NIMBLE User Manual

NIMBLE Development Team

Version 1.0.0

https://r-nimble.org
https://github.com/nimble-dev/nimble

Contents

Part I

Introduction

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 version 1.0.0, 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 (see R-nimble.org/more/issues-and-groups) 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 our GitHub repository.

If you use NIMBLE in your work, please cite us, as this helps justify past and future funding for the development of NIMBLE. For more information, please call `citation('nimble')` in R.

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[1].
2. A library of algorithms such as MCMC.
3. A language, called NIMBLE, embedded within and similar in style to R, for writing algorithms that operate on models written in BUGS.

[1] See Chapter 5 for information about NIMBLE's version of 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++.

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[2].

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

1.2 How to use this manual

We suggest everyone start with the Lightning Introduction in Chapter 2.

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.

[2] The packages Rcpp and RcppEigen provide different ways of connecting C++, the Eigen library and R. In those packages you program directly in C++, while in NIMBLE you program in 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[1]. 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.

```
pumpCode <- nimbleCode({
  for (i in 1:N){
      theta[i] ~ dgamma(alpha,beta)
      lambda[i] <- theta[i]*t[i]
      x[i] ~ dpois(lambda[i])
  }
  alpha ~ dexp(1.0)
  beta ~ dgamma(0.1,1.0)
})
```

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

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

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

pumpInits <- list(alpha = 1, beta = 1,
                  theta = rep(0.1, pumpConsts$N))
```

Here x[i] is the number of failures recorded during a time duration of length t[i] for the ith 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.

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

pump$getNodeNames()
```

```
##  [1] "alpha"               "beta"                "lifted_d1_over_beta"
##  [4] "theta[1]"            "theta[2]"            "theta[3]"
##  [7] "theta[4]"            "theta[5]"            "theta[6]"
## [10] "theta[7]"            "theta[8]"            "theta[9]"
## [13] "theta[10]"           "lambda[1]"           "lambda[2]"
## [16] "lambda[3]"           "lambda[4]"           "lambda[5]"
## [19] "lambda[6]"           "lambda[7]"           "lambda[8]"
## [22] "lambda[9]"           "lambda[10]"          "x[1]"
## [25] "x[2]"                "x[3]"                "x[4]"
## [28] "x[5]"                "x[6]"                "x[7]"
## [31] "x[8]"                "x[9]"                "x[10]"
```

```
pump$x
```

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

```
pump$logProb_x
```

```
##  [1]  -2.998011  -1.118924  -1.882686  -2.319466  -4.254550 -20.739651
##  [7]  -2.358795  -2.358795  -9.630645 -48.447798
```

```
pump$alpha
```

```
## [1] 1
```

```
pump$theta
```

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

```
pump$lambda
```

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

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

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

```
pump$plotGraph()
```

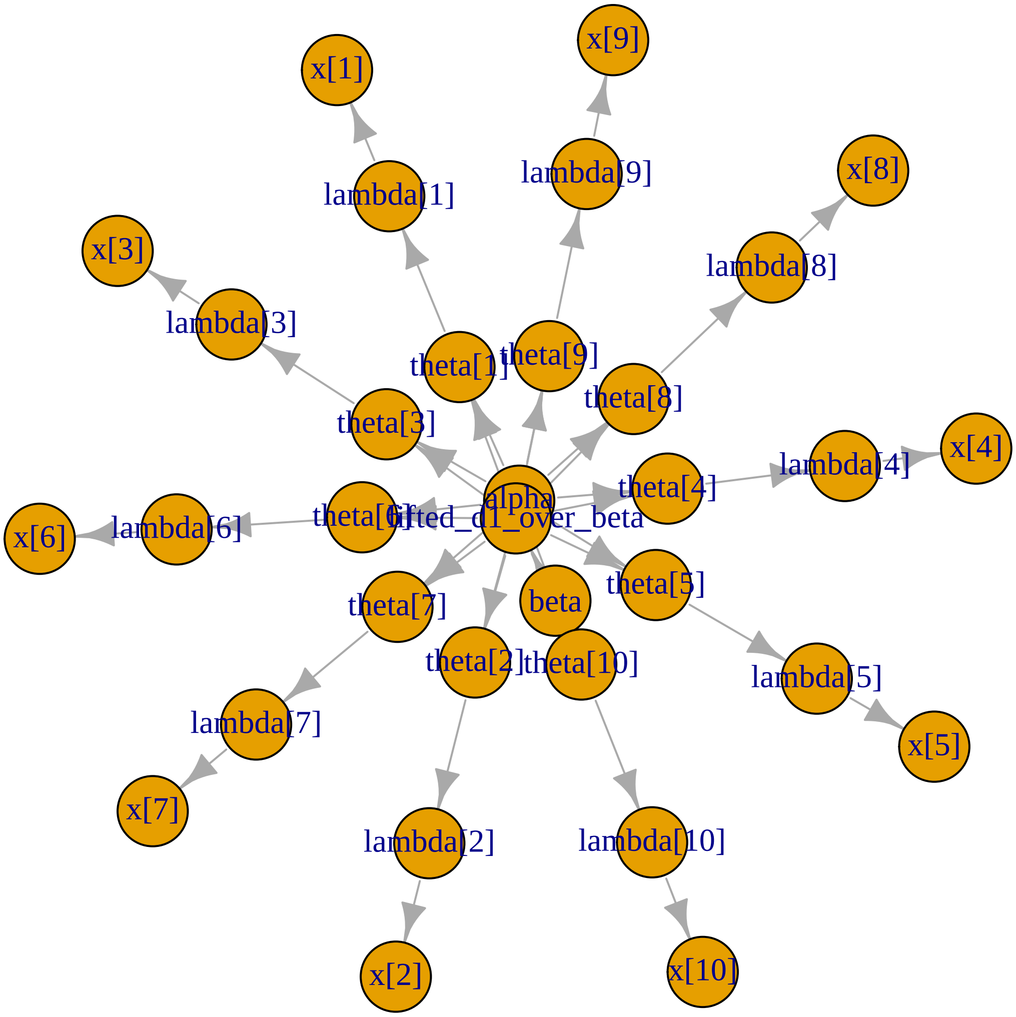


Figure 2.1: Directed Acyclic Graph plot of the pump model, thanks to the igraph package

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.

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

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

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

```
## [1] "lifted_d1_over_beta"
```

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

```
## [1] 1
```

```
# Now let's simulate new theta values
set.seed(1) # This makes the simulations here reproducible
pump$simulate("theta")
pump$theta   # the new theta values
```

```
##  [1] 0.15514136 1.88240160 1.80451250 0.83617765 1.22254365 1.15835525
##  [7] 0.99001994 0.30737332 0.09461909 0.15720154
```

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

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

```
pump$logProb_x
```

```
##  [1]  -2.998011  -1.118924  -1.882686  -2.319466  -4.254550 -20.739651
##  [7]  -2.358795  -2.358795  -9.630645 -48.447798
```

```
pump$getLogProb("x") # The sum of logProb_x
```

```
## [1] -96.10932
```

```
pump$calculate(pump$getDependencies(c("theta")))
```

```
## [1] -262.204
```

```
pump$lambda  # Now they have.
```

```
##  [1]  14.6298299  29.5537051 113.5038360 105.3583839   6.4061287  36.3723548
##  [7]   1.0395209   0.3227420   0.1987001   1.6506161
```

```
pump$logProb_x
```

```
##  [1]  -6.002009 -26.167496 -94.632145 -65.346457  -2.626123  -7.429868
##  [7]  -1.000761  -1.453644  -9.840589 -39.096527
```

Notice that the first getDependencies call returned dependencies from alpha and beta down to the next stochastic nodes in the model. The second call requested only deterministic dependencies. The call to 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.

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

```
##  [1] 0.15514136 1.88240160 1.80451250 0.83617765 1.22254365 1.15835525
##  [7] 0.99001994 0.30737332 0.09461909 0.15720154
```

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 One-line invocation of MCMC

The most direct approach to invoking NIMBLE's MCMC engine is using the nimbleMCMC function. This function would generally take the code, data, constants, and initial values as input, but it can also accept the (compiled or uncompiled) model object as an argument. It provides a variety of options for executing and controlling multiple chains of NIMBLE's default MCMC algorithm, and returning posterior samples, posterior summary statistics, and/or WAIC values.

For example, to execute two MCMC chains of 10,000 samples each, and return samples, summary statistics, and WAIC values:

```
mcmc.out <- nimbleMCMC(code = pumpCode, constants = pumpConsts,
                       data = pumpData, inits = pumpInits,
                       nchains = 2, niter = 10000,
                       summary = TRUE, WAIC = TRUE,
                       monitors = c('alpha','beta','theta'))
```

```
##   [Warning] There are 7 individual pWAIC values that are greater than 0.4. This may indicate
```

```
names(mcmc.out)
```

```
## [1] "samples" "summary" "WAIC"
```

```
mcmc.out$summary
```

```
## $chain1
##                  Mean     Median     St.Dev.   95%CI_low 95%CI_upp
## alpha     0.69804352 0.65835063 0.27037676 0.287898244 1.3140461
## beta      0.92862598 0.82156847 0.54969128 0.183699137 2.2872696
## theta[1]  0.06019274 0.05676327 0.02544956 0.021069950 0.1199230
## theta[2]  0.10157737 0.08203988 0.07905076 0.008066869 0.3034085
## theta[3]  0.08874755 0.08396502 0.03760562 0.031186960 0.1769982
## theta[4]  0.11567784 0.11301465 0.03012598 0.064170937 0.1824525
## theta[5]  0.60382223 0.54935089 0.31219612 0.159731108 1.3640771
## theta[6]  0.61204831 0.60085518 0.13803302 0.372712375 0.9135269
## theta[7]  0.90263434 0.70803389 0.73960182 0.074122175 2.7598261
## theta[8]  0.89021051 0.70774794 0.72668155 0.072571029 2.8189252
## theta[9]  1.57678136 1.44390008 0.76825189 0.455195149 3.4297368
## theta[10] 1.98954127 1.96171250 0.42409802 1.241383787 2.9012192
##
## $chain2
##                 Mean     Median    St.Dev.   95%CI_low 95%CI_upp
## alpha     0.69101961 0.65803654 0.26548378 0.277195564 1.2858148
## beta      0.91627273 0.81434426 0.53750825 0.185772263 2.2702428
## theta[1]  0.05937364 0.05611283 0.02461866 0.020956151 0.1161870
## theta[2]  0.10017726 0.08116259 0.07855024 0.008266343 0.3010355
## theta[3]  0.08908126 0.08390782 0.03704170 0.031330829 0.1736876
## theta[4]  0.11592652 0.11356920 0.03064645 0.063595333 0.1829574
## theta[5]  0.59755632 0.54329373 0.31871551 0.149286703 1.3748728
## theta[6]  0.61080189 0.59946693 0.13804343 0.371373877 0.9097319
## theta[7]  0.89902759 0.70901502 0.72930369 0.076243503 2.7441445
## theta[8]  0.89954594 0.70727079 0.73345905 0.071250926 2.8054633
## theta[9]  1.57530029 1.45005738 0.75242164 0.469959364 3.3502795
## theta[10] 1.98911473 1.96227061 0.42298189 1.246910723 2.9102326
##
## $all.chains
##                 Mean     Median    St.Dev.  95%CI_low 95%CI_upp
## alpha     0.69453156 0.65803654 0.26795776 0.28329854 1.2999319
## beta      0.92244935 0.81828160 0.54365539 0.18549077 2.2785444
## theta[1]  0.05978319 0.05646474 0.02504028 0.02102807 0.1183433
## theta[2]  0.10087731 0.08162361 0.07880204 0.00811108 0.3017967
## theta[3]  0.08891440 0.08394667 0.03732417 0.03123228 0.1749967
## theta[4]  0.11580218 0.11326039 0.03038683 0.06385253 0.1827382
## theta[5]  0.60068928 0.54668011 0.31548032 0.15363752 1.3686801
## theta[6]  0.61142510 0.60015416 0.13803618 0.37203765 0.9122467
## theta[7]  0.90083096 0.70852800 0.73445465 0.07550465 2.7534885
## theta[8]  0.89487822 0.70761105 0.73007484 0.07211191 2.8067373
## theta[9]  1.57604083 1.44719278 0.76035931 0.46374515 3.3866706
## theta[10] 1.98932800 1.96195345 0.42352979 1.24334249 2.9068229
```

```
mcmc.out$WAIC
```

```
## nimbleList object of type waicNimbleList
## Field "WAIC":
## [1] 48.69896
## Field "lppd":
## [1] -19.99653
## Field "pWAIC":
## [1] 4.352945
```

See Section 7.1 or `help(nimbleMCMC)` for more details about using `nimbleMCMC`.

Note that the WAIC value varies depending on what quantities are treated as parameters. See Section 7.8 or `help(waic)` for more details.

2.5 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.

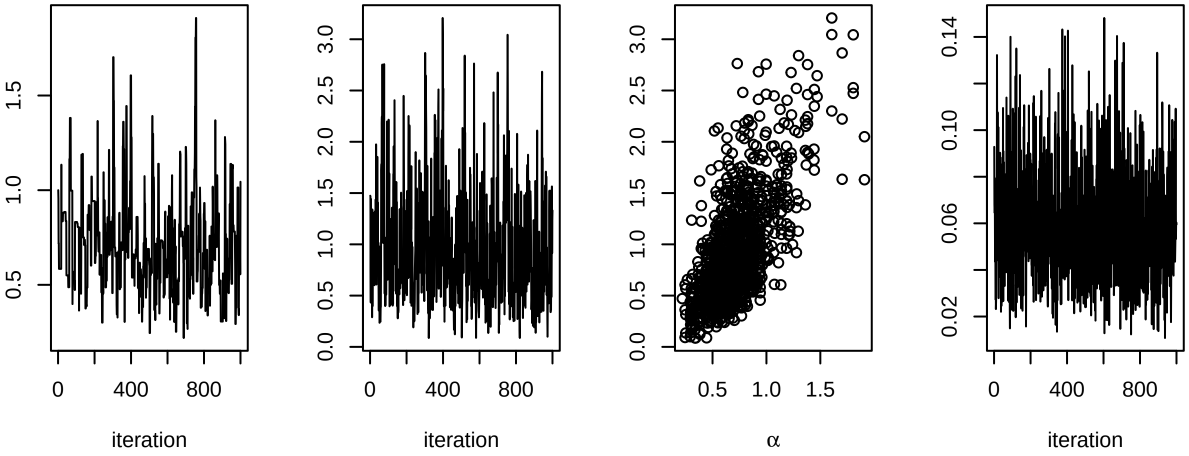
```
pumpConf <- configureMCMC(pump, print = TRUE)
```

```
## ===== Monitors =====
## thin = 1: alpha, beta
## ===== Samplers =====
## RW sampler (1)
##   - alpha
## conjugate sampler (11)
##   - beta
##   - theta[]  (10 elements)
```

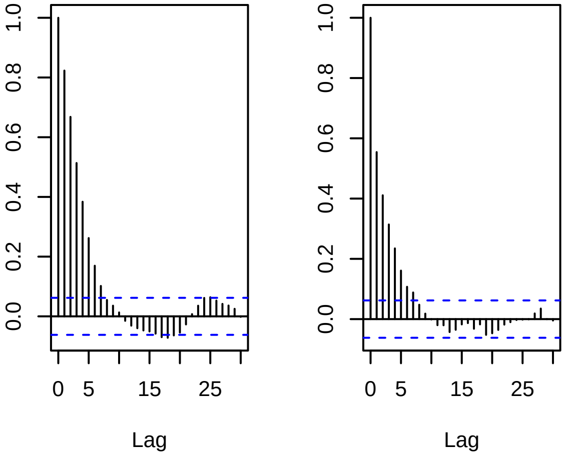
```
pumpConf$addMonitors(c("alpha", "beta", "theta"))
```

```
## thin = 1: alpha, beta, theta
```

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

niter <- 1000
set.seed(1)
samples <- runMCMC(CpumpMCMC, niter = niter)

par(mfrow = c(1, 4), mai = c(.6, .4, .1, .2))
plot(samples[ , "alpha"], type = "l", xlab = "iteration",
     ylab = expression(alpha))
plot(samples[ , "beta"], type = "l", xlab = "iteration",
     ylab = expression(beta))
plot(samples[ , "alpha"], samples[ , "beta"], xlab = expression(alpha),
     ylab = expression(beta))
plot(samples[ , "theta[1]"], type = "l", xlab = "iteration",
     ylab = expression(theta[1]))
```



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



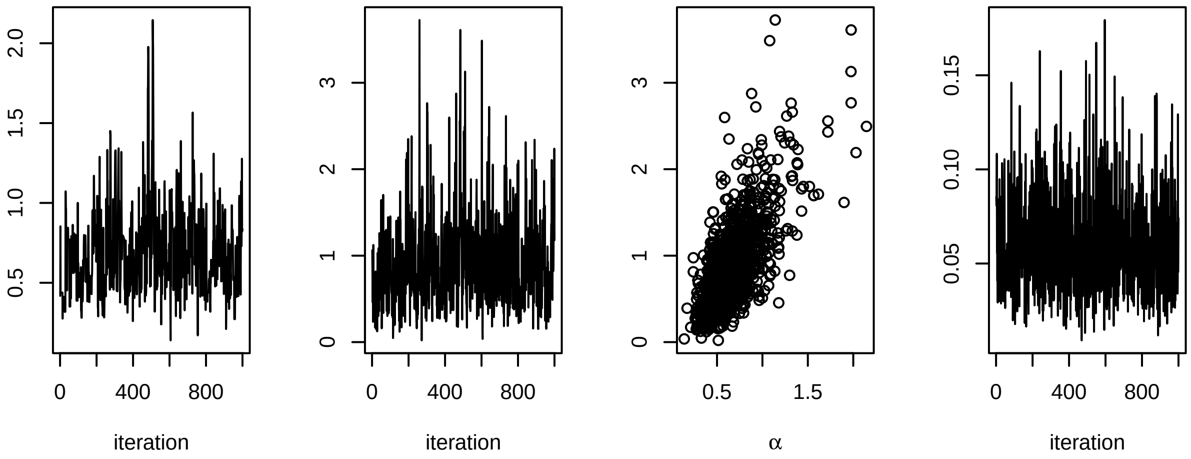
Notice the posterior correlation between `alpha` and `beta`. A measure of the mixing for each is the autocorrelation for each parameter, shown by the `acf` plots.

2.6 Customizing the MCMC

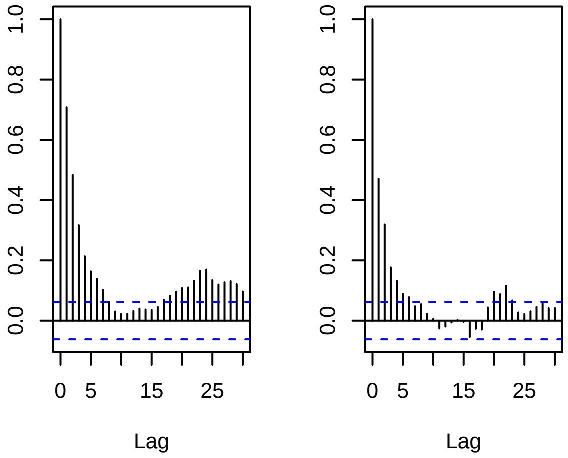
Let's add an adaptive block sampler on `alpha` and `beta` jointly and see if that improves the mixing.

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

pumpMCMC2 <- buildMCMC(pumpConf)

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

set.seed(1)
CpumpNewMCMC$run(niter)
samplesNew <- as.matrix(CpumpNewMCMC$mvSamples)
```

```r
par(mfrow = c(1, 4), mai = c(.6, .4, .1, .2))
plot(samplesNew[ , "alpha"], type = "l", xlab = "iteration",
     ylab = expression(alpha))
plot(samplesNew[ , "beta"], type = "l", xlab = "iteration",
     ylab = expression(beta))
plot(samplesNew[ , "alpha"], samplesNew[ , "beta"], xlab = expression(alpha),
     ylab = expression(beta))
plot(samplesNew[ , "theta[1]"], type = "l", xlab = "iteration",
     ylab = expression(theta[1]))
```



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



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

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

2.7 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[2].

```
pump2 <- pump$newModel()

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

pumpMCEM <- buildMCEM(model = pump2, latentNodes = "theta[1:10]",
                      boxConstraints = box)
pumpMLE <- pumpMCEM$run()
```

```
## Iteration Number: 1.
## Current number of MCMC iterations: 1000.
## Parameter Estimates:
##     alpha      beta
## 0.8160625 1.1230921
## Convergence Criterion: 1.001.
## Iteration Number: 2.
## Current number of MCMC iterations: 1000.
## Parameter Estimates:
##     alpha      beta
## 0.8045037 1.1993128
## Convergence Criterion: 0.0223464.
## Iteration Number: 3.
## Current number of MCMC iterations: 1250.
## Parameter Estimates:
##     alpha      beta
## 0.8203178 1.2497067
## Convergence Criterion: 0.004913688.
## Iteration Number: 4.
## Current number of MCMC iterations: 3032.
## Parameter Estimates:
##     alpha      beta
## 0.8226618 1.2602452
## Convergence Criterion: 0.0004201045.
```

```
pumpMLE
```

```
##     alpha      beta
## 0.8226618 1.2602452
```

Both estimates are within 0.01 of the values reported by George et al. (1993)[3]. Some discrepancy is to be expected since it is a Monte Carlo algorithm.

