

# NIMBLE User Manual

NIMBLE Development Team

Version 0.6-3



[R-nimble.org](http://R-nimble.org)

# Contents

<b>I</b>	<b>Introduction and Building Models in NIMBLE</b>	<b>5</b>
<b>1</b>	<b>Welcome to NIMBLE</b>	<b>6</b>
1.1	What does NIMBLE do? . . . . .	6
1.2	How to use this manual . . . . .	7
<b>2</b>	<b>Lightning introduction</b>	<b>8</b>
2.1	A brief example . . . . .	8
2.2	Creating a model . . . . .	8
2.3	Compiling the model . . . . .	9
2.4	Creating, compiling and running a basic MCMC configuration . . . . .	9
2.5	Customizing the MCMC . . . . .	10
2.6	Running MCEM . . . . .	10
2.7	Creating your own functions . . . . .	10
<b>3</b>	<b>More introduction</b>	<b>12</b>
3.1	NIMBLE adopts and extends the BUGS language for specifying models . . . .	12
3.2	nimbleFunctions for writing algorithms . . . . .	13
3.3	The NIMBLE algorithm library . . . . .	14
<b>4</b>	<b>Installing NIMBLE</b>	<b>16</b>
4.1	Requirements to run NIMBLE . . . . .	16
4.2	Installing a C++ compiler for NIMBLE to use . . . . .	16
4.2.1	OS X . . . . .	17
4.2.2	Linux . . . . .	17
4.2.3	Windows . . . . .	17
4.3	Installing the NIMBLE package . . . . .	17
4.3.1	Problems with installation . . . . .	18
4.4	Customizing your installation . . . . .	18
4.4.1	Using your own copy of Eigen . . . . .	18
4.4.2	Using libnimble . . . . .	18
4.4.3	LAPACK and BLAS . . . . .	19
4.4.4	Customizing compilation of the NIMBLE-generated C++ . . . . .	19

## II Models in NIMBLE 20

### 5 Writing models in NIMBLE's dialect of BUGS 21

5.1	Comparison to BUGS dialects supported by WinBUGS, OpenBUGS and JAGS	21
5.1.1	Supported features of BUGS and JAGS	21
5.1.2	NIMBLE's Extensions to BUGS and JAGS	21
5.1.3	Not-yet-supported features of BUGS and JAGS	22
5.2	Writing models	22
5.2.1	Declaring stochastic and deterministic nodes	23
5.2.2	More kinds of BUGS declarations	23
5.2.3	Vectorized versus scalar declarations	25
5.2.4	Available distributions	26
5.2.5	Available BUGS language functions	30
5.2.6	Available link functions	32
5.2.7	Truncation, censoring, and constraints	32
5.2.8	Understanding the variables and nodes of a model	35

### 6 Building and using models 36

6.1	Creating model objects	36
6.1.1	Using <code>nimbleModel</code> to create a model	36
6.1.2	Creating a model from standard BUGS and JAGS input files	39
6.1.3	Making multiple instances from the same model definition	40
6.2	NIMBLE models are objects you can query and manipulate	41
6.2.1	Determining the nodes and variables in a model	41
6.2.2	Determining model structure	42

## III Algorithms in NIMBLE 43

### 7 MCMC 44

7.1	The MCMC configuration	45
7.1.1	Default MCMC configuration	45
7.1.2	Customizing the MCMC configuration	47
7.2	Building and compiling the MCMC algorithm	51
7.3	Executing the MCMC algorithm	52
7.4	Extracting MCMC samples	53
7.5	Running multiple MCMC chains	53
7.6	Recording model log-probabilities	55
7.7	Measuring sampler computation times	55
7.8	Sampler Algorithms provided with NIMBLE	55
7.8.1	Conjugate (Gibbs) samplers	56
7.8.2	<code>binary</code> (Gibbs) sampler	57
7.8.3	<code>RW_multinomial</code> sampler	57
7.8.4	<code>RW_dirichlet</code> sampler	57
7.8.5	Scalar Metropolis-Hastings random walk <code>RW</code> sampler	57

7.8.6	Multivariate Metropolis-Hastings <code>RW_block</code> sampler . . . . .	58
7.8.7	<code>slice</code> sampler . . . . .	58
7.8.8	Elliptical slice sampling: <code>ess</code> sampler . . . . .	59
7.8.9	Hierarchical <code>crossLevel</code> sampler . . . . .	59
7.8.10	Customized log likelihood evaluations: <code>RW_llFunction</code> sampler . . . . .	60
7.8.11	Terminal node <code>posterior_predictive</code> sampler . . . . .	61
7.8.12	Particle MCMC <code>PMCMC</code> sampler . . . . .	61
7.9	Detailed MCMC example: <code>litters</code> . . . . .	61
7.10	Comparing different MCMC engines with <code>MCMCsuite</code> and <code>compareMCMCs</code> . . . . .	64
7.10.1	MCMC Suite example: <code>litters</code> . . . . .	65
7.10.2	MCMC Suite outputs . . . . .	65
7.10.3	Customizing MCMC Suite . . . . .	67
<b>8</b>	<b>Sequential Monte Carlo and MCEM</b> . . . . .	<b>69</b>
8.1	Particle Filters / Sequential Monte Carlo . . . . .	69
8.1.1	Filtering Algorithms . . . . .	69
8.1.2	Particle MCMC (PMCMC) . . . . .	72
8.2	Monte Carlo Expectation Maximization (MCEM) . . . . .	73
<b>IV</b>	<b>Programming with NIMBLE</b> . . . . .	<b>75</b>
<b>9</b>	<b>Introduction to programming in NIMBLE</b> . . . . .	<b>76</b>
<b>10</b>	<b>Writing simple nimbleFunctions using the NIMBLE language</b> . . . . .	<b>77</b>
10.1	Introduction to simple nimbleFunctions . . . . .	77
10.2	Using and compiling simple nimbleFunctions . . . . .	77
10.3	The NIMBLE language . . . . .	78
10.3.1	Basics . . . . .	78
10.3.2	How numeric types work in NIMBLE . . . . .	78
10.3.3	Declaring argument types and the return type . . . . .	79
10.3.4	Creating non-scalar variables: <code>numeric</code> , <code>integer</code> , <code>matrix</code> , and <code>array</code> . . . . .	79
10.3.5	Querying sizes . . . . .	82
10.3.6	Basic flow control: <code>if-then-else</code> , <code>for</code> , and <code>while</code> . . . . .	82
10.3.7	Using functions . . . . .	83
10.3.8	Basic math and linear algebra . . . . .	83
10.3.9	Distribution functions . . . . .	86
10.3.10	<code>print</code> and <code>stop</code> . . . . .	87
10.3.11	Checking for user interrupts . . . . .	87
10.3.12	Alternative keywords for some functions . . . . .	87
<b>11</b>	<b>Extending BUGS with user-defined distributions and functions</b> . . . . .	<b>89</b>
11.1	User-defined functions . . . . .	89
11.2	User-defined distributions . . . . .	90

11.2.1	Using <code>registerDistributions</code> for alternative parameterizations and providing other information . . . . .	93
<b>12</b>	<b>Working with NIMBLE models</b>	<b>96</b>
12.1	Variables and nodes in NIMBLE models . . . . .	96
12.1.1	Determining the variables and nodes in a model . . . . .	96
12.1.2	Understanding lifted nodes . . . . .	97
12.2	Accessing information about nodes and variables . . . . .	97
12.2.1	Determining dependencies in a model . . . . .	97
12.2.2	Getting distributional information about a node . . . . .	98
12.2.3	Getting information about a distribution . . . . .	98
12.2.4	Getting distribution parameter values for a node . . . . .	98
12.2.5	Getting distribution bounds for a node . . . . .	99
12.3	Carrying out model calculations . . . . .	100
12.3.1	Core model operations: calculation and simulation . . . . .	100
12.3.2	<code>simNodes</code> , <code>calcNodes</code> , and <code>getLogProbs</code> . . . . .	100
12.3.3	Example: simulating arbitrary collections of nodes . . . . .	102
12.3.4	Accessing log probabilities via <code>logProb</code> variables . . . . .	102
12.4	The <i>modelValues</i> data structure . . . . .	102
12.4.1	Creating <code>modelValues</code> objects . . . . .	103
12.4.2	Accessing contents of <code>modelValues</code> . . . . .	104
<b>13</b>	<b>Writing nimbleFunctions that interact with models</b>	<b>107</b>
13.1	Overview . . . . .	107
13.2	Using and compiling <code>nimbleFunctions</code> . . . . .	108
13.3	Writing setup code . . . . .	108
13.3.1	Useful tools for <code>setup</code> functions . . . . .	108
13.3.2	Accessing and modifying numeric values from <code>setup</code> . . . . .	109
13.3.3	Determining numeric types in <code>nimbleFunctions</code> . . . . .	109
13.3.4	Control of <code>setup</code> outputs . . . . .	109
13.4	Writing run code . . . . .	109
13.4.1	Driving models: <code>calculate</code> , <code>calculateDiff</code> , <code>simulate</code> , <code>getLogProb</code> . . . . .	110
13.4.2	Getting and setting more than one model node or variable at a time using <code>values</code> . . . . .	110
13.4.3	Using <code>modelValues</code> objects . . . . .	111
13.4.4	Using model variables and <code>modelValues</code> in expressions . . . . .	111
13.4.5	Including other methods in a <code>nimbleFunction</code> . . . . .	112
13.4.6	Using other <code>nimbleFunctions</code> . . . . .	112
13.4.7	Virtual <code>nimbleFunctions</code> and <code>nimbleFunctionLists</code> . . . . .	112
13.4.8	Character objects . . . . .	112
13.4.9	User-defined data structures . . . . .	113
13.5	Example: writing user-defined samplers to extend NIMBLE's MCMC engine . . . . .	113
13.6	Copying <code>nimbleFunctions</code> (and NIMBLE models) . . . . .	115
13.7	Debugging <code>nimbleFunctions</code> . . . . .	116
13.8	Some options for reducing memory usage . . . . .	116

# Part I

## Introduction and Building Models in NIMBLE

# Chapter 1

## Welcome to NIMBLE

NIMBLE is a system for building and sharing analysis methods for statistical models from R, especially for hierarchical models and computationally-intensive methods. While NIMBLE is embedded in R, it goes beyond R by supporting separate programming of models and algorithms along with compilation for fast execution.

As of 0.6-3, NIMBLE has been around for a while and is reasonably stable, but we have a lot of plans to expand and improve it. The algorithm library provides MCMC with a lot of user control and ability to write new samplers easily. Other algorithms include particle filtering (sequential Monte Carlo) and Monte Carlo Expectation Maximization (MCEM).

But NIMBLE is about much more than providing an algorithm library. It provides a language for writing model-generic algorithms. We hope you will program in NIMBLE and make an R package providing your method. Of course, NIMBLE is open-source, so we also hope you'll contribute to its development.

Please join the mailing lists ([R-nimble.org](http://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. You can also follow and contribute to developer discussions on the [wiki of our GitHub repository](#).

### 1.1 What does NIMBLE do?

NIMBLE makes it easier to program statistical algorithms that will run efficiently and work on many different models from R.

You can think of NIMBLE as comprising four pieces:

1. A system for writing statistical models flexibly, which is an extension of the BUGS language<sup>1</sup>.
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 models written in BUGS.
4. A compiler that generates C++ for your models and algorithms, compiles that C++, and lets you use it seamlessly from R without knowing anything about C++.

---

<sup>1</sup>See Section 5.1 for information about NIMBLE's version of BUGS.

NIMBLE stands for Numerical Inference for statistical Models for Bayesian and Likelihood Estimation.

Although NIMBLE was motivated by algorithms for hierarchical statistical models, it's useful for other goals too. You could use it for simpler models. And since NIMBLE can automatically compile R-like functions into C++ that use the Eigen library for fast linear algebra, you can use it to program fast numerical functions without any model involved<sup>2</sup>

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.

## 1.2 How to use this manual

We suggest everyone start with the Lightning Introduction.

Then, if you want to jump into using NIMBLE's algorithms without learning about NIMBLE's programming system, go to Part [II](#) to learn how to build your model and Part [III](#) to learn how to apply NIMBLE's built-in algorithms to your model.

If you want to learn about NIMBLE programming (nimbleFunctions), go to Part [IV](#). This teaches how to program user-defined function or distributions to use in BUGS code, compile your R code for faster operations, and write algorithms with NIMBLE. These algorithms could be specific algorithms for your particular model (such as a user-defined MCMC sampler for a parameter in your model) or general algorithms you can distribute to others. In fact the algorithms provided as part of NIMBLE and described in Part [III](#) are written as nimbleFunctions.

---

<sup>2</sup>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 R in a `nimbleFunction` and the NIMBLE compiler turns it into C++.



# 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<sup>1</sup>. We could load the model from the standard BUGS example file formats (Section 6.1.2), 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 configuration for the pump model.
4. Compile and run the MCMC
5. Customize the MCMC configuration 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.

Here `x[i]` is the number of failures recorded during a time duration of length `t[i]` for the  $i^{th}$  pump. `theta[i]` is a failure rate, and the goal is estimate parameters `alpha` and `beta`. Now let's create the model and look at some of its nodes.

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

---

<sup>1</sup>The data set describes failure rates of some pumps.

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).

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 are 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.

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 `pump$simulate('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.

Note that the compiled model is used when running any NIMBLE algorithms via C++, so the model needs to be compiled before (or at the same time as) any compilation of algorithms, such as the compilation of the MCMC done in the next section.

## 2.4 Creating, compiling and running a basic MCMC configuration

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`, on which the default sampler is a random walk Metropolis sampler.

Notice the posterior correlation between `alpha` and `beta`. A measure of the mixing for each is the autocorrelation for each parameter, 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.

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<sup>2</sup>.

Both estimates are within 0.01 of the values reported by George et al. [2]<sup>3</sup>. 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. More details on programming in NIMBLE are in Part IV.

Here is our `nimbleFunction`:

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.

---

<sup>2</sup>Note that for this model, one could analytically integrate over `theta` and then numerically maximize the resulting marginal likelihood.

<sup>3</sup>Table 2 of the paper accidentally swapped the two estimates.

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).

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.

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.

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.

One of the most important concepts behind NIMBLE is to allow a combination of high-level processing in R and low-level processing in 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 C++. This separation between *setup* and *run* steps will become clearer as we go.

### 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 became widely used 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.

NIMBLE extends BUGS by:

1. allowing you to write new functions and distributions and use them in BUGS models;
2. allowing you to define multiple models in the same code using conditionals evaluated when the BUGS code is processed;
3. supporting a variety of more flexible syntax such as R-like named parameters and more general algebraic expressions.

By supporting new functions and distributions, NIMBLE makes BUGS an extensible language, which is a major departure from previous packages that implement BUGS.

We adopted BUGS because it has been so successful, with over 30,000 users by the time they stopped counting [3]. Many papers and books provide BUGS code as a way to document their statistical models. We describe NIMBLE's version of BUGS later. The web sites for WinBUGS, OpenBUGS and JAGS provide other useful documentation on writing models in BUGS. For the most part, if you have BUGS code, you can try NIMBLE.

NIMBLE does several things with BUGS code:

1. NIMBLE creates a *model definition* object that knows everything about the variables and their relationships written in the BUGS code. Usually you'll ignore the *model definition* and let NIMBLE's default options take you directly to the next step.
2. NIMBLE creates a model object<sup>1</sup>. 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. When you're ready, NIMBLE can generate customized C++ code representing the model, compile the C++, load it back into R, and provide a new model object that uses the compiled model internally. We use the word "compile" to refer to all of these steps together.

As an example of how radical a departure NIMBLE is from previous BUGS implementations, consider a situation where you want to simulate new data from a model written in BUGS code. Since NIMBLE creates model objects that you can control from R, simulating new data is trivial. With previous BUGS-based packages, this isn't possible.

More information about specifying and manipulating models is in Chapters 6 and 12.

## 3.2 nimbleFunctions for writing algorithms

NIMBLE provides *nimbleFunctions* for writing functions that can (but don't have to) use BUGS models. The main ways they can use BUGS models are:

1. inspecting the structure of a model, such as determining the dependencies between variables, in order to do the right calculations with each model;
2. accessing values of the model's variables;
3. controlling execution of the model's probability calculations or corresponding simulations;
4. managing *modelValues* data structures for multiple sets of model values and probabilities.

