . 78
7.7.1 MCMC Suite example: `litters` . . . 78
7.7.2 MCMC Suite outputs . . . 79
7.7.3 Arguments to MCMC Suite . . . 80
7.8 Advanced topics . . . 83
7.8.1 Custom sampler functions . . . 83

8 Other algorithms provided by NIMBLE 86
8.1 Basic Utilities . . . 86
8.1.1 `simNodes`, `calcNodes`, and `getLogProbs` . . . 86
8.1.2 `simNodesMV`, `calcNodesMV`, and `getLogProbsMV` . . . 88
8.2 Particle Filter . . . 89
8.3 Monte Carlo Expectation Maximization (MCEM) . . . 90

9 Writing `nimbleFunctions` 93
9.1 Writing nimbleFunctions . . . 93
9.2 Using and compiling nimbleFunctions . . . 95
9.2.1 Accessing and modifying numeric values from setup . . . 95
9.3 `nimbleFunction`s without `setup` code . . . 96
9.4 Useful tools for `setup` functions . . . 97
9.4.1 Control of `setup` outputs . . . 97
9.5 NIMBLE language components . . . 98
9.5.1 Basics . . . 98
9.5.2 Declaring argument types and the return type . . . 98
9.5.3 Driving models: `calculate`, `calculateDiff`, `simulate`, and `getLogProb` 99
9.5.4 Accessing model and modelValues variables and using `copy` . . . 99
9.5.5 Using model variables and modelValues in expressions . . . 103
9.5.6 Getting and setting more than one model node or variable at a time using `values` . . . 103
9.5.7 Basic flow control: if-then-else, for, and while . . . 104
9.5.8 How numeric types work . . . 104
9.5.9 Querying and changing sizes . . . 105
9.5.10 Basic math and linear algebra . . . 105
9.5.11 Including other methods in a nimbleFunction . . . 106
9.5.12 Using other nimbleFunctions . . . 107
9.5.13 Virtual nimbleFunctions and nimbleFunctionLists . . . 108
9.5.14 print and stop . . . 110
9.5.15 Character objects . . . 110
9.5.16 Alternative keywords for some functions . . . 111

9.5.17 User-defined data structures 111
9.5.18 Distribution functions 111
9.5.19 rankSample 112
9.5.20 DSL vectors and arrays 112

Chapter 1

Welcome to NIMBLE

NIMBLE is a system for building and sharing analysis methods for statistical models, especially for hierarchical models and computationally-intensive methods. This is an early version, 0.4. You can do quite a bit with it, but it has some rough edges and gaps, and we plan to keep expanding it. If you want to analyze data, we hope you will find something already useful. If you want to build algorithms, we hope you will program in NIMBLE and make an R package providing your method. We also hope you will join the mailing lists (R-nimble.org) and help improve NIMBLE by telling us what you want to do with it, what you like, and what could be better. We have a lot of ideas for how to improve it, but we want your help and ideas too.

1.1 Why something new?

There is a lot of statistical software out there. Why did we build something new? More and more, statistical models are being customized to the details of each project. That means it is often difficult to find a package whose set of available models and methods includes what you need. And more and more, statistical models are hierarchical, meaning they have some unobserved random variables between the parameters and the data. These may be random effects, shared frailties, latent states, or other such things. Or a model may be hierarchical simply due to putting Bayesian priors on parameters. Except for simple cases, hierarchical statistical models are often analyzed with computationally-intensive algorithms, the best known of which is Markov chain Monte Carlo (MCMC).

Several existing software systems have become widely used by providing a flexible way to say what the model is and then automatically providing an algorithm such as MCMC. When these work, and when MCMC is what you want, that's great. Unfortunately, there are a lot of hard models out there for which default MCMCs don't work very well. And there are also a lot of useful new and old algorithms that are not MCMC, but they can be hard to find implemented for the model you need, and you may have to go learn a new system to use a new algorithm. That's why we wanted to create a system that combines a flexible system for model specification – the BUGS language – with the ability to program with those models. That's the goal of NIMBLE.

1.2 What does NIMBLE do?

NIMBLE stands for Numerical Inference of statistical Models for Bayesian and Likelihood Estimation. Although NIMBLE was motivated by algorithms for hierarchical statistical models, you could use it for simpler models too.

You can think of NIMBLE as comprising three pieces:

1. A system for writing statistical models flexibly, which is an extension of the BUGS language[1].
2. A library of algorithms such as MCMC.
3. A language, called NIMBLE, embedded within and similar in style to R, for writing algorithms that operate on BUGS models.

Both BUGS models and NIMBLE algorithms are automatically processed into C++ code, compiled, and loaded back into R with seamless interfaces.

Since NIMBLE can compile R-like functions into C++ that use the Eigen library for fast linear algebra, it can be useful for making fast numerical functions with or without BUGS models involved[2]

One of the beauties of R is that many of the high-level analysis functions are themselves written in R, so it is easy to see their code and modify them. The same is true for NIMBLE: the algorithms are themselves written in the NIMBLE language.

1.3 How to use this manual

We emphasize that you can use NIMBLE for data analysis with the algorithms provided by NIMBLE without ever using the NIMBLE language to write algorithms. So as you get started, feel free to focus on Chapters 2-8. The algorithm library in v0.4 is just a start, so we hope you'll let us know what you want to see and consider writing it in NIMBLE. More about NIMBLE programming comes in 9.

[1] But see Section 5.1.3 for information about limitations and extensions to how NIMBLE handles BUGS right now.

[2] The packages Rcpp and RcppEigen provide different ways of connecting C++, the Eigen library and R. In those packages you program directly in C++, while in NIMBLE you program in an R-like fashion and the NIMBLE compiler turns it into C++. Programming directly in C++ allows full access to C++, while programming in NIMBLE allows simpler code.

Chapter 2

Lightning introduction

2.1 A brief example

Here we'll give a simple example of building a model and running some algorithms on the model, as well as creating our own user-specified algorithm. The goal is to give you a sense for what one can do in the system. Later sections will provide more detail.

We'll use the *pump* model example from BUGS[1]. We could load the model from the standard BUGS example file formats (5.3.2), but instead we'll show how to enter it directly in R.

In this "lightning introduction" we will:

1. Create the model for the pump example.
2. Compile the model.
3. Create a basic MCMC specification for the pump model.
4. Compile and run the MCMC
5. Customize the MCMC specification and compile and run that.
6. Create, compile and run a Monte Carlo Expectation Maximization (MCEM) algorithm, which illustrates some of the flexibility NIMBLE provides to combine R and NIMBLE.
7. Write a short `nimbleFunction` to generate simulations from designated nodes of any model.

2.2 Creating a model

First we define the model code, its constants, data, and initial values for MCMC.

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

[1]The data set describes failure times of some pumps.

```
  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))
```

Now let's create the model and look at some of its nodes.

```
pump <- nimbleModel(code = pumpCode, name = 'pump', constants = pumpConsts,
                    data = pumpData, inits = pumpInits)

pump$getNodeNames()

##  [1] "alpha"                "beta"
##  [3] "lifted_d1_over_beta" "theta[1]"
##  [5] "theta[2]"             "theta[3]"
##  [7] "theta[4]"             "theta[5]"
##  [9] "theta[6]"             "theta[7]"
## [11] "theta[8]"             "theta[9]"
## [13] "theta[10]"            "lambda[1]"
## [15] "lambda[2]"            "lambda[3]"
## [17] "lambda[4]"            "lambda[5]"
## [19] "lambda[6]"            "lambda[7]"
## [21] "lambda[8]"            "lambda[9]"
## [23] "lambda[10]"           "x[1]"
## [25] "x[2]"                 "x[3]"
## [27] "x[4]"                 "x[5]"
## [29] "x[6]"                 "x[7]"
## [31] "x[8]"                 "x[9]"
## [33] "x[10]"

pump$x

##  [1]  5  1  5 14  3 19  1  1  4 22

pump$logProb_x

##  [1]  -2.998011  -1.118924  -1.882686  -2.319466  -4.254550
##  [6] -20.739651  -2.358795  -2.358795  -9.630645 -48.447798
```

```
pump$alpha

## [1] 1

pump$theta

##  [1] 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1

pump$lambda

##  [1]  9.430  1.570  6.290 12.600  0.524  3.140  0.105  0.105
##  [9]  0.210  1.050
```

Notice that in the list of nodes, NIMBLE has introduced a new node, `lifted_d1_over_beta`. We call this a "lifted" node. Like R, NIMBLE allows alternative parameterizations, such as the scale or rate parameterization of the gamma distribution. Choice of parameterization can generate a lifted node, as can using a link function or a distribution argument that is an expression. It's helpful to know why they exist, but you shouldn't need to worry about them.

Thanks to the plotting capabilities of the `igraph` package that NIMBLE uses to represent the directed acyclic graph, we can plot the model (figure 2.1).

```
plot(pump$graph)
```

You are in control of the model. By default, `nimbleModel` does its best to initialize a model, but let's say you want to re-initialize `theta`. To simulate from the prior for `theta` (overwriting the initial values previously in the model) we first need to be sure the parent nodes of all `theta[i]` nodes were fully initialized, including any non-stochastic nodes such as lifted nodes. We then use the `simulate` function to simulate from the distribution for `theta`. Finally we use the `calculate` function to calculate the dependencies of `theta`, namely `lambda` and the log probabilities of `x` to ensure all parts of the model are up to date. First we show how to use the model's `getDependencies` method to query information about its graph.

```
## Show all dependencies of alpha and beta terminating in stochastic nodes
pump$getDependencies(c('alpha', 'beta'))

##  [1] "alpha"               "beta"
##  [3] "lifted_d1_over_beta" "theta[1]"
##  [5] "theta[2]"            "theta[3]"
##  [7] "theta[4]"            "theta[5]"
##  [9] "theta[6]"            "theta[7]"
## [11] "theta[8]"            "theta[9]"
## [13] "theta[10]"
```

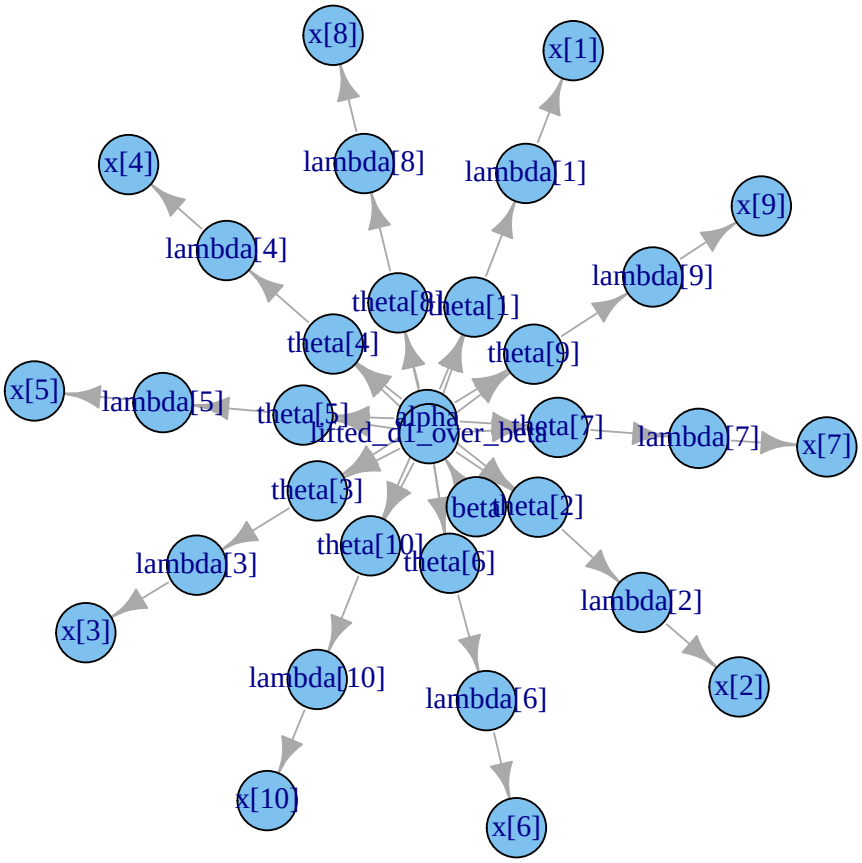


Figure 2.1: Directed Acyclic Graph plot of the pump model, thanks to the igraph package

```
## Now show only the deterministic dependencies
pump$getDependencies(c('alpha', 'beta'), determOnly = TRUE)

## [1] "lifted_d1_over_beta"

## Check that the lifted node was initialized.
pump[["lifted_d1_over_beta"]] ## It was.

## [1] 1

## Now let's simulate new theta values
set.seed(0) ## This makes the simulations here reproducible
simulate(pump, 'theta')
pump$theta   ## the new theta values

##  [1] 1.79180692 0.29592523 0.08369014 0.83617765 1.22254365
##  [6] 1.15835525 0.99001994 0.30737332 0.09461909 0.15720154

## lambda and logProb_x haven't been re-calculated yet
pump$lambda ## these are the same values as above

##  [1]  9.430  1.570  6.290 12.600  0.524  3.140  0.105  0.105
##  [9]  0.210  1.050

pump$logProb_x

##  [1]  -2.998011  -1.118924  -1.882686  -2.319466  -4.254550
##  [6] -20.739651  -2.358795  -2.358795  -9.630645 -48.447798

getLogProb(pump, 'x') ## The sum of pump\£logProb_x

## [1] -96.10932

calculate(pump, pump$getDependencies(c('theta')))

## [1] -286.6951

pump$lambda  ## Now they have.

##  [1] 168.9673926   4.6460261   5.2641096 105.3583839   6.4061287
##  [6]  36.3723548   1.0395209   0.3227420   0.1987001   1.6506161

pump$logProb_x

##  [1] -148.106356   -3.110014   -1.747041  -65.346457   -2.626123
##  [6]   -7.429868   -1.000761   -1.453644   -9.840589  -39.096527
```

Notice that the first `getDependencies` call returned dependencies from `alpha` and `beta` down to the next stochastic nodes in the model. The second call requested only deterministic dependencies. The call to `calculate(pump, 'theta')` expands `'theta'` to include all nodes in `theta`. After simulating into `theta`, we can see that `lambda` and the log probabilities of `x` still reflect the old values of `theta`, so we `calculate` them and then see that they have been updated.

2.3 Compiling the model

Next we compile the model, which means generating C++ code, compiling that code, and loading it back into R with an object that can be used just like the uncompiled model. The values in the compiled model will be initialized from those of the original model in R, but the original and compiled models are distinct objects so any subsequent changes in one will not be reflected in the other.

```
Cpump <- compileNimble(pump)
Cpump$theta

##  [1] 1.79180692 0.29592523 0.08369014 0.83617765 1.22254365
##  [6] 1.15835525 0.99001994 0.30737332 0.09461909 0.15720154
```

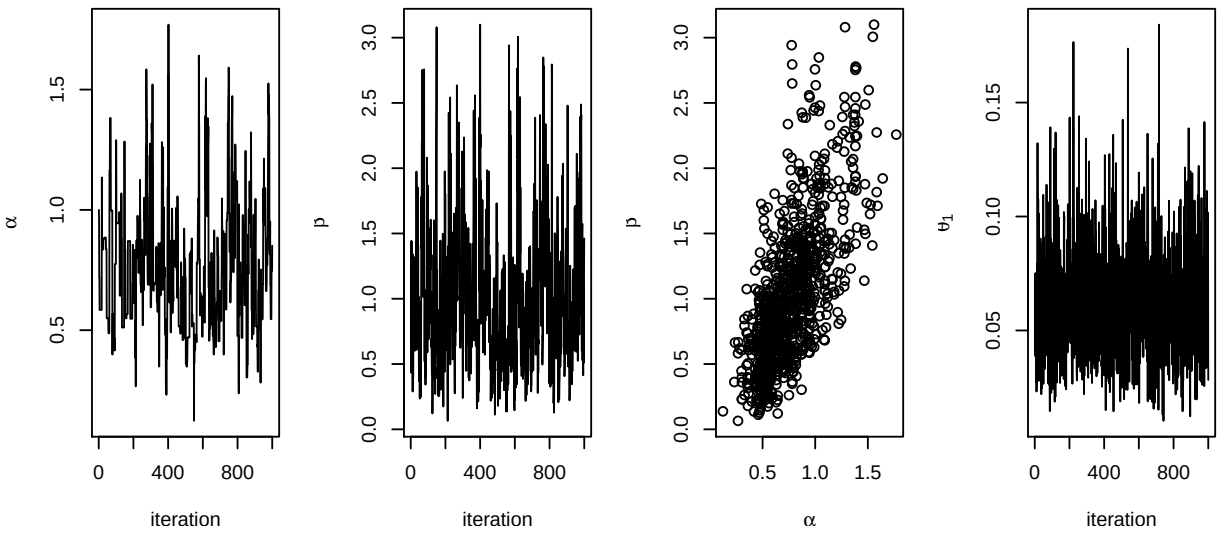
2.4 Creating, compiling and running a basic MCMC specification

At this point we have initial values for all of the nodes in the model and we have both the original and compiled versions of the model. As a first algorithm to try on our model, let's use NIMBLE's default MCMC. Note that conjugate relationships are detected for all nodes except for `alpha`[2], on which the default sampler is a random walk Metropolis sampler.

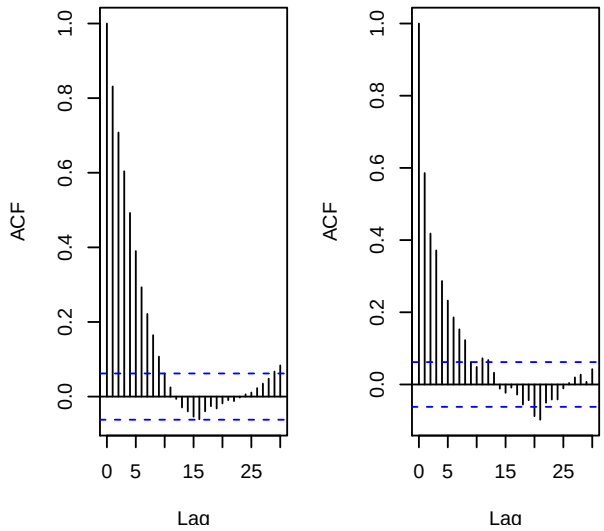
```
pumpSpec <- configureMCMC(pump, print = TRUE)

## [1] RW sampler: alpha,  adaptive: TRUE,  adaptInterval: 200,  scale: 1
## [2] conjugate_dgamma sampler: beta,  dependents_dgamma: theta[1], theta[2], theta[3],
## [3] conjugate_dgamma sampler: theta[1],  dependents_dpois: x[1]
## [4] conjugate_dgamma sampler: theta[2],  dependents_dpois: x[2]
## [5] conjugate_dgamma sampler: theta[3],  dependents_dpois: x[3]
## [6] conjugate_dgamma sampler: theta[4],  dependents_dpois: x[4]
## [7] conjugate_dgamma sampler: theta[5],  dependents_dpois: x[5]
## [8] conjugate_dgamma sampler: theta[6],  dependents_dpois: x[6]
## [9] conjugate_dgamma sampler: theta[7],  dependents_dpois: x[7]
## [10] conjugate_dgamma sampler: theta[8],  dependents_dpois: x[8]
```

[2] We haven't set up conjugate relationships for an exponential yet.

```
## [11] conjugate_dgamma sampler: theta[9],  dependents_dpois: x[9]
## [12] conjugate_dgamma sampler: theta[10],  dependents_dpois: x[10]

pumpSpec$addMonitors(c('alpha', 'beta', 'theta'))

## thin = 1: alpha, beta, theta

pumpMCMC <- buildMCMC(pumpSpec)
CpumpMCMC <- compileNimble(pumpMCMC, project = pump)

niter <- 1000
set.seed(0)
CpumpMCMC$run(niter)

## NULL

samples <- as.matrix(CpumpMCMC$mvSamples)

par(mfrow = c(1, 4), mai = c(.5, .5, .1, .2))
plot(samples[ , 'alpha'], type = 'l', xlab = 'iteration',
     ylab = expression(alpha))
plot(samples[ , 'beta'], type = 'l', xlab = 'iteration',
     ylab = expression(beta))
plot(samples[ , 'alpha'], samples[ , 'beta'], xlab = expression(alpha),
     ylab = expression(beta))
plot(samples[ , 'theta[1]'], type = 'l', xlab = 'iteration',
     ylab = expression(theta[1]))
```



```
acf(samples[, 'alpha']) ## plot autocorrelation of alpha sample
acf(samples[, 'beta'])  ## plot autocorrelation of beta  sample
```



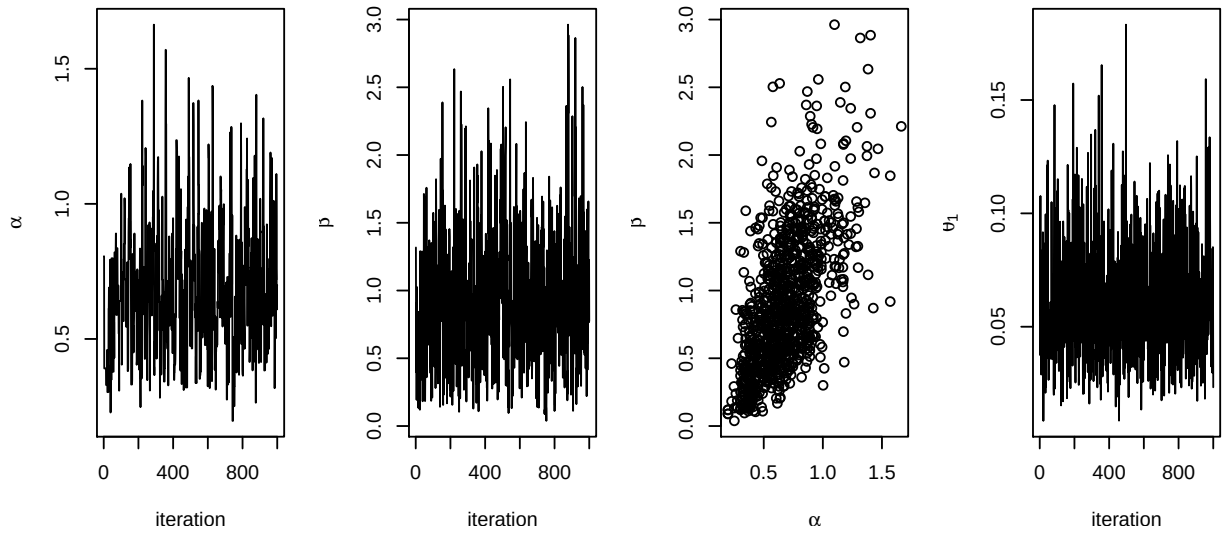
Notice the posterior correlation between `alpha` and `beta`. And a measure of the mixing for each is the autocorrelation for each, shown by the `acf` plots.

2.5 Customizing the MCMC

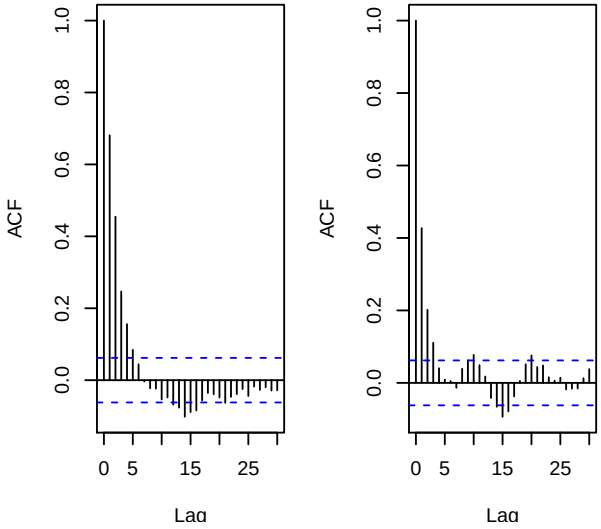
Let's add an adaptive block sampler on `alpha` and `beta` jointly and see if that improves the mixing.

```
pumpSpec$addSampler(target = c('alpha', 'beta'), type = 'RW_block',
                    control = list(adaptInterval = 100))

## [13] RW_block sampler: alpha, beta,  adaptive: TRUE,  adaptScaleOnly: FALSE,  adaptIn

pumpMCMC2 <- buildMCMC(pumpSpec)

# need to reset the nimbleFunctions in order to add the new MCMC
CpumpNewMCMC <- compileNimble(pumpMCMC2, project  = pump,
                              resetFunctions = TRUE)

set.seed(0);
CpumpNewMCMC$run(niter)

## NULL

samplesNew <- as.matrix(CpumpNewMCMC$mvSamples)

par(mfrow = c(1, 4), mai = c(.5, .5, .1, .2))
```

```
plot(samplesNew[ , 'alpha'], type = 'l', xlab = 'iteration',
     ylab = expression(alpha))
plot(samplesNew[ , 'beta'], type = 'l', xlab = 'iteration',
     ylab = expression(beta))
plot(samplesNew[ , 'alpha'], samplesNew[ , 'beta'], xlab = expression(alpha),
     ylab = expression(beta))
plot(samplesNew[ , 'theta[1]'], type = 'l', xlab = 'iteration',
     ylab = expression(theta[1]))
```




```
acf(samplesNew[, 'alpha']) ## plot autocorrelation of alpha sample
acf(samplesNew[, 'beta'])  ## plot autocorrelation of beta  sample
```



We can see that the block sampler has decreased the autocorrelation for both `alpha` and `beta`. Of course these are just short runs, and what we are really interested in is the effective sample size of the MCMC per computation time, but that's not the point of this example.