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

[3] Table 2 of the paper accidentally swapped the two estimates.

2.8 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*:

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

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

Here are a few things to notice about the nimbleFunction.

1. The `setup` function is written in R. It creates relevant information specific to our model for use in the run-time code.

2. The `setup` code creates a *modelValues* object to hold multiple sets of values for variables in the model provided.
3. The `run` function is written in NIMBLE. It carries out the calculations using the information determined once for each set of `model` and `nodes` arguments by the setup code. The run-time code is what will be compiled.
4. The `run` code requires type information about the argument `n`. In this case it is a scalar integer.

5. The for-loop looks just like R, but only sequential integer iteration is allowed.
6. The functions `calculate` and `simulate`, which were introduced above in R, can be used in NIMBLE.
7. The special function `copy` is used here to record values from the model into the modelValues object.

8. Multiple instances, or 'specializations', can be made by calling `simNodesMany` with different arguments. Above, `simNodesTheta1to5` has been made by calling `simNodesMany` with the `pump`

model and nodes "theta[1:5]" as inputs to the setup function, while simNodesTheta6to10 differs by providing "theta[6:10]" as an argument. The returned objects are objects of a uniquely generated R reference class with fields (member data) for the results of the setup code and a run method (member function).

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

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

```
set.seed(1)  # make the calculation repeatable
pump$alpha <- pumpMLE[1]
pump$beta <- pumpMLE[2]
# make sure to update deterministic dependencies of the altered nodes
pump$calculate(pump$getDependencies(c("alpha","beta"), determOnly = TRUE))
```

```
## [1] 0
```

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

```
## [[1]]
##  [1] 0.21829875 1.93210970 0.62296551 0.34197267 3.45729603 1.15835525
##  [7] 0.99001994 0.30737332 0.09461909 0.15720154
##
## [[2]]
##  [1] 0.82759982 0.08784057 0.34414959 0.29521943 0.14183505 1.15835525
##  [7] 0.99001994 0.30737332 0.09461909 0.15720154
```

```
simNodesTheta1to5$mv[["logProb_x"]][1:2]
```

```
## [[1]]
##  [1] -10.250111 -26.921849 -25.630612 -15.594173 -11.217566  -7.429868
##  [7]  -1.000761  -1.453644  -9.840589 -39.096527
##
## [[2]]
##  [1] -61.043876  -1.057668 -11.060164 -11.761432  -3.425282  -7.429868
##  [7]  -1.000761  -1.453644  -9.840589 -39.096527
```

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

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

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

```
Cpump$calculate(Cpump$getDependencies(c("alpha","beta"), determOnly = TRUE))
```

```
## [1] 0
```

```
Cpump$theta <- saveTheta

set.seed(1)
CsimNodesTheta1to5$run(10)
CsimNodesTheta1to5$mv[["theta"]][1:2]
```

```
## [[1]]
##  [1] 0.21829875 1.93210970 0.62296551 0.34197267 3.45729603 1.15835525
##  [7] 0.99001994 0.30737332 0.09461909 0.15720154
##
## [[2]]
##  [1] 0.82759982 0.08784057 0.34414959 0.29521943 0.14183505 1.15835525
##  [7] 0.99001994 0.30737332 0.09461909 0.15720154
```

```
CsimNodesTheta1to5$mv[["logProb_x"]][1:2]
```

```
## [[1]]
##  [1] -10.250111 -26.921849 -25.630612 -15.594173 -11.217566  -2.782156
##  [7]  -1.042151  -1.004362  -1.894675  -3.081102
##
## [[2]]
##  [1] -61.043876  -1.057668 -11.060164 -11.761432  -3.425282  -2.782156
##  [7]  -1.042151  -1.004362  -1.894675  -3.081102
```

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 (Lunn et al., 2009). 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 documntation 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[1]. 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 13.

3.2 nimbleFunctions for writing algorithms

NIMBLE provides *nimbleFunction*s for writing functions that can (but don't have to) use BUGS models. The main ways that nimbleFunctions 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[2].

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.

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] or multiple model objects

[2] That's why it's easy to use new functions and distributions written as nimbleFunctions in BUGS code.

1. NIMBLE creates a working R version of the nimbleFunction. This is most useful for debugging (Section 15.7).
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.

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.

More about writing algorithms is in Chapter 15.

3.3 The NIMBLE algorithm library

In Version 1.0.0, 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, among others. You can modify sampler choices and configurations from R before compiling the MCMC. You can also write new samplers as nimbleFunctions.

2. Reversible jump MCMC for variable selection.

3. WAIC calculation for model comparison after an MCMC algorithm has been run.

4. A set of particle filter (sequential Monte Carlo) methods including a basic bootstrap filter, auxiliary particle filter, ensemble Kalman Filter, iterated filtering 2 filter (IF2), and Liu-West filter.

5. An ascent-based Monte Carlo Expectation Maximization (MCEM) algorithm.

6. A variety of basic functions 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.

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, MacOS, or Windows.

The following are required to run NIMBLE.

1. R, of course.
2. The igraph, coda and R6 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, but see Section 4.5.3 for how to use a faster version of the BLAS.

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.[1]

4.2.1 MacOS

On MacOS, 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

[1] 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 MacOS, the package arrives already built to your system.

Store.

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 the GNU Fortran compiler package following these instructions from CRAN.

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 https://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).

Important: You must set the path so that the installer will add the location of the C++ compiler and related tools to your system's PATH variable, ensuring that R can find them. For R version 4.0 or greater (`Rtools42` or `Rtools40`) be sure to follow the instructions in the section Putting Rtools on the PATH. For R version 3.6.3 or lesser (i.e., using `Rtools35.exe`) make sure to check the box labelled "Add rtools to system PATH" (page 5 of the installation pages) (it should be checked by default). 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.

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`.

4.4 Troubleshooting installation problems

We have tested the installation on the three commonly used platforms – MacOS, Linux, Windows[2]. We don't anticipate problems with installation, but we want to hear about any and help resolve them.

The following are some troubleshooting tips that have helped users in some situations.

For Windows:

- Be sure you have set `PATH` when installing Rtools (see instructions above in the Windows installation section). Alternatively, one can set the `PATH` manually using syntax similar to

[2] We've tested NIMBLE on Windows 7, 8 and 10.

this (after changing `C:\\Rtools\\bin;C:\\Rtools\\mingw_64\\bin` to be appropriate for your system):

```
path <- Sys.getenv('PATH')
newPath <- paste("C:\\Rtools\\bin;C:\\Rtools\\mingw_64\\bin;",
             path, sep = "")
Sys.setenv(PATH = newPath)
```

- Be sure the `Rtools.exe` version matches the R version.
- Try re-installing `Rtools` followed by re-installing NIMBLE.
- If you're using `Rtools42` or `Rtools40`, make sure you are using a newer version of RStudio (at least 1.2.5042).
- If there are filesystem or permissions issues, it is possible to install NIMBLE in a local directory using the `lib` argument to `install.packages`.
- In the past we've heard reports from Windows users of problems when their filesystem involved a space in a directory name in the path to `RHOME`. We think this problem has been resolved.
- We've also heard reports from Windows users of problems when R is installed on a network drive. A work-around is to install locally in a directory on a drive physically on the machine.

For MacOS:

- Newly installed Xcode/command line tools may need to be started once manually to provide a one-time permission before they will work from NIMBLE.
- Upgrading your MacOS version may result in an error while installing or using NIMBLE. Please try running the following in the Terminal to reinstall Xcode/command line tools:

```
xcode-select --install
```

- If multiple C++ compilers are present on a system, be sure the `PATH` will find the right one.

All operating systems:

- If problems arise from generating and compiling C++ files from the default location in R's `tempdir()`, one can use the `dirName` argument to `compileNimble` to put such files elsewhere, such as in a local working directory.

If those suggestions don't help, please post about installation problems to the nimble-users Google group or email nimble.stats@gmail.com.

4.5 Customizing your installation

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

4.5.1 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, for example: `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`.

4.5.2 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 MacOS 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 static library (do this only on MacOS 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
```

4.5.3 BLAS and LAPACK

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.

4.5.4 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.

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, or JAGS, so we start by summarizing how NIMBLE is similar to and different from 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[1] node declarations.
2. Most univariate and multivariate distributions.
3. Link functions.
4. Most mathematical functions.
5. 'for' loops for iterative declarations.
6. Arrays of nodes up to 4 dimensions.
7. Truncation and censoring as in JAGS using the `T()` notation and `dinterval`.

5.1.2 NIMBLE's Extensions to BUGS and JAGS

NIMBLE extends the BUGS language in the following ways:

1. User-defined functions and distributions – written as nimbleFunctions – can be used in model code. See Chapter 12.

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

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 (but an expression can be defined in a separate deterministic expression and the resulting variable then used).
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.[2]
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.
11. One can use stochastic/dynamic indexes, i.e., indexes can be other nodes or functions of other nodes. For a given dimension of a node being indexed, if the index is not constant, it must be a scalar value. So expressions such as `mu[k[i], 3]` or `mu[k[i], 1:3]` or `mu[k[i], j[i]]` are allowed, but not `mu[k[i]:(k[i]+1)]`[3] Nested dynamic indexes such as `mu[k[j[i]]]` are also allowed.

5.1.3 Not-supported features of BUGS and JAGS

The following are not supported.

1. 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).
2. 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.
3. 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`).
4. Use of non-sequential indexes in the definition of a for loop. JAGS allows syntax such as `for(i in expr)` when `expr` evaluates to an integer vector. In NIMBLE one can use the work-around of defining a constant vector, say `k`, and using `k[i]` in the body of the for loop.
5. Use of non-sequential indexes to subset variables. JAGS allows syntax such as `sum(mu[expr])` when `expr` evaluates to an integer vector. In NIMBLE one can use the work-around of defining a nimbleFunction that takes `mu` and the index vector as arguments and does the calculation of interest.

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

[3] In some cases, one can write a nimbleFunction to achieve the desired result, such as replacing `sum(mu[start[i]:end[i]])` with a nimbleFunction that takes `mu`, `start[i]`, and `end[i]` as arguments and does the needed summation.

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.

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[4]. 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.

Here is the code to define and create a simple linear regression model with four observations.

```
library(nimble)
mc <- nimbleCode({
    intercept ~ dnorm(0, sd = 1000)
    slope ~ dnorm(0, sd = 1000)
    sigma ~ dunif(0, 100)
    for(i in 1:4) {
        predicted.y[i] <- intercept + slope * x[i]
        y[i] ~ dnorm(predicted.y[i], sd = sigma)
    }
})

model <- nimbleModel(mc, data = list(y = rnorm(4)))

library(igraph)

layout <- matrix(ncol = 2, byrow = TRUE,
   # These seem to be rescaled to fit in the plot area,
   # so I'll just use 0-100 as the scale
              data = c(33, 100,
                       66, 100,
                       50, 0, # first three are parameters
                       15, 50, 35, 50, 55, 50, 75, 50, # x's
                       20, 75, 40, 75, 60, 75, 80, 75, # predicted.y's
                       25, 25, 45, 25, 65, 25, 85, 25) # y's
              )

sizes <- c(45, 30, 30,
           rep(20, 4),
           rep(50, 4),
```

[4] Technically, a *directed acyclic graph*

```
           rep(20, 4))

edge.color <- "black"
    # c(
    # rep("green", 8),
    # rep("red", 4),
    # rep("blue", 4),
    # rep("purple", 4))
stoch.color <- "deepskyblue2"
det.color <- "orchid3"
rhs.color <- "gray73"
fill.color <- c(
    rep(stoch.color, 3),
    rep(rhs.color, 4),
    rep(det.color, 4),
    rep(stoch.color, 4)
)


plot(model$graph, vertex.shape = "crectangle",
     vertex.size = sizes,
     vertex.size2 = 20,
     layout = layout,
     vertex.label.cex = 3.0,
     vertex.color = fill.color,
     edge.width = 3,
     asp = 0.5,
     edge.color = edge.color)
```

The graph representing the model is shown in Figure 5.1. 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:

```
{
    intercept ~ dnorm(0, sd = 1000)
    slope ~ dnorm(0, sd = 1000)
    sigma ~ dunif(0, 100)
    for(i in 1:4) {
        predicted.y[i] <- intercept + slope * x[i]
        y[i] ~ dnorm(predicted.y[i], sd = sigma)
    }
}
```

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]`

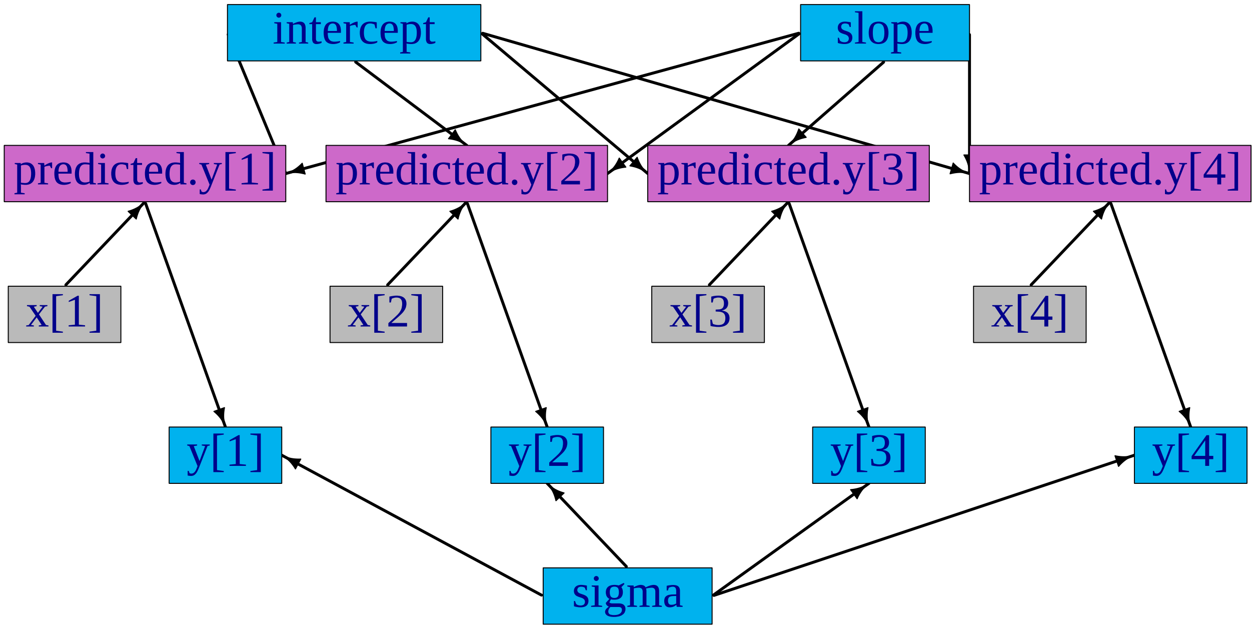


Figure 5.1: Graph of a linear regression model

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 5.1, 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:

```
{
    intercept ~ dnorm(0, sd = 1000)
    slope ~ dnorm(0, sd = 1000)
    sigma ~ dunif(0, 100)
    for(i in 1:4) {
        y[i] ~ dnorm(intercept + slope * x[i], sd = sigma)
    }
}
```

In this case, the `predicted.y[i]` nodes in Figure 5.1 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) {
        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 can be provided via `constants` (in which case it can never be changed) or via `data` or `inits`, as discussed 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. Variables that define for-loop indices (`N` and `M`) must also 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 12.
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 12.

5.2.2.1 More about indexing

Examples of allowed indexing include:

- `x[i]` # a single index
- `x[i:j]` # a range of indices
- `x[i:j,k:l]` # 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
- `x[k[i]+1]` # a dynamic (and computed) index
- `x[k[j[i]]]` # nested dynamic indexes
- `x[k[i], 1:3]` # nested indexing of rows or columns

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[5]. Section 6.1.1.5 discusses how to provide NIMBLE with dimensions of `x` when needed.

One cannot provide a vector of indices that are not constant. For example, `x[start[i]:end[i]]` is allowed only if `start` and `end` are provided as constants. Also, one cannot provide a vector, or expression evaluating to a vector (apart from use of `:`) as an index. For example, `x[inds]` is not allowed. Often one can write a nimbleFunction to achieve the desired result, such as defining a nimbleFunction that takes `x`, `start[i]`, and `end[i]` or takes `x` and `inds` as arguments and does the subsetting (possibly in combination with some other calculation). Note that `x[L:U]` and `x[inds]` are allowed in nimbleFunction code.

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 1×1 matrix (use `inprod` to get it as a scalar, which it 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 1×3 matrix. More about indexing and dimensions is discussed in Section 11.3.2.7.

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])
}
```

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,

[5] In `nimbleFunctions`, as explained in later chapters, square brackets with blank indices are not necessary for multivariate objects.

`logY[1:10]`. If each `logY[i]` is used separately by an algorithm, it may be more efficient computationally if they are declared as scalars. If they are all used together, it will often make sense to declare them as a vector.