In fact, the calculations of the model are themselves constructed as *nimbleFunctions*, as are the algorithms provided in NIMBLE's algorithm library<sup>2</sup>.

Programming with *nimbleFunctions* involves a fundamental distinction between two stages of processing:

1. A *setup* function within a *nimbleFunction* gives the steps that need to happen only once for each new situation (e.g., for each new model). Typically such steps include inspecting the model's variables and their relationships, such as determining which parts of a model will need to be calculated for a MCMC sampler. *setup* functions are executed in R and never compiled.

---

<sup>1</sup>or multiple model objects

<sup>2</sup>That's why it's easy to use new functions and distributions written as *nimbleFunctions* in BUGS code.

2. One or more *run* functions within a `nimbleFunction` give steps that need to happen multiple times using the results of the *setup* function, such as the iterations of a MCMC sampler. Formally, *run* code is written in the NIMBLE language, which you can think of as a small subset of R along with features for operating models and related data structures. The NIMBLE language is what the NIMBLE compiler can automatically turn into C++ as part of a compiled `nimbleFunction`.

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`. This is most useful for debugging.
2. When you are ready, NIMBLE can generate C++ code, compile it, load it back into R and give you new objects that use the compiled C++ internally. Again, we refer to these steps all together as “compilation.” The behavior of compiled `nimbleFunctions` is usually very similar, but not identical, to their uncompiled counterparts.

More about writing algorithms is in Chapter 13.

If you are familiar with object-oriented programming, you can think of a `nimbleFunction` as a class definition. The *setup* function initializes a new object and *run* functions are class methods. Member data are determined automatically as the objects from a *setup* function needed in *run* functions. If no `setup` function is provided, the `nimbleFunction` corresponds to a simple (compilable) function rather than a class.

### 3.3 The NIMBLE algorithm library

In Version 0.6-3, the NIMBLE algorithm library includes:

1. MCMC with samplers including conjugate (Gibbs), slice, adaptive random walk (with options for reflection or sampling on a log scale), adaptive block random walk, and elliptical slice. You can modify sampler choices and configurations from R before compiling the MCMC. You can also write new samplers as `nimbleFunctions`.
2. A set of particle filter (sequential Monte Carlo) methods including a basic bootstrap filter, auxiliary particle filter, and Liu-West filter.
3. An ascent-based Monte Carlo Expectation Maximization (MCEM) algorithm.
4. A variety of basic algorithms that can be used as programming tools for larger algorithms. These include:
  - (a) A likelihood function for arbitrary parts of any model.
  - (b) Functions to simulate one or many sets of values for arbitrary parts of any model.
  - (c) Functions to calculate the summed log probability (density) for one or many sets of values for arbitrary parts of any model along with stochastic dependencies in the model structure.
  - (d) Other functions described in Chapter 8.

Add references where appropriate..

CJP - so is the only difference between `calculate` and `calcNodes` (and similarly for `simNodes`, `getLogProbNodes`) that `calcNodes` etc. include operations on dependencies? If so, I propose we extend `calculate`, `simulate`, `getLogProb` to have an argument `includeDependencies` with default `FALSE`. PdV - There were a couple of things conflated here, but the question is still valid. The suggestion seems like a slippery slope towards trying to make `calculate` (etc.) smarter, which could get away from the core design ideas.

More about the NIMBLE algorithm library is in Chapter 8.



# Chapter 4

## Installing NIMBLE

### 4.1 Requirements to run NIMBLE

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

The following are required to run NIMBLE.

1. [R](#), of course.
2. The [igraph](#) and [coda](#) R packages.
3. A working C++ compiler that NIMBLE can use from R on your system. There are standard open-source C++ compilers that the R community has already made easy to install. See Section [4.2](#) for instructions. You don't need to know anything about C++ to use NIMBLE. This must be done before installing 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 Installing a C++ compiler for NIMBLE to use

NIMBLE needs a C++ compiler and the standard utility *make* in order to generate and compile C++ for models and algorithms. <sup>1</sup>

---

<sup>1</sup>This differs from most packages, which might need a C++ compiler only when the package is built. If you normally install R packages using `install.packages` on Windows or OS X, the package arrives already built to your system.

### 4.2.1 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](#).

[PdV – Is this link to App Store really stable?](#)

For the compiler to work correctly for OS X, the installed R must be for the correct version of OS X. For example, R for Snow Leopard (OS X version 10.8) will attempt to use an incorrect C++ compiler if the installed OS X is actually version 10.9 or higher.

[PdV – rewrote this – need to check example.](#)

In the somewhat unlikely event you want to install from the source package rather than the CRAN binary package, the easiest approach is to use the source package provided at [R-nimble.org](#). If you do want to install from the source package provided by CRAN, you'll need to install [this gfortran package](#).

### 4.2.2 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
```

### 4.2.3 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 (and follow the urge to update your version of R if you notice it is not the most recent). This installer leads you through several “pages”. We think you can accept the defaults with one exception: check the PATH checkbox (page 5) so that the installer will add the location of the C++ compiler and related tools to your system's PATH, ensuring that R can find them. After you click “Next”, you will get a page with a window for customizing the new PATH variable. You shouldn't need to do anything there, so you can simply click “Next” again.

The checkbox for the “R 2.15+ toolchain” (page 4) must be checked (in order to have *gcc/g++*, *make*, etc. installed). This should be checked by default.

## 4.3 Installing the NIMBLE package

Since NIMBLE is an R package, you can install it in the usual way, via `install.packages('nimble')` in R or using the R CMD INSTALL method if you download the package source directly.

NIMBLE can also be obtained from [the NIMBLE website](#). To install from our website, please see our [Download page](#) for the specific invocation of `install.packages`.

### 4.3.1 Problems with installation

We have tested the installation on the three commonly used platforms – OS X, Linux, Windows<sup>2</sup>. 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](mailto:nimble.stats@gmail.com).

Add links to [nimble-users](#) and [nimble.stats@gmail.com](mailto:nimble.stats@gmail.com)

## 4.4 Customizing your installation

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

### 4.4.1 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_VERSION.tar.gz
```

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/include/eigen3`. It is not the full path to the Eigen directory itself, i.e., NOT `/usr/include/eigen3/Eigen`.

### 4.4.2 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 the library code is compiled once as a linkable library - *libnimble.so*. This single instance of the library is then linked with the code for each generated model. In contrast, the default for Windows and Mac OS X is to compile the library code as a static library - *libnimble.a* - that is compiled into each model's and each algorithm'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. If one would like to enable the linkable library in place of the

---

<sup>2</sup>We've tested NIMBLE on Windows 7, 8 and 10

static library (do this only on Mac OS X and other UNIX variants and not on Windows), one can install the source package with the configuration argument `--enable-dylib` set to true. First obtain the NIMBLE source package (which will have the extension `.tar.gz` from [our website](#) and then install as follows, replacing `VERSION` with the appropriate version number:

```
R CMD INSTALL --configure-args='--enable-dylib=true' nimble_VERSION.tar.gz
```

### 4.4.3 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 available on CRAN for more details on providing a fast BLAS for your R installation.

### 4.4.4 Customizing compilation of the NIMBLE-generated C++

For each model or `nimbleFunction`, NIMBLE can generate and compile C++. To compile generated C++, NIMBLE makes system calls starting with `R CMD SHLIB` and therefore uses 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 as usual.

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. Such an alternative file 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*.

Advanced users can also change their default compilers by editing the *Makevars* file, see Section 1.2.1 of the Writing R Extensions manual available on CRAN.

[PdV – Can we combine the final Makevars point, which I moved here, with the previous paragraph.](#)

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.

# **Part II**

## **Models in NIMBLE**

# Chapter 5

## Writing models in NIMBLE’s dialect of BUGS

Models in NIMBLE are written using a variation on the BUGS language. From BUGS code, NIMBLE creates a model object. This chapter describes NIMBLE’s version of BUGS. The next chapter explains how to build and manipulate model objects.

### 5.1 Comparison to BUGS dialects supported by WinBUGS, OpenBUGS and JAGS

Many users will come to NIMBLE with some familiarity with WinBUGS, OpenBUGS and/or JAGS, so we start by summarizing how NIMBLE is similar and different to those before documenting NIMBLE’s version of BUGS more completely. In general, NIMBLE aims to be compatible with the original BUGS language and also JAGS’ version. However, at this point, there are some features not supported by NIMBLE, and there are some extensions that are planned but not implemented.

#### 5.1.1 Supported features of BUGS and JAGS

1. Stochastic and deterministic<sup>1</sup> 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:

---

<sup>1</sup>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.

1. User-defined functions and distributions – written as `nimbleFunctions` – can be used in model code. See Chapter 11.
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.
4. Linear algebra, including for vectorized calculations of simple algebra, can be used in deterministic declarations.
5. Distribution parameters can be expressions, as in JAGS but not in WinBUGS. Caveat: parameters to *multivariate* distributions (e.g., `dmnorm`) cannot be expressions.
6. Alternative models can be defined from the same model code by using if-then-else statements that are evaluated when the model is defined.
7. 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.
8. More general constraints can be declared using `dconstraint`, which extends the concept of JAGS’ `dinterval`.
9. Link functions can be used in stochastic, as well as deterministic, declarations.<sup>2</sup>
10. Data values can be reset, and which parts of a model are flagged as data can be changed, 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 Chapter 11 for a description of how you could handle some cases with user-defined functions or 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. Multivariate nodes must appear with brackets, even if they are empty. E.g., `x` cannot be multivariate but `x[]` or `x[2:5]` can be.
4. 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[])`, and if the dimensions of the variable are not clearly defined in other declarations, NIMBLE currently requires that the dimensions of `x` be provided when the model object is created (via `nimbleModel`).

## 5.2 Writing models

Here we introduce NIMBLE’s version of BUGS. The WinBUGS, OpenBUGS and JAGS manuals are also useful resources for writing BUGS models, including many examples.

---

<sup>2</sup>But beware of the possibility of needing to set values for “lifted” nodes created by NIMBLE.

### 5.2.1 Declaring stochastic and deterministic nodes

BUGS is a declarative language for graphical (or hierarchical) models. Most programming languages are imperative, which means a series of commands will be executed in the order they are written. A declarative language like BUGS is more like building a machine before using it. Each line declares that a component should be plugged into the machine, but it doesn’t matter in what order they are declared as long as all the right components are plugged in by the end of the code.

The machine in this case is a graphical model<sup>3</sup>. A *node* (sometimes called a *vertex*) holds one value, which may be a scalar or a vector. *Edges* define the relationships between nodes. A huge variety of statistical models can be thought of as graphs.

For example, a simple linear regression with four observations is shown in figure ?? . Each observation, `y[i]`, is a node whose edges say that it follows a normal distribution depending on a predicted value, `predicted.y[i]`, and standard deviation, `sigma`, which are each nodes. Each predicted value is a node whose edges say how it is calculated from `slope`, `intercept`, and one value of an explanatory variable, `x[i]`, which are each nodes.

This graph is created from the following BUGS code:

In this code, stochastic relationships are declared with “`~`” and deterministic relationships are declared with “`<-`”. For example, each `y[i]` follows a normal distribution with mean `predicted.y[i]` and standard deviation `sigma`. Each `predicted.y[i]` is the result of `intercept + slope * x[i]`. The for-loop yields the equivalent of writing four lines of code, each with a different value of `i`. It does not matter in what order the nodes are declared. Imagine that each line of code draws part of figure ?? , and all that matters is that the everything gets drawn in the end. Available distributions, default and alternative parameterizations, and functions are listed in section 5.2.4

An equivalent graph can be created by this BUGS code:

In this case, the `predicted.y[i]` nodes in figure ?? will be created automatically by NIMBLE and will have a different name, generated by NIMBLE.

### 5.2.2 More kinds of BUGS declarations

Here are some examples of valid lines of BUGS code. This code does not describe a sensible or complete model, and it includes some arbitrary indices (e.g. `mvx[8:10, i]`) to illustrate flexibility. Instead the purpose of each line is to illustrate a feature of NIMBLE’s version of BUGS.

```
{
  ## 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)
  ## 3. For-loop and nested indexing
  for(i in 1:N) {
```

---

<sup>3</sup>Technically, a *directed acyclic graph*



```

    for(j in 1:M[i]) {
      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)
}

```

When a variable appears only on the right-hand side, it must be provided in the `data` or `constants` argument to `nimbleModel`, covered in chapter 6.

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. As 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 Chapter 11.
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 Chapter 11.

## More about indexing

Examples of allowed indexing include:

- `x[i]` # a single index
- `x[i:j]` # a range of indices
- `x[i:j,k:1]` # multiple single indices or ranges for higher dimensional arrays

- `x[i:j, ]` # blank indices indicating the full range
- `x[3*i+7]` # computed indices
- `x[(3*i):(5*i+1)]` # computed lower and upper ends of an index range

NIMBLE does not allow multivariate nodes to be used 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[])` (if `x` is a vector) or `xbar <- mean(x[,])` (if `x` is a matrix) for NIMBLE.<sup>4</sup> Ways to provide NIMBLE with dimensions of `x` in such cases, if necessary, is covered in section ??.

Generally NIMBLE supports R-like linear algebra expressions and attempts to follow the same rules as R about dimensions (although in some cases this is not possible). For example, `x[1:3] %*% y[1:3]` converts `x[1:3]` into a row vector and thus computes the inner product, which is returned as a  $1 \times 1$  matrix (use `inprod` to get it as a scalar, which is typically easier). Like in R, a scalar index will result in dropping a dimension unless the argument `drop=FALSE` is provided. For example, `mymatrix[i, 1:3]` will be a vector of length 3, but `mymatrix[i, 1:3, drop=FALSE]` will be a  $1 \times 3$  matrix. More about indexing and dimensions is discussed in section (10.3.8).

### 5.2.3 Vectorized versus scalar declarations

Suppose you need nodes `logY[i]` that should be the log of the corresponding `Y[i]`, say for `i` from 1 to 10. Conventionally this would be created with a for loop:

```
{
  for(i in 1:10) {
    logY[i] <- log(Y[i])
  }
}
```

Since NIMBLE supports R-like algebraic expressions, an alternative in NIMBLE's dialect of BUGS is to use a vectorized declaration like this:

```
{
  logY[1:10] <- log(Y[1:10])
}
```

Supported functions from BUGS and JAGS are listed in Section 5.2.5. Support for more general R expressions is covered in Chapter 13 about programming with `nimbleFunctions`.

There is an important difference between the models that are created by the above two methods. The first creates 10 scalar nodes, `logY[1] ... logY[10]`. The second creates one vector node, `logY[1:10]`. If each `logY[i]` is used separately by an algorithm, it may be beneficial to declare them as scalars. If they are all used together, it will often make sense to declare them as a vector.

---

<sup>4</sup>In `nimbleFunctions` explained in later chapters, square brackets with blank indices are not necessary for multivariate objects.

## 5.2.4 Available distributions

### Distributions

NIMBLE supports most of the distributions allowed in BUGS and JAGS. Table 5.1 lists the distributions that are currently supported, with their default parameterizations, which match those of BUGS<sup>5</sup>. NIMBLE also allows one to use alternative parameterizations for a variety of distributions as described next. See section ?? to learn how to write new distributions using `nimbleFunctions`.