Once you learn the MCMC system, you can write your own samplers and include them. The entire system is written in nimbleFunctions.

2.6 Running MCEM

NIMBLE is a system for working with algorithms, not just an MCMC engine. So let's try maximizing the marginal likelihood for `alpha` and `beta` using Monte Carlo Expectation Maximization[3].

```
pump2 <- pump$newModel()

nodes <- pump2$getNodeNames(stochOnly = TRUE)

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

pumpMCEM <- buildMCEM(model = pump2, latentNodes = 'theta[1:10]',
                      boxConstraints = box)

pumpMLE <- pumpMCEM()
# Note: buildMCEM returns an R function that contains a
# nimbleFunction rather than a nimble function. That is why
# pumpMCEM() is used instead of pumpMCEM$run().

pumpMLE

##     alpha      beta
## 0.8230659 1.2600147
```

Both estimates are within 0.01 of the values reported by George et al. (1993)[4]. Some discrepancy is to be expected since it is a Monte Carlo algorithm.

2.7 Creating your own functions

Now let's see an example of writing our own algorithm and using it on the model. We'll do something simple: simulating multiple values for a designated set of nodes and calculating every part of the model that depends on them.

Here is our `nimbleFunction`:

[3]Note that for this model, one could analytically integrate over `theta` and then numerically maximize the resulting marginal likelihood.

[4]George, E.I., Makov, U.E. & Smith, A.F.M. 1993. Conjugate likelihood distributions. *Scand. J. Statist.* **20**:147-156. Their numbers were accidentally swapped in Table 2.

```
simNodesMany <- nimbleFunction(
    setup = function(model, nodes) {
        mv <- modelValues(model)
        deps <- model$getDependencies(nodes)
        allNodes <- model$getNodeNames()
    },
    run = function(n = integer()) {
        resize(mv, n)
        for(i in 1:n) {
            simulate(model, nodes)
            calculate(model, deps)
            copy(from = model, nodes = allNodes,
                 to = mv, rowTo = i, logProb = TRUE)
        }
    })

simNodesTheta1to5 <- simNodesMany(pump, 'theta[1:5]')
simNodesTheta6to10 <- simNodesMany(pump, 'theta[6:10]')
```

Here are a few things to notice about the `nimbleFunction`

1. The `setup` function is written in R. It creates relevant information specific to our model for use in the run-time code.
2. The `setup` code creates a `modelValues` object to hold multiple sets of values for variables in the model provided.
3. The `run` function is written in NIMBLE. It carries out the calculations using the information determined once for each set of `model` and `nodes` arguments by the setup code. The run-time code is what will be compiled.
4. The `run` code requires type information about the argument `n`. In this case it is a scalar integer.
5. The for-loop looks just like R, but only sequential integer iteration is allowed.
6. The functions `calculate` and `simulate`, which were introduced above in R, can be used in NIMBLE.
7. The special function `copy` is used here to record values from the model into the modelValues object.
8. Multiple instances, or "specializations", can be made by calling `simNodesMany` with different arguments. Above `simNodesTheta1to5`, has been made by calling `simNodesMany` with the `pump` model and nodes `'theta[1:5]'` as inputs to the `setup` function, while `simNodesTheta6to10` differs by providing `'theta[6:10]'` as an argument. The returned objects are objects of a uniquely generated R reference class with fields (member data) for the results of the `setup` code and a `run` method (member function). Arbitrary other methods can be provided with a `methods` argument, following the syntax of R's `setRefClass` function.

By the way, `simNodesMany` is very similar to a standard `nimbleFunction` provided with nimble, `simNodesMV`.

Now let's execute this `nimbleFunction` in R, before compiling it.

```
set.seed(0)  ## make the calculation repeatable
pump$alpha <- pumpMLE[1]
pump$beta <- pumpMLE[2]
calculate(pump, pump$getDependencies(c('alpha','beta'), determOnly = TRUE))

## [1] 0

saveTheta <- pump$theta
simNodesTheta1to5$run(10)
simNodesTheta1to5$mv[['theta']][1:2]

## [[1]]
##  [1] 1.43717729 1.53093576 1.45028779 0.03716752 0.13310071
##  [6] 1.15835525 0.99001994 0.30737332 0.09461909 0.15720154
##
## [[2]]
##  [1] 0.34222406 3.45822771 0.82805042 0.08796383 0.34440151
##  [6] 1.15835525 0.99001994 0.30737332 0.09461909 0.15720154

simNodesTheta1to5$mv[['logProb_x']][1:2]

## [[1]]
##  [1] -115.767499  -20.856152  -73.444053   -8.258863   -3.570190
##  [6]   -7.429868   -1.000761   -1.453644   -9.840589  -39.096527
##
## [[2]]
##  [1] -19.688263 -50.299758 -37.107538  -2.598331  -1.825300
##  [6]  -7.429868  -1.000761  -1.453644  -9.840589 -39.096527
```

In this code we have initialized the values of `alpha` and `beta` to their MLE and then recorded the `theta` values to use below. Then we have requested 10 simulations from `simNodesTheta1to5`. Shown are the first two simulation results for `theta` and the log probabilities of `x`. Notice that `theta[6:10]` and the corresponding log probabilities for `x[6:10]` are unchanged because the nodes being simulated are only `theta[1:5]`. In R, this function runs slowly.

Finally, let's compile the function and run that version.

```
CsimNodesTheta1to5 <- compileNimble(simNodesTheta1to5,
                                    project  = pump, resetFunctions = TRUE)
Cpump$alpha <- pumpMLE[1]
Cpump$beta <- pumpMLE[2]
calculate(Cpump, Cpump$getDependencies(c('alpha','beta'), determOnly = TRUE))

## [1] 0
```

```
Cpump$theta <- saveTheta

set.seed(0)
CsimNodesTheta1to5$run(10)

## NULL

CsimNodesTheta1to5$mv[['theta']][1:2]

## [[1]]
##  [1] 1.43717729 1.53093576 1.45028779 0.03716752 0.13310071
##  [6] 1.15835525 0.99001994 0.30737332 0.09461909 0.15720154
##
## [[2]]
##  [1] 0.34222406 3.45822771 0.82805042 0.08796383 0.34440151
##  [6] 1.15835525 0.99001994 0.30737332 0.09461909 0.15720154

CsimNodesTheta1to5$mv[['logProb_x']][1:2]

## [[1]]
##  [1] -115.767499  -20.856152  -73.444053   -8.258863   -3.570190
##  [6]   -2.593423   -1.006239   -1.180023   -1.757379   -2.531520
##
## [[2]]
##  [1] -19.688263 -50.299758 -37.107538  -2.598331  -1.825300
##  [6]  -2.593423  -1.006239  -1.180023  -1.757379  -2.531520
```

Given the same initial values and the same random number generator seed, we got identical results for `theta[1:5]` and their dependencies, but it happened much faster.

Chapter 3

More Introduction

Now that we have shown a brief example, we will introduce more about the concepts and design of NIMBLE. Subsequent chapters will go into more detail about working with models and programming in NIMBLE.

One of the most important concepts behind NIMBLE is to allow a combination of high-level processing in R and low-level processing in compiled C++. For example, when we write a Metropolis-Hastings MCMC sampler in the NIMBLE language, the inspection of the model structure related to one node is done in R, and the actual sampler calculations are done in compiled C++. The theme of separating one-time high-level processing and repeated low-level processing will become clearer as we introduce more about NIMBLE's components.

3.1 NIMBLE adopts and extends the BUGS language for specifying models

We adopted the BUGS language, and we have extended it to make it more flexible. The BUGS language originally appeared in WinBUGS, then in OpenBUGS and JAGS. These systems all provide automatically-generated MCMC algorithms, but we have adopted only the language for describing models, not their systems for generating MCMCs. In fact, if you want to use those or other MCMCs in combination with NIMBLE's other algorithms, you can[1]. We adopted BUGS because it has been so successful, with over 30,000 registered users by the time they stopped counting, and with many papers and books that provide BUGS code as a way to document their statistical models. To learn the basics of BUGS, we refer you to the OpenBUGS or JAGS web sites. For the most part, if you have BUGS code, you can try NIMBLE.

NIMBLE takes BUGS code and does several things with it:

1. NIMBLE extracts all the declarations in the BUGS code to create a *model definition*. This includes a directed acyclic graph (DAG) representing the model and functions that can inspect the graph and model relationships. Usually you'll ignore the *model definition* and let NIMBLE's default options take you directly to the next step.

[1] and will be able to do so more thoroughly in the future

2. From the *model definition*, NIMBLE builds a working model in R. This can be used to manipulate variables and operate the model from R. Operating the model includes calculating, simulating, or querying the log probability value of model nodes. These basic capabilities, along with the tools to query model structure, allow one to write programs that use the model and adapt to its structure.
3. From the working model, NIMBLE generates customized C++ code representing the model, compiles the C++, loads it back into R, and provides an R object that interfaces to it. We often call the uncompiled model the "R-model" and the compiled model the "C-model." The C-model can be used identically to the R-model, so code written to use one will work with the other. We use the word "compile" to refer to the entire process of generating C++ code, compiling it and loading it into R.

More about specifying and manipulating models is in Chapter 5-6.

3.2 The NIMBLE language for writing algorithms

NIMBLE provides a language, embedded within and similar in style to R, for writing algorithms that can operate on BUGS models. The algorithms can use NIMBLE's utilities for inspecting the structure of a model, such as determining the dependencies between variables. And the algorithms can control the model, changing values of its variables and controlling execution of its probability calculations or corresponding simulations. Finally, the algorithms can use automatically generated data structures to manage sets of model values and probabilities. In fact, the calculations of the model are themselves constructed as functions in the NIMBLE language, as are the algorithms provided in NIMBLE's algorithm library. This means that you can BUGS with new distributions and new functions written in NIMBLE.

Like the models themselves, functions in the NIMBLE language are turned into C++, which is compiled, loaded, and interfaced to R.

Programming in NIMBLE involves a fundamental distinction between:

1. the steps for an algorithm that need to happen only once, at the beginning, such as inspecting the model; and
2. the steps that need to happen each time a function is called, such as MCMC iterations.

Programming in NIMBLE allows, and indeed requires, these steps to be given separately. When one writes a `nimbleFunction`, each of these parts can be provided. The former, if needed, are given in a *setup function*, and they are executed directly in R, allowing any feature of R to be used. The latter are in one or more *run-time functions*, and they are turned into C++. Run-time code is written in the NIMBLE language, which you can think of as a carefully controlled, small subset of R along with some special functions for handling models and NIMBLE's data structures.

What NIMBLE does with a `nimbleFunction` is similar to what it does with a BUGS model:

1. NIMBLE creates a working R version of the `nimbleFunction`, which you can use with an R-model or a C-model.
2. NIMBLE generates C++ code for the run-time function(s), compiles it, and loads it back into R with an interface nearly identical to the R version of the `nimbleFunction`. As for models, we refer to the uncompiled and compiled versions as R-nimbleFunctions

and C-nimbleFunctions, respectively. In v0.4, the behavior of `nimbleFunction`s is usually very similar, but not identical, between the two versions.

More about writing algorithms is in Chapter 9.

3.3 The NIMBLE algorithm library

In v0.4, the NIMBLE algorithm library is fairly limited. It includes:

1. MCMC with samplers including conjugate, slice, adaptive random walk, and adaptive block. NIMBLE's MCMC system illustrates the flexibility of combining R and C++. An R function inspects the model object and creates an MCMC specification object representing choices of which kind of sampler to use for each node. This MCMC specification can be modified in R, such as adding new samplers for particular nodes, before compiling the algorithm. Since each sampler is written in NIMBLE, you can use its source code or write new samplers to insert into the MCMC. And if you want to build an entire MCMC system differently, you could do that too.
2. A `nimbleFunction` that provides a likelihood function for arbitrary sets of nodes in any model. This can be useful for simple maximum likelihood estimation of non-hierarchical models using R's optimization functions. And it can be useful for other R packages that run algorithms on any likelihood function.
3. A `nimbleFunction` that provides ability to simulate, calculate, or retrieve the summed log probability (density) of many sets of values for arbitrary sets of nodes.
4. A basic Monte Carlo Expectation Maximization (MCEM) algorithm. MCEM has its issues as an algorithm, such as potentially slow convergence to maximum likelihood (i.e. empirical Bayes in this context) estimates, but we chose it as a good illustration of how NIMBLE can be used. Each MCMC step uses NIMBLE's MCMC; the objective function for maximization is another `nimbleFunction`; and the actual maximization is done through R's `optim` function[2].

More about the NIMBLE algorithm library is in Chapter 8.

[2]In the future we plan to provide direct access to R's optimizers from within `nimbleFunction`s

Chapter 4

Getting started

4.1 Requirements to run NIMBLE

You can run NIMBLE on any of the three common operating systems: Linux, Mac, or Windows.

The following are required to run NIMBLE.

1. R, of course.
2. The igraph R package.
3. A working C++ compiler that R can use on your system. There are standard open-source C++ compilers that the R community has already made easy to install. You don't need to know anything about C++ to use NIMBLE.

NIMBLE also uses a couple of C++ libraries that you don't need to install, as they will already be on your system or are provided by NIMBLE.

1. The Eigen C++ library for linear algebra. This comes with NIMBLE, or you can use your own copy.
2. The BLAS and LAPACK numerical libraries. These come with R.

Most fairly recent versions of these requirements should work.

4.2 Installation

Since NIMBLE is an R package, you can install it in the usual way, via `install.packages()` or related mechanisms. We have not yet put in on CRAN, so you'll have to find it at R-nimble.org.

For most installations, you can ignore low-level details. However, there are some options that some users may want to utilize.

4.2.1 Using your own copy of Eigen

NIMBLE uses the Eigen C++ template library for linear algebra. Version 3.2.1 of Eigen is included in the NIMBLE package and that version will be used unless the package's configuration script finds another version on the machine. This works well, and the following is only relevant if you want to use a different (e.g., newer) version.

The configuration script looks in the standard include directories, e.g. `/usr/include` and `/usr/local/include` for the header file `Eigen/Dense`. You can specify a particular location in either of two ways:

1. Set the environment variable `EIGEN_DIR` before installing the R package, e.g., `export EIGEN_DIR=/usr/include/eigen3` in the bash shell.
2. Use `R CMD INSTALL --configure-args='--with-eigen=/path/to/eigen' nimble` or `install.packages("nimble", configure.args = "--with-eigen=/path/to/eigen")`.

In these cases, the directory should be the full path to the directory that contains the Eigen directory, e.g. `/usr/local/include`. It is not the full path to the Eigen directory itself, i.e., NOT `/usr/local/include/Eigen`.

4.2.2 Using libnimble

NIMBLE generates specialized C++ code for user-specified models and nimbleFunctions. This code uses some NIMBLE C++ library classes and functions. By default, on Linux and OS X, the library code is compiled once as a linkable library - *libnimble*. This single instance of the library is then linked with the code for each generated model. Alternatively, one can have the library code recompiled in each model's own dynamically loadable library (DLL). This does repeat the same code across models and so occupies more memory. There may be a marginal speed advantage. This is currently what happens on Windows. One can disable using *libnimble* via the configuration argument `--enable-lib`, e.g.

```
R CMD INSTALL --configure-args='--enable-lib=false' nimble
```

4.2.3 LAPACK and BLAS

NIMBLE also uses BLAS and LAPACK for some of its linear algebra (in particular calculating density values and generating random samples from multivariate distributions). NIMBLE will use the same BLAS and LAPACK installed on your system that R uses. Note that a fast (and where appropriate, threaded) BLAS can greatly increase the speed of linear algebra calculations. See Section A.3.1 of the R Installation and Administration manual for more details on providing a fast BLAS for your R installation.

4.2.4 Problems with Installation

We have tested the installation on the three commonly used platforms – OS X, Linux, Windows 7. We don't anticipate problems with installation, but we want to hear about any and help resolve them. Please post about installation problems to the nimble-users Google group or email `nimble.stats@gmail.com`.

4.3 Installing a C++ compiler for R to use

In addition to needing a C++ compiler to install the package (from source), you also need to have a C++ compiler and the utility *make* at run-time. This is needed during the R session to compile the C++ code that NIMBLE generates for a user's models and algorithms.

4.3.1 OS X

On OS X, you should install Xcode. The command-line tools, which are available as a smaller installation, should be sufficient. This is freely available from the Apple developer site and the App Store.

For the compiler to work correctly for OS X, it is very important that the installed R be matched to the correct OS, i.e. R for snowleopard will attempt to use the incorrect compiler if the user has OS 10.9 or higher.

4.3.2 Linux

On Linux, you can install the GNU compiler suite (*gcc/g++*). You can use the package manager to install pre-built binaries. On Ubuntu, the following command will install or update *make*, *gcc* and *libc*.

```
sudo apt-get install build-essential
```

4.3.3 Windows

On Windows, you should download and install *Rtools.exe* available from http://cran.r-project.org/bin/windows/Rtools/. Select the appropriate executable corresponding to your version of R. (We strongly recommend using the most recent version of R, currently 3.1.0, and hence *Rtools31.exe*). This installer leads you through several "pages". You can accept all of the defaults. It is essential the checkbox for the "R 2.15+ toolchain" (page 4) is enabled in order to have *gcc/g++*, *make*, etc. installed. Also, we recommend that you check the PATH checkbox (page 5). This will ensure that R can locate these commands.

Advanced users may wish to change their default compilers. This can be done by editing the Makevars file, see Writing R Externsions: 1.2.1.

4.4 Customizing Compilation of the NIMBLE-generated Code

For each model or nimbleFunction, the NIMBLE package generates and compiles C++ code. This uses classes and routines available through the NIMBLE run-time library and also the Eigen library. The compilation mechanism uses R's SHLIB functionality and so the regular R configuration in `${R_HOME}/etc${R_ARCH}/Makeconf`. NIMBLE places a *Makevars* file in the directory in which the code is generated and `R CMD SHLIB` uses this file.

In all but specialized cases, the general compilation mechanism will suffice. However, one can customize this. One can specify the location of an alternative *Makevars* (or *Makevars.win*) file to use. That should define the variables `PKG_CPPFLAGS` and `PKG_LIBS`. These should contain, respectively, the pre-processor flag to locate the NIMBLE include directory, and the necessary libraries to link against (and their location as necessary), e.g., *Rlapack* and *Rblas* on Windows, and *libnimble*.

Use of this file allows users to specify additional compilation and linking flags. See the Writing R Extensions manual for more details of how this can be used and what it can contain.

Chapter 5

Building models

NIMBLE aims to be compatible with the original BUGS language and also the version used by the popular JAGS package, as well as to extend the BUGS language. However, at this point, there are some BUGS features not supported by NIMBLE, and there are some extensions that are planned but not implemented.

For readers familiar with BUGS, we begin with an overview of supported features and extensions.

5.1 Overview of supported features and extensions of BUGS and JAGS

5.1.1 Supported features of BUGS and JAGS

1. Stochastic and deterministic[1] node declarations.
2. Most univariate and multivariate distributions
3. Link functions
4. Most mathematical functions
5. "for" loops for iterative declarations.
6. Arrays of nodes up to 4 dimensions.
7. Truncation and censoring as in JAGS using the `T()` notation and `dinterval()`.

5.1.2 NIMBLE's Extensions to BUGS and JAGS

NIMBLE extends the BUGS language in the following ways:

1. User-defined functions and distributions – written as `nimbleFunctions` – can be used in model code. See 5.2.5.
2. Multiple parameterizations for distributions, similar to those in R, can be used.
3. Named parameters for distributions and functions, similar to R function calls, can be used.

[1] NIMBLE calls non-stochastic nodes "deterministic", whereas BUGS calls them "logical". NIMBLE uses "logical" in the way R does, to refer to boolean (TRUE/FALSE) variables.

4. Distribution parameters can be expressions, as in JAGS but not in WinBUGS[2]. Caveat: parameters to *multivariate* distributions (*e.g.,* `dmnorm()`) cannot be expressions.
5. Alternative models can be defined from the same model code by using if-then-else statements that are evaluated when the model is defined.
6. More flexible indexing of vector nodes within larger variables is allowed. For example one can place a multivariate normal vector arbitrarily within a higher-dimensional object, not just in the last index.
7. More general constraints can be declared using `dconstraint()`, which extends the concept of JAGS' `dinterval()`. See Section 5.2.8.
8. Link functions can be used in stochastic, as well as deterministic, declarations.[3]
9. Data values can be reset, and which parts of a model are flagged as data can be reset, allowing one model to be used for different data sets without rebuilding the model each time.

5.1.3 Not-yet-supported features of BUGS and JAGS

In this release, the following are not supported.

1. Stochastic indices (but see 5.2.5 for a description of how you could handle some cases with user-defined distributions).
2. The appearance of the same node on the left-hand side of both a `<-` and a ~ declaration (used in WinBUGS for data assignment for the value of a stochastic node).
3. Like BUGS, NIMBLE generally determines the dimensionality and sizes of variables from the BUGS code. However, when a variable appears with blank indices, such as in `x.sum <- sum(x[])`, NIMBLE currently requires that the dimensions of x be provided.

5.2 Writing models

5.2.1 Declaring stochastic and deterministic nodes

The WinBUGS, OpenBUGS and JAGS manuals are useful resources for writing BUGS models. Here we will just introduce the basics of the BUGS language – and some of NIMBLE's extensions – with a block of code showing a variety of declarations:

```
exampleCode <- nimbleCode({
    ## 1. normal distribution with BUGS parameter order
    x ~ dnorm(a + b * c, tau)
    ## 2. normal distribution with a named parameter
    y ~ dnorm(a + b * c, sd = sigma)
    for(i in 1:N) {
        for(j in 1:M[i]) {
            ## 3. For-loop and nested indexing
```

[2] e.g., `y ~ dnorm(5 + mu, 3 * exp(tau))`

[3] But beware of the possibility of needing to set values for "lifted" nodes created by NIMBLE.

```
            z[i,j] ~ dexp(r[ blockID[i] ])
        }
    }
    ## 4. multivariate distribution with arbitrary indexing
    for(i in 1:3)
        mvx[8:10, i] ~ dmnorm(mvMean[3:5], cov = mvCov[1:3, 1:3, i])
    ## 5. User-provided distribution
    w ~ dMyDistribution(hello = x, world = y)
    ## 6. Simple deterministic node
    d1 <- a + b
    ## 7. Vector deterministic node with matrix multiplication
    d2[] <- A[,] %*% mvMean[1:5]
    ## 8. Deterministic node with user-provided function
    d3 <- foo(x, hooray = y)
})
```

This code does not show a complete model and includes some arbitrary indices (e.g. `mvx[8:10, i]`) to illustrate flexibility. When a variable appears only on the right-hand side, it must be provided in data or constants. Notes on the comment-numbered lines are:

1. `x` follows a normal distribution with mean `a + b*c` and precision `tau` (default BUGS second parameter for `dnorm`).
2. `y` follows a normal distribution with the same mean as `x` but a named standard deviation parameter instead of a precision parameter (sd = 1/sqrt(precision)).
3. `z[i, j]` follows an exponential distribution with parameter `r[blockID[i] ]`. This shows how for-loops can be used for indexing of variables containing multiple nodes. Nested indexing can be used if the nested indices (`blockID`) are provided as constants when the model is defined (via `nimbleModel` or `readBUGSmodel`). Variables that define for-loop indices (`N` and `M`) must be provided as constants.
4. The arbitrary block `mvx[8:10, i]` follows a multivariate normal distribution, with a named covariance matrix instead of BUGS' default of a precision matrix. Like in R, curly braces for for-loop contents are only needed if there is more than one line.
5. `w` follows a user-defined distribution. See 5.2.5.
6. `d1` is a scalar deterministic node that, when calculated, will be set to `a + b`.
7. `d2` is a vector deterministic node using matrix multiplication in R's syntax.
8. `d3` is a deterministic node using a user-provided function. See 5.2.5.

More about indexing

Examples of allowed indexing include:

- `x[i]`
- `x[i:j]`
- `x[i:j,k:l]` and indexing of higher dimensional arrays
- `x[i:j, ]`
- `x[3*i+7]`
- `x[(3*i):(5*i+1)]`

When calling functions such as `mean` and `sum` on a vector variable, the square brackets are required but can have blank indices, e.g. `xbar <- mean(x[])` if `x` is a vector and `xbar <- mean(x[,])` if `x` is a matrix [4].

NIMBLE does not allow multivariate nodes to be indicated without square brackets, which is an incompatibility with JAGS. Therefore a statement like `xbar <- mean(x)` in JAGS must be converted to `xbar <- mean(x[])` for NIMBLE.