5.2.4 Available distributions

5.2.4.1 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[6]. NIMBLE also allows one to use alternative parameterizations for a variety of distributions as described next. See Section 12.2 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	`dbern(prob = p)` $0 < p < 1$	$p^x(1-p)^{1-x}$	0	1
Beta	`dbeta(shape1 = a, shape2 = b)`, $a > 0$, $b > 0$	$\frac{x^{a-1}(1-x)^{b-1}}{\beta(a,b)}$	0	1
Binomial	`dbin(prob = p, size = n)` $0 < p < 1$, $n \in \mathbb{N}^*$	$\binom{n}{x} p^x (1-p)^{n-x}$	0	n
CAR (intrinsic)	`dcar_normal(adj, weights, num, tau, c, zero_mean)`	see chapter 9 for details		
CAR (proper)	`dcar_proper(mu, C, adj, num, M, tau, gamma)`	see chapter 9 for details		
Categorical	`dcat(prob = p)` $p \in (\mathbb{R}^+)^N$	$\frac{p_x}{\sum_i p_i}$	1	N
Chi-square	`dchisq(df = k)`, $k > 0$	$\frac{x^{\frac{k}{2}-1}\exp(-x/2)}{2^{\frac{k}{2}}\Gamma(\frac{k}{2})}$	0	
Double exponential	`ddexp(location = `μ`, rate = `τ`)`,$\tau > 0$	$\frac{\tau}{2}\exp(-\tau\|x-\mu\|)$		
Dirichlet	`ddirch(alpha = `α`)` $\alpha_j \geq 0$	$\Gamma(\sum_j \alpha_j)\prod_j \frac{x_j^{\alpha_j - 1}}{\Gamma(\alpha_j)}$	0	
Exponential	`dexp(rate = `λ`)`, $\lambda > 0$	$\lambda \exp(-\lambda x)$	0	
Flat	`dflat()`	$\propto 1$ (improper)		
Gamma	`dgamma(shape = r, rate = `λ`)` $\lambda > 0$, $r > 0$	$\frac{\lambda^r x^{r-1}\exp(-\lambda x)}{\Gamma(r)}$	0	
Half flat	`dhalfflat()`	$\propto 1$ (improper)	0	
Inverse gamma	`dinvgamma(shape = r, scale = `λ`)` $\lambda > 0$, $r > 0$	$\frac{\lambda^r x^{-(r+1)}\exp(-\lambda/x)}{\Gamma(r)}$	0	
LKJ Correl'n	`dlkj_corr_cholesky(eta = `η`,`	$\prod_{i=2}^p x_{ii}^{p-i+2\eta-2}$		

[6] 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 11.2.4 for how to use distributions in `nimbleFunctions`.

Name	Usage	Density	Lower	Upper
Cholesky	size = p), $\eta > 0$			
Logistic	dlogis(location = μ, rate = τ),$\tau > 0$	$\frac{\tau \exp\{(x-\mu)\tau\}}{\left[1+\exp\{(x-\mu)\tau\}\right]^2}$		
Log-normal	dlnorm(meanlog = μ, taulog = τ), $\tau > 0$	$\left(\frac{\tau}{2\pi}\right)^{\frac{1}{2}} x^{-1} \exp\left\{-\tau(\log(x)-\mu)^2/2\right\}$	0	
Multinomial	dmulti(prob = p, size = n) $\sum_j x_j = n$	$n! \prod_j \frac{p_j^{x_j}}{x_j!}$		
Multivariate normal	dmnorm(mean = μ, prec = Λ) Λ positive definite	$(2\pi)^{-\frac{d}{2}} \lvert\Lambda\rvert^{\frac{1}{2}} \exp\{-\frac{(x-\mu)^T\Lambda(x-\mu)}{2}\}$		
Multivariate Student t	dmvt(mu = μ, prec = Λ“ df = ν), Λ positive def.	$\frac{\Gamma(\frac{\nu+d}{2})}{\Gamma(\frac{\nu}{2})(\nu\pi)^{d/2}} \lvert\Lambda\rvert^{1/2}(1+\frac{(x-\mu)^T\Lambda(x-\mu)}{\nu})^{-\frac{\nu+d}{2}}$		
Negative binomial	dnegbin(prob = p, size = r) $0 < p \leq 1$, $r \geq 0$	$\binom{x+r-1}{x} p^r (1-p)^x$	0	
Normal	dnorm(mean = μ, tau = τ) $\tau > 0$	$\left(\frac{\tau}{2\pi}\right)^{\frac{1}{2}} \exp\{-\tau(x-\mu)^2/2\}$		
Poisson	dpois(lambda = λ) $\lambda > 0$	$\frac{\exp(-\lambda)\lambda^x}{x!}$	0	
Student t	dt(mu = μ, tau = τ, df = k) $\tau > 0$, $k > 0$	$\frac{\Gamma(\frac{k+1}{2})}{\Gamma(\frac{k}{2})} \left(\frac{\tau}{k\pi}\right)^{\frac{1}{2}} \left\{1+\frac{\tau(x-\mu)^2}{k}\right\}^{-\frac{(k+1)}{2}}$		
Uniform	dunif(min = a, max = b) $a < b$	$\frac{1}{b-a}$	a	b
Weibull	dweib(shape = v, lambda = λ) $v > 0$, $\lambda > 0$	$v\lambda x^{v-1} \exp(-\lambda x^v)$	0	
Wishart	dwish(R = R, df = k) R $p \times p$ pos. def., $k \geq p$	$\frac{\lvert x\rvert^{(k-p-1)/2} \lvert R\rvert^{k/2} \exp\{-\mathrm{tr}(Rx)/2\}}{2^{pk/2}\pi^{p(p-1)/4} \prod_{i=1}^{p} \Gamma((k+1-i)/2)}$		
Inverse Wishart	dinvwish(S = S, df = k) S $p \times p$ pos. def., $k \geq p$	$\frac{\lvert x\rvert^{-(k+p+1)/2} \lvert S\rvert^{k/2} \exp\{-\mathrm{tr}(Sx^{-1})/2\}}{2^{pk/2}\pi^{p(p-1)/4} \prod_{i=1}^{p} \Gamma((k+1-i)/2)}$		

5.2.4.1.1 Improper distributions Note that dcar_normal, dflat and dhalfflat specify improper prior distributions and should only be used when the posterior distribution of the model is known to be proper. Also for these distributions, the density function returns the unnormalized density and the simulation function returns NaN so these distributions are not appropriate for algorithms that need to simulate from the prior or require proper (normalized) densities.

5.2.4.1.2 LKJ distribution for correlation matrices NIMBLE provides the LKJ distribution via dlkj_corr_cholesky for correlation matrices, discussed in Section 24 of Stan Development Team (2021a) and based on Lewandowski et al. (2009). This distribution has various advantages (both from a modeling perspective and an MCMC fitting perspective) over other prior distributions for correlation or covariance matrices such as the inverse Wishart distribution.

For computational efficiency, the distribution is on the Cholesky factor of the correlation matrix rather than the correlation matrix itself. As a result using this distribution in a model involves a bit of care, including efficiently creating a covariance matrix from the correlation matrix and a set of standard deviations.

Here is some example code illustrating how to use the distribution in a model.

```
code <- nimbleCode({
    Ustar[1:p,1:p] ~ dlkj_corr_cholesky(1.3, p)
    U[1:p,1:p] <- uppertri_mult_diag(Ustar[1:p, 1:p], sds[1:p])
    for(i in 1:n)
        y[i, 1:p] ~ dmnorm(mu[1:p], cholesky = U[1:p, 1:p], prec_param = 0)
})
```

Note that we need to take the upper triangular Cholesky factor (`Ustar`) of the correlation matrix and convert it to the upper triangular Cholesky factor of a covariance matrix (`U`), based on a vector of standard deviations (`sds`), and use that in the parameterization of `dmnorm` that directly (and therefore efficiently) uses the Cholesky of the covariance.

Other approaches are possible, but unless care is taken they are likely to be less computationally efficient than the template above.

The template uses a user-defined nimbleFunction to efficiently combine the Cholesky of the correlation matrix with a vector of standard deviations to produce the Cholesky of the covariance matrix. That function is given here:

```
uppertri_mult_diag <- nimbleFunction(
    run = function(mat = double(2), vec = double(1)) {
        returnType(double(2))
        p <- length(vec)
        out <- matrix(nrow = p, ncol = p, init = FALSE)
        for(i in 1:p)
            out[ , i] <- mat[ , i] * vec[i]
        return(out)
    })
```

5.2.4.2 Alternative parameterizations for distributions

NIMBLE allows one to specify distributions in model code using a variety of parameterizations, including the BUGS parameterizations. Available parameterizations are 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[7]. For `gamma`, this is (shape, scale).

2. The BUGS parameterization is the one defined in the original BUGS language. In general this is the parameterization for which conjugate MCMC samplers can be executed most efficiently. For `dgamma`, this is (shape, rate).
3. An *alternative* parameterization is one that must be converted into the *canonical* parameterization. For `dgamma`, NIMBLE provides both (shape, rate) and (mean, sd) parameterization

[7] Usually this is the parameterization in the `Rmath` header of R's C implementation of distributions.

and creates nodes to calculate (shape, scale) from either (shape, rate) or (mean, sd). In the case of `dgamma`, 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).

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^2 * (1-mean) / sd^2 - mean, shape2 = mean * (1 - mean)^2 / sd^2 + mean - 1)`
`dbin(prob, size)`	canonical
`dcat(prob)`	canonical
`dchisq(df)`	canonical
`ddexp(location, scale)`	canonical
`ddexp(location, rate)`	`ddexp(location, scale = 1 / rate)`
`ddexp(location, var)`	`ddexp(location, scale = sqrt(var / 2))`
`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^2/sd^2, scale = sd^2/mean)`
`dinvgamma(shape, rate)`	canonical
`dinvgamma(shape, scale)`	`dgamma(shape, rate = 1 / scale)`
`dlkj_corr_cholesky(eta)`	canonical
`dlogis(location, scale)`	canonical
`dlogis(location, rate)`	`dlogis(location, scale = 1 / rate)`
`dlnorm(meanlog, sdlog)`	canonical
`dlnorm(meanlog, taulog)`	`dlnorm(meanlog, sdlog = 1 / sqrt(taulog))`
`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)`

Parameterization	NIMBLE re-parameterization
`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)`
`dlkj_corr_cholesky( ...shape, size)`	canonical
`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^(- 1 / shape))`
`dwish(cholesky, df, ...scale_param=1)`	canonical (scale)
`dwish(cholesky, df, ...scale_param=0)`	canonical (inverse scale)
`dwish(R, df)`	`dwish(cholesky = chol(R), df, scale_param = 0)`
`dwish(S, df)`	`dwish(cholesky = chol(S), df, scale_param = 1)`
`dinvwish(cholesky, df, ...scale_param=1)`	canonical (scale)
`dinvwish(cholesky, df, ...scale_param=0)`	canonical (inverse scale)
`dinvwish(R, df)`	`dinvwish(cholesky = chol(R), df, scale_param = 0)`
`dinvwish(S, df)`	`dinvwish(cholesky = chol(S), df, scale_param = 1)`

Note that for multivariate normal, multivariate t, Wishart, and Inverse 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.

Table 5.3: Distributions with alternative names (aliases)

Distribution	Canonical name	Alias
Binomial	dbin	dbinom
Chi-square	dchisq	dchisqr
Double exponential	ddexp	dlaplace
Dirichlet	ddirch	ddirich
Multinomial	dmulti	dmultinom
Negative binomial	dnegbin	dnbinom
Weibull	dweib	dweibull
Wishart	dwish	dwishart

We plan to, but do not currently, include the following distributions as part of core NIMBLE: beta-binomial, Dirichlet-multinomial, F, Pareto, 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. Support for more general R expressions is covered in Chapter 11 about programming with nimbleFunctions.

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 11.2.4 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. Vector input column indicates if the function can take a vector as an argument (i.e., if the function is vectorized).

Usage	Description	Comments	Status	Vector input
`x | y`, `x & y`	logical OR (|) and AND(&)		yes	yes
`!x`	logical not		yes	yes
`x > y, x >= y`	greater than (and or equal to)		yes	yes
`x < y, x <= y`	less than (and or equal to)		yes	yes
`x != y, x == y`	(not) equals		yes	yes
`x + y, x - y, x * y`	component-wise operators	mix of scalar and vector	yes	yes
`x / y`	component-wise division	vector x and scalar y	yes	yes
`x^y, pow(x, y)`	power	x^y; vector x,scalar y	yes	yes
`x %% y`	modulo (remainder)		yes	no

`min(x1, x2),`
`max(x1, x2)`
	min. (max.) of two scalars		yes	See `pmin`,
				`pmax`

Usage	Description	Comments	Status	Vector input
`exp(x)`	exponential		yes	yes
`log(x)`	natural logarithm		yes	yes
`sqrt(x)`	square root		yes	yes
`abs(x)`	absolute value		yes	yes
`step(x)`	step function at 0	0 if $x < 0$, 1 if $x >= 0$	yes	yes
`equals(x, y)`	equality of two scalars	1 if $x == y$, 0 if $x! = y$	yes	yes
`cube(x)`	third power	x^3	yes	yes
`sin(x), cos(x), tan(x)`	trigonometric functions		yes	yes
`asin(x), acos(x), atan(x)`	inverse trigonometric functions		yes	yes
`asinh(x), acosh(x), atanh(x)`	inv. hyperbolic trig. functions		yes	yes
`logit(x)`	logit	$\log(x/(1-x))$	yes	yes
`ilogit(x), expit(x)`	inverse logit	$\exp(x)/(1+\exp(x))$	yes	yes
`probit(x)`	probit (Gaussian quantile)	$\Phi^{-1}(x)$	yes	yes
`iprobit(x), phi(x)`	inverse probit (Gaussian CDF)	$\Phi(x)$	yes	yes
`cloglog(x)`	complementary log log	$\log(-\log(1-x))$	yes	yes
`icloglog(x)`	inverse complementary log log	$1-\exp(-\exp(x))$	yes	yes
`ceiling(x)`	ceiling function	$\lceil(x)\rceil$	yes	yes
`floor(x)`	floor function	$\lfloor(x)\rfloor$	yes	yes
`round(x)`	round to integer		yes	yes
`trunc(x)`	truncation to integer		yes	yes
`lgamma(x), loggam(x)`	log gamma function	$\log\Gamma(x)$	yes	yes
`besselK(k, nu, ...expon.scaled)`	modified bessel function of the second kind	returns vector even if k a matrix/array	yes	yes
`log1p(x)`	log of 1 + x	$\log(1+x)$	yes	yes
`lfactorial(x), logfact(x)`	log factorial	$\log x!$	yes	yes
`qDIST(x, PARAMS)`	"q" distribution functions	canonical parameterization	yes	yes
`pDIST(x, PARAMS)`	"p" distribution functions	canonical parameterization	yes	yes
`rDIST(x, PARAMS)`	"r" distribution functions	canonical parameterization	yes	yes
`dDIST(x, PARAMS)`	"d" distribution functions	canonical parameterization	yes	yes
`sort(x)`			no	
`rank(x, s)`			no	
`ranked(x, s)`			no	
`order(x)`			no	

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

Usage	Description	Comments	Status
`inverse(x)`	matrix inverse	x symmetric, positive def.	yes
`chol(x)`	matrix Cholesky factorization	x symmetric, positive def.	yes

Usage	Description	Comments	Status
		returns upper triang. matrix	
`t(x)`	matrix transpose	$x^\top$	yes
`x%*%y`	matrix multiply	xy; x, y conformant	yes
`inprod(x, y)`	dot product	$x^\top y$; x and y vectors	yes
`solve(x,y)`	solve system of equations	$x^{-1}y$; y matrix or vector	yes
`forwardsolve(x, y)`	solve lower-triangular system of equations	$x^{-1}y$; x lower-triangular	yes
`backsolve(x, y)`	solve upper-triangular system of equations	$x^{-1}y$; x upper-triangular	yes
`logdet(x)`	log matrix determinant	$\log \lvert x \rvert$	yes
`asRow(x)`	convert vector to 1-row matrix	sometimes automatic	yes
`asCol(x)`	convert vector to 1-column matrix	sometimes automatic	yes
`sum(x)`	sum of elements of `x`		yes
`mean(x)`	mean of elements of `x`		yes
`sd(x)`	standard deviation of elements of `x`		yes
`prod(x)`	product of elements of `x`		yes
`min(x), max(x)`	min. (max.) of elements of `x`		yes
`pmin(x, y), pmax(x,y)`	vector of mins (maxs) of elements of `x` and `y`		yes
`interp.lin(x, v1, v2)`	linear interpolation		no
`eigen(x)$values`	matrix eigenvalues	x symmetric	yes
`eigen(x)$vectors`	matrix eigenvectors	x symmetric	yes
`svd(x)$d`	matrix singular values		yes
`svd(x)$u`	matrix left singular vectors		yes
`svd(x)$v`	matrix right singular vectors		yes

See Section 12.1 to learn how to use nimbleFunctions to write new functions for use in BUGS code.

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

Link functions are specified as functions applied to a 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 following 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[i]` 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:

5.2.7.1 Truncation

Either of the following forms,

```
x ~ dnorm(0, sd = 10) T(0, a)
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 `x` are limited a priori, hence the probability density of `x` must be normalized[8]. In this example it would be the normal probability density divided by its integral from 0 to `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.

5.2.7.2 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 an individual still surviving at the end of a study, the age of death is not known and hence is 'censored' (right-censoring). NIMBLE adopts JAGS syntax for censoring, as follows:

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

In the case of right-censoring, `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 vector for `c` should give the censoring times corresponding to censored entries and a value above the observed times for uncensored entries (e.g., `Inf`). The values for `c` can be

[8] NIMBLE uses the CDF and inverse CDF (quantile) functions of a distribution to do this; in some cases if one uses truncation to include only the extreme tail of a distribution, numerical difficulties can arise.

provided via `constants`, `inits`, or `data`. We recommend providing via `constants` for values that are fixed and via `inits` for values that are to be estimated or might be changed for some reason.

Left-censored observations would be specified by setting `censored[i]` to 0 and `t[i]` to `NA`.

Important: The value of `t[i]` corresponding to each censored data point should be given a valid initial value (via `inits`) that is consistent with `censored[i]`. Otherwise, NIMBLE will initialize from the prior, which may not produce a valid initial value.

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:

`censored[i] = 0` if `t[i]` $<=$ `c[i, 1]`

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

`censored[i] = M` if `c[i, M] < t[i]`

(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.

5.2.7.3 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`.

Important: the model should be initialized so that the constraints are satisfied. Otherwise, NIMBLE will initialize from the prior, which may not produce a valid initial value and may cause algorithms (particularly) MCMC to fail.

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.

5.2.7.3.1 Ordering To specify an ordering of parameters, such as $\alpha_1 <= \alpha_2 <= \alpha_3$ 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 `alpha1` $\leq$ `alpha2` $\leq$ `alpha3`, but it does not treat the three parameters symmetrically. Instead it puts a marginal prior on `alpha1` that is standard normal and then constrains `alpha2` and `alpha3` to follow truncated normal distributions. This is not equivalent to a symmetric prior on the three `alphas` that assigns zero probability density when values are not in order.

NIMBLE does not support the JAGS `sort` syntax.

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:

```
pumpCode <- nimbleCode({
  for (i in 1:N){
      theta[i] ~ dgamma(alpha,beta);
      lambda[i] <- theta[i]*t[i];
      x[i] ~ dpois(lambda[i])
  }
  alpha ~ dexp(1.0);
  beta ~ dgamma(0.1,1.0);
})

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

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

```
pumpInits <- list(alpha = 1, beta = 1,
              theta = rep(0.1, pumpConsts$N))

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

6.1.1.1 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 constants or as data or initial values. There is no real difference between providing as data or initial values and the values can be added after building a model via `setInits` or `setData`.

6.1.1.2 Providing (or changing) data and initial values for an existing model

Whereas constants must be provided during the call to `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 (stochastic) 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()`.

Data nodes cannot be deterministic, and using `setData` on deterministic nodes (or passing values for deterministic nodes via the `data` argument to `nimbleModel`) will produce an error.

To change data values without any modification of which nodes are flagged as containing data, simply use R's usual assignment syntax to assign values in a compiled (or, more rarely, an uncompiled) model, e.g.,

```
cModel$y <- c(1.5, 2.5, 1.7)
```

This can be useful for running an MCMC with a different dataset of the same size (and of course the same pattern of missingness, if any) without having to rebuild and recompile the MCMC, such as in a simulation study. It is possible to change the data values in a compiled model using `setData`, but we don't recommend doing this because `setData` won't modify which nodes are flagged as containing data in the already-constructed MCMC, thereby potentially introducing confusion.

6.1.1.3 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)
```

```
## ===== Monitors =====
## thin = 1: alpha, beta
## ===== Samplers =====
## RW sampler (1)
##   - alpha
## posterior_predictive sampler (2)
##   - theta[]  (2 elements)
## conjugate sampler (9)
##   - beta
##   - theta[]  (8 elements)
## ===== Comments =====
```

```
pumpMissConf$addMonitors('x', 'alpha', 'beta', 'theta')
```

```
## thin = 1: alpha, beta, theta, x
```

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

```
niter <- 10
set.seed(0)
Cobj$pumpMissMCMC$run(niter)
samples <- as.matrix(Cobj$pumpMissMCMC$mvSamples)

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

```
##      x[1] x[2] x[3] x[4] x[5]
## [1,]   12    1    0   14    3
## [2,]    7    1   35   14    3
## [3,]   19    1    4   14    3
## [4,]   61    1   36   14    3
## [5,]   70    1   61   14    3
```

Missing values may also occur in explanatory/predictor 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, for example so that an MCMC would impute the missing values.

6.1.1.4 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()
```

```
## [1] "intercept"                        "slope1"
## [3] "predictedY"                       "sigmaY"
## [5] "lifted_d1_over_sqrt_oPsigmaY_cP" "Y"
## [7] "x1"
```

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

```
## [1] "intercept"                              "slope1"
## [3] "slope2"                                 "predictedY"
## [5] "sigmaY"                                 "lifted_d1_over_sqrt_oPsigmaY_cP"
## [7] "Y"                                      "x1"
## [9] "x2"
```

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.