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	<code>dbern(prob = p)</code> $0 < p < 1$	$p^x(1-p)^{1-x}$	0	1
Beta	<code>dbeta(shape1 = a, shape2 = b)</code> $a > 0, b > 0$	$\frac{x^{a-1}(1-x)^{b-1}}{\beta(a, b)}$	0	1
Binomial	<code>dbin(prob = p, size = n)</code> $0 < p < 1, n \in \mathbb{N}^*$	$\binom{n}{x} p^x (1-p)^{n-x}$	0	$n$
Categorical	<code>dcat(prob = p)</code> $p \in (\mathbb{R}^+)^N$	$\frac{p_x}{\sum_i p_i}$	1	$N$
Chi-square	<code>dchisq(df = k)</code> $k > 0$	$\frac{x^{\frac{k}{2}-1} \exp(-x/2)}{2^{\frac{k}{2}} \Gamma(\frac{k}{2})}$	0	
Dirichlet	<code>ddirch(alpha = <math>\alpha</math>)</code> $\alpha_j \geq 0$	$\frac{\prod_j x_j^{\alpha_j-1}}{\Gamma(\sum_i \alpha_i) \prod_j \Gamma(\alpha_j)}$	0	
Exponential	<code>dexp(rate = <math>\lambda</math>)</code> $\lambda > 0$	$\lambda \exp(-\lambda x)$	0	
Gamma	<code>dgamma(shape = r, rate = <math>\lambda</math>)</code> $\lambda > 0, r > 0$	$\frac{\lambda^r x^{r-1} \exp(-\lambda x)}{\Gamma(r)}$	0	
Logistic	<code>dlogis(location = <math>\mu</math>, rate = <math>\tau</math>)</code> $\tau > 0$	$\frac{\tau \exp\{(x-\mu)\tau\}}{[1 + \exp\{(x-\mu)\tau\}]^2}$		
Log-normal	<code>dlnorm(meanlog = <math>\mu</math>, tauolog = <math>\tau</math>)</code> $\tau > 0$	$\left(\frac{\tau}{2\pi}\right)^{\frac{1}{2}} x^{-1} \exp\{-\tau(\log(x) - \mu)^2/2\}$	0	
Multinomial	<code>dmulti(prob = p, size = n)</code> $\sum_j x_j = n$	$n! \prod_j \frac{p_j^{x_j}}{x_j!}$		
Multivariate normal	<code>dnorm(mean = <math>\mu</math>, prec = <math>\Lambda</math>)</code> $\Lambda$ positive definite	$(2\pi)^{-\frac{d}{2}}  \Lambda ^{\frac{1}{2}} \exp\left\{-\frac{(x-\mu)^T \Lambda (x-\mu)}{2}\right\}$		
Multivariate Student t	<code>dmvt(mu = <math>\mu</math>, prec = <math>\Lambda</math>, df = <math>\nu</math>)</code> $\Lambda$ positive definite	$\frac{\Gamma(\frac{\nu+d}{2})}{\Gamma(\frac{\nu}{2})(\nu\pi)^{d/2}}  \Lambda ^{1/2} \left(1 + \frac{(x-\mu)^T \Lambda (x-\mu)}{\nu}\right)^{-\frac{\nu+d}{2}}$		
Negative binomial	<code>dnegbin(prob = p, size = r)</code> $0 < p \leq 1, r \geq 0$	$\binom{x+r-1}{x} p^r (1-p)^x$		0
Normal	<code>dnorm(mean = <math>\mu</math>, tau = <math>\tau</math>)</code>	$\left(\frac{\tau}{2\pi}\right)^{\frac{1}{2}} \exp\{-\tau(x-\mu)^2/2\}$		

<sup>5</sup>Note that the same distributions are available for writing `nimbleFunctions`, but in that case the default parameterizations and function names match R’s when possible. Please see Section 10.3.9 for how to use distributions in `nimbleFunctions`.

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
	$\tau > 0$			
Poisson	<code>dpois(lambda = <math>\lambda</math>)</code> $\lambda > 0$	$\frac{\exp(-\lambda)\lambda^x}{x!}$	0	
Student t	<code>dt(mu = <math>\mu</math>, tau = <math>\tau</math>, df = <math>k</math>)</code> $\tau > 0, k > 0$	$\frac{\Gamma(\frac{k+1}{2})}{\Gamma(\frac{k}{2})} \left(\frac{\tau}{k\pi}\right)^{\frac{1}{2}} \left\{1 + \frac{\tau(x-\mu)^2}{k}\right\}^{-\frac{(k+1)}{2}}$		
Uniform	<code>dunif(min = <math>a</math>, max = <math>b</math>)</code> $a < b$	$\frac{1}{b-a}$	$a$	$b$
Weibull	<code>dweib(shape = <math>v</math>, lambda = <math>\lambda</math>)</code> $v > 0, \lambda > 0$	$v\lambda x^{v-1} \exp(-\lambda x^v)$	0	
Wishart	<code>dwish(R = <math>R</math>, df = <math>k</math>)</code> $R$ $p \times p$ pos. def., $k \geq p$	$\frac{ x ^{(k-p-1)/2}  R ^{k/2} \exp\{-\text{tr}(Rx/2)\}}{2^{pk/2} \Gamma_p(k/2)}$		

### Alternative parameterizations for distributions

NIMBLE allows one to specify distributions in model code using a variety of parameterizations, including the BUGS parameterizations. Available parameterizations is 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<sup>6</sup>. 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 `gamma`, NIMBLE provides both (shape, rate) and (mean, sd) parameterization and creates nodes to calculate (shape, scale) from either (shape, rate) or (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).

<sup>6</sup>Usually this is the parameterization in the `Rmath` header of R’s C implementation of distributions.

Table 5.2: Distribution parameterizations allowed in NIMBLE. The first column indicates the supported parameterizations for distributions given in Table 5.1. The second column indicates the relationship to the *canonical* parameterization used in NIMBLE.

Parameterization	NIMBLE re-parameterization
dbern(prob)	dbin(size = 1, prob)
dbeta(shape1, shape2)	canonical
dbeta(mean, sd)	dbeta(shape1 = mean <sup>2</sup> * (1-mean) / sd <sup>2</sup> - mean, shape2 = mean * (1 - mean) <sup>2</sup> / sd <sup>2</sup> + mean - 1)
dbin(prob, size)	canonical
dcat(prob)	canonical
dchisq(df)	canonical
ddirch(alpha)	canonical
dexp(rate)	canonical
dexp(scale)	dexp(rate = 1/scale)
dgamma(shape, scale)	canonical
dgamma(shape, rate)	dgamma(shape, scale = 1 / rate)
dgamma(mean, sd)	dgamma(shape = mean <sup>2</sup> /sd <sup>2</sup> , scale = sd <sup>2</sup> /mean)
dlogis(location, scale)	canonical
dlogis(location, rate)	dlogis(location, scale = 1 / rate)
dlnorm(meanlog, sdlog)	canonical
dlnorm(meanlog, tauog)	dlnorm(meanlog, sdlog = 1 / sqrt(tauog))
dlnorm(meanlog, varlog)	dlnorm(meanlog, sdlog = sqrt(varlog))
dmulti(prob, size)	canonical
dmnorm(mean, cholesky, prec_param=1)	canonical (precision)
dmnorm(mean, cholesky, prec_param=0)	canonical (covariance)
dmnorm(mean, prec)	dmnorm(mean, cholesky = chol(prec), prec_param=1)
dmnorm(mean, cov)	dmnorm(mean, cholesky = chol(cov), prec_param=0)
dmvt(mu, cholesky, df, prec_param=1)	canonical (precision/inverse scale)
dmvt(mu, cholesky, df, prec_param=0)	canonical (scale)
dmvt(mu, prec, df)	dmvt(mu, cholesky = chol(prec), df, prec_param=1)
dmvt(mu, scale, df)	dmvt(mu, cholesky = chol(scale), df, prec_param=0)
dnegbin(prob, size)	canonical
dnorm(mean, sd)	canonical
dnorm(mean, tau)	dnorm(mean, sd = 1 / sqrt(var))
dnorm(mean, var)	dnorm(mean, sd = sqrt(var))
dpois(lambda)	canonical
dt(mu, sigma, df)	canonical
dt(mu, tau, df)	dt(mu, sigma = 1 / sqrt(tau), df)
dt(mu, sigma2, df)	dt(mu, sigma = sqrt(sigma2), df)
dunif(min, max)	canonical
dweib(shape, scale)	canonical
dweib(shape, rate)	dweib(shape, scale = 1 / rate)
dweib(shape, lambda)	dweib(shape, scale = lambda <sup>(- 1 / shape)</sup> )
dwish(R, df)	dwish(cholesky = chol(R), df, scale_param = 0)

Table 5.2: Distribution parameterizations allowed in NIMBLE. The first column indicates the supported parameterizations for distributions given in Table 5.1. The second column indicates the relationship to the *canonical* parameterization used in NIMBLE.

Parameterization	NIMBLE re-parameterization
<code>dwish(S, df)</code>	<code>dwish(cholesky = chol(S), df, scale_param = 1)</code>

Note that for multivariate normal, multivariate t, and Wishart, the canonical parameterization uses the Cholesky decomposition of one of the precision/inverse scale or covariance/scale matrix. For example, for the multivariate normal, if `prec_param=TRUE`, the `cholesky` argument is treated as the Cholesky decomposition of a precision matrix. Otherwise it is treated as the Cholesky decomposition of a covariance matrix.

In addition, NIMBLE supports alternative distribution names, known as aliases, as in JAGS, as specified in Table 5.3.

Distribution	Canonical name	Alias
Binomial	<code>dbin</code>	<code>dbinom</code>
Chi-square	<code>dchisq</code>	<code>dchisqr</code>
Dirichlet	<code>ddirch</code>	<code>ddirich</code>
Multinomial	<code>dmulti</code>	<code>dmultinom</code>
Negative binomial	<code>dnegbin</code>	<code>dnbinom</code>
Weibull	<code>dweib</code>	<code>dweibull</code>
Wishart	<code>dwish</code>	<code>dwishart</code>

Table 5.3: Distributions with alternative names (aliases).

We plan to, but do not currently, include the following distributions as part of core NIMBLE: double exponential (Laplace), beta-binomial, Dirichlet-multinomial, F, inverse gamma, Pareto, inverse Wishart, or forms of the multivariate t other than the standard one provided.

### 5.2.5 Available BUGS language functions

Tables 5.4-5.5 show the available operators and functions. **This table needs updating.**

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. Note that we provide distribution functions for use in calculations, namely the “p”, “q”, and “d” functions. See Section 10.3.9 for details on the syntax for using distribution functions as functions in deterministic calculations, as only some parameterizations are allowed and the names of some distributions differ from those used to define stochastic nodes in a model.

Table 5.4: 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
$x \mid y, x \& y$	logical OR ( ) and AND(&)		✓	
$!x$	logical not		✓	
$x > y, x \geq y$	greater than (and or equal to)		✓	
$x < y, x \leq y$	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	✓	✓
$x^y, \text{pow}(x, y)$	power	$x^y$ ; vector $x$ and scalar $y$ ok	✓	✓
$x \% y$	modulo (remainder)		✓	
$\text{min}(x1, x2), \text{max}(x1, x2)$	min. (max.) of two scalars		✓	
$\text{exp}(x)$	exponential		✓	✓
$\log(x)$	natural logarithm		✓	✓
$\text{sqrt}(x)$	square root		✓	✓
$\text{abs}(x)$	absolute value		✓	✓
$\text{step}(x)$	step function at 0	0 if $x < 0$ , 1 if $x > 0$	✓	✓
$\text{equals}(x, y)$	equality of two scalars	1 if $x == y$ , 0 if $x != y$	✓	
$\text{cube}(x)$	third power	$x^3$	✓	✓
$\sin(x), \cos(x), \tan(x)$	trigonometric functions		✓	✓
$\text{asin}(x), \text{acos}(x), \text{atan}(x)$	inverse trigonometric functions		✓	✓
$\text{asinh}(x), \text{acosh}(x), \text{atanh}(x)$	inv. hyperbolic trig. functions		✓	✓
$\text{logit}(x)$	logit	$\log(x/(1-x))$	✓	✓
$\text{ilogit}(x), \text{expit}(x)$	inverse logit	$\exp(x)/(1+\exp(x))$	✓	✓
$\text{probit}(x)$	probit (Gaussian quantile)	$\Phi^{-1}(x)$	✓	✓
$\text{iprobit}(x), \text{phi}(x)$	inverse probit (Gaussian CDF)	$\Phi(x)$	✓	✓
$\text{cloglog}(x)$	complementary log log	$\log(-\log(1-x))$	✓	✓
$\text{icloglog}(x)$	inverse complementary log log	$1 - \exp(-\exp(x))$	✓	✓
$\text{ceiling}(x)$	ceiling function	$\lceil x \rceil$	✓	✓

Table 5.4: 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
<code>floor(x)</code>	floor function	$\lfloor (x) \rfloor$	✓	✓
<code>round(x)</code>	round to integer		✓	✓
<code>trunc(x)</code>	truncation to integer		✓	✓
<code>lgamma(x)</code> , <code>loggam(x)</code>	log gamma function	$\log \Gamma(x)$	✓	✓
<code>log1p(x)</code>	log of $1 + x$	$\log(1 + x)$	✓	✓
<code>lfactorial(x)</code> , <code>logfact(x)</code>	log factorial	$\log x!$	✓	✓
<code>log1p(x)</code>	log one-plus	$\log(x + 1)$	✓	✓
<code>qDIST(x, PARAMS)</code>	“q” distribution functions	canonical parameterization	✓	
<code>pDIST(x, PARAMS)</code>	“p” distribution functions	canonical parameterization	✓	
<code>rDIST(1, PARAMS)</code>	“r” distribution functions	canonical parameterization	✓	
<code>dDIST(x, PARAMS)</code>	“d” distribution functions	canonical parameterization	✓	
<code>sort(x)</code>				
<code>rank(x, s)</code>				
<code>ranked(x, s)</code>				
<code>order(x)</code>				

This table needs updating, at least for `eigen` and `svd`.

Table 5.5: Functions operating on vectors and matrices. Status column indicates if the function is currently provided in NIMBLE.

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



### 5.2.6 Available link functions

NIMBLE allows the link functions listed in Table 5.6.

Table 5.6: Link functions

Link function	Description	Range	Inverse
<code>cloglog(y) &lt;- x</code>	Complementary log log	$0 < y < 1$	<code>y &lt;- icloglog(x)</code>
<code>log(y) &lt;- x</code>	Log	$0 < y$	<code>y &lt;- exp(x)</code>
<code>logit(y) &lt;- x</code>	Logit	$0 < y < 1$	<code>y &lt;- expit(x)</code>
<code>probit(y) &lt;- x</code>	Probit	$0 < y < 1$	<code>y &lt;- iprobit(x)</code>

Link functions are specified as functions applied to a node on the left hand side of a BUGS expression. To handle link functions in deterministic declarations, NIMBLE converts the declaration into an equivalent inverse declaration. For example, `log(y) <- x` is converted into `y <- exp(x)`. In other words, the link function is just a simple variant for conceptual clarity.

To handle link functions in a stochastic declaration, 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`.

Note that NIMBLE does 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.

### 5.2.7 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 BUGS. The three methods are:

#### Truncation

Either of the following forms,

- `x ~ dnorm(0, sd = 10) T(0, a)`, or
- `x ~ T(dnorm(0, sd = 10), 0, a)`,

declares that `x` follows a normal distribution between 0 and `a` (inclusive of 0 and `a`). Either boundary may be omitted or may be another node, such as `a` in this example. The first form

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  $\mathbf{x}$  are limited a priori, hence the probability density of  $\mathbf{x}$  must be normalized. In this example it would be the normal probability density divided by its integral from 0 to  $\mathbf{a}$ . 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 BUGS.

## 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 analysis, when for individuals still surviving at the end of a study, their age of death is not known and hence is “censored” (right-censoring). NIMBLE adopts JAGS syntax for censoring, as follows (using right-censoring as an example):

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

where `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 a value below the observed times for uncensored entries (e.g., 0, assuming `t[i] > 0`). Left-censoring would be specified by setting `censored[i]` to 0 and `t[i]` to `NA`.

The `dinterval` is not really a distribution but rather a trick: in the above example when `censored[i] = 1` it gives a “probability” of 1 if `t[i] > c[i]` and 0 otherwise. This means that `t[i] ≤ c[i]` is treated as impossible. More generally than simple right- or left-censoring, `censored[i] ~ 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 `censored[i]`-th cutpoint interval. This is done by setting data `censored[i]` as follows:

$$\begin{aligned} \text{censored}[i] &= 0 \quad \text{if } t[i] \leq c[i, 1] \\ \text{censored}[i] &= m \quad \text{if } c[i, m] < t[i] \leq c[i, m+1] \text{ for } 1 \leq m \leq M \\ \text{censored}[i] &= M \quad \text{if } c[i, M] < t[i]. \end{aligned}$$

(The `i` index is provided only for consistency with the previous example.) The most common uses 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.

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

Censoring differs from truncation because censoring an observation involves bounds 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.

## Constraints and ordering

NIMBLE provides a more general way to enforce constraints using `dconstraint(cond)`. 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 `constraint_data` set (as `data`) to 1. This is equivalent to a half-normal distribution on the half-plane  $\mu_1 + \mu_2 > 0$ . Nodes following `dconstraint` should be provided as data for the same reason of avoiding unintended initialization described above for `dinterval`.