Here's an example of indexing in the context of multivariate nodes, showing two ways to do the indexing. The first provides indices, so no `dimensions` argument is needed, while the second omits the indices and provides a `dimensions` argument instead.

```
code <- nimbleCode({
  y[1:K] ~ dmulti(p[1:K], n)
  p[1:K] ~ ddirch(alpha[1:K])
  log(alpha[1:K]) ~ dmnorm(alpha0[1:K], R[1:K, 1:K])
})

K <- 5
model <- nimbleModel(code, constants = list(n = 3, K = K,
                           alpha0 = rep(0, K), R = diag(K)))

## Warning:  NAs were detected in model variables:  log_alpha, logProb_log_alpha,
alpha, p, logProb_p, y, logProb_y

codeAlt <- nimbleCode({
  y[] ~ dmulti(p[], n)
  p[] ~ ddirch(alpha[])
  log(alpha[]) ~ dmnorm(alpha0[], R[ , ])
})

model <- nimbleModel(codeAlt, constants = list(n = 3, K = K,
                           alpha0 = rep(0, K), R = diag(K)),
                  dimensions = list(y = K, p = K, alpha = K))

## Warning:  NAs were detected in model variables:  log_alpha, logProb_log_alpha,
alpha, p, logProb_p, y, logProb_y
```

5.2.2 Available distributions and functions

Distributions

NIMBLE supports most of the distributions allowed in BUGS and JAGS. Table 5.1 lists the distributions currently supported with their default parameterizations, which match those of BUGS. The same distributions are available for writing `nimbleFunctions`, but in that case the default parameterizations match R's when possible (see Chapter 9).

[4] This is a case where the dimension of x must be provided when defining the model.

TODO: WE NEED TO PUT MISSING ARGUMENT NAMES IN THIS TABLE

Table 5.1: Distributions with their default order of parameters. The value of the random variable is denoted by x.

Name	Usage	Density	Lower	Upper
Bernoulli	`dbern(prob = p)` $0 < p < 1$	$p^x(1-p)^{1-x}$	0	1
Beta	`dbeta(shape1 = a, shape2 = b)` $a > 0$, $b > 0$	$\frac{x^{a-1}(1-x)^{b-1}}{\beta(a,b)}$	0	1
Binomial	`dbin(prob = p, size = n)` $0 < p < 1$, $n \in \mathbb{N}^*$	$\binom{n}{x} p^x (1-p)^{n-x}$	0	n
Categorical	`dcat(prob = p)` $p \in (\mathbb{R}^+)^N$	$\frac{p_x}{\sum_i p_i}$	1	N
Chi-square	`dchisq(df = k)` $k > 0$	$\frac{x^{\frac{k}{2}-1}\exp(-x/2)}{2^{\frac{k}{2}}\Gamma(\frac{k}{2})}$	0	
Dirichlet	`ddirch(alpha)` $\alpha_j \geq 0$	$\Gamma(\sum_i \alpha_i)\prod_j \frac{x_j^{\alpha_j - 1}}{\Gamma(\alpha_j)}$	0	
Exponential	`dexp(rate = lambda)` $\lambda > 0$	$\lambda \exp(-\lambda x)$	0	
Gamma	`dgamma(shape = r, rate = λ)` $\lambda > 0$, $r > 0$	$\frac{\lambda^r x^{r-1}\exp(-\lambda x)}{\Gamma(r)}$	0	
Logistic	`dlogis(location = μ, rate = τ)` $\tau > 0$	$\frac{\tau \exp\{(x-\mu)\tau\}}{[1+\exp\{(x-\mu)\tau\}]^2}$		
Log-normal	`dlnorm(mu, tau)` $\tau > 0$	$\left(\frac{\tau}{2\pi}\right)^{\frac{1}{2}} x^{-1} \exp\left\{-\tau(\log(x)-\mu)^2/2\right\}$	0	
Multinomial	`dmulti(prob = b,size = n)` $\sum_j x_j = n$	$n! \prod_j \frac{p_j^{x_j}}{x_j!}$		
Multivariate normal	`dmnorm(mu, prec = Λ)` Λ positive definite	$(2\pi)^{-\frac{d}{2}}\lvert\Lambda\rvert^{\frac{1}{2}} \exp\{-(x-\mu)^T\Lambda(x-\mu)/2\}$		
Negative binomial	`dnegbin(prob = p, size = r)` $0 < p \leq 1$, $r \geq 0$	$\binom{x+r-1}{x} p^r (1-p)^x$	0	
Normal	`dnorm(mu, tau)` $\tau > 0$	$\left(\frac{\tau}{2\pi}\right)^{\frac{1}{2}} \exp\{-\tau(x-\mu)^2/2\}$		
Poisson	`dpois(lambda)` $\lambda > 0$	$\frac{\exp(-\lambda)\lambda^x}{x!}$	0	
Student t	`dt(mu, tau, df = k)` $\tau > 0$, $k > 0$	$\frac{\Gamma(\frac{k+1}{2})}{\Gamma(\frac{k}{2})}\left(\frac{\tau}{k\pi}\right)^{\frac{1}{2}}\left\{1+\frac{\tau(x-\mu)^2}{k}\right\}^{-\frac{(k+1)}{2}}$		
Uniform	`dunif(min = a,max = b)` $a < b$	$\frac{1}{b-a}$	a	b
Weibull	`dweib(shape = v, lambda)` $v > 0$, $\lambda > 0$	$v\lambda x^{v-1}\exp(-\lambda x^v)$	0	
Wishart	`dwish(R, df = k)` R $p \times p$ pos. def., $k \geq p$	$\frac{\lvert x\rvert^{(k-p-1)/2}\lvert R\rvert^{k/2}\exp\{-\mathrm{tr}(Rx/2)\}}{2^{pk/2}\Gamma_p(k/2)}$		

Alternative parameterizations for distributions

NIMBLE provides additional parameterizations listed in Table 5.2. To understand how NIMBLE handles alternative parameterizations, it is useful to distinguish three cases, using the `gamma` distribution as an example:

1. A *canonical* parameterization is used directly for computations. Usually this is the parameterization in the `Rmath` header of R's C implementation of distributions. For `gamma`, this is (shape, scale).
2. The *BUGS* parameterization is the one defined in the original BUGS language. For `gamma`, this is (shape, rate).
3. An *alternative* parameterization is one that must be converted into the *canonical* parameterization. For example, NIMBLE provides a (mean, sd) parameterization and creates nodes to calculate (shape, scale) from (mean, sd). In the case of `gamma`, the *BUGS* parameterization is also an *alternative* parameterization.

Since NIMBLE provides compatibility with existing BUGS and JAGS code, the order of parameters places the *BUGS* parameterization first. For example, the order of parameters for `dgamma` is `dgamma(shape, rate, scale, mean, sd)`. Like R, if parameter names are not given, they are taken in order, so that (shape, rate) is the default. This happens to match R's order of parameters, but it need not. If names are given, they can be given in any order. NIMBLE knows that rate is an alternative to scale and that (mean, sd) are an alternative to (shape, scale or rate).

The file *distsDefs_table.R* in the R directory of the source package is the definitive source for the NIMBLE's distributions and parameterizations.

TODO: WHAT IS THE STATUS OF THE NEXT TWO STATEMENTS?:

We plan to, but do not currently, handle the following distributions: double exponential (Laplace), beta-binomial, Dirichlet-multinomial, F, Pareto, inverse Wishart, and various forms of multivariate t.

We will shortly add the distribution aliases allowed in JAGS: `dbinom`, `dnbinom`, `dweibull`, `ddirich`.

TODO: SHOULD WE COMMENT ON MULTIVARIATE NORMAL USE OF CHOLESKY? E.G. PROVIDE IT EXPLICITLY FOR USERS WHO WANT TO PROVIDE CHOLESKY MATRIX DIRECTLY?

Table 5.2: Alternative distribution parameterizations. The first column indicates the supported parameterizations for distributions given in Table 5.1 with their *BUGS* parameterization. The second column indicates the relationship to the *canonical* parameterization used in NIMBLE. In cases where the the *BUGS* parameterization is not the *canonical* one, the latter is listed in this table with *none* in the second column.

Alternative Parameterization	NIMBLE re-parameterization
`dbeta(mean, sd)`	`dbeta(shape1 = mean^2 * (1 - mean) / sd^2 - mean, shape2 = mean * (1 - mean)^2 / sd^2 + mean - 1)`

Table 5.2: Alternative distribution parameterizations. The first column indicates the supported parameterizations for distributions given in Table 5.1 with their *BUGS* parameterization. The second column indicates the relationship to the *canonical* parameterization used in NIMBLE. In cases where the the *BUGS* parameterization is not the *canonical* one, the latter is listed in this table with *none* in the second column.

Alternative Parameterization	NIMBLE re-parameterization
`dexp(scale)`	`dexp(rate = 1/scale)`
`dgamma(shape, scale)`	*none*
`dgamma(mean, sd)`	`dgamma(shape = mean^2 / sd^2, scale = sd^2 / mean)`
`dweib(shape, scale)`	*none*
`dweib(shape, rate)`	`dweib(shape, scale = 1/rate)`
`dlnorm(mu, sd)`	*none*
`dlogis(location, scale)`	*none*
`dnorm(mu, sd)`	*none*
`dnorm(mu, var)`	`dnorm(mu, sd = sqrt(var))`
`dt(mu, sigma, df)`	*none*
`dmnorm(mu, cov)`	`dnorm(mu, chol = chol(cov), prec_param = FALSE)`
`dmnorm(mu, chol)`	`dnorm(mu, chol, prec_param = FALSE)`
`dwish(S, df)`	`dwish(chol = chol(S), df, scale_param = TRUE)`

5.2.3 Available BUGS language functions

Tables 5.3-5.4 show the available operators and functions. These are also available for `nimbleFunction` programming (see Chapter 9). In fact, BUGS model nodes are implemented as `nimbleFunction`s that are custom-generated from BUGS declarations, so it would be more correct to say that functions and operators available for `nimbleFunction`s are also available for the model declarations.

TODO: CHECK ALL OF THE FOLLOWING STATEMENTS.

For the most part NIMBLE supports the functions used in BUGS and JAGS, with exceptions indicated in the table. Additional functions provided by NIMBLE are also listed. For distribution functions, currently "r" and "d" functions are provided for each distribution, and when called from `nimbleFunctions` only the BUGS parameterization is available in v0.4, and in fact sometimes that is provided differently. See 9.5.18. Currently "r" functions only return one random draw at a time, and the first argument must always be 1. For multivariate distribution functions the `prec_param` or `scale_param` argument must be provided, indicating when a covariance or precision matrix has been given. In a future release we will provide a variety of distribution functions, including density, cumulative distribution and quantile functions, using the same syntax as `dnorm`, `pnorm`, `qnorm`. We will also extend the alternative parameterizations with named parameters to `nimbleFunctions`.

Table 5.3: Functions operating on scalars, many of which can operate on each element (component-wise) of vectors and matrices. Status column indicates if the function is currently provided in NIMBLE.

Usage	Description	Comments	Status	Accepts vector input
`x | y`, `x & y`	logical OR (|) and AND(&)		✓	
`!x`	logical not		✓	
`x > y`, `x >= y`	greater than (and or equal to)		✓	
`x < y`, `x <= y`	less than (and or equal to)		✓	
`x != y`, `x == y`	(not) equals		✓	
`x + y`, `x - y`, `x * y`	component-wise operators	mix of scalar and vector ok	✓	✓
`x / y`,	component-wise division	vector x and scalar y ok	✓	✓
`x^y`, `pow(x, y)`	power	x^y; vector x and scalar y ok	✓	✓
`x %% y`	modulo (remainder)		✓	
`min(x1, x2)`, `max(x1, x2)`	min. (max.) of two scalars		✓	
`exp(x)`	exponential		✓	✓
`log(x)`	natural logarithm		✓	✓
`sqrt(x)`	square root		✓	✓
`abs(x)`	absolute value		✓	✓
`step(x)`	step function at 0	0 if $x < 0$, 1 if $x > 0$	✓	✓
`equals(x, y)`	equality of two scalars	1 if $x == y$, 0 if $x! = y$	✓	
`cube(x)`	third power	x^3	✓	✓
`sin(x)`, `cos(x)`, `tan(x)`	trigonometric functions		✓	✓
`asin(x)`, `acos(x)`, `atan(x)`	inverse trigonometric functions		✓	✓
`asinh(x)`, `acosh(x)`, `atanh(x)`	inv. hyperbolic trig. functions		✓	✓
`logit(x)`	logit	$\log(x/(1-x))$	✓	✓
`ilogit(x)`, `expit(x)`	inverse logit	$\exp(x)/(1+\exp(x))$	✓	✓
`probit(x)`	probit (Gaussian quantile)	$\Phi^{-1}(x)$	✓	✓
`iprobit(x)`, `phi(x)`	inverse probit (Gaussian CDF)	$\Phi(x)$	✓	✓
`cloglog(x)`	complementary log log	$\log(-\log(1-x))$	✓	✓
`icloglog(x)`	inverse complementary log log	$1-\exp(-\exp(x))$	✓	✓
`ceiling(x)`	ceiling function	$\lceil(x)\rceil$	✓	✓
`floor(x)`	floor function	$\lfloor(x)\rfloor$	✓	✓
`round(x)`	round to integer		✓	✓
`trunc(x)`	truncation to integer		✓	✓
`lgamma(x)`, `loggam(x)`	log gamma function	$\log\Gamma(x)$	✓	✓
`log1p(x)`	log of 1 + x	$\log(1+x)$	✓	✓
`lfactorial(x)`, `logfact(x)`	log factorial	$\log x!$	✓	✓
`log1p(x)`	log one-plus	$log(x+1)$	✓	✓
`dDIST(x, PARAMS)`	"d" distribution functions	canonical parameterization	✓	
`rDIST(1, PARAMS)`	"r" distribution functions	canonical parameterization	✓	
`sort(x)`				
`rank(x, s)`				
`ranked(x, s)`				
`order(x)`				

Table 5.4: Functions operating on vectors and matrices. Status column indicates if the function is currently provided in NIMBLE.

Usage	Description	Comments	Status
`inverse(x)`	matrix inverse	`x` symmetric, positive definite	✓
`chol(x)`	matrix Cholesky factorization	`x` symmetric, positive definite	✓
`eigen(x)`	matrix eigendecomposition		
`svd(x)`	matrix singular value decomposition		
`t(x)`	matrix transpose	$x^\top$	✓
`x%*%y`	matrix multiply	xy; x, y conformant	✓
`inprod(x, y)`	dot product	$x^\top y$; `x` and `y` vectors	✓
`logdet(x)`	log matrix determinant	$\log \lvert x \rvert$	✓
`asRow(x)`	convert vector `x` to 1-row matrix	sometimes automatic	✓
`asCol(x)`	convert vector `x` to 1-column matrix	sometimes automatic	✓
`sum(x)`	sum of elements of `x`		✓
`mean(x)`	mean of elements of `x`		✓
`sd(x)`	standard deviation of elements of `x`		✓
`prod(x)`	product of elements of `x`		✓
`min(x)`, `max(x)`	min. (max.) of elements of `x`		✓
`pmin(x, y)`, `pmax(x, y)`	vector of mins (maxs) of elements of `x` and `y`		✓
`interp.lin(x, v1, v2)`	linear interpolation		

5.2.4 Available link functions

NIMBLE allows the link functions listed in Table 5.5.

Table 5.5: Link functions

Link function	Description	Range	Inverse
`cloglog(y) <- x`	Complementary log log	$0 < y < 1$	`y <- icloglog(x)`
`log(y) <- x`	Log	$0 < y$	`y <- exp(x)`
`logit(y) <- x`	Logit	$0 < y < 1$	`y <- expit(x)`
`probit(y) <- x`	Probit	$0 < y < 1$	`y <- iprobit(x)`

Link functions are specified as functions applied to a variable on the left hand side of a BUGS expression. To handle link functions, NIMBLE does some processing that inserts an additional node into the model. For example, the declaration `logit(p[i])` ∼ `dnorm(mu[i],1)`, is equivalent to the follow two declarations:

- `logit_p[i]` ∼ `dnorm(mu[i], 1)`,
- `p[i] <- expit(logit_p[i])`

where `expit` is the inverse of `logit`. When the BUGS expression defines a deterministic node, such as `logit(p) <- b0 + b1*x`, the same operations are performed except that `logit_p` is a deterministic node.

Note that we do not provide an automatic way of initializing the additional node(`logit_p` in this case), which is a parent node of the explicit node (`p[i]`), without explicitly referring to the additional node by the name that NIMBLE generates. For deterministic declarations, this is of little import, but for stochastic declarations, it requires care.

5.2.5 Adding user-defined distributions and functions

As of Version 0.4, NIMBLE now allows you to define your own functions and distributions as `nimbleFunction`s for use in BUGS code. As a result, NIMBLE frees you from being constrained to the functions and distributions listed in Section **??**. For example, instead of setting up a Dirichlet prior with multinomial data and needing to use MCMC, one could recognize that this results in a Dirichlet-multinomial distribution and provide that as a user-defined distribution instead.

Further, while NIMBLE at the moment does not allow the use of random indices, such as is common in clustering contexts, you may be able to analytically integrate over the random indices, resulting in a mixture distribution that you could implement as a user-defined distribution. For example, one could implement the *dnormmix* distribution provided in JAGS as a user-defined distribution in NIMBLE.

User-defined functions

To provide a new function for use in BUGS code, simply create a `nimbleFunction` that has no `setup` code. Then use it in your BUGS code. That's it.

Writing `nimbleFunction`s requires that you declare the dimensionality of arguments and the returned object (). Make sure that the dimensionality specified in your `nimbleFunction` matches how you use it in BUGS code. For example, if you define scalar parameters in your BUGS code you will want to define `nimbleFunctions` that take scalar arguments. Here is an example that returns twice its input argument:

```
timesTwo <- nimbleFunction(
    run = function(x = double(0)) {
        returnType(double(0))
        return(2*x)
    })

code <- BUGScode({
    for(i in 1:3) {
        mu[i] ~ dnorm(0, 1)
        mu_times_two[i] <- timesTwo(mu[i])
    }
})
```

The `x = double(0)` argument and `returnType(double(0))` establish that the input and output will both be 0-dimensional (scalar) numbers.

You can define `nimbleFunctions` that take inputs and outputs with more dimensions. Here is an example that takes a vector (1-dimensional) as input and returns a vector with twice the input values:

```
vectorTimesTwo  <- nimbleFunction(
    run = function(x = double(1)) {
        returnType(double(1))
        return(2*x)
    }
    )
code <- BUGScode({
    for(i in 1:3) {
        mu[i] ~ dnorm(0, 1)
    }
    mu_times_two[1:3] <- vectorTimesTwo(mu[1:3])
})
```

There is a subtle difference between the `mu_times_two` variables in the two examples. In the first example, there are individual nodes for each `mu_times_two[i]`. In the second example, there is a single multivariate node, `mu_times_two[1:3]`. Each implementation could be more efficient for different needs. For example, suppose an algorithm modifies the value of `mu[2]` and then updates nodes that depend on it. In the first example, `mu_times_two[2]` would be updated. In the second example `mu_times_two[1:3]` would be updated because it is a single, vector node.

At present you cannot provide a scalar argument where a `nimbleFunction` expects a vector; unlike in R, scalars are not simply vectors of length 1.

User-defined distributions

To provide a user-defined distribution, you need to, at a minimum, do the following:

1. define density ("d") and simulation ("r") `nimbleFunctions` without setup code for your distribution,
2. register the distribution using `registerDistributions`, and
3. use your distribution in BUGS code.

You can optionally provide distribution ("p") and quantile ("q") functions, which will allow truncation to be applied to a user-defined distribution. You can also provide a list of alternative parameterizations.

Here is an extended example of providing a univariate exponential distribution (although this is already provided by NIMBLE) and a multivariate Dirichlet-multinomial distribution.

```
dmyexp <- nimbleFunction(
    run = function(x = double(0), rate = double(0, default = 1),
        log = integer(0, default = 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, default = 1)) {
        returnType(double(0))
        if(n != 1) print("rmyexp only allows n = 1; using n = 1.")
        dev <- runif(1, 0, 1)
        return(-log(1-dev) / rate)
    })

pmyexp <- nimbleFunction(
    run = function(q = double(0), rate = double(0, default = 1),
        lower.tail = integer(0, default = 1),
        log.p = integer(0, default = 0)) {
        returnType(double(0))
        if(!lower.tail) {
            logp <- -rate * q
            if(log.p) return(logp)
            else return(exp(logp))
        } else {
            p <- 1 - exp(-rate * q)
            if(!log.p) return(p)
            else return(log(p))
        }
    })

qmyexp <- nimbleFunction(
    run = function(p = double(0), rate = double(0, default = 1),
        lower.tail = integer(0, default = 1),
        log.p = integer(0, default = 0)) {
        returnType(double(0))
        if(log.p) p <- exp(p)
        if(!lower.tail) p <- 1 - p
        return(-log(1 - p) / rate)
    })

ddirchmulti <- nimbleFunction(
    run = function(x = double(1), alpha = double(1), size = double(0),
        log = integer(0)) {
        returnType(double(0))
        logProb <- 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(1))
        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(
    dmyexp = list(
        BUGSdist = "dmyexp(rate, scale)",
        Rdist = "dmyexp(rate = 1/scale)",
        altParams = c("scale = 1/rate", "mean = 1/rate"),
        pqAvail = TRUE,
        range = c(0, Inf)
        ),
    ddirchmulti = list(
        BUGSdist = "ddirchmulti(alpha, size)",
        types = c('value = double(1)', 'alpha = double(1)'),
        )
    ))

code <- BUGScode({
     y[1:K] ~ ddirchmulti(alpha[1:K], n)
     for(i in 1:K) {
         alpha[i] ~ T(dmyexp(scale = 3), 0, 100)
      }
     })
```

When you write your user-defined distribution-related functions, you should write them to take as input the parameters for a single parameterization, which will be the standard parameterization that NIMBLE will use. If you would like to allow for multiple parameterizations, you can do this via the `Rdist` element of the list provided to `registerDistributions` as illustrated. If you provide CDF("p") and inverse CDF (quantile, i.e. "q") functions, be sure to specify `pqAvail = TRUE` when you call `registerDistributions`.

Here are more details on the requirements for distribution-related `nimbleFunctions`, which follow R's conventions:

- Your distribution-related functions must have names that begin with "d", "r", "p" and "q". The name of the distribution must not be identical to any of the NIMBLE-provided distributions.

- The function name in the `BUGSdist` entry in the list provided to `registerDistributions` will be the name you can use in BUGS code.
- The name of your `nimbleFunction`s must match the function name in the `Rdist` entry. If missing, the `Rdist` entry defaults to be the same as the `BUGSdist` entry.
- All simulation ("r") functions must take `n` as their first argument. Note that you may simply have your function only handle `n=1` and return an warning for other values of `n`.
- Your distribution-related functions must take as arguments the default parameters, starting as the second argument and in the order used in the parameterizations in the `Rdist` argument to `registerDistributions` or the `BUGSdist` argument if there are no alternative parameterizations. NIMBLE uses doubles for numerical calculations, so we suggest simply using doubles in general, even for integer-valued parameters or values of random variables.
- All density functions must have as their last argument `log` and implement return of the log probability density. NIMBLE algorithms typically use only `log = 1`, but we recommend you implement the `log = 0` case for completeness.
- All distribution and quantile functions must have their last two arguments be (in order), `lower.tail` and `log.p`. These functions must work for `lower.tail = 1` (i.e., TRUE) and `log.p = 0` (i.e., FALSE), as these are the inputs we use when working with truncated distributions. It is your choice whether you implement the necessary calculations for other combinations of these inputs, but again we recommend doing so for completeness.
- Define the `nimbleFunctions` in R's global environment. Don't expect R's standard scoping to work. [5].

Further details on using `registerDistributions` can be found by `help(registerDistributions)`. NIMBLE uses the same list format as `registerDistributions` to define its distributions. This list can be found in the `R/distributions_inputList.R` in the package source code `R` directory.

5.2.6 Data and constants

NIMBLE makes a distinction between data and constants that would both be considered "data" in BUGS and JAGS.

- *Constants* can never be changed and must be provided when a model is defined. For example, a vector of known index values, such as for block indices, helps define the model graph itself and must be provided as constants. Variables used in the index ranges of for-loops must also be provided as constants.
- *Data* is a label for the role a node plays in the model. Nodes marked as data will by default be protected from any functions that would simulate over their values (see `simulate` in 9), but it is possible to over-ride that default or to change their values by direct assignment. This allows an algorithm to be applied to many data sets in the same model without re-creating the model each time. It also allows simulation of data

[5] NIMBLE can't use R's standard scoping because it doesn't work for R reference classes, and `nimbleFunction`s are implemented as custom-generated reference classes.

in a model. Data must be provided when an instance of a model is created from the model definition, although they can also be provided earlier when a model is defined.