6.1.1.5 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)
```

In that example, since `alpha0` and `R` are provided as constants, we don't need to specify their dimensions.

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
simpleModelCopy3 <- simpleModelCopy2$newModel(check = FALSE)
```

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 13 covers more topics for writing algorithms that use models.

6.2.1 What are variables and nodes?

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

Suppose we have created a model from the following BUGS code.

```
mc <- nimbleCode({
    a ~ dnorm(0, 0.001)
    for(i in 1:5) {
        y[i] ~ dnorm(a, sd = 0.1)
        for(j in 1:3)
            z[i,j] ~ dnorm(y[i], tau)
    }
    tau ~ dunif(0, 20)
    y.squared[1:5] <- y[1:5]^2
})

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

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]`.

6.2.2 Determining the nodes and variables in a model

One can determine the variables in a model using `getVarNames` and the nodes in a model using `getNodeNames`. Optional arguments to `getNodeNames` allow you to select only certain types of nodes, as discussed in Section 13.1.1 and in the R help for `getNodeNames`.

```
model$getVarNames()
```

```
## [1] "a"                              "y"
## [3] "lifted_d1_over_sqrt_oPtau_cP" "z"
## [5] "tau"                            "y.squared"
```

```
model$getNodeNames()
```

```
##  [1] "a"                              "tau"
##  [3] "y[1]"                           "y[2]"
##  [5] "y[3]"                           "y[4]"
##  [7] "y[5]"                           "lifted_d1_over_sqrt_oPtau_cP"
##  [9] "y.squared[1:5]"                 "z[1, 1]"
## [11] "z[1, 2]"                        "z[1, 3]"
## [13] "z[2, 1]"                        "z[2, 2]"
## [15] "z[2, 3]"                        "z[3, 1]"
## [17] "z[3, 2]"                        "z[3, 3]"
## [19] "z[4, 1]"                        "z[4, 2]"
## [21] "z[4, 3]"                        "z[5, 1]"
## [23] "z[5, 2]"                        "z[5, 3]"
```

Note that some of the nodes may be 'lifted' nodes introduced by NIMBLE (Section 13.1.2). In this case `lifted_d1_over_sqrt_oPtau_cP` (this is a node for the standard deviation of the `z` nodes using NIMBLE's canonical parameterization of the normal distribution) is the only lifted node in the model.

To determine the dependencies of one or more nodes in the model, you can use `getDependencies` as discussed in Section 13.1.3.

6.2.3 Accessing nodes

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

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

```
## [1] 5
```

```
model[["a"]]
```

```
## [1] 5
```

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

```
## [1]         NA -0.7317482  0.8303732 -1.2080828         NA
```

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

```
## [1] -1.0479844 -0.7317482  0.8303732 -1.2080828  1.4411577
```

```
model$z[1,]
```

```
## [1]  2.0752450  0.2199248 -0.7660820
```

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:

```
model[["y[2]"]]
```

```
## [1] -0.7317482
```

```
model[["y[2]"]] <- -5
model$y
```

```
## [1] -1.0479844 -5.0000000  0.8303732 -1.2080828  1.4411577
```

```
model[["z[2, 3]"]]
```

```
## [1] -0.4302118
```

```
model[["z[2:4, 1:2]"]][1, 2]
```

```
## [1] -1.46725
```

```
model$z[2, 2]
```

```
## [1] -1.46725
```

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]`.

6.2.4 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 previously.

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]':

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

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

multiVarModel$expandNodeNames("X[1,1:5]")
```

```
## [1] "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]])
```

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

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

6.2.5 Why use node names?

Syntax like `model[["z[2, 3]"]]` may seem strange at first, because the natural habit of an R user would be `model[["z"]][2,3]`. To see its utility, consider the example of writing the nimbleFunction given in Section 2.8. 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.

6.2.6 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:

```
model$isData('z[1]')
```

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

```
model$isData(c('z[1]', 'z[2]', 'a'))
```

```
## [1]  TRUE  TRUE FALSE
```

```
model$isData('z')
```

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

```
model$isData('z[1:3, 1]')
```

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

6.3 Using models in parallel

NIMBLE uses Reference Classes and R6 classes for models and algorithms. Objects of these classes are passed by reference and copies of such objects are simply new variable names that reference the same underlying object.

Thus to run an algorithm in parallel on a given model, one must create multiple copies of the model and algorithm, and compiled versions of these, by calling `nimbleModel`, `buildMCMC`, `compileNimble`, etc. once for each copy. In other words all such calls should be within the parallelized block of code.

For a worked example in the context of MCMC, please see the parallelization example on our webpage.

Part III

Algorithms in NIMBLE

Chapter 7

MCMC

NIMBLE provides a variety of paths to creating and executing an MCMC algorithm, which differ greatly in their simplicity of use, and also in the options available and customizability.

The most direct approach to invoking the MCMC engine is using the `nimbleMCMC` function (Section 7.1). This one-line call creates and executes an MCMC, and provides a wide range of options for controlling the MCMC: specifying monitors, burn-in, and thinning, running multiple MCMC chains with different initial values, and returning posterior samples, summary statistics, and/or a WAIC value. However, this approach is restricted to using NIMBLE's default MCMC algorithm; further customization of, for example, the specific samplers employed, is not possible.

The lengthier and more customizable approach to invoking the MCMC engine on a particular NIMBLE model object involves the following steps:

1. (Optional) Create and customize an MCMC configuration for a particular model:

    a. Use `configureMCMC` to create an MCMC configuration (see Section 7.2). The configuration contains a list of samplers with the node(s) they will sample.

    b. (Optional) Customize the MCMC configuration:

        i. Add, remove, or re-order the list of samplers (Section 7.11 and `help(samplers)` in R for details), including adding your own samplers (Section 15.5);
        ii. Change the tuning parameters or adaptive properties of individual samplers;
        iii. Change the variables to monitor (record for output) and thinning intervals for MCMC samples.

2. Use `buildMCMC` to build the MCMC object and its samplers either from the model (using default MCMC configuration) or from a customized MCMC configuration (Section 7.3).

3. Compile the MCMC object (and the model), unless one is debugging and wishes to run the uncompiled MCMC.

4. Run the MCMC and extract the samples (Sections 7.5, 7.6 and 7.7).

5. Optionally, calculate the WAIC (Section 7.8).

Prior to version 0.8.0, NIMBLE provided two additional functions, `MCMCsuite` and `compareMCMCs`, to facilitate comparison of multiple MCMC algorithms, either internal or external to NIMBLE.

Those capabilities have been redesigned and moved into a separate package called `compareMCMCs`.

End-to-end examples of MCMC in NIMBLE can be found in Sections 2.5-2.6 and Section 7.12.

7.1 One-line invocation of MCMC: *nimbleMCMC*

The most direct approach to executing an MCMC algorithm in NIMBLE is using `nimbleMCMC`. This single function can be used to create an underlying model and associated MCMC algorithm, compile both of these, execute the MCMC, and return samples, summary statistics, and a WAIC value. This approach circumvents the longer (and more flexible) approach using `nimbleModel`, `configureMCMC`, `buildMCMC`, `compileNimble`, and `runMCMC`, which is described subsequently.

The `nimbleMCMC` function provides control over the:

- number of MCMC iterations in each chain;
- number of MCMC chains to execute;
- number of burn-in samples to discard from each chain;
- thinning interval on which samples should be recorded;
- model variables to monitor and return posterior samples;
- initial values, or a function for generating initial values for each chain;
- setting the random number seed;
- returning posterior samples as a matrix or a `coda` `mcmc` object;
- returning posterior summary statistics; and
- returning a WAIC value calculated using post-burn-in samples from all chains.

This entry point for using `nimbleMCMC` is the `code`, `constants`, `data`, and `inits` arguments that are used for building a NIMBLE model (see Chapters 5 and 6). However, when using `nimbleMCMC`, the `inits` argument can also specify a list of lists of initial values that will be used for each MCMC chain, or a function that generates a list of initial values, which will be generated at the onset of each chain. As an alternative entry point, a NIMBLE `model` object can also be supplied to `nimbleMCMC`, in which case this model will be used to build the MCMC algorithm.

Based on its arguments, `nimbleMCMC` optionally returns any combination of

- Posterior samples,
- Posterior summary statistics, and
- WAIC value.

The above are calculated and returned for each MCMC chain, using the post-burn-in and thinned samples. Additionally, posterior summary statistics are calculated for all chains combined when multiple chains are run.

Several example usages of `nimbleMCMC` are shown below:

```
code <- nimbleCode({
    mu ~ dnorm(0, sd = 1000)
    sigma ~ dunif(0, 1000)
    for(i in 1:10)
        x[i] ~ dnorm(mu, sd = sigma)
})
data <- list(x = c(2, 5, 3, 4, 1, 0, 1, 3, 5, 3))
initsFunction <- function() list(mu = rnorm(1,0,1), sigma = runif(1,0,10))
```

```
# execute one MCMC chain, monitoring the "mu" and "sigma" variables,
# with thinning interval 10.  fix the random number seed for reproducible
# results.  by default, only returns posterior samples.
mcmc.out <- nimbleMCMC(code = code, data = data, inits = initsFunction,
                       monitors = c("mu", "sigma"), thin = 10,
                       niter = 20000, nchains = 1, setSeed = TRUE)

# note that the inits argument to nimbleModel must be a list of
# initial values, whereas nimbleMCMC can accept inits as a function
# for generating new initial values for each chain.
initsList <- initsFunction()
Rmodel <- nimbleModel(code, data = data, inits = initsList)

# using the existing Rmodel object, execute three MCMC chains with
# specified burn-in.  return samples, summary statistics, and WAIC.
mcmc.out <- nimbleMCMC(model = Rmodel,
                       niter = 20000, nchains = 3, nburnin = 2000,
                       summary = TRUE, WAIC = TRUE)

# run ten chains, generating random initial values for each
# chain using the inits function specified above.
# only return summary statistics from each chain; not all the samples.
mcmc.out <- nimbleMCMC(model = Rmodel, nchains = 10, inits = initsFunction,
                       samples = FALSE, summary = TRUE)
```

See `help(nimbleMCMC)` for further details.

7.2 The MCMC configuration

The MCMC configuration contains information needed for building an MCMC. When no customization is needed, one can jump directly to the `buildMCMC` step below. An MCMC configuration is an object of class `MCMCconf`, which includes:

- The model on which the MCMC will operate
- The model nodes which will be sampled (updated) by the MCMC
- The samplers and their internal configurations, called control parameters
- 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

7.2.1 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.

7.2.1.1 Default assignment of sampler algorithms

The default sampler assigned to a stochastic node is determined by the following, in order of precedence:

1. If the node has no data nodes in its entire downstream dependency network, a `posterior_predictive` sampler is assigned. This sampler updates the node in question and all downstream stochastic nodes by simulating new values from each node's conditional distribution. As of version 0.13.0, the operation of the `posterior_predictive` sampler has changed in order to improve MCMC mixing. See Section 7.2.1.2.
2. If the node has a conjugate relationship between its prior distribution and the distributions of its stochastic dependents, a `conjugate` ('Gibbs') sampler is assigned.
3. If the node follows a multinomial distribution, then a `RW_multinomial` sampler is assigned. This is a discrete random-walk sampler in the space of multinomial outcomes.
4. If a node follows a Dirichlet distribution, then a `RW_dirichlet` sampler is assigned. This is a random walk sampler in the space of the simplex defined by the Dirichlet.
5. If a node follows an LKJ correlation distribution, then a `RW_block_lkj_corr_cholesky` sampler is assigned. This is a block random walk sampler in a transformed space where the transformation uses the signed stickbreaking approach described in Section 10.12 of Stan Development Team (2021b).
6. If the node follows any other 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 (Roberts and Sahu, 1997).
7. If the node is binary-valued (strictly taking values 0 or 1), then a `binary` sampler is assigned. This sampler calculates the conditional probability for both possible node values and draws the new node value from the conditional distribution, in effect making a Gibbs sampler.
8. If the node is otherwise discrete-valued, then a `slice` sampler is assigned (Neal, 2003).
9. 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.

Details of each sampler and its control parameters can be found by invoking `help(samplers)`.

7.2.1.2 Sampling posterior predictive nodes

A posterior predictive node is a node that is not itself data and has no data nodes in its entire downstream (descendant) dependency network. Such nodes play no role in inference for model parameters but have often been included in BUGS models to accomplish posterior predictive checks and similar calculations.

As of version 0.13.0, NIMBLE's handling of posterior predictive nodes in MCMC sampling has changed in order to improve MCMC mixing. Samplers for nodes that are not posterior predictive nodes no longer condition on the values of the posterior predictive nodes. This produces a valid MCMC over the posterior distribution marginalizing over the posterior predictive nodes. This MCMC will generally mix better than an MCMC that conditions on the values of posterior predictive nodes, by reducing the dimensionality of the parameter space and removing the dependence between the sampled nodes and the posterior predictive nodes. At the end of each MCMC iteration, the posterior predictive nodes are sampled by `posterior_predictive` sampler(s) based on their conditional distribution(s).

7.2.1.3 Options to control default sampler assignments

Very basic control of default sampler assignments is provided via two arguments to `configureMCMC`. The `useConjugacy` argument controls whether conjugate samplers are assigned when possible, and the `multivariateNodesAsScalars` argument controls whether scalar elements of multivariate nodes are sampled individually. See `help(configureMCMC)` for usage details.

7.2.1.4 Default monitors

The default MCMC configuration includes monitors on all top-level stochastic nodes of the model. Only variables that are monitored will have their samples saved for use outside of the MCMC. MCMC configurations include two sets of monitors, each with different thinning intervals. By default, the second set of monitors (`monitors2`) is empty.

7.2.1.5 Automated parameter blocking

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

The default configuration may be replaced by one 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)
```

Note that this using `autoBlock = TRUE` compiles and runs MCMCs, progressively exploring different sampler assignments, so it takes some time and generates some output. It is most useful for determining effective blocking strategies that can be re-used for later runs. The additional control argument `autoIt` may also be provided to indicate the number of MCMC samples to be used in each trial of the automated blocking procedure (default 20,000).

7.2.2 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 methods of an existing `MCMCconf` object.

7.2.2.1 Controlling which nodes to sample

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. The order of samplers will match the order of `nodes`. Any deterministic nodes will be ignored.

If a data node is included in `nodes`, *it will be assigned a sampler*. This is the only way in which a default sampler may be placed on a data node and will result in overwriting data values in the node.

7.2.2.2 Creating an empty configuration

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 any of `nodes = NULL`, `nodes = character()`, or `nodes = list()`.

7.2.2.3 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 and any subsequent sampler added via `addSampler` (see below). 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))
```

When adding samplers to a configuration using `addSampler`, the default control list can also be over-ridden.

7.2.2.4 Adding samplers to the configuration: *addSampler*

Additional samplers may be added to a configuration using the `addSampler` method of the `MCMCconf` object. The `addSampler` method has two modes of operation, determined by the `default` argument.

When `default = TRUE`, `addSampler` will assign NIMBLE's default sampling algorithm for each node specified following the same protocol as `configureMCMC`. This may include conjugate samplers, multivariate samplers, or otherwise, also using additional arguments to guide the selection process (for example, `useConjugacy` and `multivariateNodesAsScalars`). In this mode of operation, the `type` argument is not used.

When `default = FALSE`, or when this argument is omitted, `addSampler` uses the `type` argument to specify the precise sampler to assign. Instances of this particular sampler are assigned to all nodes specified. The `type` argument may be provided as a character string or a nimbleFunction object. Valid character strings are indicated in `help(samplers)` (do not include `"sampler_"`). Added samplers can be labeled with a `name` argument, which is used in output of `printSamplers`. Writing a new sampler as a nimbleFunction is covered in Section 15.5.

Regardless of the mode of operation, nodes are specified using either the `target` or the `nodes` argument. The `target` argument does not undergo expansion to constituent nodes (unless `default = TRUE`), and thus only a single sampler is added. The `nodes` argument is always expanded to the underlying nodes, and separate samplers are added for each node. Eithe argument is provided as a character vector. Newly added samplers will be appended to the end of current sampler list. Adding a sampler for a node will *not* remove existing samplers operating on that node.

The hierarchy of precedence for control list elements for added samplers is:

1. The `control` list argument provided to `addSampler`;
2. The original `control` list argument provided to `configureMCMC`;
3. The default values, as defined in the sampling algorithm `setup` function.

See `help(addSampler)` for more details.

7.2.2.5 Printing, re-ordering, modifying and removing samplers: *printSamplers*, *removeSamplers*, *setSamplers*, and *getSamplerDefinition*

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 `printSamplers` method accepts arguments controlling the level of detail displayed as discussed in its R help information.

```
# Print all samplers
mcmcConf$printSamplers()

# Print all samplers operating on node "a[1]",
# or any of the "beta[]" variables
mcmcConf$printSamplers(c("a[1]", "beta"))

# Print all conjugate and slice samplers
mcmcConf$printSamplers(type = c("conjugate", "slice"))

# Print all RW samplers operating on "x"
mcmcConf$printSamplers("x", type = "RW")

# Print the first 100 samplers
mcmcConf$printSamplers(1:100)

# Print all samplers in their order of execution
mcmcConf$printSamplers(executionOrder = TRUE)
```

Samplers may be removed from the configuration object using `removeSamplers`, which accepts a character vector of node or variable names, or a numeric vector of indices.

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

# Remove all samplers acting on "alpha[1]" and "beta[1]"
mcmcConf$removeSamplers(c("alpha[1]", "beta[1]"))

# Remove the first five samplers
mcmcConf$removeSamplers(1:5)

# Providing no argument removes all samplers
mcmcConf$removeSamplers()
```

Samplers to retain may be specified reordered using `setSamplers`, which also accepts a character vector of node or variable names, or a numeric vector of indices.