Formally, `dconstraint(cond)` is a probability distribution on  $\{0, 1\}$  such that  $P(1) = 1$  if `cond` is `TRUE` and  $P(0) = 1$  if `cond` is `FALSE`.

Of course, in many cases, parameterizing the model so that the constraints are automatically respected may be a better strategy than using `dconstraint`. One should be cautious about constraints that would make it hard for an MCMC or optimization to move through the parameter space (such as equality constraints that involve two or more parameters). For such restrictive constraints, general purpose algorithms that are not tailored to the constraints may fail or be inefficient. If constraints are used, it will generally be wise to ensure the model is initialized with values that satisfy them.

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

```
constraint_data ~ 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 does not represent a directed acyclic graph.

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  $\alpha_1 \leq \alpha_2 \leq \alpha_3$ , 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 `alphas` that assigns 0 probability density when values are not in order.

NIMBLE does not support the JAGS `sort` syntax.

### 5.2.8 Understanding the variables and nodes of a model

This section discusses some basic concepts and terminology to be able to speak about NIMBLE models clearly.

Say we have the following BUGS code

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's calculations are handled by a 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]`.

# Chapter 6

## Building and using models

This chapter explains how to build and manipulate model objects starting from BUGS code.

### 6.1 Creating model objects

NIMBLE provides two functions for creating model objects: `nimbleModel` and `readBUGSmodel`. The first, `nimbleModel`, is more general and was illustrated in Chapter 2. The second, `readBUGSmodel` provides compatibility with BUGS file formats for models, variables, data, and initial values for MCMC. In addition one can create new model objects from existing model objects.

#### 6.1.1 Using `nimbleModel` to create a model

`nimbleModel` processes BUGS code to determine all the nodes, variables, and their relationships in a model. Any constants must be provided at this step. Data and initial values can optionally be provided. BUGS code passed to `nimbleModel` must go through `nimbleCode`.

We look again at the pump example from the introduction:

#### Data and constants

NIMBLE makes a distinction between data and constants:

- *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 Chapter 13), 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 in a model.

WinBUGS, OpenBUGS and JAGS do not allow data values to be changed or different nodes to be labeled as data without starting from the beginning again. Hence they do not distinguish between constants and data.

For compatibility with BUGS and JAGS, NIMBLE allows both to be provided in the `constants` argument to `nimbleModel`, in which case NIMBLE handles values for stochastic nodes as data and everything else as constants.

Values for nodes that appear only on the right-hand side of BUGS declarations (e.g., covariates/predictors) can be provided as data or constants.

### Providing data and initial values to an existing model

Whereas constants must be provided during the call the `nimbleModel`, data and initial values can be provided later via the model member functions `setData` and `setInits`. For example, if `pumpData` is a named list of data values (as above), then `pump$setData(pumpData)` sets the named variables to the values in the list.

`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()`.

### 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. 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 configuration 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)

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

pumpMissMCMC <- buildMCMC(pumpMissConf)
Cobj <- compileNimble(pumpMiss, pumpMissMCMC)
```

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

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

Missing values may also occur in explanatory variables. 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 distributions for such explanatory variables so that values are imputed by an MCMC for missing values.

### Defining alternative models with the same code

Avoiding code duplication is a basic principle of good programming. 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 `x[2]` as 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$getVarNames()
includeX2 <- TRUE
modelWithX2 <- nimbleModel(regressionCode, constants = list(N = 30),
                           check = FALSE)
modelWithX2$getVarNames()
```

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*. The `setData` and `setInits` described above can be used for each copy of the model.

### Providing dimensions via `nimbleModel`

`nimbleModel` can usually determine the dimensions of every variable from the declarations in the BUGS code. However, it is possible to use a multivariate object only with empty indices (e.g. `x[,]`), in which case the dimensions must be provided as an argument to `nimbleModel`.

Here's an example with multivariate nodes. 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)),
                    check = FALSE)

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),
                    check = FALSE)
```

### 6.1.2 Creating a model from standard BUGS and JAGS input files

Users with BUGS and JAGS experience may have files set up in standard formats for use in BUGS and JAGS. `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 BUGS and JAGS. In either case, after processing the inputs, it calls `nimbleModel`. Note that unlike BUGS and JAGS, only a single set of initial values can be specified in creating a model. Please see `help(readBUGSmodel)` for argument details.

As an example of 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)
```

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. Also note that if `data` and `inits` are provided as files, the files 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 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.

Note that 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'`.

### 6.1.3 Making multiple instances from the same model definition

Sometimes it is useful to have more than one copy of the same model. For example, an algorithm (i.e., `nimbleFunction`) such as an MCMC will be bound to a particular model before it is run. 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, check = FALSE)

## Make one instance of the model
simpleModelCopy1 <- simpleModelDefinition$newModel(check = FALSE)
## Make another instance from the same definition
simpleModelCopy2 <- simpleModelDefinition$newModel(check = FALSE)
## Ask simpleModelCopy2 for another copy of itself
## small bug in version 0.6-1 means we need to set 'data' to an
## empty list for this to work; this will be fixed in 0.6-2
simpleModelCopy3 <- simpleModelCopy2$newModel(check = FALSE, data =
  list())
```

#### LET'S CHECK THE STATUS OF THE BUG DESCRIBED IN THE COMMENTS.

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; it must be a constant because it helps define the model.

## 6.2 NIMBLE models are objects you can query and manipulate

NIMBLE models are objects that can be modified and manipulated from R. In this section we introduce some basic ways to use a model object. Chapter 12 covers more topics for writing algorithms that use models.

### 6.2.1 Determining the nodes and variables in a model

One can determine the nodes and variables in a model using `getNodeNames`:

Note that some of the nodes may be “lifted” nodes introduced by NIMBLE (Section 12.1.2).

We can determine the set of nodes contained in one or more nodes or variables using `expandNodeNames`. The `returnScalarComponents` argument also allows us to return all of the scalar components of multivariate nodes.

#### Accessing nodes

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

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, such as `model[['z[2:4, 1:2]']][1, 2]`.

**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

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 the `expandNodeNames` method. For example, to get the spaces correctly inserted into “`X[1,1:5]`”:

Alternatively, for those inclined to R’s less commonly used features, a nice trick is to use its `parse` and `deparse` functions.

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

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

**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 Section 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.

## 6.2.2 Determining model structure

### Determining dependencies in a model

One can determine the node dependencies (or “descendants”) in a model using `getDependencies`. There are a variety of arguments to `getDependencies` 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 through the next stochastic node on all edges emanating from the node(s) specified as input. This is what would be needed to calculate a Metropolis-Hastings acceptance probability in MCMC, for example.

### Checking if a node holds data

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

```
pump$isData('x[1]')
pump$isData(c('x[1]', 'x[2]', 'alpha'))
pump$isData('x')
pump$isData('x[1:3]')
```

# Part III

## Algorithms in NIMBLE

# Chapter 7

## MCMC

A NIMBLE MCMC algorithm is built to operate on a particular NIMBLE model object. A high degree of customization is possible for the MCMC algorithm. There are also two different ways an MCMC algorithm can be run, which provide different options, but both result in the collection of posterior samples from model parameters and latent states. We first outline the steps and options available for creating and executing an MCMC algorithm. More details on each step are available through the rest of the chapter.

A NIMBLE model object is necessary to create an MCMC algorithm (see Chapter 6). We'll refer to this object as `model`.

### Creating an MCMC algorithm

A default MCMC algorithm can be created using `buildMCMC(model)`. See Section 7.2.

To customize properties of the MCMC algorithm – including the sampling algorithms, joint (block) sampling of dimensions, tuning and adaptive properties of sampling algorithms, variables being monitored (posterior samples are collected), and the thinning interval for sample collection – an MCMC configuration object (`conf`) must be created using `configureMCMC(model)`. Properties of the MCMC algorithm can be modified using the `conf` object, after which an MCMC algorithm (`mcmc`) can be created using `buildMCMC(conf)`. See Section 7.1.

Both the `model` and `mcmc` should generally be compiled using `compileNimble`, for significantly faster execution (Section 7.2). We'll refer to the compiled MCMC algorithm as `Cmcmc`.

### Executing an MCMC algorithm and collecting samples

There are two methods available to execute an MCMC algorithm:

- `Cmcmc$run(...)`
- `runMCMC(Cmcmc, ...)`

Using `Cmcmc$run(...)` provides lower-level options including extending an MCMC run (collecting additional samples from where it last stopped), and also collecting timing information for the internal sampling algorithms. After executing an MCMC in this manner, poste-

rior samples are extracted from within the `Cmcmc` object using `as.matrix(Cmcmc$mvSamples)`. See Sections 7.3 and 7.4.

Using `runMCMC(Cmcmc, ...)` provides higher-level options such as running multiple chains, setting initial values, removing burn-in, and returning posterior samples in the form of a `coda mcmc` object. Running an MCMC in this manner returns an array of samples, a list of sample arrays in the case of multiple chains, or optionally a `coda` object. See Section 7.5.

## Other topics

In addition to details of the steps outlined above, this chapter also includes:

- Information about the sampling algorithms provided with NIMBLE (Section 7.8)
- A detailed example of using the MCMC engine (section 7.9)
- Methods to automatically run WinBUGS, OpenBUGS, JAGS, Stan and/or multiple NIMBLE MCMCs on the same model (section 7.10)

## 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:

```
mcmcConf <- 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 *MCMCconf* reference class object. The *MCMCconf* 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:

1. If the node has no stochastic dependents, a predictive `posterior_predictive` sampler is assigned. The `posterior_predictive` 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 binary-valued (strictly taking values 0 or 1), then a `binary` (Gibbs) sampler is assigned.
4. If the node follows a multinomial distribution, then a `RW_multinomial` sampler is assigned.
5. If the node is otherwise discrete-valued, then a `slice` sampler is assigned [5].
6. If a node follows a Dirichlet distribution and there is no conjugate relationship with the distributions of its stochastic dependents, then the `RW_dirichlet` sampler is assigned.
7. 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 [6].
8. 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.

The control parameters governing each of the default sampling algorithms are defined in the NIMBLE system option `MCMCcontrolDefaultList`. These default values are described in Section 7.8, 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:

```
autoBlockConf <- 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). Note that this function compiles and runs MCMCs, progressively exploring different sampler assignments, so it takes some time and generates some output.

## 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 `MCMCconf` 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 `MCMCconf` object, and will be used as defaults for any subsequent samplers added to this same `MCMCconf` 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.

```
mcmcConf <- 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 `MCMCconf` object. The first 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 ‘posterior\_predictive’, ‘RW’, ‘RW\_block’, ‘slice’, ‘RW\_multinomial’, ‘RW\_dirichlet’, ‘binary’, ‘ess’, ‘crossLevel’, and ‘RW\_lfFunction’, 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 `printSamplers`.

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 call to `configureMCMC`
3. Those supplied in the NIMBLE system option `MCMCcontrolDefaultList`

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.

Note that adding a sampler for a node will *not* remove any existing samplers on that node. In standard practice one would first remove existing samplers on a node and then add the new sampler, though removal is optional from the perspective of having a valid MCMC configuration.

## Printing, re-ordering, and removing samplers

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

The `nimbleFunction` definition underlying a particular sampler may be viewed using the `getSamplerDefinition` method, using the sampler index as an argument. A node name argument may also be supplied, in which case the definition of the first sampler acting on that node is returned. In all cases, `getSamplerDefinition` only returns the definition of the *first* sampler specified either by index or node name.

```
## Return the definition of the third sampler in the mcmcConf object
mcmcConf$getSamplerDefinition(3)

## Return the definition of the first sampler acting on node 'x',
## or the first of any indexed nodes comprising the variable 'x'
mcmcConf$getSamplerDefinition('x')
```

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 `MCMCconf` object initially contains 10 samplers, and each example is independent of the others.

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

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

## Reverse the ordering of the samplers
mcmcConf$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.
mcmcConf$setSamplers(ind = c(1, 1, 1, 2, 3))

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

## Set the list of samplers to those acting on any components of the
## model variables 'x', 'y', or 'z'.
mcmcConf$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 `mcmcConf` 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
mcmcConf$removeSamplers(ind = 1)

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

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

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

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

The `getSamplers` method may also be used to return a list of `samplerConf` objects. Each `samplerConf` is a reference class object containing the following (required) fields: `name` (a character string), `samplerFunction` (a valid `nimbleFunction` sampler), `target` (the model node to be sampled), and `control` (list of control arguments). The list returned by `getSamplers` can be modified using the access functions provided, then passed as an argument to `setSamplers` to over-write the current list of samplers in the MCMC configuration object. However, no checking of the validity of this modified list is performed; if the list of `samplerConf` objects is corrupted to be invalid, incorrect behaviour will result at the time of calling `buildMCMC`. The fields of a `samplerConf` object can be modified using the access functions `setName(name)`, `setSamplerFunction(fun)`, `setTarget(target, model)`, and `setControl(control)`.

```
## retrieve samplerConf list
samplerConfList <- mcmcConf$getSamplers()

## change the name of the first sampler
samplerConfList[[1]]$setName('newNameForThisSampler')

## change the sampler function of the second sampler,
## assuming existence of a nimbleFunction 'anotherSamplerNF',
## which represents a valid nimbleFunction sampler.
samplerConfList[[2]]$setSamplerFunction(anotherSamplerNF)

## change the 'adaptive' element of the control list of the third sampler
control <- samplerConfList[[3]]$control
control$adaptive <- FALSE
samplerConfList[[3]]$setControl(control)

## change the target node of the fourth sampler
samplerConfList[[4]]$setTarget('y', model) ## model argument required

## use this modified list of samplerConf objects in the MCMC configuration
mcmcConf$setSamplers(samplerConfList)
```

## Monitors and thinning intervals

An MCMC configuration 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 objects called `mvSamples` and `mvSamples2`, contained within the MCMC object. These are both *modelValues* objects; *modelValues* are NIMBLE data structures used to store values of model variables. 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 argument to configureMCMC
mcmcConf <- configureMCMC(Rmodel, monitors = c('alpha', 'beta'), monitors2 = 'x')

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

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

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

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

The current lists of monitors and thinning intervals may be displayed using the `printMonitors` method. Both sets of monitors (`monitors` and `monitors2`) may be reset to empty character vectors by calling the `resetMonitors` method. The methods `getMonitors` and `getMonitors2` return the currently specified `monitors` and `monitors2` as character vectors.

## 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(mcmcConf)
```

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

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:

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

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 algorithm (either the compiled or uncompiled version) can be executed using the member method `mcmc$run()`. The `run` method has one required argument, `niter`, representing the number of iterations to run the MCMC algorithm. Calling `mcmc$run(niter)` causes the full list of samplers (as determined by the `MCMCconf` 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 `run` method 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, continuing the current run of the MCMC chain. 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.

The `run` method has an optional `simulateAll` argument. When `simulateAll = TRUE`, the `simulate` method of all stochastic nodes is called at the onset of the MCMC algorithm. This may be used to generate a new set of initial values. Note there is some danger in doing this when using non-informative priors (such as normal distributions with very large variances or gamma distributions with small parameter values), as one may easily simulate an extreme starting value.

The `run` method has an optional `time` argument. When `time = TRUE`, runtimes of the individual internal MCMC samplers will be recorded. See Section 7.7.

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]'])
```

Finally, note that all of the manipulations on *modelValues* objects discussed in Section 12.4 can be used on *mvSamples* objects that store the MCMC samples.

## 7.5 Running multiple MCMC chains

Once an MCMC algorithm has been created using `buildMCMC`, the function `runMCMC` can be used to run multiple chains and extract samples. `runMCMC` can be used to execute compiled or uncompiled algorithms, although uncompiled algorithms will be much slower.

`runMCMC` has one mandatory argument: the (compiled or uncompiled) MCMC algorithm. Other arguments to `runMCMC` are available to control:

- Number of iterations in each chain
- Number of chains
- Number of burnin samples to discard from each chain
- Initial values, or a function for generating initial values for each chain

- Setting the random number seed
- Suppressing messages and output
- Returning the samples as a `coda mcmc` object

See `help(runMCMC)` for details of these arguments.

Other features of the MCMC can still be customized when using `runMCMC`, which is done using the MCMC configuration object. This includes setting monitors, customizing samplers, or setting a thinning interval (see Section 7.1.2). The MCMC configuration object is then used to create an MCMC algorithm, which is compiled and passed as an argument to `runMCMC`.

When running a single chain, `runMCMC` returns a matrix of MCMC samples (as described in Section 7.4). When running multiple chains, a list of sample matrices is returned. Using the argument `returnCodaMCMC=TRUE`, a `coda mcmc` object, or in the case of multiple chains, a `coda mcmc.list` object is returned instead.

The following examples demonstrate some uses of `runMCMC`, and assume the existence of `Cmcmc`, a compiled MCMC algorithm.