We encourage users to distinguish between data and constants when building a model via `nimbleModel()`. However, for compatibility with BUGS and JAGS, NIMBLE allows both to provided in the the `constants` argument to `nimbleModel()`, in which case NIMBLE determines which are which, based on which variables appear on the left-hand side of BUGS declarations.

THIS IS CONFUSING: One can also provide variables appearing only on the right-hand side of BUGS declarations (e.g., covariates/predictors) via the `data` argument to `nimbleModel()` and these will appear as variables in the model, but will not be nodes in the model. A user can change these values via direct assignment if desired.

Missing data values

Sometimes one needs a model variable to have a mix of data and non-data, often due to missing data values. In NIMBLE, when data values are provided, any nodes with `NA` values will *not* be labeled as data. The result will be that nodes with non-`NA` values will be flagged as data nodes, while nodes with `NA` values will not. A node following a multivariate distribution must be either entirely observed or entirely missing.

Here's an example of running an MCMC on the *pump* model, with two of the observations taken to be missing. Some of the steps in this example are documented more below. NIMBLE's default MCMC specification will treat the missing values as unknowns to be sampled, as can be seen in the MCMC output here.

```
pumpMiss <- pump$newModel()
pumpMiss$resetData()
pumpDataNew <- pumpData
pumpDataNew$x[c(1, 3)] <- NA
pumpMiss$setData(pumpDataNew)

pumpMissSpec <- configureMCMC(pumpMiss)
pumpMissSpec$addMonitors(c('x', 'alpha', 'beta', 'theta'))

pumpMissMCMC <- buildMCMC(pumpMissSpec)
Cobj <- compileNimble(pumpMiss, pumpMissMCMC)

niter <- 1000
set.seed(0)
Cobj$pumpMissMCMC$run(niter)
samples <- as.matrix(Cobj$pumpMissMCMC$mvSamples)

samples[1:5, 13:17]
```

TODO: IS THIS REDUNDANT: Missing values may also occur in variables appearing on the right-hand side of BUGS declarations. Values for such variables should be passed in

via the `data` argument to `nimbleModel`, with NA for the missing values. In some contexts, one would want to specify (prior) distributions for the elements with missing values.

5.2.7 Defining alternative models with the same code

Avoiding code duplication is a basic principle of good programming. In BUGS and JAGS, if one wants to consider one model that is slightly different from another, one needs to create completely different model code for each one. At best this can lead to excess code. At worst it can lead to errors.

In NIMBLE, one can use definition-time if-then-else statements to create different models from the same code. As a simple example, say we have a linear regression model and want to consider including or omitting an explanatory variable:

```
regressionCode <- nimbleCode({
    intercept ~ dnorm(0, sd = 1000)
    slope1 ~ dnorm(0, sd = 1000)
    if(includeX2) {
        slope2 ~ dnorm(0, sd = 1000)
        for(i in 1:N)
            predictedY[i] <- intercept + slope1 * x1[i] + slope2 * x2[i]
    } else {
        for(i in 1:N) predictedY[i] <- intercept + slope1 * x1[i]
    }
    sigmaY ~ dunif(0, 100)
    for(i in 1:N) Y[i] ~ dnorm(predictedY[i], sigmaY)
})

includeX2 <- FALSE
modelWithoutX2 <- nimbleModel(regressionCode, constants = list(N = 30), check=FALSE)
modelWithoutX2$getVarNames()

## [1] "intercept"
## [2] "slope1"
## [3] "predictedY"
## [4] "sigmaY"
## [5] "lifted_d1_over_sqrt_oPsigmaY_cP"
## [6] "Y"
## [7] "x1"

includeX2 <- TRUE
modelWithX2 <- nimbleModel(regressionCode, constants = list(N = 30), check = FALSE)
modelWithX2$getVarNames()

## [1] "intercept"
## [2] "slope1"
```

```
## [3] "slope2"
## [4] "predictedY"
## [5] "sigmaY"
## [6] "lifted_d1_over_sqrt_oPsigmaY_cP"
## [7] "Y"
## [8] "x1"
## [9] "x2"
```

5.2.8 Truncation, censoring, and constraints

NIMBLE provides three ways to declare boundaries on the value of a variable, each for different situations. We introduce these and comment on their relationships to related features of JAGS and WinBUGS/OpenBUGS. The three methods are:

Truncation

`x ~ dnorm(0, sd = 10) T(0, a)` or, equivalently, `x ~ T(dnorm(0, sd = 10), 0, a)` declares that `x` follows a normal distribution between 0 and `a`. Either boundary may be omitted or may be another node, such as `a` in this example. The first syntax is compatible with JAGS, but in NIMBLE it can only be used when reading code from a text file. When writing model code in R, the second version must be used.

Truncation means the possible values of `x` are limited a priori, hence the probability density of `x` must be normalized. In this example it would be the normal probability density divided by its integral from 0 to `a`[6]. Like JAGS, NIMBLE also provides `I` as a synonym for `T` to accommodate older BUGS code, but `T` is preferred because it disambiguates multiple usages of `I` in WinBUGS.

Note that in NIMBLE, `mu ~ dfoo(theta) T(L, U)` restricts X to lie in $(L, U]$ whereas in JAGS it restricts X to lie in $[L, U]$. This distinction makes no difference for continuous random variables.

TODO: NEED TO DISCUSS THIS.

Censoring

Censoring refers to the situation where one datum gives the lower or upper bound on an unobserved random variable. This is common in survival time studies, when we may know at least how long some patients lived (right-censoring) but not their ages of death, so their age of death is "censored". NIMBLE adopts JAGS syntax for censoring, as follows (using right-censoring as an example):

```
is.censored[i] ~ dinterval(t[i], c[i])
t[i] ~ dweib(r, mu[i])
```

[6] If you have a model object `model`, you can see exactly the calculation used by typing `model$nodes[['x']]$calculate`

where `is.censored[i]` should be given as data with a value of 1 if `t[i]` is right-censored (`t[i]` $>$ `c[i]`) and 0 if it is observed. The data vector for `t` should have `NA` (indicating missing data) for any censored `t[i]` entries. (As a result, these nodes will be sampled in an MCMC.) The data vector for `c` should give the censoring times corresponding to censored entries and `0` for uncensored entries (assuming `t[i]` $>$ `0`).

Censoring differs from truncation because in censoring an observation gives an upper or lower bound on a random variable that could have taken any value, while in truncation we know a priori that a datum could not have occurred outside the truncation range.

The `dinterval` is not really a distribution but rather a trick: in the above example it gives a "probability" of 1 if `t[i]` $>$ `c[i]` and 0 otherwise. This means that `t[i]` $\leq$ `c[i]` is treated as impossible. More generally, `is.censored[i]` $\sim$ `dinterval(t[i], c[i,])` is defined such that for a vector of increasing cutpoints, `c[i,]`, `t[i]` is enforced to fall within the `is.censored[i]`-th cutpoint interval. This is done by setting data `is.censored[i]` as follows:

`is.censored[i]` = `0` if `t[i]` $\leq$ `c[i, 1]`
`is.censored[i]` = `m` if `c[i, m]` $<$ `t[i]` $\leq$ `c[i, m+1]` for $1 <= m <= M$
`is.censored[i]` = `M` if `c[i, M]` $<$ `t[i]`.

(The `i` index is provided only for consistency with the previous example.) The most common usage of `dinterval` will be for left- and right-censored data, in which case `c[i,]` will be a single value (and typically given as simply `c[i]`), and for interval-censored data, in which case `c[i,]` will be a vector of two values. For left-censoring, one can use `x[i]` $\sim$ `dinterval(t[i], c[i])` with `x[i]` set to 0 or `x[i]` $\sim$ `dinterval(c[i], t[i])` with `x[i]` set to 1.

Nodes following a `dinterval` distribution should normally be set as data with known values. Otherwise, the node may simulated during some algorithm's (e.g. MCMC's) initialization and thereby establish a permanent, perhaps unintended, constraint.

Constraints and ordering

NIMBLE provides a more general way to enforce constraints using `dconstraint(condition)`. For example, we could specify that the sum of `mu1` and `mu2` must be positive like this:

```
mu1 ~ dnorm(0, 1)
mu2 ~ dnorm(0, 1)
constraint.data ~ dconstraint( mu1 + mu2 > 0 )
```

with data `constraint.data` set (as data) to 1. Formally, `dconstraint(condition)` is a probability distribution on $\{0, 1\}$ such that $P(1) = 1$ if `constraint` is `TRUE` and $P(0) = 1$ if `constraint` is `FALSE`. Like `dinterval`, `dconstraint` results in distributions that are not normalized (e.g. for (`mu1`, `mu2`)), which makes most sense if the constraint is observed rather than established a priori. Nodes following `dconstraint` should be provided as data for the same reason of avoiding unintended initialization described above for `dinterval`.

Ordering To specify an ordering of parameters, such as $\alpha_1 <= \alpha_2 <= \alpha_3$ one can use `dconstraint` as follows:

```
const ~ dconstraint( alpha1 <= alpha2 & alpha2 <= alpha3 )
```

Note that unlike in BUGS, one cannot specify prior ordering using syntax such as

```
alpha[1] ~ dnorm(0, 1) I(, alpha[2])
alpha[2] ~ dnorm(0, 1) I(alpha[1], alpha[3])
alpha[3] ~ dnorm(0, 1) I(alpha[2], )
```

as this is does not represent a directed acyclic graph. CHRIS, WOULDN'T THIS WORK WITH `alpha[1]` $\sim$ `dnorm(0, 1)`? DOES BUGS REALLY ALLOW THIS NON-DAG?

Also note that specifying prior ordering using `T(,)` can result in possibly unexpected results. For example:

```
alpha1 ~ dnorm(0, 1)
alpha2 ~ dnorm(0, 1) T(alpha1, )
alpha3 ~ dnorm(0, 1) T(alpha2, )
```

will enforce `alpha1` $\leq$ `alpha2` $\leq$ `alpha3`, but it does not treat the three parameters symmetrically. Instead it puts a marginal prior on `alpha1` that is standard normal and then constrains `alpha2` and `alpha3` to follow truncated normal distributions. This is not equivalent to a symmetric prior on the three `alpha`s that assigns 0 probability density when values are not in order.

Of course, in many cases, parameterizing the model so that the constraints are automatically respected may be a better strategy than using `dconstraint`. If constraints are used, it will generally be wise to ensure the model is initialized with values that satisfy them.

NIMBLE does not support the JAGS `dsort` syntax.

5.2.9 Understanding lifted nodes

In some cases, NIMBLE introduces new nodes into the model that were not specified in the BUGS code for the model, such as the `lifted_d1_over_beta` node in the introductory example. For this reason, it is important that programs written to adapt to different model structures use NIMBLE's systems for querying the model graph. For example, a call to `pump$getDependencies("beta")` will correctly include `lifted_d1_over_beta` in the results. If one skips this step and assumes the nodes are only those that appear in the BUGS code, one may not get correct results.

It can be helpful to know the situations in which lifted nodes are generated. These include:

- When distribution parameters are expressions, NIMBLE creates a new deterministic node that contains the expression for a given parameter. The node is then a direct descendant of the new deterministic node. This is an optional feature, but it is currently enabled in all cases.

- As discussed in 5.2.4 the use of link functions causes new nodes to be introduced. In particular, this can cause headaches in terms of initializing values in stochastic declarations with link functions.
- Use of alternative parameterizations of distributions. For example when a user provides the precision of a normal distribution as `tau`, NIMBLE creates a new node `sd <- 1/sqrt(tau)` and uses `sd` as a parameter in the normal distribution. If many nodes use the same `tau`, only one new `sd` node will be created, so the computation `1/sqrt(tau)` will not be repeated redundantly. More about NIMBLE's parameterizations is described below.

5.3 Creating model objects

NIMBLE provides two functions for creating model object: `nimbleModel` and `readBUGSmodel`. It is also possible to use a model definition object to create a model object directly.

Here we describe in detail two ways to provide a BUGS model for use by NIMBLE. The first, The second, `readBUGSmodel` provides compatibility with BUGS file formats for models, variables, data, and initial values for MCMC.

5.3.1 Using `nimbleModel()` to specify a model

`nimbleModel` is the primary way to create models and was illustrated in Chapter 2. The R help page (`?nimbleModel`) provides more details on `nimbleModel` arguments.

5.3.2 Specifying a model from standard `R2WinBUGS` and `rjags` input files

Users with BUGS and JAGS experience may have files set up in standard formats for the `bugs()` and `jags()` functions in the `R2WinBUGS` and `R2jags` packages. `readBUGSmodel()` can read in the model, data/constant values and initial values in those formats. It can also take information directly from R objects somewhat more flexibly than `nimbleModel()`, specifically allowing inputs set up similarly to those for `R2WinBUGS` and `rjags`. In either case, after processing the inputs, it calls `nimbleModel()`.

Please see `help(readBUGSmodel)` for argument details.

TODO: CHRIS IT IS UNCLEAR IF THIS IS DIFFERENT OR THE SAME AS R2WINBUGS AND/OR RJAGS. If the `data` and `inits` are provided as files, the file should contain R code that creates objects analogous to what would populate the list if a list were provided instead. Please see the JAGS manual examples or the *classic_bugs* directory in the NIMBLE package for example syntax. NIMBLE does not handle formatting such as in some of the original BUGS examples in which data was indicated with syntax such as `data x in 'x.txt'`. Only a single set of initial values can specified in creating a model.

Example using `readBUGSmodel`

Let's create a model for the *pump* example from BUGS.

```
pumpDir <- system.file('classic-bugs', 'vol1', 'pump', package = 'nimble')
pumpModel <- readBUGSmodel('pump.bug', data = 'pump-data.R',
                           inits = 'pump-init.R', dir = pumpDir)

## Detected x as data within 'constants'.
## Adding x as data for building model.
```

Note that `readBUGSmodel()` allows one to include `var` and `data` blocks in the model file as in some of the BUGS examples (such as `inhaler`). The `data` block pre-computes constant and data values. NIMBLE by and large does not need the information given in a `var` block but occasionally this is used to determine dimensionality, such as in the case of syntax like `xbar <- mean(x[])` where `x` is a variable that appears only on the right-hand side of BUGS expressions.

Providing data via setData

Whereas the *constants* are a property of the *model definition* – since they may help determine the model structure itself – *data* nodes can be different in different copies of the model generated from the same *model definition*. For this reason, *data* is not required to be provided when the model code is processed. It can be provided later via the model member function `setData`. e.g., `pump$setData(pumpData)`, where `pumpData` is a named list of data values.

`setData` does two things: it sets the values of the data nodes, and it flags those nodes as containing data. `nimbleFunction` programmers can then use that information to control whether an algorithm should over-write data or not. For example, NIMBLE's `simulate()` functions by default do not overwrite data values but can be told to do so. Values of data variables can be replaced, and the indication of which nodes should be treated as data can be reset by using the `resetData` method, e.g. `pump$resetData()`.

Making multiple instances from the same model definition

Sometimes it is useful to have more than one copy of the same model. For example, `nimbleFunctions` are often bound to a particular model as a result of `setup` code. A user could build multiple algorithms to use the same model instance, or they may want each algorithm to have its own instance of the model.

There are two ways to create new instances of a model, shown in this example:

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

## Return the model definition only, not a built model
simpleModelDefinition <- nimbleModel(simpleCode, constants = list(N = 10),
                                     returnDef = TRUE)
## Make one instance of the model
simpleModelCopy1 <- simpleModelDefinition$newModel()
```

```
## Warning:  NAs were detected in model variables:  x, logProb_x

## Make another instance from the same definition
simpleModelCopy2 <- simpleModelDefinition$newModel()

## Warning:  NAs were detected in model variables:  x, logProb_x

## Ask simpleModelCopy2 for another copy of itself
simpleModelCopy3 <- simpleModelCopy2$newModel()

## Warning:  NAs were detected in model variables:  x, logProb_x
```

Each copy of the model can have different nodes flagged as data and different values in any nodes. They cannot have different values of `N` because that is a constant, and it must be a constant because it helps define the model.

Chapter 6

Using NIMBLE models from R

6.1 Some basic concepts and terminology

Before going further, we need some basic concepts and terminology to be able to speak about NIMBLE clearly.

Say we have the following BUGS code

```
mc <- nimbleCode({
    a ~ dnorm(0, 0.001)
    for(i in 1:5) {
        y[i] ~ dnorm(a, 0.1)
        for(j in 1:3)
            z[i,j] ~ dnorm(y[i], sd = 0.1)
    }
    y.squared[1:5] <- y[1:5]^2
})

model <- nimbleModel(mc, data = list(z = matrix(rnorm(15), nrow = 5)))

## Warning:  NAs were detected in model variables:  a, logProb_a, y, logProb_y,
logProb_z, y.squared
```

In NIMBLE terminology:

- The *variables* of this model are `a`, `y`, `z`, and `y.squared`.
- The *nodes* of this model are `a`, `y[1]` ... `y[5]`, `z[1,1]` ... `z[5, 3]`, and `y.squared[1:5]`. In graph terminology, nodes are vertices in the model graph.
- the *node functions* of this model are `a ~ dnorm(0, 0.001)`, `y[i] ~ dnorm(a, 0.1)`, `z[i,j] ~ dnorm(y[i], sd = 0.1)`, and `y.squared[1:5] <- y[1:5]^2`. Each node has a corresponding node function. Sometimes the distinction between nodes and node functions is important, but when it is not important we may refer to both simply as *nodes*.
- The *scalar elements* of this model include all the scalar nodes as well as the scalar elements `y.squared[1]` ... `y.squared[5]` of the multivariate node `y.squared[1:5]`.

6.2 Accessing variables

Model variables can be accessed and set just as in R using `$` and `[[ ]]`. For example

```
model$a <- 5
model$a

## [1] 5

model[['a']]

## [1] 5

model$y[2:4] <- rnorm(3)
model$y

## [1]         NA -0.6545846  1.7672873  0.7167075         NA

model[['y']][c(1, 5)] <- rnorm(2)
model$y

## [1]  0.9101742 -0.6545846  1.7672873  0.7167075  0.3841854

model$z[1,]

## [1]  0.2670988  1.5868335 -0.4734006
```

6.2.1 Accessing log probabilities via logProb variables

For each variable that contains at least one stochastic node, NIMBLE generates a model variable with the prefix "logProb_". When the stochastic node is scalar, the `logProb` variable will have the same size. For example:

```
model$logProb_y

## [1] NA NA NA NA NA

calculate(model, 'y')

## [1] -15.29134

model$logProb_y

## [1] -2.906565 -3.668947 -2.592753 -2.987561 -3.135518
```

Creation of `logProb` variables for stochastic multivariate nodes is trickier, because they can represent an arbitrary block of a larger variable. In general NIMBLE records the logProb values using the lowest possible indices. For example, if `x[5:10, 15:20]` follows a Wishart distribution, its log probability (density) value will be stored in `logProb_x[5, 15]`. When possible, NIMBLE will reduce the dimensions of the corresponding logProb variable. For example, in

```
for(i in 1:10) x[i,] ~ dmnorm(mu[], prec[,])
```

`x` may be 10×20 (dimensions must be provided), but `logProb_x` will be 10×1. For the most part you do not need to worry about how NIMBLE is storing the log probability values, because you can always get them using `getLogProb`.

6.3 Accessing nodes

While nodes that are part of a variable can be accessed as above, each node also has its own name that be used to access it directly. For example, `y[2]` has the name "y[2]" and can be accessed by that name as follows:

```
model[['y[2]']]

## [1] -0.6545846

model[['y[2]']] <- -5
model$y

## [1]  0.9101742 -5.0000000  1.7672873  0.7167075  0.3841854

model[['z[2, 3]']]

## [1] -0.6203667

model[['z[2:4, 1:2]']][1, 2]

## [1] 0.5584864

model$z[2, 2]

## [1] 0.5584864
```

Notice that node names can include index blocks, such as `model[['z[2:4, 1:2]']]`, and these are not strictly required to correspond to actual nodes. Such blocks can be subsequently sub-indexed in the regular R manner.

6.3.1 How nodes are named

Every node has a name that is a character string including its indices, with a space after every comma. For example, `X[1, 2, 3]` has the name "X[1, 2, 3]". Nodes following multivariate distributions have names that include their index blocks. For example, a multivariate node for `X[6:10, 3]` has the name "X[6:10, 3]".

The definitive source for node names in a model is `getNodeNames()`, described below. For example

```
multiVarCode <- nimbleCode({
    X[1, 1:5] ~ dmnorm(mu[], cov[,])
    X[6:10, 3] ~ dmnorm(mu[], cov[,])
})

multiVarModel <- nimbleModel(multiVarCode, dimensions = list(mu = 5, cov = c(5,5)))

## Error in chol.default(model$cov[1:5, 1:5]):  the leading minor of order 1 is
not positive definite

multiVarModel$getNodeNames()

## Error in eval(expr, envir, enclos):  object 'multiVarModel' not found
```

You can see one lifted node for the Cholesky decomposition of `cov`, and the two multivariate normal nodes.

In the event you need to ensure that a name is formatted correctly, you can use R's `parse()` and `deparse()` functions. For example, to get the spaces correctly inserted into "X[1,1:5]":

```
deparse(parse(text = "X[1,1:5]", keep.source = FALSE)[[1]])

## [1] "X[1, 1:5]"
```

The `keep.source = FALSE` makes `parse()` more efficient.

6.3.2 Why use node names?

Syntax like `pump[["x[2, 3]"]]` may seem strange at first, because the natural habit of an R user would be `pump[["x"]][2,3]` . To see its utility, consider the example of writing the nimbleFunction given in 2.7. By giving every scalar node a name, even if it is part of a multivariate variable, one can write functions in R or NIMBLE that access any single node by a name, regardless of the dimensionality of the variable in which it is embedded. This is particularly useful for NIMBLE, which resolves how to access a particular node during the compilation process.

6.4 calculate(), simulate(), and getLogProb()

The four basic ways to operate a model are to calculate nodes, simulate into nodes, get the log probabilities (or probability densities) that have already been calculated, and the log probability of a new value to that of an old value. In more detail:

calculate For a stochastic node, `calculate` determines the log probability value, stores it in the appropriate `logProb` variable, and returns it. For a deterministic node, `calculate` executes the deterministic calculation and returns 0.

simulate For a stochastic node, `simulate` generates a random draw. For deterministic nodes, `simulate` is equivalent to `calculate` without returning 0. `simulate` always returns `NULL` (or `void` in C++).

getLogProb `getLogProb` simply returns the most recently calculated log probability value, or 0 for a deterministic node.

calculateDiff `calculateDiff` is identical to `calculate`, but it returns the new log probability value minus the one that was previously stored.

There are two ways to access `calculate`, `calculateDiff`, `simulate`, and `getLogProb`. The primary way is via the functions with those names, which can use arbitrary collections of nodes. In that case, `calculate` and `getLogProb` return the sum of the log probabilities from each node, while `calculateDiff` returns the sum of the new values minus the old values. The other way is to directly access the corresponding function for each node in a model. Normally you'll use the first way, but we'll show you both.

6.4.1 For arbitrary collections of nodes

```
model$y

## [1]  0.9101742 -5.0000000  1.7672873  0.7167075  0.3841854

simulate(model, 'y[1:3]')
model$y

## [1] 10.3195078  2.9896248  3.5401512  0.7167075  0.3841854

simulate(model, 'y')
model$y

## [1] 9.529274 3.393389 4.344205 3.757832 3.988094

model$z

##            [,1]       [,2]        [,3]
## [1,]  0.2670988  1.5868335 -0.47340064
## [2,] -0.5425200  0.5584864 -0.62036668
## [3,]  1.2078678 -1.2765922  0.04211587
## [4,]  1.1604026 -0.5732654 -0.91092165
## [5,]  0.7002136 -1.2246126  0.15802877
```

```
simulate(model, c('y[1:3]', 'z[1:5, 1:3]'))
model$y

## [1] 4.117366 6.562761 4.439232 3.757832 3.988094

model$z

##            [,1]       [,2]        [,3]
## [1,]  0.2670988  1.5868335 -0.47340064
## [2,] -0.5425200  0.5584864 -0.62036668
## [3,]  1.2078678 -1.2765922  0.04211587
## [4,]  1.1604026 -0.5732654 -0.91092165
## [5,]  0.7002136 -1.2246126  0.15802877

simulate(model, c('z[1:5, 1:3]'), includeData = TRUE)
model$z

##          [,1]     [,2]     [,3]
## [1,] 4.066770 4.251670 4.095908
## [2,] 6.544805 6.552742 6.634027
## [3,] 4.431875 4.435468 4.371066
## [4,] 3.725405 3.763848 3.698943
## [5,] 4.041243 3.836254 4.018749
```

Notice that

1. inputs like `'y[1:3]'` are automatically expanded into `c('y[1]', 'y[2]', 'y[3]')`. In fact, simply `'y'` will be expanded into all nodes within `y`.
2. an abitrary number of nodes can be provided as a character vector
3. simulations will be done in the order provided, so in practice the nodes will often be obtained by functions like `getDependencies` described below. These return nodes in topologically sorted order, which means no node comes before something it depends on.
4. The data nodes `z` were not simulated into until `includeData = TRUE` was used.

Use of `calculate`, `calculateDiff` and `getLogProb` is simliar to `simulate`, except that they return the sum of the log probabilities (densities) of the nodes requested, and they have no `includeData` argument.

6.4.2 Direct access to each node's functions

Access to the underlying `calculate`, `calculateDiff`, `simulate`, and `getLogProb` functions built by NIMBLE can be had as follows:

```
y2lp <- model$nodes[['y[2]']]$calculate()
y2lp
```

```
## [1] -2.192342

model$nodes[['y[2]']]$getLogProb()

## [1] -2.192342
```

6.5 Querying model structure

NIMBLE provides functions for asking a model about its structure. These can be used from R, including from the setup code of a `nimbleFunction` (9).

Here we demonstrate this functionality using the *pump* example because it has a few more interesting components than the example above.

6.5.1 `getNodeNames()` and `getVarNames()`

First we'll see how to determine the nodes and variables in a model.