```
# 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"))

# Set the list of samplers to only those acting on model nodes
# "alpha[1]", "alpha[2]", ..., "alpha[10]"
mcmcConf$setSamplers("alpha[1:10]")

# Truncate the current list of samplers to the first 10 and the 100th
mcmcConf$setSamplers(ind = c(1:10, 100))
```

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")
```

7.2.2.6 Customizing individual sampler configurations: *getSamplers*, *setSamplers*, *setName*, *setSamplerFunction*, *setTarget*, and *setControl*

Each sampler in an `MCMCconf` object is represented by a sampler configuration as a `samplerConf` object. 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 `MCMCconf` method `getSamplers` allows access to the `samplerConf` objects. These can be modified and 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 behavior 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)`.

Here are some examples:

```
# 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 existance 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)
```

7.2.2.7 Customizing the sampler execution order

The ordering of sampler execution can be controlled as well. This allows for sampler functions to execute multiple times within a single MCMC iteration, or the execution of different sampler functions to be interleaved with one another.

The sampler execution order is set using the function `setSamplerExecutionOrder`, and the current ordering of execution is retrieved using `getSamplerExecutionOrder`. For example, assuming the MCMC configuration object `mcmcConf` contains five samplers:

```
# first sampler to execute twice, in succession:
mcmcConf$setSamplerExecutionOrder(c(1, 1, 2, 3, 4, 5))

# first sampler to execute multiple times, interleaved:
mcmcConf$setSamplerExecutionOrder(c(1, 2, 1, 3, 1, 4, 1, 5))

# fourth sampler to execute 10 times, only
mcmcConf$setSamplerExecutionOrder(rep(4, 10))

# omitting the argument to setSamplerExecutionOrder()
# resets the ordering to each sampler executing once, sequentially
mcmcConf$setSamplerExecutionOrder()

# retrieve the current ordering of sampler execution
ordering <- mcmcConf$getSamplerExecutionOrder()

# print the sampler functions in the order of execution
mcmcConf$printSamplers(executionOrder = TRUE)
```

7.2.2.8 Monitors and thinning intervals: *printMonitors*, *getMonitors*, *setMonitors*, *addMonitors*, *resetMonitors* and *setThin*

An MCMC configuration object contains two independent sets of variables to monitor, each with their own thinning interval: `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 multiple sets of values of model variables[1]. These can be accessed as the member data `mvSamples` and `mvSamples2` of the MCMC object, and they can be converted to matrices using `as.matrix` or lists using `as.list` (see Section 7.7).

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")
```

A new set of monitor variables can be added to the MCMC configuration, overwriting the current monitors, using the `setMonitors` method:

```
# Replace old monitors, now monitor "delta" and "gamma" only
mcmcConf$setMonitors("gamma", "delta")
```

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.

7.2.2.9 Monitoring model log-probabilities

To record model log-probabilities from an MCMC, one can add monitors for *logProb* variables (which begin with the prefix `logProb_`) that correspond to variables with (any) stochastic nodes. For example, to record and extract log-probabilities for the variables `alpha`, `sigma_mu`, and `Y`:

```
mcmcConf <- configureMCMC(Rmodel, enableWAIC = TRUE)
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)
```

[1] See Section 14.1 for general information on modelValues.

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

7.3 Building and compiling the MCMC

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 nimbleFunction specific to configuration `mcmcConf` (and of course its associated model).

Note that if you would like to be able to calculate the WAIC of the model, you should usually set `enableWAIC = TRUE` as an argument to`configureMCMC` (or to `buildMCMC` if not using `configureMCMC`), or set `nimbleOptions(MCMCenableWAIC = TRUE)`, which will enable WAIC calculations for all subsequently built MCMC functions. For more information on WAIC calculations, including situations in which you can calculate WAIC without having set `enableWAIC = TRUE` see Section 7.8 or `help(waic)` in R.

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 want to compile the MCMC function to C++ (as is the case for other NIMBLE functions). To do so, we use `compileNimble`. If the model has already been compiled, it should be provided as the `project` argument so the MCMC will be part of the same compiled project. 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)
```

Note that if you compile the MCMC with another object (the model in this case), you'll need to explicitly refer to the MCMC component of the resulting object to be able to run the MCMC:

```
Cmcmc$Rmcmc$run(niter = 1000)
```

7.4 Initializing MCMC

To see how to provide initial values, see `help(runMCMC)`, `help(nimbleMCMC)`, or `help(nimbleModel)`.

It is often important to provide valid and reasonable initial values to an MCMC, either as fixed values or via an initialization function.

Not doing so can cause slow convergence or even failure of an MCMC algorithm (e.g., if the values the model is initialized with are not valid). When starting an MCMC, when NIMBLE encounters a missing parameter value, it simulates from the prior distribution. NIMBLE can be more sensitive to missing or bad starting values than some MCMC packages.

The following cases are particularly important to consider when initializing: - In a model with flat priors (i.e., using `x~dflat()` or `x~dhalfflat()`), NIMBLE cannot generate initial values from those priors. - In a model with diffuse priors, initializing from the prior can give unreasonable/extreme initial values. - In a model with stochastic indices (e.g., `x[idx[i]]` with `idx[i]` unknown), those indices should have (valid) initial values. - In a model with constraints (via `dconstraint`), the model should be initialized such that the constraints are satisfied. - In a model with censoring (via `dinterval`), the initial value(s) of `t` in `dinterval` for the censored node(s) should be consistent with the censoring that is specified.

That said, some MCMC algorithms (such a conjugate samplers) don't use the initial values, so flat or diffuse priors on nodes assigned such samplers will not be a problem with regard to initialization.

Many user questions and problems end up being related to missing or bad initial values. Some suggestions for diagnosing initialization problems include:

- Run `model$initializeInfo()` to find uninitialized nodes.
- Run `model$calculate()` to see if the full model log probability density can be calculated (values of `-Inf`, `NA`, `NaN` indicate there are missing or invalid initial values).
- To debug the specific node(s) whose initial value(s) are causing problems, run `model$calculate(nodes)` on specific `nodes` to find their log probability density values, looking look for values of `-Inf`, `NA`, or `NaN`.
- For the node(s) with invalid log probability density, inspect the values of the node(s) and parameter(s) of the distribution(s) assigned to those node(s). You can inspect values of a variable in the model using natural R syntax, `model$variable_name`.
- With a compiled or uncompiled model, you can run the model initialization (but no MCMC iterations) with `mcmc$run(niter = 0)`. Then inspect the values of model variables to see the results of initialization.

7.5 User-friendly execution of MCMC algorithms: *runMCMC*

Once an MCMC algorithm has been created using `buildMCMC`, the function `runMCMC` can be used to run multiple chains and extract posterior samples, summary statistics and/or a WAIC value. This is a simpler approach to executing an MCMC algorithm, than the process of executing and extracting samples as described in Sections 7.6 and 7.7.

`runMCMC` also provides several user-friendly options such as burn-in, thinning, running multiple chains, and different initial values for each chain. However, using `runMCMC` does not support several lower-level options, such as timing the individual samplers internal to the MCMC, continuing an existing MCMC run (picking up where it left off), or modifying the sampler execution ordering.

`runMCMC` takes arguments that will control the following aspects of the MCMC:

- Number of iterations in each chain;
- Number of chains;
- Number of burn-in samples to discard from each chain;
- Thinning interval for recording samples;
- Initial values, or a function for generating initial values for each chain;
- Setting the random number seed;

- Returning the posterior samples as a `coda` `mcmc` object;
- Returning summary statistics calculated from each chains; and
- Returning a WAIC value calculated using (post-burn-in) samples from all chains.

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

```
# run a single chain, and return a matrix of samples
mcmc.out <- runMCMC(Cmcmc)

# run three chains of 10000 samples, discard initial burn-in of 1000,
# record samples thereafter using a thinning interval of 10,
# and return of list of sample matrices
mcmc.out <- runMCMC(Cmcmc, niter=10000, nburnin=1000, thin=10, nchains=3)

# run three chains, returning posterior samples, summary statistics,
# and the WAIC value
mcmc.out <- runMCMC(Cmcmc, nchains = 3, summary = TRUE, WAIC = TRUE)

# run two chains, and specify the initial values for each
initsList <- list(list(mu = 1, sigma = 1),
                  list(mu = 2, sigma = 10))
mcmc.out <- runMCMC(Cmcmc, nchains = 2, inits = initsList)

# run ten chains of 100,000 iterations each, using a function to
# generate initial values and a fixed random number seed for each chain.
# only return summary statistics from each chain; not all the samples.
initsFunction <- function()
    list(mu = rnorm(1,0,1), sigma = runif(1,0,100))
mcmc.out <- runMCMC(Cmcmc, niter = 100000, nchains = 10,
                    inits = initsFunction, setSeed = TRUE,
                    samples = FALSE, summary = TRUE)
```

See `help(runMCMC)` for further details.

7.6 Running the MCMC

The MCMC algorithm (either the compiled or uncompiled version) can be executed using the member method `mcmc$run` (see `help(buildMCMC)` in R). The `run` method has one required argument, `niter`, the number of iterations to be run.

The `run` method has optional arguments `nburnin`, `thin` and `thin2`. These can be used to specify the number of pre-thinning burn-in samples to discard, and the post-burnin thinning intervals for recording samples (corresponding to `monitors` and `monitors2`). If either `thin` and `thin2` are provided, they will override the thinning intervals that were specified in the original MCMC configuration object.

7.6.1 Rerunning versus restarting an MCMC

The `run` method has an optional `reset` argument. When `reset = TRUE` (the default value), the following occurs prior to running the MCMC:

1. All model nodes are checked and filled or updated as needed, in valid (topological) order. If a stochastic node is missing a value, it is populated using a call to `simulate` and its log probability value is calculated. The values of deterministic nodes are calculated from their parent nodes. If any right-hand-side-only nodes (e.g., explanatory variables) are missing a value, an error results.
2. 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.
3. 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).

This means that one can begin a new run of an existing MCMC without having to rebuild or recompile the model or the MCMC. This can be helpful if one wants to use the same model and MCMC configuration, but with different initial values, different values of data nodes (but which nodes are data nodes must be the same), changes to covariate values(or other non-data, non-parameter values) in the model, or a different number of MCMC iterations, thinning interval or burn-in.

In contrast, when `mcmc$run(niter, reset = FALSE)` is called, the MCMC picks up from where it left off, continuing the previous chain and expanding the output as needed. No values in the model are checked or altered, and sampler functions are not reset to their initial states. (Similarly, if using WAIC, one can use `resetWAIC = FALSE` so that the WAIC calculation is based on all the samples from the expanded set of samples.)

The `run` method also has an optional `resetMV` argument. This argument is only considered when`'reset` is set to `FALSE`. When `mcmc$run(niter, reset = FALSE, resetMV = TRUE)` is called 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) and the MCMC carries on from where it left off. In other words, the previously obtained samples are deleted (e.g. to reduce memory usage) prior to continuing the MCMC. The default value of `resetMV` is `FALSE`.

7.6.2 Measuring sampler computation times: *getTimes*

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 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.

7.6.3 Assessing the adaption process of *RW* and *RW_block* samplers

If you'd like to see the evolution (over the iterations) of the acceptance proportion and proposal scale information, you can use some internal methods provided by NIMBLE, after setting two options to make the history accessible. Here we assume that `cMCMC` is the compiled MCMC object and `idx` is the numeric index of the sampler function you want to access from amongst the list of sampler functions that are part of the MCMC.

```
## set options to make history accessible
nimbleOptions(buildInterfacesForCompiledNestedNimbleFunctions = TRUE)
nimbleOptions(MCMCsaveHistory = TRUE)
## Next, set up and run your MCMC.
## Now access the history information:
Cmcmc$samplerFunctions[[idx]]$getScaleHistory()
Cmcmc$samplerFunctions[[idx]]$getAcceptanceHistory()
Cmcmc$samplerFunctions[[idx]]$getPropCovHistory()   ## only for RW_block
```

Note that modifying elements of the control list may greatly affect the performance of the `RW_block` sampler. In particular, the sampler can take a long time to find a good proposal covariance when the elements being sampled are not on the same scale. We recommend providing an informed value for 'propCov' in this case (possibly simply a diagonal matrix that approximates the relative scales), as well as possibly providing a value of 'scale' that errs on the side of being too small. You may also consider decreasing 'adaptFactorExponent' and/or 'adaptInterval', as doing so has greatly improved performance in some cases.

7.7 Extracting MCMC samples

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

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

These *modelValues* objects can be converted into matrices using `as.matrix` or lists using `as.list`:

```
samplesMatrix <- as.matrix(mvSamples)
samplesList <- as.list(mvSamples)
samplesMatrix2 <- as.matrix(mvSamples2)
samplesList2 <- as.list(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]"])
```

The list version will contain an element for each variable that will be the size and shape of the variable with an additional index for MCMC iteration. By default MCMC iteration will be the first index, but including `iterationAsLastIndex = TRUE` will make it the last index.

Obtaining samples as matrices or lists is most common, but see Section 14.1 for more about programming with modelValues objects, especially if you want to write nimbleFunctions to use the samples.

7.8 Calculating WAIC

The WAIC (Watanabe, 2010) can be calculated from the posterior samples produced during the MCMC algorithm. Users have two options to calculate WAIC. The main approach requires enabling WAIC when setting up the MCMC and allows access to a variety of ways to calculate WAIC. This approach does not require that any specific monitors be set[2] because the WAIC is calculated in an online manner, accumulating the necessary quantities to calculate WAIC as the MCMC is run.

The second approach allows one to calculate the WAIC after an MCMC has been run using an MCMC object or matrix (or dataframe) containing the posterior samples, but it requires the user to have correctly specified the monitored variables and only provides the default way to calculate WAIC. An advantage of the second approach is that one can specify additional burnin beyond that specified for the MCMC.

Here we first discuss the main approach and then close the section by showing the second approach. Specific details of the syntax are provided in `help(waic)`.

To enable WAIC for the first approach, the argument `enableWAIC = TRUE` must be supplied to `configureMCMC` (or to `buildMCMC` if not using `configureWAIC`), or the `MCMCenableWAIC` NIMBLE option must have been set to `TRUE`.

The WAIC (as well as the pWAIC and lppd values) is extracted by the member method `mcmc$getWAIC` (see `help(waic)` in R for more details) or is available as the `WAIC` element of the `runMCMC` or `nimbleMCMC` output lists. One can use the member method `mcmc$getWAICdetails` for additional quantities related to WAIC that are discussed below.

Note that there is not a unique value of WAIC for a model. WAIC is calculated from Equations 5, 12, and 13 in Gelman et al. (2014) (i.e., using pWAIC2), as discussed in detail in Hug and Paciorek (2021). Therefore, NIMBLE provides user control over how WAIC is calculated in two ways.

First, by default NIMBLE provides the conditional WAIC, namely the version of WAIC where all parameters directly involved in the likelihood are treated as θ for the purposes of Equation 5 from Gelman et al. (2014). Users can request the marginal WAIC (see Ariyo et al. (2020)), namely the version of WAIC where latent variables are integrated over (i.e., using a marginal likelihood). This is done by providing the `waicControl` list with a `marginalizeNodes` element to `configureMCMC` or `buildMCMC` (when providing a model as the argument to `buildMCMC`). See `help(waic)` for more details.

Second, WAIC relies on a partition of the observations, i.e., 'pointwise' prediction. By default, in NIMBLE the sum over log pointwise predictive density values treats each data node as contributing a single value to the sum. When a data node is multivariate, that data node contributes a single value to the sum based on the joint density of the elements in the node. If one wants to group data nodes such that the joint density within each group is used, one can provide the `waicControl` list with a `dataGroups` element to `configureMCMC` or `buildMCMC` (when providing a model as the argument to `buildMCMC`). See `help(waic)` for more details.

Note that based on a limited set of simulation experiments in Hug and Paciorek (2021), our tentative recommendation is that users only use marginal WAIC if also using grouping.

Marginal WAIC requires using Monte Carlo simulation at each iteration of the MCMC to average

[2] Prior to version 0.12.0, NIMBLE required specific monitors, because WAIC was calculated at the end of the MCMC using the entire set of MCMC samples.

over draws for the latent variables. To assess the stability of the marginal WAIC to the number of Monte Carlo iterations, one can examine how the WAIC changes with increasing iterations (up to the full number of iterations specified via the `nItsMarginal` element of the `waicControl` list) based on the `WAIC_partialMC`, `lppd_partialMC`, and `pWAIC_partialMC` elements of the detailed WAIC output.

For comparing WAIC between two models, Vehtari et al. (2017) discuss using the per-observation (more generally, per-data group) contributions to the overall WAIC values to get an approximate standard error for the difference in WAIC between the models. These element-wise values are available as the `WAIC_elements`, `lppd_elements`, and `pWAIC_elements` components of the detailed WAIC output.

The second overall approach to WAIC available in NIMBLE allows one to calculate WAIC post hoc after MCMC sampling using an MCMC object or matrix (or dataframe) containing posterior samples. Simply set up the MCMC and run it without enabling WAIC (but making sure to include in the monitors all stochastic parent nodes of the data nodes) and then use the function `calculateWAIC`, as shown in this example:

```r
samples <- runMCMC(Cmcmc, niter = 10000, nburnin = 1000)
## Using calculateWAIC with an MCMC object
calculateWAIC(Cmcmc)
## Using calculateWAIC with a matrix of samples
calculateWAIC(samples, model)
## Specifying additional burnin, so only last 5000 (1000+4000) iterations used
calculateWAIC(Cmcmc, burnin = 4000)
```

This approach only provides conditional WAIC without any grouping of data nodes (though one can achieve grouping by grouping data nodes into multivariate nodes).

7.9 k-fold cross-validation

The `runCrossValidate` function in NIMBLE performs k-fold cross-validation on a `nimbleModel` fit via MCMC. More information can be found by calling `help(runCrossValidate)`.

7.10 Variable selection using Reversible Jump MCMC

A common method for Bayesian variable selection in regression-style problems is to define a space of different models and have MCMC sample over the models as well as the parameters for each model. The models differ in which explanatory variables are included. Often this idea is implemented in the BUGS language by writing the largest possible model and including indicator variables to turn regression parameters off and on (see model code below for an example of how indicator variables can be used). Foramally, that approach doesn't sample between different models, instead embedding the models in one large model. However, that approach can result in poor mixing and require compromises in choices of prior distributions because a given regression parameter will sample from its prior when the corresponding indicator is 0. It is also computationally wasteful, because sampling effort will be spent on a coefficient even if it has no effect because the corresponding indicator is 0.

A different view of the problem is that the different combinations of coefficients represent models of different dimensions. Reversible Jump MCMC (Green, 1995) (RJMCMC) is a general framework for MCMC simulation in which the dimension of the parameter space can vary between iterates of the Markov chain. The reversible jump sampler can be viewed as an extension of the Metropolis-Hastings algorithm onto more general state spaces. NIMBLE provides an implementation of RJMCMC for variable selection that requires the user to write the largest model of interest and supports use of indicator variables but formally uses RJMCMC sampling for better mixing and efficiency. When a coefficient is not in the model (or its indicator is 0), it will not be sampled, and it will therefore not follow its prior in that case.

In technical detail, given two models M_1 and M_2 of possibly different dimensions, the core idea of RJMCMC is to remove the difference in the dimensions of models M_1 and M_2 by supplementing the corresponding parameters θ_1 and θ_2 with auxiliary variables $u_{1\to 2}$ and $u_{2\to 1}$ such that $(\theta_1, u_{1\to 2})$ and $(\theta_2, u_{2\to 1})$ are in bijection, $(\theta_2, u_{2\to 1}) = \Psi(\theta_1, u_{1\to 2})$. The corresponding Metropolis-Hastings acceptance probability is generalized accounting for the proposal density for the auxiliary variables.