```
## run a single chain, return a matrix of samples
samplesMatrix <- runMCMC(Cmcmc)

## run three chains of 10,000 samples, discard a burnin of 1,000,
## and return of list of sample matrices
samplesList <- runMCMC(Cmcmc, niter=10000, nburnin=1000, nchains=3)

## run two chains, and provide initial values for each
initsList <- list(list(mu=1, sigma=1),
                  list(mu=2, sigma=10))
samplesList <- runMCMC(Cmcmc, nchains=3, inits=initsList)

## run ten chains of 100,000 iterations,
## using a function to generate initial values
initsFunction <- function()
  list(mu=rnorm(1,0,1), sigma=runif(1,0,100))
samplesList <- runMCMC(Cmcmc, niter=100000, nchains=10, inits=initsFunction)

## run three chains, using a fixed random number seed for each chain
samplesList <- runMCMC(Cmcmc, nchains=3, setSeed=TRUE)

## run three chains, return a coda mcmc.list object
codaMCMClist <- runMCMC(Cmcmc, nchains=3, returnCodaMCMC=TRUE)
```



## 7.6 Recording model log-probabilities

To record model log-probabilities during the course of an MCMC algorithm, the MCMC configuration object can be customized with monitors on model *logProb variables* (which begin with the prefix `logProb_`). During MCMC execution, log-probabilities will be recorded in the same *modelValues* objects as the MCMC samples, either `mvSamples` or `mvSamples2`.

For example, to record and extract log-probabilities for the model variables `alpha`, `sigma.mu`, and `Y`:

```
mcmcConf <- configureMCMC(Rmodel)
mcmcConf$addMonitors('logProb_alpha', 'logProb_sigma_mu', 'logProb_Y')
Rmcmc <- buildMCMC(mcmcConf)
Cmodel <- compileNimble(Rmodel)
Cmcmc <- compileNimble(Rmcmc, project = Rmodel)
Cmcmc$run(10000)
samples <- as.matrix(Cmcmc$mvSamples)
```

The `samples` matrix will contain both MCMC samples and model log-probabilities.

## 7.7 Measuring sampler computation times

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

```
Cmcmc$run(niter, time=TRUE)
Cmcmc$getTimes()
```

will return a vector of the total time spent in each sampler, measured in seconds.

## 7.8 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 = 1)
  y2 ~ dbinom(prob = p, size = 2)
})
```



### 7.8.1 Conjugate (Gibbs) samplers

By default, `configureMCMC()` and `buildMCMC()` will assign conjugate samplers to all nodes satisfying a conjugate relationship, unless the option `useConjugacy = FALSE` is specified. The conjugate relationships supported are defined in the system-level variable `conjugacyRelationshipsInputList`.

Subsequently, conjugate samplers can be added to particular nodes in an MCMC configuration object (`conf`) using `conf$addSampler(target = 'x', type = 'conjugate')`, to add a conjugate sampler for model node 'x'. If the specified node does not satisfy any conjugate relationship, then an error occurs and no sampler is added to the configuration object.

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

The current release of NIMBLE supports conjugate sampling of the relationships listed in Table 7.1.

Prior Distribution	Sampling (Dependent Node) Distribution	Parameter
Beta	Bernoulli	<code>prob</code>
	Binomial	<code>prob</code>
	Negative Binomial	<code>prob</code>
Dirichlet	Multinomial	<code>prob</code>
Gamma	Poisson	<code>lambda</code>
	Normal	<code>tau</code>
	Lognormal	<code>taulog</code>
	Gamma	<code>rate</code>
	Exponential	<code>rate</code>
Normal	Normal	<code>mean</code>
	Lognormal	<code>meanlog</code>
Multivariate Normal	Multivariate Normal	<code>mean</code>
Wishart	Multivariate Normal	<code>prec</code>

Table 7.1: Conjugate relationships supported by NIMBLE's MCMC engine.

Conjugate sampler functions may (optionally) dynamically check that their own posterior likelihood calculations are correct. If incorrect, a warning is issued. However, this functionality will roughly double the run-time required for conjugate sampling. By default, this option is disabled in NIMBLE. This option may be enabled by changing the NIMBLE option:

```
nimbleOptions(verifyConjugatePosteriors = TRUE)
```

### 7.8.2 binary (Gibbs) sampler

The `binary` sampler performs Gibbs sampling for binary-valued nodes (strictly taking values 0 or 1). This can only be used for nodes following either a `dbern(p)` or `dbinom(p, size=1)` distribution. The `binary` sampler accepts no control list arguments.

Example usage:

```
mcmcConf$addSampler(target = 'y[1]', type = 'binary')
```

### 7.8.3 RW\_multinomial sampler

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

The `RW_multinomial` sampler can be customized using `control` list arguments to set the adaptive properties of the sampler. See `help(samplers)` for details.

Example usage:

```
mcmcConf$addSampler(target = 'x[1:5]', type = 'RW_multinomial')
```

### 7.8.4 RW\_dirichlet sampler

The `RW_dirichlet` sampler operates exclusively on nodes following a Dirichlet distribution, and is designed for the non-conjugate Dirichlet case. The sampler performs independent adaptive Metropolis-Hastings updates on a reparameterization of the target distribution, expressed in terms of component univariate gamma distributions.

The `RW_dirichlet` sampler can be customized using `control` list arguments to set the adaptive properties of the sampler. See `help(samplers)` for details.

Example usage:

```
mcmcConf$addSampler(target = 'p[1:5]', type = 'RW_dirichlet')
```

### 7.8.5 Scalar Metropolis-Hastings random walk RW sampler

The `RW` sampler executes adaptive Metropolis-Hastings sampling with a normal proposal distribution, implementing the adaptation routine given in [7]. This sampler can be applied to any scalar continuous-valued stochastic node, and can optionally sample on a log scale.

The `RW` sampler can be customized using the `control` list argument to set the initial proposal distribution scale, the adaptive properties of the sampler, the reflective property of proposals, and whether to sample on a log scale. See `help(samplers)` for details.

Example usage:

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

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

mcmcConf$addSampler(target = 'p', type = 'RW',
  control = list(reflective = TRUE))
```

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. This can be avoided in some cases by using the option `reflective = TRUE`, which reflects proposal values to stay within the range of the target node distribution.

### 7.8.6 Multivariate Metropolis-Hastings RW\_block sampler

The `RW_block` sampler performs a simultaneous update of one or more model nodes, using an adaptive Metropolis-Hastings algorithm with a multivariate normal proposal distribution [6], implementing the adaptation routine given in [7]. 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 can be customized using the `control` list argument to set the initial proposal covariance, and the adaptive properties of the sampler. See `help(samplers)` for details.

Example usage:

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

### 7.8.7 slice sampler

The `slice` sampler performs slice sampling of the scalar node to which it is applied [5]. This sampler can operate on either continuous-valued or discrete-valued scalar nodes. The slice sampler performs a “stepping out” procedure, in which the slice is iteratively expanded to the left or right by an amount `sliceWidth`. This sampler is optionally adaptive, whereby the value of `sliceWidth` is adapted towards the observed absolute difference between successive samples.

The `slice` sampler can be customized using the `control` list argument to set the initial slice width, and the adaptive properties of the sampler. See `help(samplers)` for details.

Example usage:

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

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

### 7.8.8 Elliptical slice sampling: `ess` sampler

The `ess` sampler performs elliptical slice sampling of a single node, which must follow a multivariate normal distribution [4]. The algorithm is an extension of slice sampling [5], generalized to the multivariate normal context. An auxiliary variable is used to identify points on an ellipse (which passes through the current node value) as candidate samples, which are accepted contingent upon a likelihood evaluation at that point. This algorithm requires no tuning parameters and therefore no period of adaptation, and may result in very efficient sampling from multivariate Gaussian distributions.

The `ess` sampler accepts no control list arguments. See `help(samplers)` for details.

Example usage:

```
## assuming node 'x[1:10]' follows a multivariate normal distribution
mcmcConf$addSampler(target = 'x[1:10]', type = 'ess')
```

### 7.8.9 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 descendants of node(s) at one level have conjugate relationships with their own stochastic descendants. 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 can be customized using the `control` list argument to set the initial proposal covariance and the adaptive properties for the Metropolis-Hastings sampling of the higher-level nodes. See `help(samplers)` for details.

Example usage:

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

The requirement that all stochastic descendants of the `target` nodes must themselves have only conjugate descendants 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.

### 7.8.10 Customized log likelihood evaluations: `RW_llFunction` sampler

Sometimes it is useful to control the log likelihood calculations used for an MCMC updater instead of simply using the model. For example, one could use a sampler with a log likelihood that analytically (or numerically) integrates over latent model nodes. Or one could use a sampler with a log likelihood that comes from a stochastic approximation such as a particle filter (see below), allowing composition of a particle MCMC (PMCMC) algorithm [1]. The `RW_llFunction` sampler handles this by using a Metropolis-Hastings algorithm with a normal proposal distribution and a user-provided log-likelihood function. To allow compiled execution, the log-likelihood function must be provided as a specialized instance of a `nimbleFunction`. The log-likelihood function may use the same model as the MCMC as a setup argument (as does the example below), but if so the state of the model should be unchanged during execution of the function (or you must understand the implications otherwise).

The `RW_llFunction` sampler can be customized using the `control` list argument to set the initial proposal distribution scale and the adaptive properties for the Metropolis-Hastings sampling. In addition, the `control` list argument must contain a named `llFunction` element, which is specialized `nimbleFunction` that accepts no arguments and returns a scalar double number. The return value must be the total log-likelihood of all stochastic dependents of the `target` nodes – and, if `includesTarget = TRUE`, of the target node(s) themselves – or whatever surrogate is being used for the total log-likelihood. This is a required `control` list element with no default. See `help(samplers)` for details.

Complete example of correct usage:

```
code <- nimbleCode({
  p ~ dunif(0, 1)
  y ~ dbin(p, n)
})

Rmodel <- nimbleModel(code, data = list(y=3), inits = list(p=0.5, n=10))

llFun <- nimbleFunction(
  setup = function(model) { },
  run = function() {
    y <- model$y
    p <- model$p
    n <- model$n
    ll <- lfactorial(n) - lfactorial(y) - lfactorial(n-y) +
      y * log(p) + (n-y) * log(1-p)
    returnType(double())
    return(ll)
  }
)
```

```

RllFun <- llFun(Rmodel)

mcmcConf <- configureMCMC(Rmodel, nodes = NULL)

mcmcConf$addSampler(target = 'p', type = 'RW_llFunction',
  control = list(llFunction = RllFun, includesTarget = FALSE))

Rmcmc <- buildMCMC(mcmcConf)

```

### 7.8.11 Terminal node posterior\_predictive sampler

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

Example usage:

```

mcmcConf$addSampler(target = 'y[1]', type = 'posterior_predictive')

```

### 7.8.12 Particle MCMC PMCMC sampler

For state space models, a particle MCMC (PMCMC) sampler can be specified for top-level parameters. This sampler is described in Section [8.1.2](#).

## 7.9 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(a = c(1, 1),
              b = c(1, 1),
              p = matrix(0.5, nrow = 2, ncol = 16),
              mu = c(.5, .5),
              theta = c(.5, .5))

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

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

```

```

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

## check the samplers assigned by default MCMC configuration
mcmcConf$printSamplers()

## double-check our monitors, and thinning interval
mcmcConf$printMonitors()

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

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

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

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

## let's check the samplers
mcmcConf_CL$printSamplers()

## build this second executable R MCMC function
mcmc_CL <- buildMCMC(mcmcConf_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$run(1000)
cSamplesMatrix <- as.matrix(Cmcmc$mvSamples)
mean(cSamplesMatrix[, 'mu[1]'])

## run the crossLevel MCMC function,
## and examine the mean of mu[1]
Cmcmc_CL$run(1000)
cSamplesMatrix_CL <- as.matrix(Cmcmc_CL$mvSamples)
mean(cSamplesMatrix_CL[, 'mu[1]'])

#####
#### run multiple MCMC chains ####
#####

## run 3 chains of the crossLevel MCMC
samplesList <- runMCMC(Cmcmc_CL, niter=1000, nchains=3)

lapply(samplesList, dim)
```

## 7.10 Comparing different MCMC engines with MCMCsuite and compareMCMCs

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. NIMBLE’s `compareMCMCs` manages calls to `MCMCsuite` for multiple sets of comparisons and organizes the output(s) for generating html pages summarizing results. It also allows multiple results to be combined and allows some different options for how results are processed, such as how effective sample size is estimated.

We show how to use `MCMCsuite` for the same `litters` example used in Section 7.9. Subsequently, additional details of the `MCMCsuite` are given. Since use of `compareMCMCs` is similar, we refer readers to `help(compareMCMCs)` and the functions listed under “See also” on that R help page.

### 7.10.1 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('winbugs', 'jags', 'nimble', 'nimble_slice', 'nimble_CL'),
  MCMCdefs = list(
    nimble_CL = quote({
      mcmcConf <- configureMCMC(Rmodel, nodes = NULL)
      mcmcConf$addSampler(target = c('a[1]', 'b[1]'), type = 'crossLevel')
      mcmcConf$addSampler(target = c('a[2]', 'b[2]'), type = 'crossLevel')
      mcmcConf
    })),
  plotName = 'littersSuite'
)
```

### 7.10.2 MCMC Suite outputs

Executing the MCMC Suite returns a named list containing various outputs, as well as generates and saves traceplots and posterior density plots. The default 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
# winbugs      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
# winbugs      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 seconds.

## Efficiency

Using the MCMC Suite option `calculateEfficiency = TRUE` will also provide several measures of MCMC sampling efficiency. Additional summary statistics are provided for each node: the total number of samples collected ( $n$ ), the effective sample size resulting from these samples ( $ess$ ), and the effective sample size per second of algorithm runtime (*efficiency*).

In addition to these node-by-node measures of efficiency, an additional return list element is also provided. This element, *efficiency*, is itself a named list containing two elements: *min* and *mean*, which contain the minimal and mean efficiencies (effective sample size per second of algorithm runtime) across all monitored nodes, separately for each algorithm.

## 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.

### 7.10.3 Customizing MCMC Suite

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

- MCMC algorithms to execute, optionally including WinBUGS, OpenBUGS, JAGS, Stan, and various flavours of NIMBLE’s MCMC
- Custom-configured NIMBLE MCMC algorithms
- Automated parameter blocking for efficient MCMC sampling
- Nodes to monitor
- Number of MCMC iterations
- Thinning interval
- Burn-in
- Summary statistics to report
- Calculating sampling efficiency (effective sample size per second of algorithm runtime)
- Generating and saving plots

NIMBLE MCMC algorithms may be specified using the `MCMCs` argument to `MCMCsuite`, which is character vector defining the MCMC algorithms to run. The `MCMCs` argument may include any of the following algorithms:

```

‘winbugs’ WinBUGS MCMC algorithm
‘openbugs’ OpenBUGS MCMC algorithm
‘jags’ JAGS MCMC algorithm
‘Stan’ Stan MCMC algorithm
‘nimble’ NIMBLE MCMC using the default configuration
‘nimble_noConj’ NIMBLE MCMC using the default configuration with useConjugacy =
    FALSE
‘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 default value for the `MCMCs` argument is `‘nimble’`, which specifies only the default NIMBLE MCMC algorithm.