```
pump$getNodeNames()

##  [1] "alpha"               "beta"
##  [3] "lifted_d1_over_beta" "theta[1]"
##  [5] "theta[2]"            "theta[3]"
##  [7] "theta[4]"            "theta[5]"
##  [9] "theta[6]"            "theta[7]"
## [11] "theta[8]"            "theta[9]"
## [13] "theta[10]"           "lambda[1]"
## [15] "lambda[2]"           "lambda[3]"
## [17] "lambda[4]"           "lambda[5]"
## [19] "lambda[6]"           "lambda[7]"
## [21] "lambda[8]"           "lambda[9]"
## [23] "lambda[10]"          "x[1]"
## [25] "x[2]"                "x[3]"
## [27] "x[4]"                "x[5]"
## [29] "x[6]"                "x[7]"
## [31] "x[8]"                "x[9]"
## [33] "x[10]"

pump$getNodeNames(determOnly = TRUE)

##  [1] "lifted_d1_over_beta" "lambda[1]"
##  [3] "lambda[2]"           "lambda[3]"
##  [5] "lambda[4]"           "lambda[5]"
##  [7] "lambda[6]"           "lambda[7]"
##  [9] "lambda[8]"           "lambda[9]"
## [11] "lambda[10]"
```

```
pump$getNodeNames(stochOnly = TRUE)

##  [1] "alpha"     "beta"      "theta[1]"  "theta[2]"  "theta[3]"
##  [6] "theta[4]"  "theta[5]"  "theta[6]"  "theta[7]"  "theta[8]"
## [11] "theta[9]"  "theta[10]" "x[1]"      "x[2]"      "x[3]"
## [16] "x[4]"      "x[5]"      "x[6]"      "x[7]"      "x[8]"
## [21] "x[9]"      "x[10]"

pump$getNodeNames(dataOnly = TRUE)

##  [1] "x[1]"  "x[2]"  "x[3]"  "x[4]"  "x[5]"  "x[6]"  "x[7]"
##  [8] "x[8]"  "x[9]"  "x[10]"

pump$getVarNames()

## [1] "lifted_d1_over_beta" "theta"
## [3] "lambda"              "x"
## [5] "alpha"               "beta"
```

Note that some of the nodes may be "lifted" nodes introduced by NIMBLE (5.2.9).

6.5.2 getDependencies()

Next we'll see how to determine the node dependencies (or "descendents") in a model. There are a variety of arguments that allow one to specify whether to include the node itself, whether to include deterministic or stochastic or data dependents, etc. By default `getDependencies` returns descendants up to the next stochastic node on all edges of the graph. This is what would be needed to calculate a Metropolis-Hastings acceptance probability in MCMC, for example.

```
pump$getDependencies('alpha')

##  [1] "alpha"     "theta[1]"  "theta[2]"  "theta[3]"  "theta[4]"
##  [6] "theta[5]"  "theta[6]"  "theta[7]"  "theta[8]"  "theta[9]"
## [11] "theta[10]"

pump$getDependencies(c('alpha', 'beta'))

##  [1] "alpha"               "beta"
##  [3] "lifted_d1_over_beta" "theta[1]"
##  [5] "theta[2]"            "theta[3]"
##  [7] "theta[4]"            "theta[5]"
##  [9] "theta[6]"            "theta[7]"
## [11] "theta[8]"            "theta[9]"
## [13] "theta[10]"
```

```
pump$getDependencies('theta[1:3]', self = FALSE)

## [1] "lambda[1]" "lambda[2]" "lambda[3]" "x[1]"      "x[2]"
## [6] "x[3]"

pump$getDependencies('theta[1:3]', stochOnly = TRUE, self = FALSE)

## [1] "x[1]" "x[2]" "x[3]"

# get all dependencies, not just the direct descendants
pump$getDependencies('alpha', downstream = TRUE)

##  [1] "alpha"      "theta[1]"   "theta[2]"   "theta[3]"
##  [5] "theta[4]"   "theta[5]"   "theta[6]"   "theta[7]"
##  [9] "theta[8]"   "theta[9]"   "theta[10]"  "lambda[1]"
## [13] "lambda[2]"  "lambda[3]"  "lambda[4]"  "lambda[5]"
## [17] "lambda[6]"  "lambda[7]"  "lambda[8]"  "lambda[9]"
## [21] "lambda[10]" "x[1]"       "x[2]"       "x[3]"
## [25] "x[4]"       "x[5]"       "x[6]"       "x[7]"
## [29] "x[8]"       "x[9]"       "x[10]"

pump$getDependencies('alpha', downstream = TRUE, dataOnly = TRUE)

##  [1] "x[1]"  "x[2]"  "x[3]"  "x[4]"  "x[5]"  "x[6]"  "x[7]"
##  [8] "x[8]"  "x[9]"  "x[10]"
```

6.5.3 isData()

Finally, you can query whether a node is flagged as data using the `isData` method applied to one or more nodes:

```
pump$isData('x[1]')

## [1] TRUE

pump$isData(c('x[1]', 'x[2]', 'alpha'))

## [1]  TRUE  TRUE FALSE
```

TODO: IT LOOKS LIKE WE DON'T EXPLAIN expandNodeNames anywhere?

You can also query variables to determine if the nodes that are part of a variable are data nodes.

```
pump$isData('x')

##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

pump$isData('x[1:3]')

## [1] TRUE TRUE TRUE
```

6.6 The *modelValues* data structure

In the NIMBLE framework, `modelValues` are containers designed for storing values for models. They may be used for model outputs or model inputs. A modelValues object will contain *rows* of variables. Each row represents a single value of a variable from a model and will be an array (i.e. scalar, vector, matrix or three-dimensional array) from the same dimension[1]. The simplest way to build a modelValues object is from a model object. This will create a modelValues object with the same variables as the model. Although they were motivated by models, one is free to set up a `modelValues` with any variables one wants.

```
pumpModelValues = modelValues(pumpModel, m = 2)
pumpModel$x

##  [1]  5  1  5 14  3 19  1  1  4 22

pumpModelValues$x

## [[1]]
##  [1] NA NA NA NA NA NA NA NA NA NA
##
## [[2]]
##  [1] NA NA NA NA NA NA NA NA NA NA
```

In this example, `pumpModelValues` has the same variables as `pumpModel`, and we set `pumpModelValues` to have `m = 2` rows. As you can see, the rows are stored as elements of a list.

Alternatively, one can define a modelValues object manually via the `modelValuesSpec()` function, like this:

```
mvSpec = modelValuesSpec(vars = c('a', 'b', 'c'),
                         type = c('double', 'int', 'double'),
                         size = list( a = 2, b =c(2,2)  , c = 1) )

customMV = modelValues(mvSpec, m = 2 )
customMV$a
```

[1] In v0.4, NIMBLE is limited to four dimensions.

```
## [[1]]
## [1] NA NA
##
## [[2]]
## [1] NA NA
```

The arguments to `modelValuesSpec` are matching lists of variable names, types, and sizes. See `help(modelValuesSpec)` for more details. Note that in R execution, the types are not enforced. But they will be the types created in C++ code during compilation, so they should be specified carefully.

The object returned by `modelValues()` is an uncompiled modelValues. When a nimbleFunction is compiled, any modelValues objects it uses are also compiled. A NIMBLE model always contains a modelValues that it uses as a default location to store its variables.

Here is an example where the `customMV` created above is used as the setup argument for a nimbleFunction, which is then compiled. Its compiled mv is then accessed with `$`.

```
# Simple nimbleFunction that uses a modelValues object
resizeFunction_Gen <- nimbleFunction(
  setup = function(mv){},
  run = function(k = integer() ){
    resize(mv,k)})

rResize <- resizeFunction_Gen(customMV)
cResize <- compileNimble(rResize)
cCustomMV <- cResize$mv
# cCustomMV is a C++ modelValues object
```

Compiled modelValues objects can be accessed and altered in all the same ways as uncompiled ones. However, only uncompiled modelValues can be used as arguments to setup code in nimbleFunctions.

6.6.1 Accessing contents of modelValues

The values in a modelValues object can be accessed in several ways from R, and in fewer ways from NIMBLE.

```
# Sets the first row of a to (0, 1).  R only.
customMV[['a']][[1]] <- c(0,1)

# Sets the second row of a to (2, 3)
customMV['a', 2] <- c(2,3)

#Can access subsets of each row in standard R manner
customMV['a', 2][2] <- 4
```

```
# Accesses all values of 'a'. Output is a list.  R only.
customMV[['a']]

## [[1]]
## [1] 0 1
##
## [[2]]
## [1] 2 4

# Sets the first row of b to a matrix with values 1. R only.
customMV[['b']][[1]] <- matrix(1, nrow = 2, ncol = 2)

# Sets the second row of b.  R only.
customMV[['b']][[2]] <- matrix(2, nrow = 2, ncol = 2)

# Make sure the size of inputs is correct
# customMV['a', 1] <- 1:10
# Problem: dimension of 'a' is 2, not 10!
# Will cause problems when compiling nimbleFunction using customMV
```

Currently, only the syntax `customMV[‘a’, 2]` works in the NIMBLE language, not `customMV[[‘a’]][[2]]`. Also note that `c()` does not work in NIMBLE, but one can do `customMV[‘a’, 2] <- X[1:2]`.

We can query and change the number of rows using `getsize()` and `resize()`, respectively. These work in both R and NIMBLE. Note that we don’t specify the variables in this case: all variables in a modelValues object will have the same number of rows.

```
getsize(customMV)

## [1] 2

resize(customMV, 3)
getsize(customMV)

## [1] 3

customMV$a

## [[1]]
## [1] 0 1
##
## [[2]]
## [1] 2 4
##
## [[3]]
## [1] NA NA
```

Often it is useful to convert a modelValues object to a matrix for use in R. For example, we may want to convert MCMC output into a matrix for use with the `coda` package for processing MCMC samples. This can be done with the `as.matrix` method for modelValues objects. This will generate column names from every scalar element of variables (e.g. "x[1, 1]" ,"x[2, 1]", etc.). The rows of the modelValues will be the rows of the matrix, with any matrices or arrays converted to a vector based on column-major ordering.

```
as.matrix(customMV, 'a')   # convert 'a'

##      a[1] a[2]
## [1,]    0    1
## [2,]    2    4
## [3,]   NA   NA

as.matrix(customMV)        # convert all variables

##      a[1] a[2] b[1, 1] b[2, 1] b[1, 2] b[2, 2] c[1]
## [1,]    0    1       1       1       1       1   NA
## [2,]    2    4       2       2       2       2   NA
## [3,]   NA   NA      NA      NA      NA      NA   NA
```

If a variable is a scalar, using `unlist()` in R to extract all rows as a vector can be useful.

```
customMV['c', 1] <- 1
customMV['c', 2] <- 2
customMV['c', 3] <- 3
unlist(customMV['c', ])

## [1] 1 2 3
```

Once we have a modelValues object, we can see the structure of object based on the `varNames` and `sizes` components of the object.

```
customMV$varNames

## [1] "a" "b" "c"

customMV$sizes

## $a
## [1] 2
##
## $b
## [1] 2 2
##
## $c
## [1] 1
```

As with most NIMBLE objects, modelValues are passed by reference, not by value. That means any modifications of modelValues objects in either R functions or `nimbleFunctions` will persist outside of the function. This allows for more efficient computation, as stored values are immediately shared among `nimbleFunctions`.

```
alter_a <- function(mv){
  mv['a',1][1] <- 1
}
customMV['a', 1]

## [1] 0 1

alter_a(customMV)
customMV['a',1]

## [1] 1 1

#Note that the first row was changed
```

However, when you retrieve a variable from a modelValues object, the result is a standard R list, which is subsequently passed by value, as usual in R.

6.7 NIMBLE passes objects by reference

NIMBLE relies heavily on R's reference class system. When models, modelValues, and nimbleFunctions with setup code are created, NIMBLE generates a new, customized reference class definition for each. As a result, objects of these types are passed by reference and hence modified in place by most NIMBLE operations. This is necessary to avoid a great deal of copying and returning and having to reassign large objects, both in processing model and nimbleFunctions and in running algorithms.

One cannot generally copy NIMBLE models or nimbleFunctions (specializations or generators) in a safe fashion, because of the references to other objects embedded within NIMBLE objects. However, the model member function `newModel` that will create a new copy of the model from the same model definition (5.3.2). This new model can then be used with newly instantiated nimbleFunctions.

The reliable way to create new copies of nimbleFunctions is to re-run the R function called `nimbleFunction()` and record the result in a new object. For example, say you have a `nimbleFunction` called `foo` and 1000 instances of `foo` are compiled as part of an algorithm related to a model called `model1`. If you then need to use `foo` in an algorithm for another model, `model2`, doing so may work without any problems. However, there are cases where the NIMBLE compiler will tell you during compilation that the second set of `foo` instances cannot be built from the previous compiled version. A solution is to re-define `foo` from the beginning – i.e. call `nimbleFunction` again – and then proceed with building and compiling the algorithm for `model2`.

Chapter 7

MCMC

Doing MCMC in NIMBLE consists of several steps:

- Building an MCMC configuration and an MCMC function from it. These can be done in one step, but when a user wants to customize the MCMC, they can be done by:
  - Creating a configuration for the MCMC algorithm for a specific model
  - Customizing the choices of samplers or ordering, including new samplers
  - Building the MCMC function from the configuration
- Compiling and running the MCMC function
- Extracting the posterior samples

This chapter also discusses:

- Sampling algorithms provided with NIMBLE
- Default sampler assignments in an MCMC configuration
- Writing new samplers that conform to NIMBLE's MCMC system.
- Using `MCMCsuite` to automatically run WinBUGS, OpenBUGS, JAGS, Stan and/or multiple NIMBLE MCMCs on the same model.
- Using NIMBLE's algorithm to search blocks of nodes for efficient joint (block) sampling.

7.1 The MCMC configuration

The MCMC configuration contains information needed for building an MCMC. We will show how to create this information as a first step so it can be customized before moving ahead, but when no customization is needed one can jump directly to the `buildMCMC` step below. An MCMC configuration includes:

- The model on which the MCMC will operate
- The model nodes which will be sampled (updated) during execution of the MCMC
- The particular sampling algorithms for each of these nodes, including any control parameters required by each sampling algorithm
- Two sets of variables that will be monitored (recorded) during execution of the MCMC and thinning intervals for how often each set will be recorded. Two sets are allowed because it can be useful to monitor different variables at different intervals.

7.1.1 Default MCMC configuration

Assuming we have a model named `Rmodel`, the following will generate a default MCMC configuration:

```
mcmcspec <- configureMCMC(Rmodel)
```

The default configuration will contain a single sampler for each node in the model, and the default ordering follows the topological ordering of the model. `configureMCMC()` creates an *MCMCspec* reference class object. The *MCMCspec* reference class has a number of methods, such as `addSampler()` that are described later.

Default assignment of sampler algorithms

The default sampling algorithm assigned to each stochastic node is determined by the following, in order of precedence:

TODO: ADD REFERENCES FOR THE SAMPLERS

1. If the node has no stochastic dependents, a predictive *end* sampler is assigned. The *end* sampling algorithm merely calls `simulate()` on the particular node.
2. The node is checked for presence of a conjugate relationship between its prior distribution and the distributions of its stochastic dependents. If it is determined to be in a conjugate relationship, then the corresponding *conjugate* (Gibbs) sampler is assigned.
3. If the node is discrete-valued, then a *slice* sampler is assigned.
4. If the node follows a multivariate distribution, then a *RW_block* sampler is assigned for all elements. This is a Metropolis-Hastings adaptive random-walk sampler with a multivariate normal proposal.
5. If none of the above criteria are satisfied, then a *RW* sampler is assigned. This is a Metropolis-Hastings adaptive random-walk sampler with a univariate normal proposal distribution.

TODO: CAN WE PLEASE PUT CONTROL DEFAULT LIST INTO NIMBLEOPTIONS, OR IS THERE A REASON TO KEEP IT SEPARATE?

The control parameters governing each of the default sampling algorithms are dictated by the global variable `controlDefaultList`. These default values are described in Section 7.5, along with the related sampling algorithms.

Modifying the default sampler assignments `configureMCMC` accepts control arguments `useConjugacy`, `onlyRW`, `onlySlice`, and `multivariateNodesAsScalars` to modify default sampler assignments. See `help(configureMCMC)` for usage details.

Default monitors

The default MCMC configuration includes monitors on all top-level stochastic nodes of the model.

Automated parameter blocking

The default configuration may be replaced by that generated from an automated parameter blocking algorithm. This algorithm determines groupings of model nodes that, when jointly sampled with a `RW_block` sampler, increase overall MCMC efficiency. Overall efficiency is defined as the effective sample size of the slowest-mixing node divided by computation time. This is done by:

```
autoBlockSpec <- configureMCMC(Rmodel, autoBlock = TRUE)
```

In this usage, the additional control argument `autoIt` may also be provided to indicate the number of MCMC samples to be used in the automated blocking procedure (default 20,000).

7.1.2 Customizing the MCMC configuration

The MCMC configuration may be customized in a variety of ways, either through additional named arguments to `configureMCMC()` or by calling member methods of an existing `MCMCspec` object.

Default samplers for particular nodes

One can create an MCMC configuration with default samplers on just a particular set of nodes using the `nodes` argument to `configureMCMC()`. The value for the `nodes` argument may be a character vector containing node and/or variable names. In the case of a variable name, a default sampler will be added for all stochastic nodes in the variable.

If the `nodes` argument is provided, default samplers are created only for the *stochastic* nodes specified by this argument (possibly including data nodes), and the ordering of these sampling algorithms matches the ordering within the `nodes` argument. It is worthwhile to note this is the *only* way in which a sampler may be placed on a data node, which upon execution of the MCMC will overwrite any value stored in the data node.

Creating a configuration with no samplers

If you plan to customize the choice of all samplers, it can be useful to obtain a configuration with no sampler assignments at all. This can be done by providing the `nodes` argument with the value `NULL`, `character()`, or `list()`.

Overriding the default sampler control list values

The default values of control list elements for all sampling algorithms may be overridden through use of the `control` argument to `configureMCMC()`, which should be a named list. Named elements in the `control` argument will be used for all default samplers added. In addition, they are retained in the `MCMCspec` object, and will be used as defaults for any subsequent samplers added to this same `MCMCspec` object. For example, the following will create the default MCMC configuration, except all `RW` samplers will have their initial `scale` set to 3, and none of the samplers (*RW*, or otherwise) will be adaptive.

```
mcmcspec <- configureMCMC(Rmodel, control = list(scale = 3, adaptive = FALSE))
```

Note that when adding individual samplers (next), the default control list can be overridden.

Adding samplers to the configuration

Samplers may be added to a configuration using the `addSampler()` method of the `MCMCspec` object. The frst argument gives the node(s) to be sampled, called the `target`, as a character vector. The second argument gives the types of sampler, which may be provided as a character string or a `nimbleFunction` object. Valid character strings include 'RW', 'RW_block', 'slice', 'end', and 'crossLevel', which are described below. Requirements for writing a `nimbleFunction` that can be used as a sampler are also described below, and new samplers can be labeled with a `name` argument, which is used in output of `getSamplers`.

When a `control` argument is provided in a call to `addSampler()`, the `control` list elements specified will have the highest priority. The hierarchy of precedence for control list elements for samplers is:

1. Those supplied in the `control` list argument to `addSampler()`
2. Those supplied in the `control` list argument in the preceding preceding call to `configureMCMC()`
3. Those in the system-level `controlDefaultList` variable

A call to `addSampler()` results in a single instance of the specified sampler, which will be specialized to the specified `target` model node or nodes, being added at end of the current sampler ordering.

Printing, re-ordering, and removing samplers

The current, ordered, list of all samplers in the MCMC configuration may be printed by calling the `getSamplers()` method. When you want to see only samplers acting on specific model nodes or variables, provide those names as an argument to `getSamplers()`.

The existing samplers may be re-ordered using the `setSamplers()` method. The `ind` argument is a vector of sampler indices, or a character vector of model node or variable names. The samplers in the MCMC configuration will be replaced by the samplers corresponding to the indices provided, or those samplers acting on the target nodes specified. Here are a few examples. Each example assumes the `MCMCspec` object initially contains 10 samplers, and each example is independent of the others.

```
## Truncate the current list of samplers to the first 5
mcmcspec$setSamplers(ind = 1:5)

## Retain only the third sampler, which will subsequently
## become the first sampler
mcmcspec$setSamplers(ind = 3)

## Reverse the ordering of the samplers
mcmcspec$setSamplers(ind = 10:1)
```

```
## The new set of samplers becomes the
## {first, first, first, second, third} from the current list.
## Upon each iteration of the MCMC, the 'first' sampler will
## be executed 3 times, however each instance of the sampler
## will be independent in terms of scale, adaptation, etc.
mcmcspec$setSamplers(ind = c(1, 1, 1, 2, 3))

## Set the list of samplers to only those acting on model node 'alpha'
mcmcspec$setSamplers("alpha")

## Set the list of samplers to those acting on any components of the
## model variables 'x', 'y', or 'z'.
mcmcspec$setSamplers(c("x", "y", "z"))
```

Samplers may be removed from the current sampler ordering with the `removeSamplers()` method. The following examples demonstrate this usage, where again each example assumes that `mcmcspec` initially contains 10 samplers, and each example is independent of the others. `removeSamplers()` may also accept a character vector argument, and will remove all samplers acting on these target model nodes.