NIMBLE implements RJMCMC for variable selection using a univariate normal distribution centered on 0 (or some fixed value) as the proposal density for parameters being added to the model. Two ways to set up models for RJMCMC are supported, which differ by whether the inclusion probabilities for each parameter are assumed known or must be written in the model:

- If the inclusion probabilities are assumed known, then RJMCMC may be used with regular model code, i.e. model code written without heed to variable selection.
- If the inclusion probability is a parameter in the model, perhaps with its own prior, then RJMCMC requires that indicator variables be written in the model code. The indicator variables will be sampled using RJMCMC, but otherwise they are like any other nodes in the model.

The steps to set up variable selection using RJMCMC are:

1. Write the model with indicator variables if needed.
2. Configure the MCMC as usual with `configureMCMC`.
3. Modify the resulting MCMC configuration object with `configureRJ`.

The `configureRJ` function modifies the MCMC configuration to (1) assign samplers that turn on and off variable in the model and (2) modify the existing samplers for the regression coefficients to use 'toggled' versions. The toggled versions invoke the samplers only when corresponding variable is currently in the model. In the case where indicator variables are included in the model, sampling to turn on and off variables is done using `RJ_indicator` samplers. In the case where indicator variables are not included, sampling is done using `RJ_fixed_prior` samplers.

In the following we provide an example for the two different model specifications.

7.10.1 Using indicator variables

Here we consider a normal linear regression in which two covariates `x1` and `x2` are candidates to be included in the model. The two corresponding coefficients are `beta1` and `beta2`. The indicator variables are `z1` and `z2`, and their inclusion probability is `psi`. As described below, one can also use vectors for a set of coefficients and corresponding indicator variables.

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

## simulate some data
set.seed(1)
N <- 100
x1 <- runif(N, -1, 1)
x2 <- runif(N, -1, 1) ## this covariate is not included
Y <- rnorm(N, 1.5 + 2 * x1, sd = 1)


## build the model and configure default MCMC

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

RJexampleConf <- configureMCMC(RJexampleModel)
```

```
## For illustration, look at the default sampler assignments
RJexampleConf$printSamplers()
```

```
## [1] conjugate_dnorm_dnorm_additive sampler: beta0
## [2] conjugate_dnorm_dnorm_linear sampler: beta1
## [3] conjugate_dnorm_dnorm_linear sampler: beta2
## [4] RW sampler: sigma
## [5] conjugate_dbeta_dbern_identity sampler: psi
## [6] binary sampler: z1
## [7] binary sampler: z2
```

At this point we can modify the MCMC configuration object, `RJexampleConf`, to use reversible jump samplers for selection on `beta1` and `beta2`.

```
configureRJ(conf = RJexampleConf,
            targetNodes = c("beta1", "beta2"),
            indicatorNodes = c('z1', 'z2'),
            control = list(mean = c(0, 0), scale = 2))
```

The `targetNodes` argument gives the coefficients (nodes) for which we want to do variable selection.

The indicatorNodes gives the corresponding indicator nodes, ordered to match targetNodes. The control list gives the means and scale (standard deviation) for normal reversible jump proposals for targetNodes. The means will typically be 0 (by default mean = 0 and scale = 1), but they could be anything. An optional control element called fixedValue can be provided in the non-indicator setting; this gives the value taken by nodes in targetNodes when they are out of the model (by default, fixedValue is 0). All control elements can be scalars or vectors. A scalar values will be used for all targetNodes. A vector value must be of equal length as targetNodes and will be used in order.

To use RJMCMC on a vector of coefficients with a corresponding vector of indicator variables, simply provide the variable names for targetNodes and indicatorNodes. For example, targetNodes = "beta" is equivalent to targetNodes = c("beta[1]", "beta[2]") and indicatorNodes = "z" is equivalent to indicatorNodes = c("z[1]", "z[2]"). Expansion of variable names into a vector of node names will occur as described in Section 13.3.1.1. When using this method, *both* arguments must be provided as variable names to be expanded.

Next we can see the result of configureRJ by printing the modified list of samplers:

```
RJexampleConf$printSamplers()
```

```
## [1] conjugate_dnorm_dnorm_additive sampler: beta0
## [2] RW sampler: sigma
## [3] conjugate_dbeta_dbern_identity sampler: psi
## [4] RJ_indicator sampler: z1,  mean: 0,  scale: 2,  targetNode: beta1
## [5] RJ_toggled sampler: beta1,  samplerType: conjugate_dnorm_dnorm_linear
## [6] RJ_indicator sampler: z2,  mean: 0,  scale: 2,  targetNode: beta2
## [7] RJ_toggled sampler: beta2,  samplerType: conjugate_dnorm_dnorm_linear
```

An RJ_indicator sampler was assigned to each of z[1] and z[2] in place of the binary sampler, while the samplers for beta[1]and beta[2] have been changed to RJ_toggled samplers. The latter samplers contain the original samplers, in this case conjugate_dnorm_dnorm samplers, and use them only when the corresponding indicator variable is equal to 1 (i.e., when the coefficient is in the model).

Notice that the order of the samplers has changed, since configureRJ calls removeSampler for nodes in targetNodes and indicatorNodes, and subsequently addSampler, which appends the sampler to the end of current sampler list. Order can be modified by using setSamplers.

Also note that configureRJ modifies the MCMC configuration (first argument) and returns NULL.

7.10.2 Without indicator variables

We consider the same regression setting, but without the use of indicator variables and with fixed probabilities of including each coefficient in the model.

```
## Linear regression with intercept and two covariates
code <- nimbleCode({
  beta0 ~ dnorm(0, sd = 100)
  beta1 ~ dnorm(0, sd = 100)
  beta2 ~ dnorm(0, sd = 100)
  sigma ~ dunif(0, 100)
  for(i in 1:N) {
    Ypred[i] <- beta0 + beta1 * x1[i] + beta2 * x2[i]
    Y[i] ~ dnorm(Ypred[i], sd = sigma)
  }
})

## build the model
RJexampleModel2 <- nimbleModel(code, constants = list(N = N),
                               data = list(Y = Y, x1 = x1, x2 = x2),
                               inits = list(beta0 = 0, beta1 = 0, beta2 = 0,
                                            sigma = sd(Y)))

RJexampleConf2 <- configureMCMC(RJexampleModel2)
```

```
## print NIMBLE default samplers
RJexampleConf2$printSamplers()
```

```
## [1] conjugate_dnorm_dnorm_additive sampler: beta0
## [2] conjugate_dnorm_dnorm_linear sampler: beta1
## [3] conjugate_dnorm_dnorm_linear sampler: beta2
## [4] RW sampler: sigma
```

In this case, since there are no indicator variables, we need to pass to `configureRJ` the prior inclusion probabilities for each node in `targetNodes`, by specifying either one common value or a vector of values for the argument `priorProb`. This case does not allow for a stochastic prior.

```
configureRJ(conf = RJexampleConf2,
            targetNodes = c("beta1", "beta2"),
            priorProb = 0.5,
            control = list(mean = 0, scale = 2, fixedValue = c(1.5, 0)))
```

```
## print samplers after configureRJ
RJexampleConf2$printSamplers()
```

```
## [1] conjugate_dnorm_dnorm_additive sampler: beta0
## [2] RW sampler: sigma
## [3] RJ_fixed_prior sampler: beta1,  priorProb: 0.5,  mean: 0,  scale: 2,  fixedValue: 1.5
## [4] RJ_toggled sampler: beta1,  samplerType: conjugate_dnorm_dnorm_linear,  fixedValue: 1.5
## [5] RJ_fixed_prior sampler: beta2,  priorProb: 0.5,  mean: 0,  scale: 2,  fixedValue: 0
## [6] RJ_toggled sampler: beta2,  samplerType: conjugate_dnorm_dnorm_linear,  fixedValue: 0
```

Since there are no indicator variables, the `RJ_fixed_prior` sampler is assigned directly to each of `beta[1]` and `beta[2]` along with the `RJ_toggled` sampler. The former sets a coefficient to its `fixedValue` when it is out of the model. The latter invokes the regular sampler for the coefficient only if it is in the model at a given iteration.

If `fixedValue` is given when using `indicatorNodes` the values provided in `fixedValue` are ignored. However the same behavior can be obtained in this situation, using a different model specification. For example, the model in 7.10.1 can be modified to have `beta1` equal to 1.5 when not in the model as follows:

```
for(i in 1:N) {
  Ypred[i] <- beta0 + (1 - z1) * 1.5 * beta1 * x1[i] +
              z1 * beta1 * x1[i] + beta2 * z2 * x2[i]
  Y[i] ~ dnorm(Ypred[i], sd = sigma)
}
```

7.11 Samplers provided with NIMBLE

Most documentation of MCMC samplers provided with NIMBLE can be found by invoking `help(samplers)` in R. Here we provide additional explanation of conjugate samplers and how complete customization can be achieved by making a sampler use an arbitrary log-likelihood function, such as to build a particle MCMC algorithm.

7.11.1 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 current release of NIMBLE supports conjugate sampling of the relationships listed in Table 7.1[3].

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

Prior Distribution	Sampling (Dependent Node) Distribution	Parameter
Beta	Bernoulli	`prob`
	Binomial	`prob`
	Negative Binomial	`prob`
Dirichlet	Multinomial	`prob`
	Categorical	`prob`
Flat	Normal	`mean`
	Lognormal	`meanlog`
Gamma	Poisson	`lambda`
	Normal	`tau`
	Lognormal	`taulog`
	Gamma	`rate`
	Inverse Gamma	`scale`
	Exponential	`rate`
	Double Exponential	`rate`
	Weibull	`lambda`
Halfflat	Normal	`sd`
	Lognormal	`sdlog`
Inverse Gamma	Normal	`var`
	Lognormal	`varlog`
	Gamma	`scale`

[3] NIMBLE's internal definitions of these relationships can be viewed with `nimble:::conjugacyRelationshipsInputList`.

Prior Distribution	Sampling (Dependent Node) Distribution	Parameter
	Inverse Gamma	`rate`
	Exponential	`scale`
	Double Exponential	`scale`
Normal	Normal	`mean`
	Lognormal	`meanlog`
Multivariate Normal	Multivariate Normal	`mean`
Wishart	Multivariate Normal	`prec`
Inverse Wishart	Multivariate Normal	`cov`

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 with `nimbleOptions(verifyConjugatePosteriors = TRUE)`.

If one wants information about conjugate node relationships for other purposes, they can be obtained using the `checkConjugacy` method on a model. This returns a named list describing all conjugate relationships. The `checkConjugacy` method can also accept a character vector argument specifying a subset of node names to check for conjugacy.

7.11.2 Hamiltonian Monte Carlo (HMC)

As of version 1.0.0, NIMBLE provides support for automatic differentiation (AD), as described in Chapter 16. AD is needed for Hamiltonian Monte Carlo (HMC). The HMC algorithm is provided through the `nimbleHMC` package, which requires version 1.0.0 of the `nimble` package.

The HMC sampler in package `nimbleHMC` implements the NUTS version of HMC. HMC is an MCMC method that works by a physical analogy of frictionless motion on a surface that is the negative log posterior density. HMC often achieves good mixing, although at high computational cost. Within `nimble`, HMC is more flexible than in some implementations because HMC can be used on some parts of a model while other samplers are used on (the same or) other parts of the same model. Indeed, Neal (2011) discussed using it in this way.

At its core, HMC in NIMBLE is just another MCMC sampler that can be used in NIMBLE's general MCMC system, possibly along with other samplers. For convenience, `nimbleHMC` provides different ways to set up and run HMC that allow different degrees of control:

- `nimbleHMC` is analogous to `nimbleMCMC`: It takes model `code`, `data`, `inits` and `constants` and does all steps from building the model to running one or more MCMC chains. Use this if you want one call to get default HMC samples of the entire model.
- `configureHMC` is analogous to `configureMCMC`: It sets up a default MCMC configuration that comprises one HMC sampler for all model nodes (or nodes you choose). You can then use `addSampler` and other methods of the configuration to fully customize it before building and using the MCMC. Use `configureHMC` if you want to do MCMC with both HMC and other samplers and you find it easier to start with HMC in the configuration first.
- `buildHMC` is analogous to `buildMCMC`: It builds a configuration that, if not provided, will be created by `configureHMC`. Use this if you want to manage MCMC runs through `runMCMC` instead of `nimbleHMC` and you don't need to customize the configuration by doing `configureHMC` yourself.

- `addHMC`: This manages use of `addSampler` for an MCMC configuration to add specifically an HMC sampler for a chosen set of nodes. It optionally removes other samplers for those nodes first. Use this if you want to do MCMC with both HMC and other samplers and you find it easier to start with the other sampler in the configuration, i.e. from the result of `configureMCMC`.
- For the most fine-grained control, you can call `configureMCMC` and then use the `addSampler` with `type="HMC"` to add HMC sampler(s).

More information on these is available from the help pages, e.g. `help(configureHMC)`.

7.11.2.1 HMC example

Here we will configure and build an MCMC to use HMC on a GLMM example. In this case we'll use HMC for the whole model, i.e., all parameters and latent states.

We'll use a Poisson Generalized Linear Mixed Model (GLMM) as a simple example model for HMC (and also for Laplace approximation in 16.1). There will be 10 groups (indexed by `i`) of 5 observations (indexed by `j`) each. Each observation has a covariate, `X`, and each group has a random effect `ran_eff`. Here is the model code:

```
model_code <- nimbleCode({
  # priors
  intercept ~ dnorm(0, sd = 100)
  beta ~ dnorm(0, sd = 100)
  sigma ~ dunif(0, 10)
  # random effects and data
  for(i in 1:10) {
    # random effects
    ran_eff[i] ~ dnorm(0, sd = sigma)
    for(j in 1:5) {
      # data
      y[i,j] ~ dpois(exp(intercept + beta*X[i,j] + ran_eff[i]))
    }
  }
})
```

We'll simulate some values for `X`.

```
set.seed(123)
X <- matrix(rnorm(50), nrow = 10)
```

In order to allow an algorithm to use AD for a specific model, that model must be created with `buildDerivs = TRUE`. (Note that if you use the one-step method `nimbleHMC` described above, then you do not need to build the model yourself and so you do not need to provide this `buildDerivs` option.)

```
inits <- list(intercept = 0, beta = 0.2, sigma = 0.5)
model <- nimbleModel(model_code, constants = list(X = X), inits = inits,
      calculate = FALSE, buildDerivs = TRUE) # Here is the argument needed for AD.
```

To finish setting up the example we need to put data values into the model. We could have provided data in the call to `nimbleModel`, but instead we will simulate them using the model itself. Specifically, we will set parameter values, simulate data values, and then set those as the data to use.

```
model$calculate() # This will return NA because the model is not fully initialized.
```

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

```
model$simulate(model$getDependencies('ran_eff'))
model$calculate() # Now the model is fully initialized: all nodes have valid values.
```

```
## [1] -80.74344
```

```
model$setData('y') # Now the model has y marked as data, with values from simulation.
```

Finally, we will make a compiled version of the model.

```
Cmodel <- compileNimble(model)
```

Now we will build an HMC algorithm for the parameters and random effects in the model.

```
library(nimbleHMC)
HMC <- buildHMC(model)
```

```
## ===== Monitors =====
## thin = 1: beta, intercept, sigma
## ===== Samplers =====
## HMC sampler (1)
##   - intercept, beta, sigma, ran_eff[1], ran_eff[2], ran_eff[3], ran_eff[4], ran_eff[5], ran_
```

```
CHMC <- compileNimble(HMC, project = model)
# The compiled HMC will use the compiled model.
samples <- runMCMC(CHMC, niter = 1000, nburnin = 500) # short run for illustration
```

```
##   [Note] HMC sampler (nodes: intercept, beta, sigma, ran_eff[1], ran_eff[2], ran_eff[3], ran
##   [Note] HMC sampler (nodes: intercept, beta, sigma, ran_eff[1], ran_eff[2], ran_eff[3], ran
```

We will not investigate results in detail, but here is a summary to see that reasonable results were generated.

```
summary(coda::as.mcmc(samples))
```

```
## 
## Iterations = 1:500
## Thinning interval = 1
## Number of chains = 1
## Sample size per chain = 500
## 
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
## 
##               Mean     SD Naive SE Time-series SE
## beta       0.1901 0.1422 0.006358       0.006358
## intercept -0.1865 0.2983 0.013341       0.026606
## sigma      0.7460 0.2691 0.012036       0.020029
##
## 2. Quantiles for each variable:
##
##               2.5%      25%     50%     75%  97.5%
## beta      -0.08399  0.08895  0.1893 0.30176 0.4595
## intercept -0.79899 -0.38862 -0.1843 0.01333 0.3963
## sigma      0.31865  0.55724  0.7078 0.88552 1.3849
```

7.11.2.2 Automatic parameter transformation

In the example above, `sigma` is a parameter that must be non-negative. HMC works only on unconstrained parameter spaces. NIMBLE's HMC functionality automatically transforms any constrained parameters to be in an unconstrained parameter space, runs the MCMC, and back-transforms to the original space before providing the MCMC samples. The details of the parameter transformation system can be found in Section 16.8.

7.11.3 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 (Andrieu et al., 2010). The `RW_llFunction` sampler handles this by using a Metropolis-Hastings algorithm with a normal proposal distribution and a user-provided log-likelihood function. To allow compiled execution, the log-likelihood function must be provided as a specialized instance of a nimbleFunction. The log-likelihood function may use the same model as the MCMC as a setup argument (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. This is the specialized nimbleFunction that calculates the log-likelihood; it must accept no arguments and return 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.

Here is a complete example:

```
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[1])
    }
)

RllFun <- llFun(Rmodel)

mcmcConf <- configureMCMC(Rmodel, nodes = NULL)

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

Rmcmc <- buildMCMC(mcmcConf)
```

Note that we need to return `ll[1]` and not just `ll` because there are no scalar variables in compiled models. Hence `y` and other variables, and therefore `ll`, are of dimension 1 (i.e., vectors / one-dimensional arrays), so we need to specify the first element in order to have the return type be a scalar.

7.11.4 Particle MCMC 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.

7.12 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()
```

```r
# 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) # alternative: as.list
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)
```

```r
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)
```

7.13 Comparing different MCMCs with *MCMCsuite* and *compareMCMCs*

Please see the `compareMCMCs` package for the features previously provided by `MCMCsuite` and `compareMCMCs` in NIMBLE (until version 0.8.0). The `compareMCMCs` package provides tools to automatically run MCMC in nimble (including multiple sampler configurations), WinBUGS, OpenBUGS, JAGS, Stan, or any other engine for which you provide a simple common interface. The package makes it easy to manage comparison metrics and generate html pages with comparison figures.

7.14 Running MCMC chains in parallel

It is possible to run multiple chains in parallel using standard R parallelization packages such as `parallel`, `foreach`, and `future`. However, you must create separate copies of all model and MCMC objects using `nimbleModel`, `buildMCMC`, `compileNimble`, etc. This is because NIMBLE uses Reference Classes and R6 classes, so copying such objects simply creates a new variable name that refers to the original object.

Thus, in your parallel loop or lapply-style statement, you should run `nimbleModel` and all subsequent calls to create and compile the model and MCMC algorithm within the parallelized block of code, once for each MCMC chain being run in parallel.

For a worked example, please see the parallelization example on our webpage.