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 `MCMCdefs` may be provided as 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({  
    mcmcConf <- configureMCMC(Rmodel, ....)  
    mcmcConf$addSampler(.....)  
    mcmcConf      ## returns the MCMC configuration object  
  })  
)
```

Full details of the arguments and customization of the MCMC Suite is available through the R help using `help(MCMCsuite)`.

# Chapter 8

## Sequential Monte Carlo and MCEM

The NIMBLE algorithm library is growing and as of version 0.5-1 includes a suite of Sequential Monte Carlo algorithms as well as a more robust MCEM.

### 8.1 Particle Filters / Sequential Monte Carlo

#### 8.1.1 Filtering Algorithms

NIMBLE includes algorithms for four different types of sequential Monte Carlo (also known as particle filters), which can be used to sample from the latent states and approximate the log likelihood of a state space model. The particle filters currently implemented in NIMBLE are the bootstrap filter, the auxiliary particle filter, the Liu-West filter, and the ensemble Kalman filter, which can be built, respectively, with calls to `buildBootstrapFilter`, `buildAuxiliaryFilter`, `buildLiuWestFilter`, and `buildEnsembleKF`. Each particle filter requires setup arguments `model` and `nodes`, which is a character vector specifying latent model nodes. In addition, each particle filter can be customized using a `control` list argument. Details on the control options and specifics of the filtering algorithms can be found in the help pages for the functions.

Once built, each filter can be run by specifying the number of particles. Each filter has a model values object named `mvEWSamples` that is populated with equally-weighted samples from the posterior distribution of the latent states (and in the case of the Liu-West filter, the posterior distribution of the top level parameters as well) as the filter is run. The bootstrap, auxiliary, and Liu-West filters also have another model values object, `mvWSamples`, which has unequally-weighted samples from the posterior distribution of the latent states, along with weights for each particle. In addition, the bootstrap and auxiliary particle filters return estimates of the log likelihood of the given state space model.

We first create a linear state-space model to use as an example for our particle filter algorithms.

```
# 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 the model
rTimeModel <- nimbleModel(timeModelCode, constants = list(t = t),
                          data <- list(y = y), check = FALSE )

#Set parameter values and compile the model
rTimeModel$a <- 0.5
rTimeModel$b <- 1
rTimeModel$c <- 1
rTimeModel$mu_0 <- 1

cTimeModel <- compileNimble(rTimeModel)

```

Here is an example of building and running the bootstrap filter. Additional information about the bootstrap filter can be found with `help(buildBootstrapFilter)`.

```

#Build bootstrap filter
rBootF <- buildBootstrapFilter(rTimeModel, "x",
                              control = list(thresh = 0.8, saveAll = TRUE,
                                              smoothing = FALSE))

#Compile filter
cBootF <- compileNimble(rBootF, project = rTimeModel)

```

```
#Set number of particles
parNum <- 5000
#Run bootstrap filter, which returns estimate of model log-likelihood
bootLLEst <- cBootF$run(parNum)
```

Next, we provide an example of building and running the auxiliary particle filter. Additional information about the auxiliary particle filter can be found with `help(buildAuxiliaryFilter)`. Note that a filter cannot be built on a model that already has a filter specialized to it, so we create a new copy of our state space model first

```
#Copy our state-space model for use with the auxiliary filter
auxTimeModel <- rTimeModel$newModel(replicate = TRUE)
compileNimble(auxTimeModel)
#Build auxiliary filter
rAuxF <- buildAuxiliaryFilter(auxTimeModel, "x",
                             control = list(thresh = 0.5, saveAll = TRUE))
#Compile filter
cAuxF <- compileNimble(rAuxF, project = auxTimeModel)
#Run auxiliary filter, which returns estimate of model log-likelihood
auxLLEst <- cAuxF$run(parNum)
```

Now we give an example of building and running the Liu and West filter, which can sample from the posterior distribution of top-level parameters as well as latent states. The Liu and West filter accepts an additional `params` argument, specifying the top-level parameters to be sampled. Additional information can be found with `help(buildLiuWestFilter)`.

```
#Copy model
LWTimeModel <- rTimeModel$newModel(replicate = TRUE)
compileNimble(LWTimeModel)
#Build Liu-West filter, also
#specifying which top level parameters to estimate
rLWF <- buildLiuWestFilter(LWTimeModel, "x", params = c("a", "b", "c"),
                           control = list(saveAll = FALSE))
#Compile filter
cLWF <- compileNimble(rLWF, project = LWTimeModel)
#Run Liu-West filter
cLWF$run(parNum)
```

Below we give an example of building and running the ensemble Kalman filter, which can sample from the posterior distribution of latent states. Additional information can be found with `help(buildEnsembleKF)`.

```
#Copy model
ENKFTTimeModel <- rTimeModel$newModel(replicate = TRUE)
```



```

compileNimble(ENKFTIMEModel)
#Build and compile ensemble Kalman filter
rENKF <- buildEnsembleKF(ENKFTIMEModel, "x",
                        control = list(saveAll = FALSE))
cENKF <- compileNimble(rENKF, project = ENKFTIMEModel)
#Run ensemble Kalman filter
cENKF$run(parNum)

```

Once each filter has been run, we can extract samples from the posterior distribution of our latent states as follows:

```

#Equally-weighted samples (available from all filters)
bootEWSamp <- as.matrix(cBootF$mvEWSamples)
auxEWSamp <- as.matrix(cAuxF$mvEWSamples)
LWFEWSamp <- as.matrix(cLWF$mvEWSamples)
ENKFEWSamp <- as.matrix(cENKF$mvEWSamples)

#Unequally-weighted samples, along with weights (available
#from bootstrap, auxiliary, and Liu and West filters)
bootWSamp <- as.matrix(cBootF$mvWSamples, 'x')
bootWts <- as.matrix(cBootF$mvWSamples, 'wts')
auxWSamp <- as.matrix(xAuxF$mvWSamples, 'x')
auxWts <- as.matrix(cAuxF$mvWSamples, 'wts')

#Liu and West filter also returns samples
#from posterior distribution of top-level parameters:
aEWSamp <- as.matrix(cLWF$mvEWSamples, 'a')

```

### 8.1.2 Particle MCMC (PMCMC)

In addition to our four particle filters, NIMBLE also has particle MCMC samplers implemented. These sample top-level parameters by using either a bootstrap filter or auxiliary particle filter to obtain estimates of the likelihood of a model for use in a Metropolis-Hastings MCMC step. The `RW_PF` sampler uses a univariate normal proposal distribution, and should be used to sample scalar top-level parameters. The `RW_PF_block` sampler uses a multivariate normal proposal distribution for vectors of top-level parameters. Each PMCMC sampler also includes an optional algorithm to estimate the optimal number of particles to use in the particle filter at each iteration, based on a trade off between computational time and efficiency. The PMCMC samplers can be specified with a call to `addSampler` with `type = 'RW_PF'` or `type = 'RW_PF_block'`, a syntax similar to the other MCMC samplers listed in 7.8.

The `RW_PF` sampler and `RW_PF_block` sampler can be customized using the `control` list argument to set the adaptive properties of the sampler and options for the particle filter

algorithm to be run. In addition, setting the `pfOptimizeNparticles` control list option to `TRUE` will allow the sampler to estimate the optimal number of particles for the bootstrap filter. See `help(samplers)` for details. The MCMC configuration for the `timeModel` in the previous section will serve as an example for the use of our PMCMC sampler. Here we use the identity matrix as our proposal covariance matrix.

```
timeConf <- configureMCMC(rTimeModel)  # default MCMC configuration

# Add random walk pmcmc sampler with particle number optimization.
timeConf$addSampler(target = c('a', 'b', 'c', 'mu_0'), type = 'RW_PF_block',
                    control <- list(propCov= diag(4), adaptScaleOnly = FALSE,
                                   latents = 'x', pfOptimizeNparticles = TRUE))
```

## 8.2 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. NIMBLE provides an ascent-based MCEM algorithm that automatically determines when the algorithm has converged by examining the size of the changes in the likelihood between each iteration (`buildMCEM`).

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

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

compileNimble(pump)

#build an MCEM algorithm with Ascent-based convergence criterion
pumpMCEM <- buildMCEM(model = pump,
                    latentNodes = 'theta', burnIn = 300,
                    mcmcControl = list(adaptInterval = 100),
                    boxConstraints = list( list( c('alpha', 'beta'),
                                                limits = c(0, Inf) ) ),
                    buffer = 1e-6)
```

The first argument to the MCEM, `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 ascent-based MCEM algorithm has a number of control options:

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 12.3.3: 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 automatically detects box constraints for the nodes that will be optimized, using NIMBLE's `getBounds()` function. It is also possible for a user to manually specify constraints via the `boxConstraints` argument. 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 two, 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, its constraints will be automatically determined by NIMBLE. These constraint arguments are passed as the `lower` and `upper` arguments to R's `optim` function, using `method = 'L-BFGS-B'`. Note that NIMBLE will give a warning if a user-provided constraint is more extreme than the constraint determined by NIMBLE.

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

In addition, the MCEM has some extra control options that can be used to further tune the convergence criterion. See `help(buildMCEM)` for more information.

Once an MCEM has been built for the model of interest, it can be run as follows. There is only one run-time argument, `initM`, which is the number of MCMC iterations to use when the algorithm is initialized.

```
pumpMLE <- pumpMCEM(initM = 1000)
pumpMLE
```

Direct maximization after analytically integrating over the latent nodes (possible for this model but often not feasible) gives estimates of  $\hat{\alpha} = 0.823$  and  $\hat{\beta} = 1.261$ , so the MCEM seems to do pretty well.

# Part IV

## Programming with NIMBLE

## Chapter 9

# Introduction to programming in NIMBLE

should this be a chapter or just a blurb before the first chapter of Part 3?

Part [IV](#) is the programmer’s guide to NIMBLE. At the heart of programming in NIMBLE are nimbleFunctions, which are written using the NIMBLE language (also called the NIMBLE DSL).

In Chapter [10](#), we describe how to write simple nimbleFunctions (nimbleFunctions that don’t interact with models) to compile parts of R for fast calculations. We then show how to use these simple nimbleFunctions to write user-defined distributions and user-defined functions that you can use within the BUGS code of a model to extend the distributions and functions build into NIMBLE (Chapter [11](#)).

We then move into the functionality in NIMBLE that allows one to program with models, namely writing nimbleFunctions that interact with models. We first describe how one interacts with models, querying model structure and carrying out model calculation (Chapter [12](#)). This functionality is central to writing nimbleFunctions that interact with models, described in Chapter [13](#). In particular one will often query model structure within nimbleFunction setup code and then carry out model calculations based on that structural information within nimbleFunction run code. Together Sections [10.3](#) and [13.4](#) describe what is allowed within the NIMBLE language, the language used for run code.

# Chapter 10

## Writing simple nimbleFunctions using the NIMBLE language

### 10.1 Introduction to simple nimbleFunctions

`nimbleFunctions` are the heart of programming in NIMBLE. In this part of the manual, you'll learn about the various ways to use `nimbleFunctions`. Here we'll start with simple `nimbleFunctions` that only contain the code to be compiled and executed, which we call the *run* code. `nimbleFunctions` are created with the `nimbleFunction` function.

Basic `nimbleFunctions` behave like R functions: `nimbleFunction` simply returns a function that can be executed and compiled. This is useful for doing math or the other kinds of processing available in NIMBLE when no model or `modelValues` is needed. The real benefits come from compiling the `nimbleFunction`; in other words, `nimbleFunctions` allow you to compile parts of R.

*run* code is written 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. In addition, some information on variable types must be provided for input arguments, the return object, and in some circumstances for local variables.

Here's a basic example.

### 10.2 Using and compiling simple nimbleFunctions

Here's a slightly more involved `nimbleFunction` that illustrates the use of linear algebra. After we create the `nimbleFunction`, we can compile a fast version of the `nimbleFunction` using `compileNimble`.

This example shows the textbook calculation of a least squares solution for regression of 100 data points on 4 explanatory variables, all generated randomly<sup>1</sup>. Such functions can be called from other `nimbleFunctions` or used in BUGS code.<sup>2</sup>

---

<sup>1</sup>Of course in general, explicitly calculating the inverse is not the recommended numerical recipe for least squares.

<sup>2</sup>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.

## 10.3 The NIMBLE language

### 10.3.1 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 (see Section 13.7).
- 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.)

- Variables in the NIMBLE language are typed and types must be declared for arguments and return values.
- BUGS model nodes are implemented as `nimbleFunctions` with member functions for `calculate`, `calculateDiff`, `simulate`, `getLogProb`, `getParam`, and `getBound`.
- `nimbleFunctions` cannot generally be copied safely. Instead rerun the `nimbleFunction()` and record the result in a new object. *useful to say here? Is there a case in which one would want to copy an RC function?*

### 10.3.2 How numeric types work in NIMBLE

Variables in the NIMBLE language are typed – a given variable can only represent a specific kinds of value, such as a real number, integer, or character string, and the dimension (scalar, vector, matrix, etc.) cannot change once it is set.

Numeric types in NIMBLE are much less flexible than in R, a reflection of the fact that

NIMBLE code can be compiled into C++<sup>3</sup>. In NIMBLE, the *type* of a numeric object refers to the number of dimensions and the numeric type of the elements. In Version 0.6-3, 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.

Arguments and return values for functions must have their types and dimensions declared by the programmer.

Types and dimensions for variables declared within run code are handled differently. When possible, NIMBLE will determine the type of a local variable that is defined in run code for you. In particular, 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. In contrast, if the first uses of a variable involves indexing, the type must be declared explicitly and the size set as discussed in Section ???. In addition, its size must be set before assigning into it.

Treatment of vectors of length 1 presents special challenges because they could be treated as scalars or vectors. Currently they are treated as scalars.

### 10.3.3 Declaring argument types and the return type

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

As illustrated in the example in Section 10.1, 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, and 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` (e.g. `double(2, c(4, 5))` declares a 4- $\times$ -5 matrix). If sizes are omitted, they will either be set when the entire object is assigned to, or an explicit call to `setSize` is needed.

In the case of scalar arguments only, a default value can be provided. For example, to provide 1.2 as a default:

```
myfun <- nimbleFunction(  
  run = function(x = double(0, default = 1.2)) {  
  })
```

### 10.3.4 Creating non-scalar variables: numeric, integer, matrix, and array

When local variables in run code are created by assignment, their types will be automatically inferred. For example, `x <- A %*% B` will create `x` as a matrix. However, if a variable is

---

<sup>3</sup>C++ is a statically typed language, which means the type of a variable cannot change.



to be filled by indexed assignment, then it must be explicitly created or re-sized first. For example, in the following code `x` must be created before being filled by indexed assignment:

```
## NOTE: must create x as a matrix here, first

for(i in 1:10)
  for(j in 1:5)
    x[i, j] <- foo(y[i, j])
```

Scalar variables never need to be created in advance.

NIMBLE provides several functions for creating non-scalar variables. These functions are similar to those of R, but they take additional arguments to set an initialization value, indicate floating-point or integer elements, and/or indicate no initialization is needed<sup>4</sup>. `numeric` and `integer` create floating-point and integer vectors (1-dimensional objects), respectively. `matrix` creates 2-dimensional objects, and `array` creates objects of 1, 2 or more dimensions. The latter two functions can be used for floating-point or integer objects.

After a variable has been created, its size may be changed either by non-indexed assignment or by `setSize`, as illustrated below. Note that `setSize` cannot change the number of dimensions of a variable, and it does *not* necessarily preserve the contents of the variable.

### numeric and integer

`numeric` or `integer` will create a 1-dimensional vector of floating-point or integer values, respectively. The `length` argument specifies the vector length (default 0), and the `value` argument specifies the initial scalar value for all vector elements (default 0). The `init` argument specifies whether or not to initialize the elements (default `TRUE`). If first use of the variable does not rely on initial values, you can use `init = FALSE`.

```
## Example of creating and resizing a floating-point vector
## myNumericVector will be of length 10, with all elements initialized to 2
myNumericVector <- numeric(10, value = 2)
```

```
## resize this numeric vector to be length 20
## both calls are equivalent
setSize(myNumericVector, 20)
setSize(myNumericVector, c(20))
```

```
## Example of creating a length-100 integer vector and filling it with the values 1, 2, ...
x <- 100
myIntegerVector <- integer(x)
for(i in 1:x)
  myIntegerVector[i] <- i
```

---

<sup>4</sup>Skipping initialization is more efficient, but this will typically be noticeable only for functions called many, many times.

**matrix and identityMatrix**

**matrix** creates a 2-dimensional matrix object of either floating-point (if **type** = 'double', the default) or integer (if **type** = 'integer') values. The **nrow** and **ncol** arguments specify the number of rows and columns, respectively. The **value** and **init** argument are used in the same way as for **numeric()** and **integer()**.

```
## Example of creating a 10-by-1 column matrix of 1's and resizing it
onesMatrix <- matrix(1, nrow = 10, ncol = 1)

## resize this matrix to be a 10-by-10 matrix
## note that contents are not necessarily preserved
## both calls are equivalent
setSize(onesMatrix, 10, 10)
setSize(onesMatrix, c(10, 10))
```

NIMBLE provides the **identityMatrix** function for quickly creating instances of an identity matrix (all 0's with 1's on the main diagonal). This function takes a single argument, specifying the number of rows and columns in the resulting matrix.

```
I5 <- identityMatrix(5)
```

The resulting matrix may have its elements modified, or be used in arbitrary mathematical expressions, as any variable in the DSL.