```
## Remove the first sampler
mcmcspec$removeSamplers(ind = 1)

## Remove the last five samplers
mcmcspec$removeSamplers(ind = 6:10)

## Remove all samplers,
## resulting in an empty MCMC configuration, containing no samplers
mcmcspec$removeSamplers(ind = 1:10)

## Remove all samplers acting on 'x' or any component of it
mcmcspec$removeSamplers("x")

## Default: providing no argument removes all samplers
mcmcspec$removeSamplers()
```

Monitors and thinning intervals

An `MCMCspec` object contains two independent lists of variables to monitor, which correspond to two independent thinning intervals: `thin` corresponding to `monitors`, and `thin2` corresponding to `monitors2`. Monitors operate at the *variable* level. Only entire model variables may be monitored. Specifying a monitor on a *node*, e.g., `x[1]`, will result in the entire variable `x` being monitored.

The variables specified in `monitors` and `monitors2` will be recorded (with thinning interval `thin`) into the `mvSamples` and `mvSamples2` – both *modelValues* objects – of the MCMC, respectively. See Section 7.4 for information about extracting these *modelValues* objects from the MCMC algorithm object.

Monitors may be added to the MCMC configuration either in the original call to `configureMCMC()` or using the `addMonitors()` method:

```
## Using an arguments to configureMCMC()
mcmcspec <- configureMCMC(Rmodel, monitors = c('alpha', 'beta'), monitors2 = 'x')

## Calling a member method of the mcmcspec object
## This results in the same monitors as above
mcmcspec$addMonitors(c('alpha', 'beta'))
mcmcspec$addMonitors2('x')
```

Similarly, either thinning interval may be set at either step:

```
## Using an argument to configureMCMC()
mcmcspec <- configureMCMC(Rmodel, thin = 1, thin2 = 100)

## Calling a member method of the mcmcspec object
## This results in the same thinning intervals as above
mcmcspec$setThin(1)
mcmcspec$setThin2(100)
```

The current lists of monitors, and thinning intervals, may be displayed using the `getMonitors()` method. Both sets of monitors (`monitors` and `monitors2`) may be reset to empty character vectors by calling the `resetMonitors()` method.

7.2 Building and compiling the MCMC algorithm

Once the MCMC configuration object has been created, and customized to one's liking, it may be used to build an MCMC function:

```
Rmcmc <- buildMCMC(mcmcspec)
```

`buildMCMC()` is a `nimbleFunction`. The returned object `Rmcmc` is an instance of the NIMBLE function specific to configuration `mcmcspec`.

When no customization is needed, one can skip `configureMCMC` and simply provide a model object to `buildMCMC`. The following two MCMC functions will be identical:

```
mcmcspec <- configureMCMC(Rmodel)    ## default MCMC configuration
Rmcmc1 <- buildMCMC(mcmcspec)

Rmcmc2 <- buildMCMC(Rmodel)    ## uses the default configuration for Rmodel
```

For speed of execution, we usually desire to compile the MCMC function to C++ (as is the case for other NIMBLE functions). To do so, we use `compileNimble()`. Care must be taken to perform this compilation in the same project that contains the underlying model and compiled model objects. A typical compilation call looks like:

```
Cmcmc <- compileNimble(Rmcmc, project = Rmodel)
```

Alternatively, if the model has not already been compiled, they can be compiled together in one line:

```
Cmcmc <- compileNimble(Rmodel, Rmcmc)
```

7.3 Executing the MCMC algorithm

The MCMC function (either the compiled or uncompiled version) has one required argument, `niter`, representing the number of iterations to run the MCMC algorithm. We'll assume the function is called `mcmc`. Calling `mcmc(niter)` causes the full list of samplers (as determined from the input `MCMCspec` object) to be executed `niter` times, and the monitored variables to be stored into the internal `mvSamples` and/or `mvSamples2` objects as governed by the corresponding thinning intervals.

The `mcmc()` function has an optional `reset` argument. When `reset = TRUE` (the default value), the following occurs at the onset of the call to `mcmc$run()`:

- All model nodes are checked that they contain values, and that model log-probabilities are up-to-date with the current node values. If a stochastic node is missing a value, it is populated using a call to `simulate()`. The values of deterministic nodes are calculated, to be consistent with their parent nodes. If any right-hand-side-only nodes are missing a value, an error results.
- All MCMC sampler functions are reset to their initial state: the initial values of any sampler control parameters (e.g., `scale`, `sliceWidth`, or `propCov`) are reset to their initial values, as were specified by the original MCMC configuration.
- The internal *modelValues* objects `mvSamples` and `mvSamples2` are each resized to the appropriate length for holding the requested number of samples (`niter/thin`, and `niter/thin2`, respectively).

When `mcmc$run(niter, reset = FALSE)` is called, the MCMC algorithm picks up from where it left off. No values in the model are checked or altered, and sampler functions are not reset to their initial states. Further, the internal *modelValues* objects containing samples are each increased in size to appropriately accommodate the additional samples.

TODO: WHERE DOES THE USER FIND R DOC PAGES ON THIS?: Further arguments and details can be found by `help(buildMCMC)`.

7.4 Extracting MCMC samples

After executing the MCMC, the output samples can be extracted as follows as:

```
mvSamples <- mcmc$mvSamples
mvSamples2 <- mcmc$mvSamples2
```

These *modelValues* objects can be converted into matrices using `as.matrix()`:

```
samplesMatrix <- as.matrix(mvSamples)
samplesMatrix2 <- as.matrix(mvSamples2)
```

The column names of the matrices will be the node names of nodes in the monitored variables. Then, for example, the mean of the samples for node `x[2]` could be calculated as:

```
mean(samplesMatrix[, 'x[2]'])
```

7.5 Sampler Algorithms provided with NIMBLE

We now describe the samplers provided with NIMBLE. The MCMC configuration for a model generated from the following model code will serve as our example for this section:

```
code <- nimbleCode({
    a ~ dgamma(1, 1)
    b ~ dgamma(1, 1)
    p ~ dbeta(a, b)
    y1 ~ dbinom(prob = p, size = 10)
    y2 ~ dbinom(prob = p, size = 20)
})
```

7.5.1 Terminal node *end* Sampler

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

The `end` sampler functions by calling the `simulate()` method of relevant node, then updating model probabilities and deterministic dependent nodes. The application of an `end` sampler to any non-terminal node will result in invalid posterior inferences. The `end` sampler will automatically be assigned to all terminal, non-data stochastic nodes in a model by the default MCMC configuration, so it is uncommon to manually assign this sampler. The `end` sampler accepts no control list arguments.

Example usage:

```
mcmcspec$addSampler(target = 'y[1]', type = 'end')
```

7.5.2 Scalar Metropolis-Hastings random walk `RW` sampler

TODO: ADD REFERENCE FOR ADAPTATION.

The `RW` sampler executes the Metropolis-Hastings algorithm with a normal proposal distribution. This sampler can be applied to any scalar continuous-valued stochastic node.

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

adaptive (default = `TRUE`) 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.

adaptInterval (default = 200) The interval on which to perform adaptation. Every `adaptInterval` MCMC iterations, 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.

scale (default = 1) The initial value of the normal proposal standard deviation. If `adaptive = FALSE`, `scale` will never change.

Example usage:

```
mcmcspec$addSampler(target = 'a', type = 'RW',
     control = list(adaptive = FALSE, scale = 3))

mcmcspec$addSampler(target = 'b', type = 'RW',
     control = list(adaptive = TRUE, adaptInterval = 200))
```

Note that because we use a simple normal proposal distribution on all nodes, negative proposals may be simulated for non-negative random variables. These will be rejected, so the only downsides to this are some inefficiency and the presence of warnings during uncompiled (but not compiled) execution indicating `NA` or `NaN` values.

7.5.3 Multivariate Metropolis-Hastings `RW_block` sampler

TODO: ADD REFERENCE FOR ADAPTATION.

TODO: HERE AND ABOVE: STATE IF THE ADAPTINTERVAL'S ARE PRE- OR POST-THINNING.

The `RW_block` sampler performs a simultaneous update of one or more model nodes, using a Metropolis-Hastings algorithm with a multivariate normal proposal distribution. 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 (default = `TRUE`) 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`.

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

adaptInterval (default = 200) The interval on which to perform adaptation. Every `adaptInterval` MCMC iterations, the `RW_block` sampler will perform its adaptation procedure, based on the past `adaptInterval` iterations.

scale (default = 1) The initial value of the scalar multiplier for `propCov`. If `adaptive = FALSE`, `scale` will never change.

propCov (default = "identity") 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.

Example usage:

```
mcmcspec$addSampler(target = c('a', 'b', 'c'), type = 'RW_block')
```

TODO: THE TERM "POSTERIOR SLICE" IS UNCLEAR, AND THE TERM "SHRINKAGE" MAY BE CONFUSING IF IT IS NOT USED IN THE STATISTICAL SENSE.

7.5.4 Slice sampler

The `slice` sampler performs slice sampling of the scalar node to which it is applied. 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`. When sampling from the posterior slice, a shrinkage procedure is employed to improve sampling efficiency. 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 (default = `TRUE`) A logical argument, specifying whether the sampler will adapt the value of `sliceWidth` throughout the course of MCMC execution.

adaptInterval (default = 200) The interval on which to perform adaptation.

sliceWidth (default = 1) The initial value of the width of each slice, and also the width of the expansion during the iterative "stepping out" procedure.

sliceMaxSteps (default = 100) The maximum number of expansions which may occur during the "stepping out" procedure.

Example usage:

```
mcmcspec$addSampler(target = 'y[1]', type = 'slice',
     control = list(adaptive = FALSE, sliceWidth = 3))

mcmcspec$addSampler(target = 'y[2]', type = 'slice',
     control = list(adaptive = TRUE, sliceMaxSteps = 1))
```

7.5.5 Hierarchical `crossLevel` sampler

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

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

adaptive (default = `TRUE`) Logical argument, specifying whether the multivariate normal proposal distribution for the `target` nodes should be adaptived.

adaptInterval (default = 200) As for `RW_block`, the interval on which to perform adaptation.

scale (default = 1) As for `RW_block`, the initial value of the scalar multiplier for `propCov`.

propCov (default = "identity") As for `RW_block`, 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.

Example usage:

```
mcmcspec$addSampler(target = c('a', 'b', 'c'), type = 'crossLevel')
```

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.

7.5.6 Using customized log likelihood evaluations via the `RW_llFunction` sampler

TODO: DO WE HAVE A BLOCK VERSION OF THIS?

TODO: ADD A MORE COMPLETE EXAMPLE OF USING THIS

TODO: ADD PMCMC REFERENCE

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 (see below), allowing composition of a particle MCMC (PMCMC) algorithm. 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 an instance of a `nimbleFunction`. It can use the same model as the MCMC if needed, but if so the state of the model should be unchanged by the time it exits (or you must understand the implications otherwise).

Options for the `RW_llFunction` sampler include:

adaptive (default = `TRUE`) A logical argument, specifying whether the sampler should adapt the `scale` (proposal standard deviation) throughout the course of MCMC execution.

adaptInterval (default = 200) The interval on which to perform adaptation.

scale (default = 1) The initial value of the normal proposal standard deviation.

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 Logical variable indicating whether the return value of `llFunction` includes the log-likelihood associated with `target`. This is a required element with no default.

Example usage:

```
mcmcspec$addSampler(target = 'p', type = 'RW_llFunction',
    control = list(llFunction = logLiky1y2, includesTarget = FALSE))
```

7.5.7 Conjugate (Gibbs) samplers

Gibbs samplers can be provided for nodes in conjugate relationships, as specified by the system-level `conjugacyRelationshipsInputList`. Conjugate samplers should not, in general, be manually added or modified by a user, since the control list requisites and syntax are lengthy, and determining conjugacy and assigning conjugate samplers is fully handled by the default MCMC configuration.

A model may be checked for conjugate relationships using `model$checkConjugacy()`. This returns a named list describing all conjugate nodes. `checkConjugacy` can also accept a character vector argument specifying a subset of model node names to check for conjugacy.

TODO: CHECK THE FOLLOWING STATEMENT

In this release, conjugacies involving multivariate distributions as well as some additional conjugate relationships are not detected.

7.6 Detailed MCMC example: `litters`

Here is a detailed example of specifying, building, compiling, and running two MCMC algorithms. We use the `litters` example from the BUGS examples.

```
################################
##### model configuration #####
################################

## define our model using BUGS syntax
litters_code <- nimbleCode({
    for (i in 1:G) {
        a[i] ~ dgamma(1, .001)
        b[i] ~ dgamma(1, .001)
        for (j in 1:N) {
            r[i,j] ~ dbin(p[i,j], n[i,j])
            p[i,j] ~ dbeta(a[i], b[i])
        }
        mu[i] <- a[i] / (a[i] + b[i])
        theta[i] <- 1 / (a[i] + b[i])
    }
})

## list of fixed constants
constants <- list(G = 2,
                  N = 16,
                  n = matrix(c(13, 12, 12, 11, 9, 10, 9,  9, 8, 11, 8, 10, 13,
                      10, 12, 9, 10,  9, 10, 5,  9,  9, 13, 7, 5, 10, 7,  6,
                      10, 10, 10, 7), nrow = 2))

## list specifying model data
data <- list(r = matrix(c(13, 12, 12, 11, 9, 10,  9, 9, 8, 10, 8, 9, 12, 9,
                 11, 8, 9,  8,  9,  4, 8,  7, 11, 4, 4, 5 , 5, 3,  7, 3,
                 7, 0), nrow = 2))

## list specifying initial values
inits <- list(p = matrix(0.5, nrow = 2, ncol = 16))

## build the R model object
Rmodel <- nimbleModel(litters_code,
                      constants = constants,
                      data      = data,
                      inits     = inits)

## Warning:  NAs were detected in model variables:  a, logProb_a, b, logProb_b,
logProb_p, mu, theta

#########################################
##### MCMC configuration and building #####
#########################################
```

```
## generate the default MCMC configuration;
## only wish to monitor the derived quantity 'mu'
mcmcspec <- configureMCMC(Rmodel, monitors = 'mu')

## check the samplers assigned by default MCMC configuration
mcmcspec$getSamplers()
```

```
## [1]  RW sampler: a[1],  adaptive: TRUE,  adaptInterval: 200,  scale: 1
## [2]  RW sampler: a[2],  adaptive: TRUE,  adaptInterval: 200,  scale: 1
## [3]  RW sampler: b[1],  adaptive: TRUE,  adaptInterval: 200,  scale: 1
## [4]  RW sampler: b[2],  adaptive: TRUE,  adaptInterval: 200,  scale: 1
## [5]  conjugate_dbeta sampler: p[1, 1],  dependents_dbin: r[1, 1]
## [6]  conjugate_dbeta sampler: p[1, 2],  dependents_dbin: r[1, 2]
## [7]  conjugate_dbeta sampler: p[1, 3],  dependents_dbin: r[1, 3]
## [8]  conjugate_dbeta sampler: p[1, 4],  dependents_dbin: r[1, 4]
## [9]  conjugate_dbeta sampler: p[1, 5],  dependents_dbin: r[1, 5]
## [10] conjugate_dbeta sampler: p[1, 6],  dependents_dbin: r[1, 6]
## [11] conjugate_dbeta sampler: p[1, 7],  dependents_dbin: r[1, 7]
## [12] conjugate_dbeta sampler: p[1, 8],  dependents_dbin: r[1, 8]
## [13] conjugate_dbeta sampler: p[1, 9],  dependents_dbin: r[1, 9]
## [14] conjugate_dbeta sampler: p[1, 10],  dependents_dbin: r[1, 10]
## [15] conjugate_dbeta sampler: p[1, 11],  dependents_dbin: r[1, 11]
## [16] conjugate_dbeta sampler: p[1, 12],  dependents_dbin: r[1, 12]
## [17] conjugate_dbeta sampler: p[1, 13],  dependents_dbin: r[1, 13]
## [18] conjugate_dbeta sampler: p[1, 14],  dependents_dbin: r[1, 14]
## [19] conjugate_dbeta sampler: p[1, 15],  dependents_dbin: r[1, 15]
## [20] conjugate_dbeta sampler: p[1, 16],  dependents_dbin: r[1, 16]
## [21] conjugate_dbeta sampler: p[2, 1],  dependents_dbin: r[2, 1]
## [22] conjugate_dbeta sampler: p[2, 2],  dependents_dbin: r[2, 2]
## [23] conjugate_dbeta sampler: p[2, 3],  dependents_dbin: r[2, 3]
## [24] conjugate_dbeta sampler: p[2, 4],  dependents_dbin: r[2, 4]
## [25] conjugate_dbeta sampler: p[2, 5],  dependents_dbin: r[2, 5]
## [26] conjugate_dbeta sampler: p[2, 6],  dependents_dbin: r[2, 6]
## [27] conjugate_dbeta sampler: p[2, 7],  dependents_dbin: r[2, 7]
## [28] conjugate_dbeta sampler: p[2, 8],  dependents_dbin: r[2, 8]
## [29] conjugate_dbeta sampler: p[2, 9],  dependents_dbin: r[2, 9]
## [30] conjugate_dbeta sampler: p[2, 10],  dependents_dbin: r[2, 10]
## [31] conjugate_dbeta sampler: p[2, 11],  dependents_dbin: r[2, 11]
## [32] conjugate_dbeta sampler: p[2, 12],  dependents_dbin: r[2, 12]
## [33] conjugate_dbeta sampler: p[2, 13],  dependents_dbin: r[2, 13]
## [34] conjugate_dbeta sampler: p[2, 14],  dependents_dbin: r[2, 14]
## [35] conjugate_dbeta sampler: p[2, 15],  dependents_dbin: r[2, 15]
## [36] conjugate_dbeta sampler: p[2, 16],  dependents_dbin: r[2, 16]
```

```
## double-check our monitors, and thinning interval
```

```
mcmcspec$getMonitors()

## thin = 1: mu

## build the executable R MCMC function
mcmc <- buildMCMC(mcmcspec)

## let's try another MCMC, as well,
## this time using the crossLevel sampler for top-level nodes

## generate an empty MCMC configuration
## we need a new copy of the model to avoid compilation errors
Rmodel2 <- Rmodel$newModel()

## Warning:  NAs were detected in model variables:  a, logProb_a, b, logProb_b,
logProb_p, mu, theta

mcmcspec_CL <- configureMCMC(Rmodel2, nodes = NULL, monitors = 'mu')

## add two crossLevel samplers
mcmcspec_CL$addSampler(target = c('a[1]', 'b[1]'), type = 'crossLevel')

## [1] crossLevel sampler: a[1], b[1],  adaptive: TRUE,  adaptScaleOnly: FALSE,  adaptIn

mcmcspec_CL$addSampler(target = c('a[2]', 'b[2]'), type = 'crossLevel')

## [2] crossLevel sampler: a[2], b[2],  adaptive: TRUE,  adaptScaleOnly: FALSE,  adaptIn

## let's check the samplers
mcmcspec_CL$getSamplers()

## [1] crossLevel sampler: a[1], b[1],  adaptive: TRUE,  adaptScaleOnly: FALSE,  adaptIn
## [2] crossLevel sampler: a[2], b[2],  adaptive: TRUE,  adaptScaleOnly: FALSE,  adaptIn

## build this second executable R MCMC function
mcmc_CL <- buildMCMC(mcmcspec_CL)


###################################
##### compile to C++, and run #####
###################################

## compile the two copies of the model
Cmodel <- compileNimble(Rmodel)
Cmodel2 <- compileNimble(Rmodel2)
```

```
## compile both MCMC algorithms, in the same
## project as the R model object
## NOTE: at this time, we recommend compiling ALL
## executable MCMC functions together
Cmcmc <- compileNimble(mcmc, project = Rmodel)
Cmcmc_CL <- compileNimble(mcmc_CL, project = Rmodel2)

## run the default MCMC function,
## and example the mean of mu[1]
Cmcmc$run(1000)

## NULL

cSamplesMatrix <- as.matrix(Cmcmc$mvSamples)
mean(cSamplesMatrix[, 'mu[1]'])

## [1] 0.8309548

## run the crossLevel MCMC function,
## and examine the mean of mu[1]
Cmcmc_CL$run(1000)

## NULL

cSamplesMatrix_CL <- as.matrix(Cmcmc_CL$mvSamples)
mean(cSamplesMatrix_CL[, 'mu[1]'])

## [1] 0.7559898
```

7.7 Comparing different MCMC engines with `MCMCsuite`

TODO: ADD OPENBUGS

NIMBLE's `MCMCsuite` function automatically runs WinBUGS, OpenBUGS, JAGS, Stan, and/or multiple NIMBLE configurations on the same model. Note that the BUGS code must be compatible with whichever BUGS packages are included, and separate Stan code must be provided.

We show how to use `MCMCsuite` for the same `litters` example used in 7.6. Subsequently, additional details of the `MCMCsuite` are given.

TODO: MIGRATE USAGE DETAILS TO DOCS.

7.7.1 MCMC Suite example: `litters`

The following code executes the following MCMC algorithms on the `litters` example:

1. WinBUGS

2. JAGS
3. NIMBLE default configuration
4. NIMBLE configuration with argument `onlySlice = TRUE`
5. NIMBLE custom configuration using two crossLevel samplers

```
output <- MCMCsuite(
    code = litters_code,
    constants = constants,
    data = data,
    inits = inits,
    monitors = 'mu',
    MCMCs = c('bugs', 'jags', 'nimble', 'nimble_slice', 'nimble_CL'),
    MCMCdefs = list(
        nimble_CL = quote({
            mcmcspec <- configureMCMC(Rmodel, nodes = NULL)
            mcmcspec$addSampler(target = c('a[1]', 'b[1]'), type = 'crossLevel')
            mcmcspec$addSampler(target = c('a[2]', 'b[2]'), type = 'crossLevel')
            mcmcspec
        })),
    plotName = 'littersSuite'
)
```

7.7.2 MCMC Suite outputs

Executing the MCMC Suite returns a named list containing three elements, as well as generates and saves traceplots and posterior density plots. The elements of this return list object are:

Samples

`samples` is a three-dimensional array, containing all MCMC samples from each algorithm. The first dimension of the `samples` array corresponds to each MCMC algorithm, and may be indexed by the name of the algorithm. The second dimension of the `samples` array corresponds to each node which was monitored, and may be indexed by the node name. The third dimension of `samples` contains the MCMC samples, and has length `niter/thin - burnin`.

Summary

The MCMC suite output contains a variety of pre-computed summary statistics, which are stored in the `summary` matrix. For each monitored node and each MCMC algorithm, the following default summary statistics are calculated: `mean`, `median`, `sd`, the 2.5% quantile, and the 97.5% quantile. These summary statistics are easily viewable, as:

```
output$summary
# , , mu[1]
#                    mean    median         sd  quant025  quant975
# bugs          0.8795868 0.8889000 0.04349589 0.7886775 0.9205025
# jags          0.8872778 0.8911989 0.02911325 0.8287991 0.9335317
# nimble        0.8562232 0.8983763 0.12501395 0.4071524 0.9299781
# nimble_slice 0.8975283 0.9000483 0.02350363 0.8451926 0.9367147
# nimble_CL     0.8871314 0.8961146 0.05243039 0.7640730 0.9620532
#
# , , mu[2]
#                    mean    median         sd  quant025  quant975
# bugs          0.7626974 0.7678000 0.04569705 0.6745975 0.8296025
# jags          0.7635539 0.7646913 0.03803033 0.6824946 0.8313314
# nimble        0.7179094 0.7246935 0.06061116 0.6058669 0.7970130
# nimble_slice 0.7665562 0.7683093 0.04051432 0.6641368 0.8350716
# nimble_CL     0.7605938 0.7655945 0.09138471 0.5822785 0.9568195
```

Timing

`timing` contains a named vector of the runtime for each MCMC algorithm, the total compile time for the NIMBLE model and MCMC algorithms, and the compile time for Stan (if specified). All run- and compile- times are given in minutes.

Plots

Executing the MCMC Suite provides and saves several plots. These include trace plots and posterior density plots for each monitored node, under each algorithm.

Note that the generation of MCMC Suite plots *in Rstudio* may result in several warning messages from R (regarding graphics devices), but will function without any problems.

7.7.3 Arguments to MCMC Suite

The following arguments to `MCMCsuite` define the model on which to run each MCMC algorithm:

argument: `code`

argument: `constants`

argument: `data`

argument: `inits`

An MCMC Suite is customizable in terms of all of the following:

- MCMC algorithms
- Nodes to monitor
- Number of MCMC iterations

- Thinning interval
- Burn-in
- Summary statistics
- Generating and saving plots

These options are controlled by the following arguments to `MCMCsuite`:

argument: `MCMCs`

A character vector, defining the MCMC algorithms to run. The `MCMCs` argument may include any of the following algorithms:
`'bugs'` WinBUGS MCMC algorithm
`'jags'` JAGS MCMC algorithm
`'Stan'` Stan MCMC algorithm
`'nimble'` NIMBLE MCMC using the default configuration
`'nimble_RW'` NIMBLE MCMC using the default configuration with `onlyRW = TRUE`
`'nimble_slice'` NIMBLE MCMC using the default configuration with `onlySlice= TRUE`
`'autoBlock'` NIMBLE MCMC algorithm with block sampling of dynamically determined parameter groups attempting to maximize sampling efficiency

The names of additional, custom, MCMC algorithms may also be provided in the `MCMCs` argument, so long as these custom algorithms are defined in the `MCMCdefs` argument. An example of this usage is given with the `crossLevel` algorithm in the `litters` MCMC Suite example.

The values for the `MCMCs` argument is `c('jags','nimble', 'nimble_RW', 'nimble_slice', 'autoBlock')`.

argument: `MCMCdefs`

A named list of definitions, for any custom MCMC algorithms specified in the `MCMCs` argument. If `MCMCs` specified an algorithm called `'myMCMC'`, then `MCMCdefs` must contain an element named `'myMCMC'`. The contents of this element must be a block of code that, when executed, returns the desired MCMC configuration object. This block of code may assume the existence of the R model object, `Rmodel`. Further, this block of code need not worry about adding monitors to the MCMC configuration; it need only specify the samplers.

As a final important point, execution of this block of code must *return* the MCMC configuration object. Therefore, elements supplied in the `MCMCdefs` argument should usually take the form:

```
MCMCdefs = list(
     myMCMC = quote({
          mcmcspec <- configureMCMC(Rmodel, ....)
          mcmcspec$addSampler(.....)
          mcmcspec     ## returns the MCMC configuration object
     })
)
```

argument: `monitors`

Character vector specifying the nodes and/or vectors to monitor.

argument: `niter`

Integer specifying the number of MCMC iterations to run.

argument: `thin`

Integer specifying the thinning interval.

argument: `burnin`

Integer specifying the number of samples to discard from all chains of MCMC samples. Samples are discarded subsequent to thinning.

argument: `summaryStats`

A character vector, providing the name of any function which operates on a numeric vector, and returns a numeric scalar. Likewise, a character string defining such a function is admissible, for example `'function(x) mean(abs(x))'`.

The default value for `summaryStats` is the set: `mean`, `median`, `sd`, the 2.5% quantile, and the 97.5% quantile.

argument: `makePlot`

A logical specifying whether to generate the trace plots and posterior density plots. Default value is `TRUE`.

argument: `savePlot`

A logical specifying whether to save the generated plots. Only used if `makePlot = TRUE`. Default value is `TRUE`.

argument: `plotName`

A character string giving the filename for saving plots. Only used if `savePlot = TRUE`. Default value is `'MCMCsuite'`.

argument: `setSeed`

A logical argument, specifying whether to set.seed(0) prior to MCMC sampling. Default value is `TRUE`.

argument: `bugs_directory`

A character string giving the path to the directory containing the WinBUGS executable. The default value is `'C:/WinBUGS14'`.

argument: `bugs_program`

A character string giving the name of the WiBUGS program to execute. This will be passed directly to the `bugs()` function. The default value is `'WinBUGS'`.

argument: `stan_model`

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

In addition, the Stan data file (`'modelName.data.R'`) must also reside in the same directory as the Stan model file.

Optionally, the Stan initial values file (`'modelName.init.R'`) may also be in this same directory; it will be used if present.

7.8 Advanced topics

7.8.1 Custom sampler functions

The following code illustrates how a NIMBLE developer would concisely implement, and instantiate a Metropolis-Hastings random walk sampler with fixed proposal standard deviation. The comments accompanying the code explain the necessary characteristics of all sampler functions.