Chapter 8

Sequential Monte Carlo, Particle MCMC, Iterated Filtering, and MCEM

The NIMBLE algorithm library is growing and currently includes a suite of sequential Monte Carlo (particle filtering) algorithms, particle MCMC for combining particle filters with MCMC, iterated filtering version 2 and Monte Carlo expectation maximization (MCEM) for maximum likelihood estimation, and k-fold cross-validation.

8.1 Particle filters / sequential Monte Carlo and iterated filtering

As of Version 0.10.0 of NIMBLE, all of NIMBLE's sequential Monte Carlo/particle filtering functionality lives in the `nimbleSMC` package. Please load this package before trying to use these algorithms.

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. These include the bootstrap filter, the auxiliary particle filter, the Liu-West filter, and the ensemble Kalman filter. The iterated filtering version 2 (IF2) is a related method for maximum-likelihood estimation. Each of these is built with the eponymous functions `buildBootstrapFilter`, `buildAuxiliaryFilter`, `buildLiuWestFilter`, `buildEnsembleKF`, and `buildIteratedFilter2`. Each method requires setup arguments `model` and `nodes`; the latter should be a character vector specifying latent model nodes. In addition, each method can be customized using a `control` list argument. Details on the control options and specifics of the 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 modelValues 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, as well as the IF2 method, also have another modelValues object, `mvWSamples`. This 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)
```

8.1.1.1 Bootstrap filter

Here is an example of building and running the bootstrap filter.

```
# 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$run(parNum)
# The bootstrap filter can also return an estimate of the effective
# sample size (ESS) at each time point
bootESS <- cBootF$returnESS()
```

8.1.1.2 Auxiliary particle filter

Next, we provide an example of building and running the auxiliary particle filter. 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)
# The auxiliary filter can also return an estimate of the effective
# sample size (ESS) at each time point
auxESS <- cAuxF$returnESS()
```

8.1.1.3 Liu and West filter

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. Note that the Liu-West filter ofen performs poorly and is provided primarily for didactic purposes. The Liu and West filter accepts an additional `params` argument, specifying the top-level parameters to be sampled.

```
# 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))
```

```
## Warning in buildLiuWestFilter(LWTimeModel, "x", params = c("a", "b", "c"), :
## The Liu-West filter ofen performs poorly and is provided primarily for didactic
## purposes.
```

```
# Compile filter
cLWF <- compileNimble(rLWF,project = LWTimeModel)
# Run Liu-West filter
cLWF$run(parNum)
```

8.1.1.4 Ensemble Kalman filter

Next we give an example of building and running the ensemble Kalman filter, which can sample from the posterior distribution of latent states.

```
# Copy model
ENKFTimeModel <- 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$mvEWSamples) # alternative: as.list
auxEWSamp <- as.matrix(cAuxF$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")
```

8.1.1.5 Iterated filtering 2 (IF2)

The IF2 method accomplishes maximum likelihood estimation using a scheme wherein both latent states and parameters are represented by particles that are weighted and resampled during the iterations. Iterations include perturbations to the parameter particles following a schedule of decreasing magnitude to yield convergence to the MLE.

Here we apply IF2 to Nile River flow data, specifying a changepoint in the year the Aswan Dam was constructed, as the dam altered river flows.

```
library(FKF)

flowCode <- nimbleCode({
    for(t in 1:n)
        y[t] ~ dnorm(x[t], sd = sigmaMeasurements)
    x[1] ~ dnorm(x0, sd = sigmaInnovations)
    for(t in 2:n)
        x[t] ~ dnorm((t-1==28)*meanShift1899 + x[t-1], sd = sigmaInnovations)
    logSigmaInnovations ~ dnorm(0, sd = 100)
    logSigmaMeasurements ~ dnorm(0, sd = 100)
    sigmaInnovations <- exp(logSigmaInnovations)
    sigmaMeasurements <- exp(logSigmaMeasurements)
    x0 ~ dnorm(1120, var = 100)
    meanShift1899 ~ dnorm(0, sd = 100)
})

flowModel <- nimbleModel(flowCode, data = list(y = Nile),
                 constants = list(n = length(Nile)),
                 inits = list(logSigmaInnovations = log(sd(Nile)),
                              logSigmaMeasurements = log(sd(Nile)),
                              meanShift1899 = -100))
```

Note that the prior distributions for the parameters are not used by IF2, except possibly to obtain boundaries of valid parameter values (not the case here).

Now we build the filter, specifying user-controlled standard deviations (in this case the same as the perturbation `sigma` values) for use in generating the initial particles for the parameters via the `control` list.

```
filter <- buildIteratedFilter2(model = flowModel,
                           nodes = 'x',
                           params = c('logSigmaInnovations',
                                      'logSigmaMeasurements',
                                      'meanShift1899'),
                           baselineNode = 'x0',
                           control = list(sigma = c(0.1, 0.1, 5),
                                          initParamSigma = c(0.1, 0.1, 5)))
cFlowModel <- compileNimble(flowModel)
cFilter  <- compileNimble(filter, project = flowModel)
```

We now run the algorithm with 1000 particles for 100 iterations with the schedule parameter equal to 0.2.

In addition to the estimates, we can extract the values of the log-likelihood, the estimates and the standard deviation of the parameter particles as they evolve over the iterations, in order to assess convergence.

```
set.seed(1)
est <- cFilter$run(m = 1000, niter = 100, alpha = 0.2)

cFilter$estimates[95:100,] ## Last 5 iterations of parameter values
```

```
##              [,1]     [,2]      [,3]
## [1,]  0.013306153 4.822613 -269.7837
## [2,] -0.013961938 4.857470 -271.4965
## [3,]  0.002183997 4.850475 -271.1919
## [4,]  0.015598044 4.851961 -272.3889
## [5,] -0.005571944 4.842719 -272.6390
## [6,]  0.042982317 4.837347 -271.1800
```

```
cFilter$logLik[90:100] ## Last 5 iterations of log likelihood values
```

```
##  [1] -627.0497 -626.9570 -626.9856 -626.9954 -626.7448 -626.7521 -626.8472
##  [8] -626.7398 -626.9869 -627.1354 -626.6648
```

Comparing to use of the the Kalman Filter from the FKF package, we see the log-likelihood is fairly similar:

```
dtpred <- matrix(0, ncol = length(Nile))
dtpred[28] <- est[3]
ct <- matrix(0)
Zt <- Tt <- matrix(1)

fkfResult <- fkf(HHt = matrix(exp(2*est[1])),
                 GGt = matrix(exp(2*est[2])),
                 yt = rbind(Nile),
                 a0 = 1120,
                 P0 = matrix(100),
                 dt = dtpred, ct = ct, Zt = Zt, Tt = Tt)
fkfResult$logLik
```

```
## [1] -626.5521
```

8.1.2 Particle MCMC (PMCMC)

Particle MCMC (PMCMC) is a method that uses MCMC for top-level model parameters and uses a particle filter to approximate the time-series likelihood for use in determining MCMC acceptance probabilities (Andrieu et al., 2010). NIMBLE implements PMCMC by providing random-walk Metropolis-Hastings samplers for model parameters that make use of particle filters in this way. These samplers can use NIMBLE's bootstrap filter or auxiliary particle filter, or they can use a user-defined filter. Whichever filter is specified will be used to obtain estimates of the likelihood of the state-space model (marginalizing over the latent states), which is used for calculation of the Metropolis-Hastings acceptance probability. The `RW_PF` sampler uses a univariate normal proposal distribution, and can be used to sample scalar top-level parameters. The `RW_PF_block` sampler uses a multivariate normal proposal distribution and can be used to jointly sample vectors of top-level parameters. 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 Section 7.11.

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 used. In addition, providing `pfOptimizeNparticles=TRUE` in the `control` list will use an experimental algorithm to estimate the optimal number of particles to use in the particle 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, nodes = NULL) # empty 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))
```

The `type = "RW_PF"` and `type = "RW_PF*block"` samplers default to using a bootstrap filter. The adapatation control parameters `adaptive`, `adaptInterval`, and `adaptScaleOnly` work in the same way as for an `RW` and `RW*block` samplers. However, it is not clear if the same approach to adaptation works well for PMCMC, so one should consider turning off adaptation and using a well-chosen proposal covariance.

It is also possible that more efficient results can be obtained by using a custom filtering algorithm. Choice of filtering algorithm can be controlled by the `pfType` control list entry. The `pfType` entry can be set either to `'bootstrap'` (the default), `'auxiliary'`, or the name of a user-defined nimbleFunction that returns a likelihood approximation.

Any user-defined filtering nimbleFunction named in the `pfType` control list entry must satsify the following:

1. The nimbleFunction must be the result of a call to `nimbleFunction()`.
2. The `nimbleFunction` must have setup code that accepts the following (and only the following) arguments:
   - `model`, the NIMBLE model object that the MCMC algorithm is defined on.
   - `latents`, a character vector specifying the latent model nodes over which the particle filter will stochastically integrate over to estimate the log-likelihood function.
   - `control`, an `R list` object. Note that the `control` list can be used to pass in any additional information or arguments that the custom filter may require.
3. The `nimbleFunction` must have a `run` function that accepts a single integer arugment (the number of particles to use), and returns a scalar double (the log-likelihood estimate).
4. The `nimbleFunction` must define, in setup code, a `modelValues` object named `mvEWSamples` that is used to contain equally weighted samples of the latent states (that is, the `latents` argument to the setup function). Each time the `run()` method of the `nimbleFunction` is called with number of particles `m`, the `mvEWSamples` `modelValues` object should be resized to be of size `m` via a call to `resize(mvEWSamples, m)`.

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 ability to combine R and NIMBLE to run an algorithm, with R managing the highest-level processing of the algorithm and calling nimbleFunctions for computations.

NIMBLE provides an ascent-based MCEM algorithm, created using `buildMCEM`, that automatically determines when the algorithm has converged by examining the size of the changes in the likelihood between each iteration. Additionally, the MCEM algorithm can provide an estimate of the asymptotic covariance matrix of the parameters. An example of calculating the asymptotic covariance can be found in Section 8.2.1.

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 `buildMCEM`, `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.

Initial values for the parameters are taken to be the values in the model at the time `buildMCEM` is called, unless the values in the compiled model are changed before running the MCEM.

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 13.3.1.1. I.e., 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 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 the boundary.

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.

The buildMCEM function returns a list with two elements. The first element is a function called run, which will use the MCEM algorithm to estimate the MLEs. The second function is called estimateCov, and is described in Section 8.2.1. The run function 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$run(initM = 1000)
```

```
## Iteration Number: 1.
## Current number of MCMC iterations: 1000.
## Parameter Estimates:
##     alpha      beta
## 0.8219803 1.1492317
## Convergence Criterion: 1.001.
## Iteration Number: 2.
## Current number of MCMC iterations: 1000.
## Parameter Estimates:
##    alpha     beta
## 0.818586 1.237039
## Convergence Criterion: 0.02393763.
## Iteration Number: 3.
## Current number of MCMC iterations: 1000.
## Parameter Estimates:
##     alpha      beta
## 0.8274042 1.2673636
## Convergence Criterion: 0.002330748.
## Iteration Number: 4.
## Current number of MCMC iterations: 3845.
## Parameter Estimates:
##     alpha      beta
## 0.8270797 1.2730042
## Convergence Criterion: 0.0002820885.
```

```
pumpMLE
```

```
##     alpha      beta 
## 0.8270797 1.2730042
```

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, though tightening the convergence criteria may be warranted in actual usage.

8.2.1 Estimating the asymptotic covariance From MCEM

The second element of the list returned by a call to `buildMCEM` is a function called `estimateCov`, which estimates the asymptotic covariance of the parameters at their MLE values. If the `run` function has been called previously, the `estimateCov` function will automatically use the MLE values produced by the `run` function to estimate the covariance. Alternatively, a user can supply their own MLE values using the `MLEs` argument, which allows the covariance to be estimated without having called the `run` function. More details about the `estimateCov` function can be found by calling `help(buildMCEM)`. Below is an example of using the `estimateCov` function.

```
pumpCov <- pumpMCEM$estimateCov()
pumpCov
```

```
##           alpha      beta
## alpha 0.1283888 0.2180972
## beta  0.2180972 0.6377524
```

```
# Alternatively, you can manually specify the MLE values as a named vector.
pumpCov <- pumpMCEM$estimateCov(MLEs = c(alpha = 0.823, beta = 1.261))
```

Chapter 9

Spatial models

NIMBLE supports two variations of conditional autoregressive (CAR) model structures: the improper intrinsic Gaussian CAR (ICAR) model, and a proper Gaussian CAR model. This includes distributions to represent these spatially-dependent model structures in a BUGS model, as well as specialized MCMC samplers for these distributions.

9.1 Intrinsic Gaussian CAR model: *dcar_normal*

The intrinsic Gaussian conditional autoregressive (ICAR) model used to model dependence of block-level values (e.g., spatial areas or temporal blocks) is implemented in NIMBLE as the `dcar_normal` distribution. Additional details for using this distribution are available using `help('CAR-Normal')`.

ICAR models are improper priors for random fields (e.g., temporal or spatial processes). The prior is a joint prior across a collection of latent process values. For more technical details on CAR models, including higher-order CAR models, please see Rue and Held (2005), Banerjee et al. (2015), and Paciorek (2009). Since the distribution is improper it should not be used as the distribution for data values, but rather to specify a prior for an unknown process. As discussed in the references above, the distribution can be seen to be a proper density in a reduced dimension subspace; thus the impropriety only holds on one or more linear combinations of the latent process values.

In addition to our focus here on CAR modeling for spatial data, the ICAR model can also be used in other contexts, such as for temporal data in a discrete time context.

9.1.1 Specification and density

NIMBLE uses the same parameterization as WinBUGS / GeoBUGS for the `dcar_normal` distribution, providing compatibility with existing WinBUGS code. NIMBLE also provides the WinBUGS name `car.normal` as an alias.

9.1.1.1 Specification

The `dcar_normal` distribution is specified for a set of `N` spatially dependent regions as:

```
x[1:N] ~ dcar_normal(adj, weights, num, tau, c, zero_mean)
```

The `adj`, `weights` and `num` parameters define the adjacency structure and associated weights of the spatially-dependent field. See `help('CAR-Normal')` for details of these parameters. When specifying a CAR distribution, these parameters must have constant values. They do not necessarily have to be specified as `constants` when creating a model object using `nimbleModel`, but they should be defined in a static way: as right-hand-side only variables with initial values provided as `constants`, `data` or `inits`, or using fixed numerical deterministic declarations. Each of these two approaches for specifying values are shown in the example.

The adjacency structure defined by `adj` and the associated `weights` must be symmetric. That is, if region i is neighbor of region j, then region j must also be a neighbor of region i. Further, the weights associated with these reciprocating relationships must be equal. NIMBLE performs a check of these symmetries and will issue an error message if asymmetry is detected.

The scalar precision `tau` may be treated as an unknown model parameter and itself assigned a prior distribution. Care should be taken in selecting a prior distribution for `tau`, and WinBUGS suggests that users be prepared to carry out a sensitivity analysis for this choice.

When specifying a higher-order CAR process, the number of constraints `c` can be explicitly provided in the model specification. This would be the case, for example, when specifying a thin-plate spline (second-order) CAR model, for which `c` should be 2 for a one-dimensional process and 3 for a two-dimensional (e.g., spatial) process, as discussed in Rue and Held (2005) and Paciorek (2009). If `c` is omitted, NIMBLE will calculate `c` as the number of disjoint groups of regions in the adjacency structure, which implicitly assumes a first-order CAR process for each group.

By default there is no zero-mean constraint imposed on the CAR process, and thus the mean is implicit within the CAR process values, with an implicit improper flat prior on the mean. To avoid non-identifiability, one should not include an additional parameter for the mean (e.g., do not include an intercept term in a simple CAR model with first-order neighborhood structure). When there are disjoint groups of regions and the constraint is not imposed, there is an implicit distinct improper flat prior on the mean for each group, and it would not make sense to impose the constraint since the constraint holds across all regions. Similarly, if one sets up a neighborhood structure for higher-order CAR models, it would not make sense to impose the zero-mean constraint as that would account for only one of the eigenvalues that are zero. Imposing this constraint (by specifying the parameter `zero_mean = 1`) allows users to model the process mean separately, and hence a separate intercept term should be included in the model.

NIMBLE provides a convenience function `as.carAdjacency` for converting other representations of the adjacency information into the required `adj`, `weights`, `num` format. This function can convert:

- A symmetric adjacency matrix of weights (with diagonal elements equal to zero), using `as.carAdjacency(weightMatrix)`
- Two length-`N` lists with numeric vector elements giving the neighboring indices and associated weights for each region, using `as.carAdjacency(neighborList, weightList)`

These conversions should be done in R, and the resulting `adj`, `weights`, `num` vectors can be passed as `constants` into `nimbleModel`.

9.1.1.2 Density

For process values $x = (x_1, \ldots, x_N)$ and precision τ, the improper CAR density is given as:

$$p(x|\tau) \propto \tau^{(N-c)/2} \; e^{-\frac{\tau}{2}\sum_{i \neq j} w_{ij}\,(x_i - x_j)^2}$$

where the summation over all (i, j) pairs, with the weight between regions i and j given by w_{ij}, is equivalent to summing over all pairs for which region i is a neighbor of region j. Note that the value of c modifies the power to which the precision is raised, accounting for the impropriety of the density based on the number of zero eigenvalues in the implicit precision matrix for x.

For the purposes of MCMC sampling the individual CAR process values, the resulting conditional prior of region i is:

$$p(x_i | x_{-i}, \tau) \sim \mathrm{N}\left(\frac{1}{w_{i+}} \sum_{j \in \mathcal{N}_i} w_{ij}\, x_j,\; w_{i+}\tau\right)$$

where x_{-i} represents all elements of x except x_i, the neighborhood $\mathcal{N}_i$ of region i is the set of all j for which region j is a neighbor of region i, $w_{i+} = \sum_{j \in \mathcal{N}_i} w_{ij}$, and the Normal distribution is parameterized in terms of precision.

9.1.2 Example

Here we provide an example model using the intrinsic Gaussian `dcar_normal` distribution. The CAR process values are used in a spatially-dependent Poisson regression.

To mimic the behavior of WinBUGS, we specify `zero_mean = 1` to enforce a zero-mean constraint on the CAR process, and therefore include a separate intercept term `alpha` in the model. Note that we do not necessarily recommend imposing this constraint, per the discussion earlier in this chapter.

```r
code <- nimbleCode({
    alpha ~ dflat()
    beta ~ dnorm(0, 0.0001)
    tau ~ dgamma(0.001, 0.001)
    s[1:N] ~ dcar_normal(adj[1:L], weights[1:L], num[1:N], tau, zero_mean = 1)
    for(i in 1:N) {
        log(lambda[i]) <- alpha + beta*x[i] + s[i]
        y[i] ~ dpois(lambda[i])
    }
})

L <- 8
constants <- list(N = 4, L = L, num = c(3, 2, 2, 1), weights = rep(1, L),
                  adj = c(2,3,4,1,3,1,2,1), x = c(0, 2, 2, 8))
data <- list(y = c(6, 9, 7, 12))
inits <- list(alpha = 0, beta = 0, tau = 1, s = c(0, 0, 0, 0))
Rmodel <- nimbleModel(code, constants, data, inits)
```

The resulting model may be carried through to MCMC sampling. NIMBLE will assign a specialized sampler to the update the elements of the CAR process. See Chapter 7 for information about NIMBLE's MCMC engine, and Section 9.3 for details on MCMC sampling of the CAR processes.

9.2 Proper Gaussian CAR model: *dcar_proper*

The proper Gaussian conditional autoregressive model used to model dependence of block-level values (e.g., spatial areas or temporal blocks) is implemented in NIMBLE as the `dcar_proper` distribution. Additional details of using this distribution are available using `help('CAR-Proper')`.