**array**

**array** creates a vector or higher-dimensional object, depending on the **dim** argument, which takes a vector of sizes for each dimension. The **type**, **value** and **init** argument behave the same as for **matrix**.

```
## the following three lines are equivalent
## each creates a length-10 vector, with elements equal to y
a <- numeric(10, value = y)
a <- array(y, dim = 10)
a <- array(y, dim = c(10))

## the following three lines are equivalent
## each creates an integer vector of length z[5], with elements equal to x+y
b <- integer(z[5], value = x + y)
b <- array(x+y, dim = z[5], type = 'integer')
b <- array(x+y, dim = c(z[5]), type = 'integer')

## the following two lines are equivalent
## each one creates a matrix of 0's of the same size as matrix x
```

```

c <- matrix(0, nrow = dim(x)[1], ncol = dim(x)[2])
c <- array(0, dim = c(dim(x)[1], dim(x)[2]))

## the following creates a 3-dimensional array of 0's
d <- array(0, dim = c(x, y, z))

## now resize this 3-dimensional array to be (x+1) by (y+1) by (z+1)
## both calls are equivalent
setSize(d, x+1, y+1, z+1)
setSize(d, c(x+1, y+1, z+1))

```

### Deprecated method of creating non-scalar objects using declare

Previous versions of NIMBLE provided a function `declare` for declaring variables. The more R-like functions `numeric`, `integer`, `matrix` and `array` are intended to replace `declare`, but `declare` is still supported for backward compatibility. In a future version of NIMBLE, `declare` may be removed.

### 10.3.5 Querying 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 should keep in mind that it behaves differently from R's `dim`.
  - A quirky limitation in Version 0.6-3: 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.

### 10.3.6 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.

### 10.3.7 Using functions

You can call another nimbleFunction from within your nimbleFunction. Simply define the two functions and call one within the other. Similarly, you can define and use a nimbleFunction within another nimbleFunction.

please confirm these statements are true for RC functions

### 10.3.8 Basic math and linear algebra

#### Available mathematical operations

Numeric scalar and matrix mathematical operations are listed in tables 10.1 and 10.2.

Table 10.1: 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
<code>x   y, x &amp; y</code>	logical OR ( ) and AND(&)		✓	
<code>!x</code>	logical not		✓	
<code>x &gt; y, x &gt;= y</code>	greater than (and or equal to)		✓	
<code>x &lt; y, x &lt;= y</code>	less than (and or equal to)		✓	
<code>x != y, x == y</code>	(not) equals		✓	
<code>x + y, x - y, x * y</code>	component-wise operators	mix of scalar and vector ok	✓	✓
<code>x / y,</code>	component-wise division	vector $x$ and scalar $y$ ok	✓	✓
<code>x^y, pow(x, y)</code>	power	$x^y$ ; vector $x$ and scalar $y$ ok	✓	✓
<code>x %% y</code>	modulo (remainder)		✓	
<code>min(x1, x2), max(x1, x2)</code>	min. (max.) of two scalars		✓	
<code>exp(x)</code>	exponential		✓	✓
<code>log(x)</code>	natural logarithm		✓	✓
<code>sqrt(x)</code>	square root		✓	✓
<code>abs(x)</code>	absolute value		✓	✓
<code>step(x)</code>	step function at 0	0 if $x < 0$ , 1 if $x > 0$	✓	✓
<code>equals(x, y)</code>	equality of two scalars	1 if $x == y$ , 0 if $x != y$	✓	
<code>cube(x)</code>	third power	$x^3$	✓	✓
<code>sin(x), cos(x), tan(x)</code>	trigonometric functions		✓	✓
<code>asin(x), acos(x), atan(x)</code>	inverse trigonometric functions		✓	✓
<code>asinh(x), acosh(x), atanh(x)</code>	inv. hyperbolic trig. functions		✓	✓
<code>logit(x)</code>	logit	$\log(x/(1-x))$	✓	✓
<code>ilogit(x), expit(x)</code>	inverse logit	$\exp(x)/(1+\exp(x))$	✓	✓
<code>probit(x)</code>	probit (Gaussian quantile)	$\Phi^{-1}(x)$	✓	✓
<code>iprobit(x), phi(x)</code>	inverse probit (Gaussian CDF)	$\Phi(x)$	✓	✓
<code>cloglog(x)</code>	complementary log log	$\log(-\log(1-x))$	✓	✓
<code>icloglog(x)</code>	inverse complementary log log	$1 - \exp(-\exp(x))$	✓	✓
<code>ceiling(x)</code>	ceiling function	$\lceil x \rceil$	✓	✓
<code>floor(x)</code>	floor function	$\lfloor x \rfloor$	✓	✓
<code>round(x)</code>	round to integer		✓	✓
<code>trunc(x)</code>	truncation to integer		✓	✓

Table 10.1: 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
<code>lgamma(x)</code> , <code>loggam(x)</code>	log gamma function	$\log \Gamma(x)$	✓	✓
<code>log1p(x)</code>	log of $1 + x$	$\log(1 + x)$	✓	✓
<code>lfactorial(x)</code> , <code>logfact(x)</code>	log factorial	$\log x!$	✓	✓
<code>log1p(x)</code>	log one-plus	$\log(x + 1)$	✓	✓
<code>qDIST(x, PARAMS)</code>	“q” distribution functions	canonical parameterization	✓	
<code>pDIST(x, PARAMS)</code>	“p” distribution functions	canonical parameterization	✓	
<code>rDIST(1, PARAMS)</code>	“r” distribution functions	canonical parameterization	✓	
<code>dDIST(x, PARAMS)</code>	“d” distribution functions	canonical parameterization	✓	
<code>sort(x)</code>				
<code>rank(x, s)</code>				
<code>ranked(x, s)</code>				
<code>order(x)</code>				

Table 10.2: Functions operating on vectors and matrices. Status column indicates if the function is currently provided in NIMBLE.

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

NIMBLE uses the *Eigen* library in C++ to accomplish linear algebra. In Version 0.6-3, we use a lot of Eigen’s capabilities, but not all of them.

No vectorized operations other than assignment are supported for more than two dimensions in v0.6-3. That means  $A = B + C$  will work only if  $B$  and  $C$  have dimensions  $\leq$

2.

## Indexing

### Managing dimensions, sizes and indices

do we want to say anything else here about indexing?

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`. If you want the inner product as a scalar, use `inprod(v1, 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. If you do not want this behavior, use `drop=FALSE` just as in R. For example, `M1[1,,drop=FALSE]` is a matrix.
- 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$x` will not work – and could crash your R session with a segmentation fault – if `Y` is not already at least 10x30 in size. This can be done by `setSize(Y, c(10, 30))`. See section (10.3.4) for more details. Note that non-indexed assignment to `Y`, such as `Y <- model$x`, will automatically set `Y` to the necessary 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-dimensional, 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 Version 0.6-3, 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 Version 0.6-3 we limit ourselves to trying to provide useful information.

### 10.3.9 Distribution functions

Distribution “d”, “r”, “p”, and “q” functions can all be used from `nimbleFunctions` (and in BUGS model code), but the care is needed in the syntax.

- We support only the canonical NIMBLE parameterization, as listed below (with a small number of exceptions, also listed).
- The names of the distributions are the names used under the hood in NIMBLE and differ from the standard BUGS distribution names.
- Currently “r” functions only return one random draw at a time, and the first argument must always be 1.
- For the multivariate normal and Wishart distributions the `prec_param` or `scale_param` argument must be provided, indicating when a covariance or precision matrix has been given.

User-defined distributions can also be used from `nimbleFunctions`. Arguments are matched by order or by name (if given). If omitted, default argument values based on the standard R distribution functions will be used. Standard arguments to distribution functions in R (`log`, `log.p`, `lower.tail`) can be used and take the usual default values as in R. User-supplied distributions are handled analogously with regard to matching by position and use of defaults (when provided via the `nimbleFunction` run-time arguments) (Chapter 11).

Supported distributions include:

- `dbinom(size, prob)`
- `dcat(prob)`
- `dmulti(size, prob)`
- `dnbinom(size, prob)`
- `dpois(lambda)`
- `dbeta(shape1, shape2)`
- `dchisq(df)`
- `dexp(rate)`
- `dexp_nimble(rate)`
- `dexp_nimble(scale)`

- `dgamma(shape, rate)`
- `dgamma(shape, scale)`
- `dlnorm(meanlog, sdlog)`
- `dlogis(location, scale)`
- `dnorm(mean, sd)`
- `dt_nonstandard(df, mu, sigma)`
- `dt(df)`
- `dunif(min, max)`
- `dweibull(shape, scale)`
- `ddirch(alpha)`
- `dmnorm_chol(mean, cholesky, prec_param)`
- `dwish_chol(cholesky, df, scale_param)`

In the last two, `cholesky` stands for Cholesky decomposition of the relevant matrix; `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 a future release, we will also extend the alternative parameterizations given in Section 5.2.4 to `nimbleFunctions`.

### 10.3.10 print and stop

As demonstrated above, the NIMBLE function `print`, or equivalently `nimPrint`, prints an arbitrary set of outputs in order and adds a newline character at the end. `cat` or `nimCat` is identical, except without a newline at the end. The NIMBLE function `stop`, or equivalently `nimStop`, throws control to R's error-handling system and can take one string (character) argument.

### 10.3.11 Checking for user interrupts

When you write algorithms that will run for a long time in C++, you may want to explicitly check whether a user has tried to interrupt the execution (e.g. by pressing Control-C). Simply include `checkInterrupt()` in `run` code to do so. If there has been an interrupt, the process will stop and return control to R.

### 10.3.12 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`
- `cat` → `nimCat`
- `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.

# Chapter 11

## Extending BUGS with user-defined distributions and functions

As of Version 0.4, NIMBLE allows you to define your own functions and distributions as `nimbleFunctions` for use in BUGS code. As a result, NIMBLE frees you from being constrained to the functions and distributions just discussed. 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.

### 11.1 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 `nimbleFunctions` requires that you declare the dimensionality of arguments and the returned object (Section 10.3.3). 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 <- nimbleCode({
```

```

    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 <- nimbleCode({
  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.

## 11.2 User-defined distributions

To provide a user-defined distribution, you need to define density (“d”) and simulation (“r”) `nimbleFunctions`, without setup code, for your distribution. In some cases you can then simply use your distribution in BUGS code as you would any distribution already provided by NIMBLE, while in others you need to explicitly register your distribution as described below.

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, but only if you explicitly register the distribution.

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, default = 0)) {
    returnType(double(0))
    logProb <- lgamma(size) - sum(lgamma(x)) + lgamma(sum(alpha)) -
      sum(lgamma(alpha)) + sum(lgamma(alpha + x)) - lgamma(sum(alpha) +
        size)

    if(log) return(logProb)
    else return(exp(logProb))
  })

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

code <- nimbleCode({
  y[1:K] ~ ddirchmulti(alpha[1:K], n)
  for(i in 1:K) {
    alpha[i] ~ dmyexp(1/3)
  }
})

model <- nimbleModel(code, constants = list(K = 5, n = 10))

## defining model...
## building model...
## running calculate on model (any error reports that follow may simply reflect
missing values in model variables) ...
##
## checking model sizes and dimensions...
## note that missing values (NAs) or non-finite values were found in model
variables: y, alpha. This is not an error, but some or all variables may need
to be initialized for certain algorithms to operate properly.
##
## model building finished.

```

The distribution-related functions should take as input the parameters for a single parameterization, which will be the canonical parameterization that NIMBLE will use.

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.
- 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`.
- 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<sup>1</sup>.

### 11.2.1 Using `registerDistributions` for alternative parameterizations and providing other information

Behind the scenes, NIMBLE uses a function called `registerDistributions` to set up new distributions for use in BUGS code. In some circumstances, you will need to call `registerDistributions` directly to provide information that NIMBLE can’t obtain directly from the `nimbleFunctions` you write.

The cases in which you’ll need to explicitly call `registerDistributions` are when you want to do any of the following:

- provide alternative parameterizations,
- indicate a distribution is discrete, and
- provide the range of possible values for a distribution.

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’s an example of using `registerDistributions` to provide an alternative parameterization and to provide the range for the user-defined exponential distribution. We can then use the alternative parameterization in our BUGS code.

---

<sup>1</sup>NIMBLE can’t use R’s standard scoping because it doesn’t work for R reference classes, and `nimbleFunctions` are implemented as custom-generated reference classes.

```

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)
  )
))

## Registering the following user-provided distributions: dmyexp .
## Overwriting the following user-supplied distributions: dmyexp .

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

model <- nimbleModel(code, constants = list(K = 5, n = 10),
  inits = list(alpha = rep(1, 5)))

## defining model...
## building model...
## setting data and initial values...
## running calculate on model (any error reports that follow may simply reflect
missing values in model variables) ...
##
## checking model sizes and dimensions...
## note that missing values (NAs) or non-finite values were found in model
variables: y. This is not an error, but some or all variables may need to
be initialized for certain algorithms to operate properly.
##
## model building finished.

```

There are a few rules for how you specify the information about a distribution that you provide to `registerDistributions`:

- 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 `nimbleFunctions` must match the function name in the `Rdist` entry. If missing, the `Rdist` entry defaults to be the same as the `BUGSdist` entry.
- Your distribution-related functions must take as arguments the parameters in default order, 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.

- You must specify a `types` entry in the list provided to `registerDistributions` if the distribution is multivariate or if any parameter is non-scalar.

Further details on using `registerDistributions` can be found via `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` file in the package source code directory.



# Chapter 12

## Working with NIMBLE models

Here we describe how one can get information about NIMBLE models and carry out operations on a model. While all of this functionality can be used from R, the primary use of getting information and operating a model occurs when writing `nimbleFunctions` (see Chapter 13). Information about node types, distributions, and dimensions can be used to determine algorithm behavior in setup code of `nimbleFunctions`. Information about parameter and bound values would generally be used for algorithm calculations in run code of `nimbleFunctions`. Similarly, carrying out numerical operations on a model would generally be done in run code.

### 12.1 Variables and nodes in NIMBLE models

Section 5.2.8 describes what variables and nodes are in a NIMBLE model and how to nodes and their dependency relationships. Here we review some of NIMBLE’s functionality for determining and accessing variables and nodes in a model.

#### 12.1.1 Determining the variables and nodes in a model

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

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

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.

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

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 the `expandNodeNames` method. For example, to get the spaces correctly inserted into “`X[1,1:5]`”:

Alternatively, for those inclined to R’s less commonly used features, a nice trick is to use its `parse` and `deparse` functions.

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

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

As discussed in Section 6.2.2, you can determine whether a node is flagged as data using `isData`.

### 12.1.2 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 Section 5.2.6 the use of link functions causes new nodes to be introduced. This requires care if you need to initialize values in stochastic declarations with link functions.
- Use of alternative parameterizations of distributions, described in Section 5.2.4. 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.

## 12.2 Accessing information about nodes and variables

### 12.2.1 Determining dependencies in a model

Next we’ll see how to determine the node dependencies (or “descendants”) in a model. There are a variety of arguments to `getDependencies` 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 emanating from the node(s) specified as input. This is what would be needed to calculate a Metropolis-Hastings acceptance probability in MCMC, for example.

### 12.2.2 Getting distributional information about a node

We briefly demonstrate some of the functionality for information about a node here, but refer readers to the R help on `modelBaseClass` for full details.

Here is an example model, with use of various functions to determine information about nodes or variables.