```
## the name of this sampler function, for the purposes of
## adding it to MCMC configurations, will be 'my_RW'
my_RW <- nimbleFunction(

    ## sampler functions must contain 'sampler_BASE'
    contains = sampler_BASE,

    ## sampler functions must have exactly these setup arguments:
    ## model, mvSaved, target, control
    setup = function(model, mvSaved, target, control) {
        ## first, extract the control list elements, which will
        ## dictate the behavior of this sampler.
        ## the setup code will be later processed to determine
        ## all named elements extracted from the control list.
        ## these will become the required elements for any
        ## control list argument to this sampler, unless they also
```

```
        ## appear in the system-level variable 'controlDefaultList'

        ## the random walk proposal standard deviation
        scale <- control$scale

        ## determine the list of all dependent nodes,
        ## up to the first layer of stochastic nodes, generally
        ## called 'calcNodes'.  The values, inputs, and logProbs
        ## of these nodes will be retrieved and/or altered
        ## by this algorithm.
        calcNodes <- model$getDependencies(target)
    },

    ## the run function must accept no arguments, execute
    ## the sampling algorithm, leave the modelValues object
    ## 'mvSaved' as an exact copy of the updated values in model,
    ## and have no return value.  initially, mvSaved contains
    ## an exact copy of the values and logProbs in the model.
    run = function() {

        ## extract the initial model logProb
        model_lp_initial <- getLogProb(model, calcNodes)

        ## generate a proposal value for target node
        proposal <- rnorm(1, model[[target]], scale)

        ## store this proposed value into the target node.
        ## notice the double assignment operator, `<<-`,
        ## necessary because 'model' is a persistent member
        ## data object of this sampler.
        model[[target]] <<- proposal

        ## calculate target_logProb, propagate the
        ## proposed value through any deterministic dependents,
        ## and calculate the logProb for any stochastic
        ## dependnets.  The total (sum) logProb is returned.
        model_lp_proposed <- calculate(model, calcNodes)

        ## calculate the log Metropolis-Hastings ratio
        log_MH_ratio <- model_lp_proposed - model_lp_initial

        ## Metropolis-Hastings step: determine whether or
        ## not to accept the newly proposed value
        u <- runif(1, 0, 1)
```

```
        if(u < exp(log_MH_ratio)) jump <- TRUE
        else                      jump <- FALSE

        ## if we accepted the proposal, then store the updated
        ## values and logProbs from 'model' into 'mvSaved'.
        ## if the proposal was not accepted, restore the values
        ## and logProbs from 'mvSaved' back into 'model'.
        if(jump) copy(from = model, to = mvSaved, row = 1,
                      nodes = calcNodes, logProb = TRUE)
        else     copy(from = mvSaved, to = model, row = 1,
                      nodes = calcNodes, logProb = TRUE)
    },

    ## sampler functions must have a member method 'reset',
    ## which takes no arguments and has no return value.
    ## this function is used to reset the sampler to its
    ## initial state.  since this sampler function maintains
    ## no internal member data variables, reset() needn't
    ## do anything.
    methods = list(
        reset = function () {}
    )
)

## now, assume the existence of an R model object 'Rmodel',
## which has a scalar-valued stochastic node 'x'

## create an MCMC configuration with no sampler functions
mcmcspec <- configureMCMC(Rmodel, nodes = NULL)

## add our custom-built random walk sampler on node 'x',
## with a fixed proposal standard deviation = 0.1
mcmcspec$addSampler(target = 'x', type = 'my_RW',
    control = list(scale = 0.1))

Rmcmc <- buildMCMC(mcmcspec)   ## etc...
```

Chapter 8

Other algorithms provided by NIMBLE

In v0.4, the NIMBLE algorithm library is quite limited beyond MCMC. It includes some basic utilities for calculating and simulating sets of nodes. And it includes a couple of algorithms, particle filters and MCEM, that illustrate the kind of programming with models that can be done with NIMBLE.

8.1 Basic Utilities

8.1.1 `simNodes`, `calcNodes`, and `getLogProbs`

`simNodes`, `calcNodes` and `getLogProb` are basic nimbleFunctions that simulate, calculate, or get the log probabilities (densities), respectively, of the same set of nodes each time they are called. Each of these takes a model and a character string of node names as inputs. If `nodes` is left blank, then all the nodes of the model are used.

For `simNodes`, the nodes provided are topologically sorted to simulate in the correct order. For `calcNodes` and `getLogProb`, the nodes are sorted and dependent nodes are included. Recall that the calculations must be up to date (from a calculate call) for `getLogProb` to return the values you are probably looking for.

```
simpleModelCode <- nimbleCode({
  for(i in 1:4){
    x[i] ~ dnorm(0,1)
    y[i] ~ dnorm(x[i], 1) #y depends on x
    z[i] ~ dnorm(y[i], 1) #z depends on y
    #z conditionally independent of x
  }
})

simpleModel <- nimbleModel(simpleModelCode)
```

```
## Warning:  NAs were detected in model variables:  x, logProb_x, y, logProb_y,
z, logProb_z

cSimpleModel <- compileNimble(simpleModel)

#simulates all the x's and y's
rSimXY <- simNodes(simpleModel, nodes = c('x', 'y') )

#calls calculate on x and its dependents (y, but not z)
rCalcXDep <- calcNodes(simpleModel, nodes = 'x')

#calls getLogProb on x's and y's
rGetLogProbXDep <- getLogProbNodes(simpleModel,
                  nodes = 'x')

#compiling the functions
cSimXY <- compileNimble(rSimXY, project = simpleModel)
cCalcXDep <- compileNimble(rCalcXDep, project = simpleModel)
cGetLogProbXDep <- compileNimble(rGetLogProbXDep,
                           project = simpleModel)

cSimpleModel$x

## [1] NA NA NA NA

cSimpleModel$y

## [1] NA NA NA NA

#simulating x and y
cSimXY$run()

## NULL

cSimpleModel$x

## [1]  0.3866900 -0.6641685  0.7236101  0.3079503

cSimpleModel$y

## [1]  0.1074077  0.2238990  1.0589795 -0.1070424

cCalcXDep$run()

## [1] -8.531732
```

```
#Gives correct answer because logProbs
#updated by 'calculate' after simulation
cGetLogProbXDep$run()

## [1] -8.531732

cSimXY$run()

## NULL

#Gives old answer because logProbs
#not updated after 'simulate'
cGetLogProbXDep$run()

## [1] -8.531732

cCalcXDep$run()

## [1] -9.287601
```

8.1.2 `simNodesMV`, `calcNodesMV`, and `getLogProbsMV`

There is a similar trio of nimbleFunctions that does each job repeatedly for different rows of a modelValues object. For example, `simNodesMV` will simulate in the model multiple times and record each simulation in a row of its modelValues. `calcNodesMV` and `getLogProbsMV` iterate over the rows of a modelValues, copy the nodes into the model, and then do their job of calculating or collecting log probabilities (densities), respectively. Each of these returns a numeric vector with the summed log probabilities of the chosen nodes from each each row. `calcNodesMV` will save the log probabilities back into the modelValues object if `saveLP == TRUE`, a run-time argument.

Here are some examples:

```
mv <- modelValues(simpleModel)
rSimManyXY <- simNodesMV(simpleModel, nodes = c('x', 'y'), mv = mv)
rCalcManyXDeps <- calcNodesMV(simpleModel, nodes = 'x', mv = mv)
rGetLogProbMany <- getLogProbNodesMV(simpleModel,
                    nodes = 'x', mv = mv)

cSimManyXY <- compileNimble(rSimManyXY, project = simpleModel)
cCalcManyXDeps <- compileNimble(rCalcManyXDeps, project = simpleModel)
cGetLogProbMany <- compileNimble(rGetLogProbMany, project = simpleModel)

cSimManyXY$run(m = 5) # simulating 5 times

## NULL
```

```
cCalcManyXDeps$run(saveLP = TRUE) # calculating

## [1] -15.087014 -18.548380 -10.665214 -12.113602  -9.558331

cGetLogProbMany$run() #

## [1] -15.087014 -18.548380 -10.665214 -12.113602  -9.558331
```

8.2 Particle Filter

NIMBLE includes an algorithm for a basic particle filter to be used for approximating the log likelihood of a state-space model. A particle filter can be built around such a model by a call to `buildPF`. This nimbleFunction requires setup arguments `model` and `nodes`, which is a character vector specifying latent model nodes. The particle filter can be run by specifying the number of particles.

Here is an example, using a linear state-space model for which we can also calculate the likelihood using the Kalman Filter to verify if the particle filter seems to be working.

```
# Building a simple linear state-space model.
# x is latent space, y is observed data
timeModelCode <- nimbleCode({
  x[1] ~ dnorm(mu_0, 1)
        y[1] ~ dnorm(x[1], 1)
        for(i in 2:t){
                x[i] ~ dnorm(x[i-1] * a + b, 1)
                y[i] ~ dnorm(x[i] * c, 1)
        }

        a ~ dunif(0, 1)
        b ~ dnorm(0, 1)
        c ~ dnorm(1,1)
        mu_0 ~ dnorm(0, 1)
})

#simulate some data
t = 25; mu_0 = 1
x = rnorm(1 ,mu_0, 1)
y = rnorm(1, x, 1)
a = 0.5; b = 1; c = 1
for(i in 2:t){
        x[i] = rnorm(1, x[i-1] * a + b, 1)
        y[i] = rnorm(1, x[i] * c, 1)
}
```

```
## build and compile the model
rTimeModel <- nimbleModel(timeModelCode, constants = list(t = t), data = list(y = y) )

## Warning:  NAs were detected in model variables:  a, logProb_a, b, logProb_b,
c, logProb_c, mu_0, logProb_mu_0, x, logProb_x, logProb_y, lifted_x_oBi_minus_1_cB_times_a_p
lifted_x_oBi_cB_times_c

cTimeModel <- compileNimble(rTimeModel)

#Build the particle filter
rPF <- buildPF(rTimeModel, "x")
cPF = compileNimble(rPF,project = rTimeModel)

#Set parameter values
cTimeModel$mu_0 = 1
cTimeModel$a = 0.5
cTimeModel$b = 1
cTimeModel$c = 1
cTimeModel$mu_0 = 1

#Run particle filter with
#5000 particles
cPF$run(m = 5000)

## [1] -46.34275
```

8.3 Monte Carlo Expectation Maximization (MCEM)

Suppose we have a model with missing data (or a layer of latent variables that can be treated as missing data) and we would like to maximize the marginal likelihood of the model, integrating over the missing data. A brute-force method for doing this is MCEM. This is an EM algorithm in which the missing data are simulated via Monte Carlo (often MCMC, when the full conditional distributions cannot be directly sampled from) at each iteration. MCEM can be slow, and there are other methods for maximizing marginal likelihoods that can be implemented in NIMBLE. The reason we started with MCEM is to explore the flexibility of NIMBLE and illustrate the combination of R and NIMBLE involved, with R managing the highest-level processing of the algorithm and calling nimbleFunctions for computations.

We will revisit the *pump* example to illustrate the use of NIMBLE's MCEM algorithm.

```
pumpMCEM <- buildMCEM(model = newPump,
                      latentNodes = 'theta',
                      burnIn = 100,
                      mcmcControl = list(adaptInterval = 20),
```

```
                        boxConstraints = list( list( c('alpha', 'beta'),
                            limits = c(0, Inf) ) ),
                        buffer = 1e-6)

## Error in buildMCEM(model = newPump, latentNodes = "theta", burnIn = 100, :
object 'newPump' not found
```

The first argument, `model`, is a NIMBLE model, which can be either the uncompiled or compiled version. At the moment, the model provided cannot be part of another MCMC sampler.

The `latentNodes` argument should indicate the nodes that will be integrated over (sampled via MCMC), rather than maximized. These nodes must be stochastic, not deterministic! `latentNodes` will be expanded as described in section 6.4.1: e.g., either `latentNodes = 'x'` or `latentNodes = c('x[1]', 'x[2]')` will treat `x[1]` and `x[2]` as latent nodes if `x` is a vector of two values. All other non-data nodes will be maximized over. Note that `latentNodes` can include discrete nodes, but the nodes to be maximized cannot.

The `burnIn` argument indicates the number of samples from the MCMC for the E-step that should be discarded when computing the expected likelihood in the M-step. Note that `burnIn` can be set to values lower than in standard MCMC computations, as each iteration will start off where the last left off.

The `mcmcControl` argument will be passed to `configureMCMC()` to define the MCMC to be used.

The MCEM algorithm allows for box constraints on the nodes that will be optimized, specified via the `boxConstraints` argument. This is highly recommended for nodes that have zero density on parts of the real line[1] Each constraint given should be a list in which the first element is the names of the nodes or variables that the constraint will be applied to and the second element is a vector of length 2, in which the first value is the lower limit and the second is the upper limit. Values of `Inf` and `-Inf` are allowed. If a node is not listed, it will be assumed that there are no constraints. These arguments are passed as `lower` and `upper` to R's `optim()` function, using `method = 'L-BFGS-B'`)

The value of the `buffer` argument shrinks the `boxConstraints` by this amount. This can help protect against non-finite values occuring when a parameter is on its boundary value.

Once the MCEM has been built for the model of interest using `buildMCEM()`, it can be run as follows.

```
pumpMCEM(maxit = 20, m1 = 250, m2 = 500)

##     alpha      beta
## 0.8287209 1.2664132

pumpMCEM(maxit = 50, m1 = 1000, m2 = 5000)

##     alpha      beta
## 0.8230237 1.2646604
```

[1] Currently NIMBLE is not able to determine this automatically.

There are three run-time arguments:

The `maxit` argument is the number of total iterations to run the algorithm. More advanced MCEM algorithms have a stopping criteria based on computing the MCMC error. Our current draft implementation of the algorithm merely runs `maxit` iterations and then terminates.

Halfway through the algorithm, the sample size used for the E-step switches from `m1` to `m2`. This provides smaller MCMC error as the algorithm converges. If `m1` or `m2` is less than or equal to `burnIn` as defined in `build_MCEM`, the MCEM algorithm will immediately quit.

When using the MCEM algorithm, we suggest first starting with small values of `m1` and `m2` to get an estimate of how long the algorithm will take for larger MCMC samples. The speed of the algorithm will be linear in `m2` (assuming that $m1 > m2$ as intended).

Chapter 9

Writing nimbleFunctions

9.1 Writing nimbleFunctions

When you write an R function, you say what the input arguments are, you provide the code for execution, and in that code you give the returned value[1]. Using the `function` keyword in R triggers the operation of creating an object that is the function.

Creating nimbleFunctions is similar, but there are two kinds of code and two steps of execution:

1. `Setup` code is provided as a regular R function, but the programmer does not control what it returns. Typically the inputs to `setup` code are objects like a model, a vector of nodes, a modelValues object or modelValuesSpec, or another nimbleFunction. The `setup` code, as its name implies, sets up information for run-time code. It is executed in R, so it can use any aspect of R.
2. `Run` code is provided in the NIMBLE language. This is similar to a narrow subset of R, but it is important to remember that it is different – defined by what can be compiled – and much more limited. `Run` code can use the objects created by the `setup` code. In addition, some information on variable types must be provided for input arguments, the return object, and in some circumstances for local variables. There are two kinds of `run` code:
   (a) There is always a primary function, given as an argument called `run`[2].
   (b) There can optionally be other functions, or "methods" in the language of object-oriented programming, that share the same objects created by the `setup` function.

Here is a small example to fix ideas:

```
logProbCalcPlus <- nimbleFunction(
    setup = function(model, node) {
        dependentNodes <- model$getDependencies(node)
        valueToAdd <- 1
    },
```

[1] normally the value of the last evaluated code, or the argument to return().
[2] This can be omitted if you don't need it.

```
    run = function(P = double(0)) {
        model[[node]] <<- P + valueToAdd
        return(calculate(model, dependentNodes))
        returnType(double(0))
    })

code <- nimbleCode({
    a ~ dnorm(0, 1); b ~ dnorm(a, 1)
})
testModel <- nimbleModel(code)

## Warning:  NAs were detected in model variables:  a, logProb_a, b, logProb_b

logProbCalcPlusA <- logProbCalcPlus(testModel, 'a')
testModel$b <- 1.5
logProbCalcPlusA$run(0.25) ## should match

## [1] -2.650377

                       ## dnorm(1.25,0,1,TRUE)+dnorm(1.5,1.25,1,TRUE)
testModel$a  ## a was set to 0.5 + valueToAdd

## [1] 1.25
```

The call to the R function called `nimbleFunction` returns a function, similarly to defining a function in R. That function, `logProbCalcPlus`, takes arguments for its `setup` function, executes it, and returns an object, `logProbCalcPlusA`, that has a *run* member function (method) accessed by `$run`. In this case, the `setup` function obtains the stochastic dependencies of the `node` using the `getDependencies` member function of the model (see 6.5.2) and stores them in `dependentNodes`. In this way, `logProbCalcPlus` can adapt to any model. It also creates a variable, `valueToAdd`, that can be used by the nimbleFunction.

The object `logProbCalcPlusA`, returned by `logProbCalcPlus`, is permanently bound to the results of the processed `setup` function. In this case, `logProbCalcPlusA$run` takes a scalar input value, `P`, assigns `P + valueToAdd` to the given node in the model, and returns the sum of the log probabilities of that node and its stochastic dependencies[3]. We say `logProbCalcPlusA` is and "instance" of `logProbCalcPlus` that is "specialized" or "bound" to `a` and `testModel`. Usually, the `setup` code will be where information about the model structure is determined, and then the `run` code can use that information without repeatedly, redundantly recomputing it. A nimbleFunction can be called repeatedly, each time returning a specialized nimbleFunction.

Readers familiar with object-oriented programming may find it useful to think in terms of class definitions and objects. `nimbleFunction` creates a class definition. Each specialized

[3]Note the use of the global assignment operator to assign into the model. This is necessary for assigning into variables from the `setup` function, at least if you want to void warnings from R. These warnings come from R's reference class system.

nimbleFunction is one object in the class. The setup arguments are used to define member data in the object.

9.2 Using and compiling nimbleFunctions

To compile the nimbleFunction, together with its model, we use `compileNimble`:

```
CnfDemo <- compileNimble(testModel, logProbCalcPlusA)
CtestModel <- CnfDemo$testModel
ClogProbCalcPlusA <- CnfDemo$logProbCalcPlusA
```

These have been initialized with the values from their uncompiled versions and can be used in the same way:

```
CtestModel$a      ## values were initialized from testModel

## [1] 1.25

CtestModel$b

## [1] 1.5

lpA <- ClogProbCalcPlusA$run(1.5)
lpA

## [1] -5.462877

## verify the answer:
dnorm(CtestModel$b, CtestModel$a, 1, log = TRUE) +
    dnorm(CtestModel$a, 0, 1, log = TRUE)

## [1] -5.462877

CtestModel$a       ## a was modified in the compiled model

## [1] 2.5

testModel$a        ## the uncompiled model was not modified

## [1] 1.25
```

9.2.1 Accessing and modifying numeric values from setup

While models and nodes created during `setup` cannot be modified[4], numeric values and modelValues (see below) can be. For example:

```
logProbCalcPlusA$valueToAdd  ## in the uncompiled version

## [1] 1

logProbCalcPlusA$valueToAdd <- 2
ClogProbCalcPlusA$valueToAdd  ## or in the compiled version

## [1] 1

ClogProbCalcPlusA$valueToAdd <- 3
ClogProbCalcPlusA$run(1.5)

## [1] -16.46288

CtestModel$a     ## a == 1.5 + 3

## [1] 4.5
```

9.3 nimbleFunctions without setup code

The `setup` function is optional. If it is omitted, then `nimbleFunction` is more like `function`: it simply returns a function that can be executed and compiled. If there is no `setup` code, there is no specialization step. This is useful for doing math or the other kinds of processing available in NIMBLE when no model or modelValues is needed.

```
solveLeastSquares <- nimbleFunction(
    run = function(X = double(2), y = double(1)) {
        ans <- inverse(t(X) %*% X) %*% (t(X) %*% y)
        return(ans)
        returnType(double(2))
    } )

X <- matrix(rnorm(400), nrow = 100)
y <- rnorm(100)
solveLeastSquares(X, y)

##              [,1]
## [1,]  0.05894295
```

[4] Actually, they can be, but only for uncompiled nimbleFunctions

```
## [2,]  0.18030704
## [3,]  0.05952284
## [4,] -0.10547889

CsolveLeastSquares <- compileNimble(solveLeastSquares)
CsolveLeastSquares(X, y)

##             [,1]
## [1,]  0.05894295
## [2,]  0.18030704
## [3,]  0.05952284
## [4,] -0.10547889
```

This example shows the textbook calculation of a least squares solution for regression of 100 data points on 4 explanatory variables, all generated randomly[5]. Such functions can be called from other nimbleFunctions or used in BUGS code. [6]

If one wants a nimbleFunction that does get specialized but has empty setup code, use `setup = function() {}` or `setup = TRUE`.

9.4 Useful tools for `setup` functions

The setup function is typically used to determine information on nodes in a model, set up any modelValues objects, set up any nimbleFunctions or nimbleFunctionLists, and set up any persistent numeric objects. For example, the `setup` code of an MCMC nimbleFunction creates the nimbleFunctionList of sampler nimbleFunctions. The values of numeric objects created in `setup` can be modified by run code and will persist across calls.

Some of the useful tools and objects to create in `setup` functions include

vectors of node names Often these are obtained from the `getNodeNames` and `getDependencies` methods of a model, described in section 6.5.1.

modelValues objects These are discussed more below.

specializations of other nimbleFunctions A useful NIMBLE programming technique is to have one nimbleFunction contain other nimbleFunctions, which it can use in its run-time code.

lists of other nimbleFunctions In addition to containing single other nimbleFunctions, a nimbleFunction can contain a list of other nimbleFunctions. These are discussed more below.

[5] Of course in general, explicitly calculating the inverse is not the recommended numerical recipe for least squares.

[6] On the machine this is being written on, the compiled version runs a few times faster than the uncompiled version. However we refrain from formal speed tests.

9.4.1 Control of `setup` outputs

Sometimes `setup` code may create variables that are not used in run-time code. By default, NIMBLE inspects run-time code and omits variables from `setup` that do not appear in run-time code from compilation. However, sometimes a programmer may want to force a numeric or character variable to be created in compilation, even if it is not used directly in run-time code. As shown below, such variables can be directly accessed in one nimbleFunction from another, which provides a way of using nimbleFunctions as general data structures. To force NIMBLE to include variables around during compilation, for example `X` and `Y`, simply include

```
setupOutputs(X, Y)
```

anywhere in the `setup` code.

9.5 NIMBLE language components

9.5.1 Basics

There are several general points that will be useful before describing the NIMBLE language in more detail.

- NIMBLE language functions are not R functions. In many cases we have used syntax identical or nearly so to R, and in most cases we have provided a matching R function, but it is important not to confuse the NIMBLE language definition with the behavior of the corresponding R function.
- As in R, function calls in NIMBLE can provide arguments by name or in a default order.
- Like R, NIMBLE uses 1-based indexing. For example, the first element of a vector `x` is `x[1]`, not `x[0]`.
- To a large extent, NIMBLE functions can be executed in R (uncompiled) or can be compiled. Using them in R will be slow, and is intended for testing and debugging algorithm logic.
- NIMBLE is the opposite of R for argument passing. R nearly always uses pass-by-value. NIMBLE nearly always uses pass-by-reference (or pointer). That means that in compiled execution only, changing the value of a variable that was a function input will change the value in the calling function. Thus it is possible to write a nimbleFunction that returns information by modifying an argument. Yes, that's a big difference in behavior!
  Although compiled nimbleFunctions can modify arguments, the R interface to a compiled nimbleFunction performs a copy to protect the original R argument from modification. (If you want to see arguments – potentially modified – as well as any return value from R, you can modify the `control` argument to compileNimble to include "`returnAsList = TRUE`". Then the returned object will be a list with the nimbleFunction's return value as the last element.)

- BUGS model nodes are implemented as nimbleFunctions with member functions for `calculate`, `calculateDiff`, `simulate`, and `getLogProb`. There are also member functions for obtaining the value of each parameter or alternative parameters (e.g. rate = 1/scale).

9.5.2 Declaring argument types and the return type

NIMBLE requires that types of arguments and a return type be explicitly declared.

The syntax for a type declaration is:

- `type(nDim, sizes)`

`type` can currently take values `double`, `integer`, `character` (for scalars or vectors only) or `logical` (for scalars only). In a `returnType` statement, a type of `void()` is valid, but you don't need to include that because it is the default if no `returnType` statement is included. `nDim` is the number of dimensions, with 0 indicating scalar. `sizes` is an optional vector of fixed, known sizes. These should use R's `c` function if `nDim` > 1. If sizes are omitted, they will either be set when the entire object is assigned to, or an explicit call to `setSize` is needed.

9.5.3 Driving models: `calculate`, `calculateDiff`, `simulate`, and `getLogProb`

These four functions are the primary ways to operate a model. Their syntax was explained in section 6.4. Except for `getLogProb`, it is usually important for the `nodes` object to be created in `setup` code such that they are sorted in topological order.

9.5.4 Accessing model and modelValues variables and using `copy`

The `modelValues` structure was introduced in section 6.6. Inside nimbleFunctions, modelValues are designed to easily save values from a model object during the running of a nimbleFunction. A `modelValues` object used in `run` code must always exist in the setup code, either by passing it in as a setup argument or creating it in the setup code.

To illustrate this, we will create a nimbleFunction for computing importance weights for importance sampling. This function will use two modelValues objects. `propModelValues` will contain a set of values simulated from the importance sampling distribution and a field `propLL` for their log probabilities (densities). `savedWeights` will contain the difference in log probability (density) between the model and the `propLL` value provided for each set of values.

```
## Accepting modelValues as a setup argument
setupFunction = function(propModelValues, model){
    ## Building a modelValues in the setup function
    savedWeightsSpec <- modelValuesSpec(vars = 'w',
                                        types = 'double',
                                        sizes = 1)
```

```
    savedWeights <- modelValues(spec = savedWeightsSpec)
    ## List of nodes to be used in run function
    modelNodes <- model$getNodeNames(stochOnly = TRUE,
                                     includeData = FALSE)
}
```

The simplest way to pass values back and forth between models and modelValues inside of a nimbleFunction is with `copy`, which has the synonym `nimCopy`. This takes arguments

`from, to` which can either be models or modelValues

`row, rowTo` which refers to the rows of a modelValues object, if either `from` or `to` is a modelValues. If `rowTo` is omitted, it is assumed to be equal to `row` if necessary.

`nodes, nodesTo` which is a vector of the names of the nodes to be copied. The node names will be expanded when variable names are provided. If `nodesTo` is omitted it will be set equal to `nodes`.

TODO: CHECK THESE USAGES

Alternatively, the values may be accessed via indexing of individual rows, using the notation `mv[var, i]`, where `mv` is a modelValues object, `var` is a variable name (not a node name), and `i` is a row number. Likewise, the `getsize` and `resize` functions can be used as discussed previously. However the function `as.matrix` does not work in `run` code.

Here is a run-time function to use these modelValues:

```
runFunction = function(){
    ## gets the number of rows of propSamples
    m <- getsize(propModelValues)

    ## resized savedWeights to have the proper rows
    resize(savedWeights, m)
    for(i in 1:m){
        ## Copying from propSamples to model.
        ## Node names of propSamples and model must match!
        nimCopy(from = propModelValues, to = model, row = i,
                nodes = modelNodes, logProb = FALSE)
        ## calculates the log likelihood of the model
        targLL <- calculate(model)
        ## retreaves the saved log likelihood from the proposed model
        propLL <- propModelValues['propLL',i][1]
        ## saves the importance weight for the i-th sample
        savedWeights['w', i][1] <<- exp(targLL - propLL)
    }
    ## does not return anything
}
```

Once the nimbleFunction is built, the modelValues object can be accessed using using `$`, which is shown in more detail below. In fact, since modelValues, like most NIMBLE objects,

are reference class objects, one can get a reference to them before the function is executed and then use that reference afterwards.

```
##   Simple model and modelValue for example
targetModelCode <- nimbleCode({
    x ~ dnorm(0,1)
    for(i in 1:4)
        y[i] ~ dnorm(0,1)
})

## Code for proposal model
propModelCode <- nimbleCode({
        x ~ dnorm(0,2)
        for(i in 1:4)
                y[i] ~ dnorm(0,2)
})

## Building R models
targetModel = nimbleModel(targetModelCode)

## Warning:  NAs were detected in model variables:  x, logProb_x, y, logProb_y

propModel = nimbleModel(propModelCode)

## Warning:  NAs were detected in model variables:  x, logProb_x, y, logProb_y

cTargetModel = compileNimble(targetModel)
cPropModel = compileNimble(propModel)


sampleMVSpec = modelValuesSpec(vars = c('x', 'y', 'propLL'),
    types = c('double', 'double', 'double'),
    sizes = list(x = 1, y = 4, propLL = 1) )

sampleMV <- modelValues(sampleMVSpec)

##   nimbleFunction for generating proposal sample
PropSamp_Gen <- nimbleFunction(
    setup = function(mv, propModel){
        nodeNames <- propModel$getNodeNames()
    },
    run = function(m = integer() ){
        resize(mv, m)
        for(i in 1:m){
            simulate(propModel)
            nimCopy(from = propModel, to = mv, nodes = nodeNames, row = i)
```

```
            mv['propLL', i][1] <<- calculate(propModel)
        }
    }
    )

## nimbleFunction for calculating importance weights
## Recylcing setupFunction and runFunction as defined in earlier example
impWeights_Gen <- nimbleFunction(setup = setupFunction,
                                 run = runFunction)


## Making instances of nimbleFunctions
## Note that both functions share the same modelValues object
RPropSamp <- PropSamp_Gen(sampleMV, propModel)
RImpWeights <- impWeights_Gen(sampleMV, targetModel)

# Compiling
CPropSamp <- compileNimble(RPropSamp, project = propModel)
CImpWeights <- compileNimble(RImpWeights, project = targetModel)

#Generating and saving proposal sample of size 10
CPropSamp$run(10)

## NULL

## Calculating the importance weights and saving to mv
CImpWeights$run()

## NULL

## Retrieving the modelValues objects
## Extracted objects are C-based modelValues objects

savedPropSamp_1 = CImpWeights$propModelValues
savedPropSamp_2 = CPropSamp$mv

# Subtle note: savedPropSamp_1 and savedPropSamp_2
# both provide interface to the same compiled modelValues objects!
# This is because they were both built from sampleMV.

savedPropSamp_1['x',1]

## [1] 0.1917976

savedPropSamp_2['x',1]
```

```
## [1] 0.1917976

savedPropSamp_1['x',1] <- 0
savedPropSamp_2['x',1]

## [1] 0

## Viewing the saved importance weights
savedWeights <- CImpWeights$savedWeights
unlist(savedWeights[['w']])

##  [1]  0.8944363  3.7901636  0.7639024 10.5478339  2.1690808
##  [6]  1.5026996  0.2507349  0.5803941  0.9772664  1.1481190

#Viewing first 3 rows. Note that savedPropSsamp_1['x', 1] was altered
as.matrix(savedPropSamp_1)[1:3, ]

##      propLL[1]       x[1]       y[1]       y[2]      y[3]
## [1,] -6.104437  0.0000000  0.3665085 -1.2574061 1.1369568
## [2,] -8.992379 -2.2103211  0.1586775 -0.4369688 0.1104045
## [3,] -5.788930 -0.4642081 -0.5126574  1.3408970 0.6180777
##            y[4]
## [1,]  0.4446988
## [2,] -1.0083280
## [3,]  0.5184339
```

Importance sampling could also be written using simple vectors for the weights, but we illustrated putting them in a modelValues object along with model variables.

9.5.5 Using model variables and modelValues in expressions

Each way of accessing a variable, node, or modelValues can be used amid mathematical expressions, including with indexing, or passed to another nimbleFunction as an argument. For example, the following two statements would be valid:

```
model[['x[2:8, ]']][2:4, 1:3] %*% Z
```

if Z is a vector or matrix, and

```
C[6:10] <- mv[v, i][1:5, k] + B
```

if B is a vector or matrix.

The NIMBLE language allows scalars, but BUGS models never have purely scalar nodes. Instead, a single node such as defined by `z ~ dnorm(0, 1)` is implemented as a vector of length 1, similar to R. When using z via `model$z` or `model[['z']]`, NIMBLE will try to do

the right thing by treating this as a scalar. In the event of problems[7], a more explicit way to access z is `model$z[1]` or `model[['z']][1]`.

9.5.6 Getting and setting more than one model node or variable at a time using values

Sometimes it is useful to set a collection of nodes or variables at one time. For example, one might want a nimbleFunction that will serve as the objective function for an optimizer. The input to the nimbleFunction would be a vector, which should be used to fill a collection of nodes in the model before calculating their log probabilities. NIMBLE has two ways to do this, one of which was set up during development and may be deprecated in the future.

The recommended newer way is:

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

where the first line would assign the collection of values from nodes into P, and the second would to the inverse. In both cases, values from nodes with 2 or more dimensions are flattened into a vector in column-wise order.

The older syntax, which may be deprecated in the future, is

```
getValues(P, model, nodes)
setValues(P, model, nodes)
```

These are equivalent to the two previous lines. Note that `getValues` modifies P in the calling environment.

With the new notation, `values(model, nodes)` may appear as a vector in other expressions, e.g. `Y <- A %*% values(model, nodes) + b`.

9.5.7 Basic flow control: if-then-else, for, and while

These basic control flow structures use the same syntax as in R. However, `for`-loops are limited to sequential integer indexing. For example, `for(i in 2:5) {...}` works as it does in R. Decreasing index sequences are not allowed.

We plan to include more flexible `for`-loops in the future, but for now we've included just one additional useful feature: `for(i in seq_along(NFL))` will work as in R, where `NFL` is a nimbleFunctionList. This is described below.

9.5.8 How numeric types work

Numeric types in NIMBLE are much less flexible than in R, a reflection of the fact that NIMBLE code can be compiled into C++[8]. In NIMBLE, the *type* of a numeric object refers

[7] please tell us!

[8] C++ is a statically typed language, which means the type of a variable cannot change.

to the number of dimensions and the numeric type of the elements. In v0.4, objects from 0 (scalar) to 4 dimensions are supported, and the numeric types integer and double are supported. In addition the type logical is supported for scalars only. While the number of dimensions cannot change during run-time, numeric objects can be resized using `setSize` or by full (non-indexed) assignment.

When possible, NIMBLE will determine the type of a variable for you. In other cases you must declare the type. The rules are as follows:

- For numeric variables from the `setup` function that appear in the `run` function or other member functions (or are declared in `setupOutputs`): the type is determined from the values created by the `setup` code. The types created by `setup` code must be consistent across all specializations of the nimbleFunction. For example if X is created as a matrix (2-dimensional double) in one specialization but as a vector (1-dimensional double) in another, there will be a problem during compilation. The sizes may differ in each specialization.
  Treatment of vectors of length 1 presents special challenges because they could be treated as scalars or vectors. Currently they are treated as scalars. If you want a vector, ensure that the length is greater than 1 in the setup code and then use `setSize` in the run-time code.
- In `run` code, when a numeric variable is created by assignment, its type is determined by that assignment. Subsequent uses of that variable must be consistent with that type.
- If the first uses of a variable involves indexing, the type must be declared explicitly, using `declare`, before using it. In addition, its size must be set before assigning into it. Sizes can be included in the declare() statement, but if so they should not subsequently change. If a variable may have its size changed during execution, then the declare statement should omit the size argument, and a separate call to `setSize` should be used to set the initial size(s).

9.5.9 Querying and changing sizes

Sizes can be queried as follows:

- `length` behaves like R's `length` function. It returns the *entire* length of X. That means if X is multivariate, `length` returns the product of the sizes in each dimension.
- `dim`, which has synonym `nimDim`, behaves like R's `dim` function for matrices or arrays, and like R's `length` function for vectors. In other words, regardless of whether the number of dimensions is 1 or more, it returns a vector of the sizes. Using `dim` vs. `nimDim` is a personal choice, but if you use `dim`, you use it you should keep in mind that it behaves differently from R's `dim`.
  - A quirky limitation in v0.4: It not currently possible to assign the results from `nimDim` to another object using vector assignment. So the only practical way to use `nimDim` is to extract elements immediately, such as `nimDim(X)[1]`, `nimDim(X)[2]`, etc.

Sizes can be changed using:

- setSize(X, sizes)

where `sizes` is a scalar if `X` is 1-dimensional and uses R's `c()` function to provide a vector of sizes if X is more than 1-dimensional.

9.5.10 Basic math and linear algebra

NIMBLE uses the *Eigen* library in C++ to accomplish linear algebra. In v0.4, we use a lot of Eigen's capabilities, but not all of them. The supported operations are given in tables 5.3-5.4.

No vectorized operations other than assignment are supported for more than two dimensions in v0.4. That means `A = B + C` will work only if B and C have dimensions $\leq$ 2.

Managing dimensions and sizes: asRow, asCol, and dropping dimensions

It can be tricky to determine the dimensions returned by a linear algebra expression. As much as possible, NIMBLE behaves like R, but in some cases this is not possible because R uses run-time information while NIMBLE must determine dimensions at compile-time.

Suppose v1 and v2 are vectors, and M1 is a matrix. Then

- `v1 + M1` generates a compilation error unless one dimension of M1 is known at compile-time to be 1. If so, then v1 is promoted to a 1-row or 1-column matrix to conform with M1, and the result is a matrix of the same sizes. This behavior occurs for all component-wise binary functions.
- `v1 %*% M1` defaults to promoting v1 to a 1-row matrix, unless it is known at compile-time that M1 has 1 row, in which case v1 is promoted to a 1-column matrix.
- `M1 %*% v1` defaults to promoting v1 to a 1-column matrix, unless it is known at compile time that M1 has 1 column, in which case v1 is promoted to a 1-row matrix.
- `v1 %*% v2` promotes v1 to a 1-row matrix and v2 to a 1-column matrix, so the returned values is a 1x1 matrix with the inner product of v1 and v2.
- `asRow(v1)` explicitly promotes v1 to a 1-row matrix. Therefore `v1 %*% asRow(v2)` gives the outer product of v1 and v2.
- `asCol(v1)` explicitly promotes v1 to a 1-column matrix.
- The default promotion for a vector is to a 1-column matrix. Therefore, `v1 %*% t(v2)` is equivalent to `v1 %*% asRow(v2)` .
- When indexing, dimensions with scalar indices will be dropped. For example, `M1[1,]` and `M1[,1]` are both vectors.
- The left-hand side of an assignment can use indexing, but if so it must already be correctly sized for the result. For example, `Y[5:10, 20:30] <- model$x` will not work – and could crash your R session with a segmentation fault – if Y is not already at least 10x30 in size.

Here are some examples to illustrate the above points, assuming M2 is a square matrix.

- `Y <- v1 + M2 %*% v2` will return a 1-column matrix. If Y is created by this statement, it will be a 2-dimensional variable. If Y already exists, it must already be 2-dimesional, and it will be automatically re-sized for the result.
- `Y <- v1 + (M2 %*% v2)[,1]` will return a vector. Y will either be created as a vector or must already exist as a vector and will be re-sized for the result.

Size warnings and the potential for crashes

For matrix algebra, NIMBLE cannot ensure perfect behavior because sizes are not known until run-time. Therefore, it is possible for you to write code that will crash your R session. In v0.4, NIMBLE attempts to issue warning if sizes are not compatible, but it does not halt execution. Therefore, if you execute `A <- M1 %*% M2`, and M1 and M2 are not compatible for matrix multiplication, NIMBLE will output a warning that the number of rows of M1 does not match the number of columns of M2. After that warning the statement will be executed and may result in a crash. Another easy way to write code that will crash is to do things like `Y[5:10, 20:30] <- model$x` without ensuring Y is at least 10x30. In the future we hope to prevent crashes, but in v0.4 we limit ourselves to trying to provide useful information.

9.5.11 Including other methods in a nimbleFunction

Other methods can be included with the `methods` argument to `nimbleFunction`. These methods can use the objects created in `setup` code in just the same ways as the `run` function. In fact, the `run` function is just a default main method name.

```
methodsDemo <- nimbleFunction(
    setup = function() {sharedValue <- 1},
    run = function(x = double(1)) {
        print('sharedValues = ', sharedValue, '\n')
        increment()
        print('sharedValues = ', sharedValue, '\n')
        A <- times(5)
        return(A * x)
        returnType(double(1))
    },
    methods = list(
        increment = function() {
            sharedValue <<- sharedValue + 1
        },
        times = function(factor = double()) {
            return(factor * sharedValue)
            returnType(double())
        }))

methodsDemo1 <- methodsDemo()
methodsDemo1$run(1:10)

## sharedValues = 1
##
## sharedValues = 2
##  [1]  10  20  30  40  50  60  70  80  90 100
```

```
methodsDemo1$sharedValue <- 1
CmethodsDemo1 <- compileNimble(methodsDemo1)
CmethodsDemo1$run(1:10)

##  [1]  10  20  30  40  50  60  70  80  90 100
```

9.5.12 Using other nimbleFunctions

One nimbleFunction can use another nimbleFunction that was passed to it as a setup argument or was created in the setup function. This can be an effective way to program. When a nimbleFunction needs to access a setup variable or method of another nimbleFunction, use `$`.

TODO: DEMONSTRATE DIRECT ACCESS to sharedValues in this demo

```
usePreviousDemo <- nimbleFunction(
    setup = function(initialSharedValue) {
        myMethodsDemo <- methodsDemo()
    },
    run = function(x = double(1)) {
        myMethodsDemo$sharedValue <<- initialSharedValue
        A <- myMethodsDemo$run(x[1:5])
        print(A)
        B <- myMethodsDemo$times(10)
        return(B)
        returnType(double())
    })

usePreviousDemo1 <- usePreviousDemo(2)
usePreviousDemo1$run(1:10)

## sharedValues = 2
##
## sharedValues = 3
##
## 15 30 45 60 75
## [1] 30

CusePreviousDemo1 <- compileNimble(usePreviousDemo1)
CusePreviousDemo1$run(1:10)

## [1] 30
```

Note that the output from the `print` calls in the compiled function match those from the uncompiled function when run in an R session. It is not shown here because this document

is created with `knitr` and for some reason output printed from C++ does not make it into `knitr` output.

9.5.13 Virtual nimbleFunctions and nimbleFunctionLists

Often it is useful for one nimbleFunction to have a list of other nimbleFunctions that have methods with the same arguments and return types. For example, NIMBLE's MCMC contains a list of samplers that are each nimbleFunctions.

To make such a list, NIMBLE provides a way to declare the arguments and return types of methods: virtual nimbleFunctions created by `nimbleFunctionVirtual`. Other nimbleFunctions can inherit from virtual nimbleFunctions, which in R is called "containing" them. Readers familiar with object oriented programming will recognize this as a simple class inheritance system. In v0.4 it is limited to simple, single-level inheritance.

Here is how it works:

```
baseClass <- nimbleFunctionVirtual(
    run = function(x = double(1)) {returnType(double())},
    methods = list(
        foo = function() {returnType(double())}
    ))

derived1 <- nimbleFunction(
    contains = baseClass,
    setup = function(){},
    run = function(x = double(1)) {
        print('run 1')
        return(sum(x))
        returnType(double())
    },
    methods = list(
        foo = function() {
        print('foo 1')
        return(rnorm(1, 0, 1))
        returnType(double())
    }))

derived2 <- nimbleFunction(
    contains = baseClass,
    setup = function(){},
    run = function(x = double(1)) {
        print('run 2')
        return(prod(x))
        returnType(double())
    },
    methods = list(
```

```
        foo = function() {
        print('foo 2')
        return(runif(1, 100, 200))
        returnType(double())
    }))

useThem <- nimbleFunction(
    setup = function() {
        nfl <- nimbleFunctionList(baseClass)
        nfl[[1]] <- derived1()
        nfl[[2]] <- derived2()
        val <- 0
    },
    run = function(x = double(1)) {
        for(i in seq_along(nfl)) {
            print( nfl[[i]]$run(x) )
            print( nfl[[i]]$foo() )
        }
    }
    )

useThem1 <- useThem()
set.seed(0)
useThem1$run(1:5)

## run 1
## 15
## foo 1
## 1.262954
## run 2
## 120
## foo 2
## 137.2124

CuseThem1 <- compileNimble(useThem1)
set.seed(0)
CuseThem1$run(1:5)

## NULL
```

As in R, the `seq_along` function is equivalent to `1:length(nimFunList)` if `length(nimFunList)` > 0, and it is an empty sequence if `length(nimFunList)` $== 0$.

Currently `seq_along` works only for nimbleFunctionLists.

Virtual nimbleFunctions cannot define `setup` values to be inherited.

9.5.14 print and stop

As demonstrated above, the NIMBLE function `print`, or equivalently `nimPrint`, prints an arbitrary set of outputs in order. The NIMBLE function `stop`, or equivalently `nimStop`, throws control to R's error-handling system and can take one string (character) argument.

9.5.15 Character objects

NIMBLE provides limited uses of character objects in `code`. Character vectors created in `setup` code will be available in `run` code, but the only thing you can really do with them is include them in a `print` or `stop` statement.

Note that character vectors of model node and variable names are completely different. For example, in `model[[node]]`, `node` may be a character object, and the NIMBLE compiler processes this differently than `print(‘‘The node name was ’’, node)`. In the former, the NIMBLE compiler sets up a C++ pointer directly to the `node` in the `model`, so that the character content of `node` is never needed in C++. In the latter, `node` is used as a C++ string and therefore is needed in C++.

9.5.16 Alternative keywords for some functions

NIMBLE uses some keywords, such as `dim` and `print`, in ways similar but not identical to R. In addition, there are some keywords in NIMBLE that have the same names as really different R functions. For example, `step` is part of the BUGS language, but it is also an R function for stepwise model selection. And `equals` is part of the BUGS language but is also used in the `testthat` package, which we use in testing NIMBLE.

The way NIMBLE handles this to try to avoid conflicts is to replace some keywords immediately upon creating a nimbleFunction. These replacements include

- `copy` → `nimCopy`
- `dim` → `nimDim`
- `print` → `nimPrint`
- `step` → `nimStep`
- `equals` → `nimEquals`
- `round` → `nimRound`
- `stop` → `nimStop`

This system gives programmers the choice between using the keywords like `nimPrint` directly, to avoid confusion in their own code about which "print" is being used, or to use the more intuitive keywords like `print` but remember that they are not the same as R's functions.

9.5.17 User-defined data structures

NIMBLE does not explicitly have user-defined data structures, but one can use nimbleFunctions to achieve a similar effect. To do so, one can define setup code with whatever variables are wanted and ensure they are compiled using `setupOutputs`. Here is an example:

TODO: FIX THIS EXAMPLE

You'll notice that

- after execution of the compiled function, access to the `X`, `Y`, and `Z` is not yet quite the same as for the uncompiled case.
- We need to take care that at the time of compilation, the `X`, `Y` and `Z` values contains doubles via `as.numeric` so that they are not compiled as integer objects.
- The `myDataNF` could be created in the setup code. We just provided it as a setup argument to illustrate that option.

9.5.18 Distribution functions

Distribution "d", "r", "p", and "q" functions can all be used from nimbleFunctions, but the care is needed with the order of arguments. For distributions available in R, nimbleFunction parameterizations match those of R, *which may differ in default order from the same-named BUGS distributions*. You can provide named arguments to be safe. For distributions available in BUGS but not R, nimbleFunctions currently support the default BUGS parameterization but not alternatives.

Supported distributions include (TODO: IS THIS UP TO DATE? SHOULD WE INCLUDE ALTERNATIVE ARGUMENT NAMES?):

- `dbinom(size, prob)`
- `dcat(prob)`
- `dmulti(size, prob)`
- `dnbinom(prob, size)`
- `dpois(lambda)`
- `dbeta(shape1, shape2)`
- `dchisq(df)`
- `dexp(rate)`
- `dgamma(shape, scale)`
- `dlnorm(meanlog, sdlog)`
- `dlogis(location, scale)`
- `dnorm(mean, sd)`
- `dt_nonstandard(df, mu, sigma)`
- `dweibull(shape, scale)`
- `ddirch(alpha)`
- `dmnorm_chol(mean, chol, prec_param)`
- `dwish_chol(chol, df, scale_param)`

In the last two, chol stands for Cholesky decomposition; prec_param indicates whether the Cholesky is of a precision matrix or covariance matrix; and scale_param indicates whether the Cholesky is of a scale matrix or an inverse scale matrix.

In addition, user-defined distributions can also be used in nimbleFunctions.

TODO: THIS SEEMS REDUNDANT: Arguments to the distributions will be matched by either position, or if named, by name. If omitted, default argument values based on the standard R distribution functions will be used. User-supplied distributions are handled analogously with regard to matching by position and use of defaults (when provided via the nimbleFunction run-time arguments) (Section 5.2.5).

Note that when calling the "r" function, `n` must be 1 and must be included.

9.5.19 rankSample

TODO: Need to write.

9.5.20 DSL vectors and arrays

TODO: FILL THIS IN
DSL functions: nimVector(), nimArray().
Can use in run function:
nimVector(value, length)
nimArray(value, nrow, ncol)