```
code <- nimbleCode({
  for(i in 1:4)
    y[i] ~ dnorm(mu, sd = sigma)
  mu ~ T(dnorm(0, 5), -20, 20)
  sigma ~ dunif(0, 10)
})
m <- nimbleModel(code, data = list(y = rnorm(4)),
                  inits = list(mu = 0, sigma = 1))

m$isEndNode('y')
m$getDistribution('sigma')
m$isDiscrete(c('y', 'mu', 'sigma'))
m$isDeterm('mu')
m$getDimension('mu')
m$getDimension('mu', includeParams = TRUE)
```

Note that any variables provided to these functions are expanded into their constituent node names, so the length of results may not be the same length as the input vector of node and variable names. However the order of the results should be preserved relative to the order of the inputs, once the expansion is accounted for.

### 12.2.3 Getting information about a distribution

One can also get generic information about a distribution based on the name of the distribution. In particular, one can determine whether a distribution was provided by the user (`isUserDefined`), whether a distribution provides CDF and quantile functions (`pqDefined`), whether a distribution is a discrete distribution (`isDiscrete`), the parameter names (include alternative parameterizations) for a distribution (`getParamNames`), and the dimension of the distribution and its parameters (`getDimension`). For more extensive information, please see the R help for `getDistributionInfo`.

### 12.2.4 Getting distribution parameter values for a node

The function `getParam` provides access to values of the parameters of a node's distribution. Like the above functions, `getParam` can be used as global function taking a model as the first argument, or it can be used as a model member function. The next two arguments must be the name of one (stochastic) node and the name of a parameter for the distribution followed by that node. The parameter does not have to be one of the parameters used when the node was declared. Alternative parameterization values can also be obtained. See section(5.2.4) for available parameterizations. (These can also be seen via `distributionsInputList`.)

Here is an example:

```
gammaModel <- nimbleModel(  
  nimbleCode({  
    a ~ dlnorm(0, 1)  
    x ~ dgamma(shape = 2, scale = a)  
  }), data = list(x = 2.4), inits = list(a = 1.2))  
getParam(gammaModel, 'x', 'scale')  
getParam(gammaModel, 'x', 'rate')  
gammaModel$getParam('x', 'rate')
```

`getParam` also works in compiled `nimbleFunctions`.

### 12.2.5 Getting distribution bounds for a node

The function `getBound` provides access to the lower and upper bounds of the distribution for a node. In most cases these bounds will be fixed based on the distribution, but for the uniform distribution the bounds are the parameters of the distribution, and when truncation (see Section 5.2.7) is used, the bounds will be determined by the truncation. Like the functions described in the previous section, `getBound` can be used as global function taking a model as the first argument, or it can be used as a model member function. The next two arguments must be the name of one (stochastic) node and either `'lower'` or `'upper'` indicating whether the lower or upper bound is desired. For multivariate nodes the bound is a scalar that is the bound for all elements of the node, as we do not handle truncation for multivariate nodes.

Here is an example:

```
exampleModel <- nimbleModel(  
  nimbleCode({  
    y ~ T(dnorm(mu, sd = sig), a, Inf)  
    a ~ dunif(-1, b)  
    b ~ dgamma(1, 1)  
  }), inits = list(a = -0.5, mu = 1, sig = 1, b = 4),  
  data = list(y = 4))  
getBound(exampleModel, 'y', 'lower')  
getBound(exampleModel, 'y', 'upper')  
exampleModel$b <- 3  
exampleModel$calculate(exampleModel$getDependencies('b'))  
getBound(exampleModel, 'a', 'upper')  
exampleModel$getBound('b', 'lower')
```

`getBound` also works in compiled `nimbleFunctions`. In fact, we anticipate that most use of `getBound` will be for algorithms, such as for the reflection version of the random walk MCMC sampler.

## 12.3 Carrying out model calculations

### 12.3.1 Core model operations: calculation and simulation

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 compare 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. This is useful when one wants to change the value or values of node(s) in the model (e.g., by setting a value or **simulate**) and then determine the change in the log probability, such as needed for a Metropolis-Hastings acceptance probability.

Each of these functions is accessed as a member function of a model object, taking a vector of node names as an argument. If there is more than one node name, **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. Next we show an example using **simulate**

### 12.3.2 **simNodes**, **calcNodes**, and **getLogProbs**

CJP commented elsewhere that not clear if these **nf**'s could be replaced with an extra argument to **calculate/simulate/getLogProb** that allows on to specify that dependencies should be included

**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 will be topologically sorted to simulate in the correct order. For **calcNodes** and **getLogProb**, the nodes will be sorted and dependent nodes will be 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, check = FALSE)
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$x
cSimpleModel$y
#simulating x and y
cSimXY$run()
cSimpleModel$x
cSimpleModel$y

cCalcXDep$run()

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

cSimXY$run()

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

```

### 12.3.3 Example: simulating arbitrary collections of nodes

Note the following.

1. `simulate(model, nodes)` is equivalent to `model$simulate(nodes)`. You can use either, but the latter is encouraged and the former may be deprecated in the future.
2. 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`.
3. An arbitrary number of nodes can be provided as a character vector.
4. Simulations will be done in the order provided, so in practice the nodes should 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.
5. The data nodes `z` were not simulated into until `includeData = TRUE` was used.

Use of `calculate`, `calculateDiff` and `getLogProb` are similar to `simulate`, except that they return a value (described above) and they have no `includeData` argument.

### 12.3.4 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:

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`.

## 12.4 The *modelValues* data structure

`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 contains one object of each variable, which may be multivariate. 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.

As with the material in the rest of this chapter, `modelValues` objects will generally be used in nimbleFunctions that interact with models (see Chapter 13, so one may want to

read this section after an initial reading of that chapter. `modelValues` objects can be defined either in setup code or separately in R (and then passed as an argument to setup code). The `modelValues` object can then be used in run code of `nimbleFunctions`.

### 12.4.1 Creating `modelValues` objects

Here is a simple example of creating a `modelValues` object:

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

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 `modelValuesConf` function, like this:

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

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

The arguments to `modelValuesConf` are matching lists of variable names, types, and sizes. See `help(modelValuesConf)` 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`.

## 12.4.2 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']]

# 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)
resize(customMV, 3)
getsize(customMV)
customMV$a
```

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'
as.matrix(customMV)         # convert all variables
```

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', ])
```

Once we have a `modelValues` object, we can see the structure of its contents via the `varNames` and `sizes` components of the object.

```
customMV$varNames
customMV$sizes
```

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]

alter_a(customMV)
customMV['a',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.

### Automating calculation and simulation using `modelValues`

The `nimbleFunctions` `simNodesMV`, `calcNodesMV`, and `getLogProbsMV` can be used to operate on a model based on rows in 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)
rCalcManyXDepts <- calcNodesMV(simpleModel, nodes = 'x', mv = mv)
rGetLogProbMany <- getLogProbNodesMV(simpleModel,
                                     nodes = 'x', mv = mv)

cSimManyXY <- compileNimble(rSimManyXY, project = simpleModel)
cCalcManyXDepts <- compileNimble(rCalcManyXDepts, project = simpleModel)
cGetLogProbMany <- compileNimble(rGetLogProbMany, project = simpleModel)

cSimManyXY$run(m = 5) # simulating 5 times
cCalcManyXDepts$run(saveLP = TRUE) # calculating
cGetLogProbMany$run() #
```

# Chapter 13

## Writing nimbleFunctions that interact with models

### 13.1 Overview

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<sup>1</sup>. 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 `modelValuesConf`, 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`<sup>2</sup>.
  - (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:

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

---

<sup>1</sup>normally the value of the last evaluated code, or the argument to `return()`.

<sup>2</sup>This can be omitted if you don’t need it.

(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 Section 12.2.1) 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<sup>3</sup>. We say `logProbCalcPlusA` is an “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 (one can think of it as a generator), 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 `nimbleFunction` is one object in the class. The `setup` arguments are used to define member data in the object.

## 13.2 Using and compiling `nimbleFunctions`

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

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

## 13.3 Writing setup code

### 13.3.1 Useful tools for setup functions

The `setup` function is typically used to determine information on nodes in a model, set up `modelValues` objects, set up (nested) `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 ??.

**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.

---

<sup>3</sup>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.

**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.

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

### 13.3.2 Accessing and modifying numeric values from setup

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

### 13.3.3 Determining numeric types in nimbleFunctions

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.

### 13.3.4 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.

## 13.4 Writing run code

In Section 10.3 we described the functionality of the NIMBLE language that could be used in run code in which models were not used. When writing nimbleFunction run code that inter-

---

<sup>4</sup>Actually, they can be, but only for uncompiled nimbleFunctions

acts with models, one can use that functionality already described as well as the functionality discussed in this section.

### 13.4.1 Driving models: `calculate`, `calculateDiff`, `simulate`, `getLogProb`

These four functions are the primary ways to operate a model. Their syntax was explained in Section 12.3. 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, and functions such as `getDependencies` and `expandNodeNames` will always do so.

It is possible to use an entire vector of node names, single elements indexed by a variable, or fixed ranges (indexed by constants) , but not ranges indexed by a variable. For example

```
myModel$calculate(nodes)
```

and

```
myModel$calculate(nodes[i])
```

and

```
myModel$calculate(nodes[1:3])
```

will all compile correctly, but

```
myModel$calculate(nodes[1:i])
```

is not allowed.

### 13.4.2 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. This can be done using `values()`:

```
## get values from a set of model nodes into a vector
P <- values(model, nodes)
## or put values from a vector into a set of 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.

`values(model, nodes)` may be used as a vector in other expressions, e.g. `Y <- A %*% values(model, nodes) + b`.

### 13.4.3 Using modelValues objects

This didn't seem to fit in Using Models chapter because of all the discussion and examples with nimbleFunctions, though some of this functionality can be used from R. Thoughts welcome.

The `modelValues` structure was introduced in Section 12.4. 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.

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`. See `help(nimCopy)` for argument details.

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` function to use these `modelValues`:

Once the `nimbleFunction` is built, the `modelValues` object can be accessed 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.

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.

### 13.4.4 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 models defined from BUGS code are never created as 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$z` or `model[['z']]`, NIMBLE will try to do the right thing by treating this as a scalar. In the event of problems<sup>5</sup>, a more explicit way to access `z` is `model$z[1]` or `model[['z']][1]`.

### 13.4.5 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. Any method can then call another method.

### 13.4.6 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 `$`.

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 may not be shown here because this document is created with `knitr` and for some reason output printed from C++ does not make it into `knitr` output.

### 13.4.7 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 Version 0.6-3 it is limited to simple, single-level inheritance.

Here is how it works:

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.

### 13.4.8 Character objects

NIMBLE provides limited uses of character objects in `run` 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.

---

<sup>5</sup>please tell us!

Note that character vectors of model node and variable names are processed during compilation. 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++.

### 13.4.9 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:

You'll notice that:

- After execution of the compiled function, access to the `X`, `Y`, and `Z` is the same as for the uncompiled case. This occurs because `CmyUseDataNF` is an interface to the compiled version of `myUseDataNF`, and it provides access to member objects and functions. In this case, one member object is `myDataNF`, which is an interface to the compiled version of `myUseDataNF$myDataNF`, which in turn provides access to `X`, `Y`, and `Z`. To reduce memory use, NIMBLE defaults to *not* providing full interfaces to nested `nimbleFunctions` like `myUseDataNF$myDataNF`. In this example we made it provide full interfaces by setting the `buildInterfacesForCompiledNestedNimbleFunctions` option via `nimbleOptions` to `TRUE`. If we had left that option `FALSE` (its default value), we could still get to the values of interest using

```
valueInCompiledNimbleFunction(CmyDataNF, 'X')
```

- We need to take care that at the time of compilation, the `X`, `Y` and `Z` values contain 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.

## 13.5 Example: writing user-defined samplers to extend NIMBLE's MCMC engine

One important use of `nimbleFunctions` is to write additional samplers that can be used in NIMBLE's MCMC engine. This allows a user to write a custom sampler for one or more nodes in a model, as well as for programmers to provide general samplers for use in addition to the library of samplers provided with NIMBLE.

The following code illustrates how a NIMBLE developer would implement and use a Metropolis-Hastings random walk sampler with fixed proposal standard deviation.

```

my_RW <- nimbleFunction(

  contains = sampler_BASE,

  setup = function(model, mvSaved, target, control) {
    scale <- control$scale    ## proposal standard deviation
    calcNodes <- model$getDependencies(target)
  },

  run = function() {
    model_lp_initial <- getLogProb(model, calcNodes) ## initial model logProb
    proposal <- rnorm(1, model[[target]], scale)      ## generate proposal
    model[[target]] <- proposal                       ## store proposal into model
    model_lp_proposed <- calculate(model, calcNodes) ## proposal model logProb

    ## 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

    ## keep the model and mvSaved objects consistent
    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)
  },

  methods = list( reset = function () {} )
)

```

The name of this sampler function, for the purposes of using it in an MCMC algorithm, is `my_RW`. Thus, this sampler can be added to an existing MCMC configuration object `conf` using:

```

mcmcConf$addSampler(target = 'x', type = 'my_RW',
                    control = list(scale = 0.1))

```

To be used within the MCMC engine, sampler functions definitions must adhere exactly to the following:

- Include the contains statement `contains = sampler_BASE`.

- The `setup` function must have the exact four arguments `model`, `mvSaved`, `target`, `control`, in that order.
- The `run` function must accept no arguments, and have no return value. Further, after execution it must leave the `mvSaved` `modelValues` object as an up-to-date copy of the values and `logProb` values in the model object.
- Have a member method called `reset`, which takes no arguments and has no return value.

The purpose of the `setup` function is generally two-fold. First, to extract control parameters from the `control` list; in the example, the proposal standard deviation `scale`. Second, to generate any sets of nodes needed in the `run` function. In many sampling algorithms, as here, `calcNodes` is used to represent the target node(s) and dependencies up to the first layer of stochastic nodes, as this is precisely what is required for calculating the Metropolis-Hastings acceptance probability. These probability calculations are done using `model$calculate(calcNodes)`.

In the `run` function, the `mvSaved` `modelValues` object is kept up-to-date with the current state of the model, depending on whether the proposed changed was accepted. This is done using the `copy` function, to copy values between the model and `mvSaved` objects.

## 13.6 Copying nimbleFunctions (and NIMBLE models)

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` will create a new copy of the model from the same model definition (Section 6.1.3). 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`.

## 13.7 Debugging nimbleFunctions

One of the main reasons that NIMBLE provides an R (uncompiled) version of each nimbleFunction is for debugging. One can call `debug` on nimbleFunction methods (in particular the main *run* method) and then step through the code in R using R’s debugger. One can also insert `browser` calls into run code and then run the nimbleFunction from R.

In contrast, directly debugging a compiled nimbleFunction is difficult, although those familiar with running R through a debugger and accessing the underlying C code may be able to operate similarly with NIMBLE code.

TODO: basic info on inserting browser in nf run code and running in R as well as using `debug(nf$run)`

## 13.8 Some options for reducing memory usage

TODO: needs to be revised so only memory use is covered here and other material is moved as appropriate

NIMBLE can make a lot of objects in its processing, and some of them use R features like reference classes that are not light in memory usage. We have noticed that building large models can use lots of memory. To help alleviate this, we provide two options, which can be controlled via `nimbleOptions`.

As noted above, the option `buildInterfacesForCompiledNestedNimbleFunctions` defaults to `FALSE`, which means NIMBLE will not build full interfaces to compiled nimbleFunctions that only appear within other nimbleFunctions. If you want access to all such nimbleFunctions, use the option `buildInterfacesForCompiledNestedNimbleFunctions = TRUE`.

The option `clearNimbleFunctionsAfterCompiling` is more drastic, and it is experimental, so “buyer beware”. This will clear much of the contents of an uncompiled nimbleFunction object after it has been compiled in an effort to free some memory. We expect to be able to keep making NIMBLE more efficient – faster execution and lower memory use – in the future.

# Bibliography

- [1] Andrieu, C., A. Doucet, and R. Holenstein (2010). Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 72(3), 269–342.
- [2] George, E. I., U. E. Makov, and A. F. M. Smith (1993). Conjugate likelihood distributions. *Scandinavian Journal of Statistics* 20(2), 147–156.
- [3] Lunn, D., D. Spiegelhalter, A. Thomas, and N. Best (2009). The BUGS project: Evolution, critique and future directions. *Statistics in Medicine* 28(25), 3049–3067.
- [4] Murray, I., R. Prescott Adams, and D. J. C. MacKay (2010). Elliptical slice sampling. *ArXiv e-prints*.
- [5] Neal, R. M. (2003). Slice sampling. *The Annals of Statistics* 31(3), 705–741.
- [6] Roberts, G. O. and S. K. Sahu (1997). Updating schemes, correlation structure, blocking and parameterization for the Gibbs sampler. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 59(2), 291–317.
- [7] Shaby, B. and M. Wells (2011). Exploring an adaptive Metropolis algorithm. *Department of Statistics, Duke University* 2011–14.