Proper CAR models are proper priors for random fields (e.g., temporal or spatial processes). The prior is a joint prior across a collection of latent process values. For more technical details on proper CAR models please see Banerjee et al. (2015), including considerations of why the improper CAR model may be preferred.

In addition to our focus here on CAR modeling for spatial data, the proper CAR model can also be used in other contexts, such as for temporal data in a discrete time context.

9.2.1 Specification and density

NIMBLE uses the same parameterization as WinBUGS / GeoBUGS for the `dcar_proper` distribution, providing compatibility with existing WinBUGS code. NIMBLE also provides the WinBUGS name `car.proper` as an alias.

9.2.1.1 Specification

The `dcar_proper` distribution is specified for a set of `N` spatially dependent regions as:

```
x[1:N] ~ dcar_proper(mu, C, adj, num, M, tau, gamma)
```

There is no option of a zero-mean constraint for proper CAR process, and instead the mean for each region is specified by the `mu` parameter. The elements of `mu` can be assigned fixed values or may be specified using one common, or multiple, prior distributions.

The `C`, `adj`, `num` and `M` parameters define the adjacency structure, normalized weights, and conditional variances of the spatially-dependent field. See `help('CAR-Proper')` for details of these parameters. When specifying a CAR distribution, these parameters must have constant values. They do not necessarily have to be specified as `constants` when creating a model object using `nimbleModel`, but they should be defined in a static way: as right-hand-side only variables with initial values provided as `constants`, `data` or `inits`, or using fixed numerical deterministic declarations.

The adjacency structure defined by `adj` must be symmetric. That is, if region i is neighbor of region j, then region j must also be a neighbor of region i. In addition, the normalized weights specified in `C` must satisfy a symmetry constraint jointly with the conditional variances given in `M`. This constraint requires that $M^{-1}C$ is symmetric, where M is a diagonal matrix of conditional variances and C is the normalized (each row sums to one) weight matrix. Equivalently, this implies that $C_{ij}M_{jj} = C_{ji}M_{ii}$ for all pairs of neighboring regions i and j. NIMBLE performs a check of these symmetries and will issue an error message if asymmetry is detected.

Two options are available to simplify the process of constructing the `C` and `M` arguments; both options are demonstrated in the example. First, these arguments may be omitted from the `dcar_proper` specification. In this case, values of `C` and `M` will be generated that correspond to all weights being equal to one, or equivalently, a symmetric weight matrix containing only zeros and ones. Note that `C` and `M` should either *both* be provided, or *both* be omitted from the specification.

Second, a convenience function as.carCM is provided to generate the C and M arguments corresponding to a specified set of symmetric unnormalized weights. If weights contains the non-zero weights corresponding to an unnormalized weight matrix (weights is precisely the argument that can be used in the dcar_normal specification), then a list containing C and M can be generated using as.carCM(adj, weights, num). In this case, the resulting C contains the row-normalized weights, and the resulting M is a vector of the inverse row-sums of the unnormalized weight matrix.

The scalar precision tau may be treated as an unknown model parameter and itself assigned a prior distribution. Care should be taken in selecting a prior distribution for tau, and WinBUGS suggests that users be prepared to carry out a sensitivity analysis for this choice.

An appropriate value of the gamma parameter ensures the propriety of the dcar_proper distribution. The value of gamma must lie between fixed bounds, which are given by the reciprocals of the largest and smallest eigenvalues of $M^{-1/2}CM^{1/2}$. These bounds can be calculated using the function carBounds or separately using the functions carMinBound and carMaxBound. For compatibility with WinBUGS, NIMBLE provides min.bound and max.bound as aliases for carMinBound and carMaxBound. Rather than selecting a fixed value of gamma within these bounds, it is recommended that gamma be assigned a uniform prior distribution over the region of permissible values.

Note that when C and M are omitted from the dcar_proper specification (and hence all weights are taken as one), or C and M are calculated from a symmetric weight matrix using the utility function as.carCM, then the bounds on gamma are necessarily $(-1, 1)$. In this case, gamma can simply be assigned a prior over that region. This approach is shown in both examples.

9.2.1.2 Density

The proper CAR density is given as:

$$p(x|\mu, C, M, \tau, \gamma) \sim \mathrm{MVN}\left(\mu,\ \tfrac{1}{\tau}(I - \gamma C)^{-1} M\right)$$

where the multivariate normal distribution is parameterized in terms of covariance.

For the purposes of MCMC sampling the individual CAR process values, the resulting conditional prior of region i is:

$$p(x_i|x_{-i}, \mu, C, M, \tau, \gamma) \sim \mathrm{N}\left(\mu_i + \textstyle\sum_{j \in \mathcal{N}_i} \gamma\, C_{ij}\,(x_j - \mu_j),\ \tfrac{M_{ii}}{\tau}\right)$$

where x_{-i} represents all elements of x except x_i, the neighborhood $\mathcal{N}_i$ of region i is the set of all j for which region j is a neighbor of region i, and the Normal distribution is parameterized in terms of variance.

9.2.2 Example

We provide two example models using the proper Gaussian dcar_proper distribution. In both, the CAR process values are used in a spatially-dependent logistic regression to model binary presence/absence data. In the first example, the C and M parameters are omitted, which uses weights equal to one for all neighbor relationships. In the second example, symmetric unnormalized weights are specified, and as.carCM is used to construct the C and M parameters to the dcar_proper distribution.

```
# omitting C and M sets all non-zero weights to one
code <- nimbleCode({
    mu0 ~ dnorm(0, 0.0001)
    tau ~ dgamma(0.001, 0.001)
    gamma ~ dunif(-1, 1)
    s[1:N] ~ dcar_proper(mu[1:N], adj=adj[1:L], num=num[1:N], tau=tau,
                         gamma=gamma)
    for(i in 1:N) {
        mu[i] <- mu0
        logit(p[i]) <- s[i]
        y[i] ~ dbern(p[i])
    }
})

adj <- c(2, 1, 3, 2, 4, 3)
num <- c(1, 2, 2, 1)
constants <- list(adj = adj, num = num, N = 4, L = 6)
data <- list(y = c(1, 0, 1, 1))
inits <- list(mu0 = 0, tau = 1, gamma = 0, s = rep(0, 4))
Rmodel <- nimbleModel(code, constants, data, inits)

# specify symmetric unnormalized weights, use as.carCM to generate C and M
code <- nimbleCode({
    mu0 ~ dnorm(0, 0.0001)
    tau ~ dgamma(0.001, 0.001)
    gamma ~ dunif(-1, 1)
    s[1:N] ~ dcar_proper(mu[1:N], C[1:L], adj[1:L], num[1:N], M[1:N], tau,
                         gamma)
    for(i in 1:N) {
        mu[i] <- mu0
        logit(p[i]) <- s[i]
        y[i] ~ dbern(p[i])
    }
})

weights <- c(2, 2, 3, 3, 4, 4)
CM <- as.carCM(adj, weights, num)
constants <- list(C = CM$C, adj = adj, num = num, M = CM$M, N = 4, L = 6)
Rmodel <- nimbleModel(code, constants, data, inits)
```

Each of the resulting models may be carried through to MCMC sampling. NIMBLE will assign a specialized sampler to update the elements of the CAR process. See Chapter 7 for information about NIMBLE's MCMC engine, and Section 9.3 for details on MCMC sampling of the CAR processes.

9.3 MCMC Sampling of CAR models

NIMBLE's MCMC engine provides specialized samplers for the dcar_normal and dcar_proper distributions. These samplers perform sequential univariate updates on the components of the CAR process. Internally, each sampler assigns one of three specialized univariate samplers to each component, based on inspection of the model structure:

1. A conjugate sampler in the case of conjugate Normal dependencies.
2. A random walk Metropolis-Hastings sampler in the case of non-conjugate dependencies.
3. A posterior predictive sampler in the case of no dependencies.

Note that these univariate CAR samplers are not the same as NIMBLE's standard conjugate, RW, and posterior_predictive samplers, but rather specialized versions for operating on a CAR distribution. Details of these assignments are strictly internal to the CAR samplers.

In future versions of NIMBLE we expect to provide block samplers that update the entire CAR process as a single sample. This may provide improved MCMC performance by accounting for dependence between elements, particularly when conjugacy is available.

9.3.1 Initial values

Valid initial values should be provided for all elements of the process specified by a CAR structure before running an MCMC. This ensures that the conditional prior distribution is well-defined for each region. A simple and safe choice of initial values is setting all components of the process equal to zero, as is done in the preceding CAR examples.

For compatibility with WinBUGS, NIMBLE also allows an initial value of NA to be provided for regions with zero neighbors. This particular initialization is required in WinBUGS, so this allows users to make use of existing WinBUGS code.

9.3.2 Zero-neighbor regions

Regions with zero neighbors (defined by a 0 appearing in the num parameter) are a special case for both the dcar_normal and dcar_proper distribution. The corresponding neighborhood $\mathcal{N}$ of such a region contains no elements, and hence the conditional prior is improper and uninformative, tantamount to a dflat prior distribution. Thus, the conditional posterior distribution of those regions is entirely determined by the dependent nodes, if any. Sampling of these zero-neighbor regions proceeds as:

1. In the conjugate case, sampling proceeds according to the conjugate posterior.
2. In the non-conjugate case, sampling proceeds using random walk Metropolis-Hastings, where the posterior is determined entirely by the dependencies.
3. In the case of no dependents, the posterior is entirely undefined. Here, no changes will be made to the process value, and it will remain equal to its initial value throughout. By virtue of having no neighbors, this region does not contribute to the density evaluation of the subsuming dcar_normal node nor to the conditional prior of any other regions, hence its value (even NA) is of no consequence.

This behavior is different from that of WinBUGS, where the value of zero-neighbor regions of car.normal nodes is set to and fixed at zero.

Also note that for the dcar_proper distribution if any regions have zero neighbors the joint density of the process cannot be calculated. As a result one cannot do MCMC sampling of unknown parameters affecting the mean (mu) of the process, though one can instead use an uncentered parameterization in which the mean is added to the process rather than the process being centered on the mean. Or one could remove such regions from the process and model them separately.

9.3.3 Zero-mean constraint

A zero-mean constraint is available for the intrinsic Gaussian dcar_normal distribution. This constraint on the ICAR process values is imposed during MCMC sampling, if the argument zero_mean = 1, mimicking the behavior of WinBUGS. Following the univariate updates on each component, the mean is subtracted away from all process values, resulting in a zero-mean process.

Note that this is *not* equivalent to sampling under the constraint that the mean is zero (see p. 36 of Rue and Held (2005)) so should be treated as an ad hoc approach and employed with caution.

Chapter 10

Bayesian nonparametric models

10.1 Bayesian nonparametric mixture models

NIMBLE provides support for Bayesian nonparametric (BNP) mixture modeling. The current implementation provides support for hierarchical specifications involving Dirichlet process (DP) mixtures (Ferguson, 1973, 1974; Lo, 1984; Escobar, 1994; Escobar and West, 1995). More specifically, a DP mixture model takes the form

$$y_i \mid G \overset{iid}{\sim} \int h(y_i \mid \theta) G(d\theta),$$

$$G \mid \alpha, G_0 \sim DP(\alpha, G_0),$$

where $h(\cdot \mid \theta)$ is a suitable kernel with parameter θ, and α and G_0 are the concentration and baseline distribution parameters of the DP, respectively. DP mixture models can be written with different levels of hierarchy, all being equivalent to the model above.

When the random measure G is integrated out from the model, the DP mixture model can be written using latent or membership variables, z_i, following a Chinese Restaurant Process (CRP) distribution (Blackwell and MacQueen, 1973), discussed in Section 10.2. The model takes the form

$$y_i \mid \tilde{\theta}, z_i \overset{ind}{\sim} h(\cdot \mid \tilde{\theta}_{z_i}),$$

$$z \mid \alpha \sim \text{CRP}(\alpha), \qquad \tilde{\theta}_j \overset{iid}{\sim} G_0,$$

where $\text{CRP}(\alpha)$ denotes the CRP distribution with concentration parameter α.

If a stick-breaking representation (Sethuraman, 1994), discussed in section 10.3, is assumed for the random measure G, then the model takes the form

$$y_i \mid \theta^{\star}, v \overset{ind}{\sim} \sum_{l=1}^{\infty} \left\{ v_l \prod_{m<l} (1 - v_m) \right\} h(\cdot \mid \theta_l^{\star}),$$

$$v_l \mid \alpha \overset{iid}{\sim} Beta(1, \alpha), \qquad \theta_l^{\star} \overset{iid}{\sim} G_0.$$

More general representations of the random measure can be specify by considering $v_l \mid \nu_l, \alpha_l \overset{ind}{\sim} Beta(\nu_l, \alpha_l)$. Finite dimensional approximations can be obtained by truncating the infinite sum to have L components.

Different representations of DP mixtures lead to different computational algorithms. NIMBLE supports sampling algorithms based on the CRP representation, as well as on the stick-breaking representation. NIMBLE includes definitions of structures required to implement the CRP and stick-breaking distributions, and the associated MCMC algorithms.

10.2 Chinese Restaurant Process model

The CRP is a distribution over the space of partitions of positive integers and is implemented in NIMBLE as the `dCRP` distribution. More details for using this distribution are available using `help(CRP)`.

The CRP can be described as a stochastic process in which customers arrive at a restaurant, potentially with an infinite number of tables. Each customer sits at an empty or occupied table according to probabilities that depend on the number of customers in the occupied tables. Thus, the CRP partitions the set of customers, through their assignment to tables in the restaurant.

10.2.1 Specification and density

NIMBLE parametrizes the `dCRP` distribution by a concentration parameter and a size parameter.

10.2.1.1 Specification

The `dCRP` distribution is specified in NIMBLE for a membership vector `z` as

```
z[1:N] ~ dCRP(conc, size)
```

The `conc` parameter is the concentration parameter of the CRP, controlling the probability of a customer sitting on a new table, i.e., creating a new cluster. The `size` parameter defines the size of the set of integers to be partitioned.

The `conc` parameter is a positive real value that can be treated as known or unknown. When a gamma prior is assumed for the `conc` parameter, a specialized sampler is assigned. See more on this in section 10.4.1.

The `size` parameter is a positive integer that has to be fixed and equal to the length of vector `z`. It defines the set of consecutive integers from `1` to `N` to be partitioned. Each element in `z` can be an integer from `1` to `N`, and repetitions are allowed.

10.2.1.2 Density

The CRP distribution partitions the set of positive integers $1, \ldots, N$, into $N^\star \leq N$ disjoint subsets, indicating to which subset each element belongs. For instance, if $N = 6$, the set $\{1, 2, 3, 4, 5, 6\}$ can be partitioned into the subsets $S_1 = \{1, 2, 6\}$, $S_2 = \{4, 5\}$, and $S_3 = \{3\}$. Note that $N^\star = 3$, and this is one partition from out of 203 possibilities. The CRP-distributed vector z encodes this partition and its observed values would be $(1, 1, 3, 2, 2, 1)$, for this example. In mixture modeling, this indicates that observations 1, 2, and 6 belong to cluster 1, observations 4 and 5 to cluster 2, and observation 3 to cluster 3. Note that this representation is not unique, vector $(2, 2, 1, 3, 3, 2)$ encodes the same partition.

The joint probability function of $z = (z_1, \dots, z_N)$, with concentration parameter α, is given by

$$p(z \mid \alpha) \propto \frac{\Gamma(\alpha)}{\Gamma(\alpha + n)} \alpha^{N^\star(z)} \prod_{k=1}^{N^\star(z)} \Gamma(m_k(z)),$$

where $m_k(z)$ denotes the number of elements in z that are equal to k, $N^\star(z)$ denotes the number of unique elements in z, and $\Gamma(\cdot)$ denotes the gamma function. The full conditional distribution for z_i given z_{-i} is

$$p(z_i = m \mid z_{-i}, \alpha) = \frac{1}{n - 1 + \alpha} \sum_{j \neq i} 1_{\{z_j\}}(m) + \frac{\alpha}{n - 1 + \alpha} 1_{\{z^{new}\}}(m),$$

where z_{-i} denotes vector z after removing its i—th component, z^{new} is a value not in z_{-i}, and 1_A denotes the indicator function at set A.

Note that the probability of creating a new cluster is proportional to α: the larger the concentration parameter, the more clusters are created.

10.2.2 Example

The following example illustrates how to use NIMBLE to perform single density estimation for real-valued data, under a BNP approach, using the `dCRP` distribution. (Note that the BNP approach is also often used to perform density estimation on random effects.) The model is given by

$$\begin{aligned} y_i \mid \tilde{\theta}, \tilde{\sigma}^2, z_i &\overset{ind}{\sim} N(\tilde{\theta}_{z_i}, \tilde{\sigma}^2_{z_i}), \quad i = 1, \dots, N, \\ z \sim \mathrm{CRP}(\alpha), &\quad \alpha \sim \mathrm{Gamma}(1, 1), \\ \tilde{\theta}_j \overset{iid}{\sim} N(0, 100), &\quad \tilde{\sigma}^2_j \overset{iid}{\sim} \mathrm{InvGamma}(1, 1), \;\; j = 1, \dots, M. \end{aligned}$$

```r
code <- nimbleCode({
  z[1:N] ~ dCRP(alpha, size = N)
  alpha ~ dgamma(1, 1)
  for(i in 1:M) {
    thetatilde[i] ~ dnorm(0, var = 100)
    s2tilde[i] ~ dinvgamma(1, 1)
  }
  for(i in 1:N)
    y[i] ~ dnorm(thetatilde[z[i]], var = s2tilde[z[i]])
})

set.seed(1)
constants <- list(N = 100, M = 50)
data <- list(y = c(rnorm(50, -5, sqrt(3)), rnorm(50, 5, sqrt(4))))
inits <- list(thetatilde = rnorm(constants$M, 0, 10),
              s2tilde = rinvgamma(constants$M, 1, 1),
              z = sample(1:10, size = constants$N, replace = TRUE),
              alpha  = 1)
model <- nimbleModel(code, constants, data, inits)
```

The model can be fitted through MCMC sampling. NIMBLE will assign a specialized sampler to update `z` and `alpha`. See Chapter 7 for information about NIMBLE's MCMC engine, and Section 10.4.1 for details on MCMC sampling of the CRP.

One of the advantages of BNP mixture models is that the number of clusters is treated as random. Therefore, in MCMC sampling, the number of cluster parameters varies with the iteration. Since NIMBLE does not currently allow dynamic length allocation, the number of unique cluster parameters, $N^\star$, has to be fixed. One safe option is to set this number to N, but this is inefficient, both in terms of computation and in terms of storage, because in practice it is often that $N^\star < N$. In addition, configuring and building the MCMC can be slow (and use a lot of memory) for large N. In an effort to mitigate these inefficiencies, we allow the user to set $N^\star = M$, with $M < N$, as seen in the example above. However, if this number is too small and is exceeded in any iteration a warning is issued.

10.2.3 Extensions

The BNP functionality in NIMBLE was extended in version 0.10.0 to more general models. These extensions enable users, for instance, to use a DP or DPM prior for the distribution of the random effects in a generalized linear mixed effects model with mutiple measurements over time or multiple trials per participant.

The following example illustrates how to use NIMBLE in a random effects model with repeated measurements per subject using a DP prior for the distribution of the subject's random effects. The model is given by

$$y_{i,j} \mid \tilde{\theta}, z_i, \sigma^2 \sim N(\tilde{\theta}_{z_i}, \sigma^2), \qquad i = 1, \dots, N, \qquad j = 1, \dots, J,$$

$$z \sim \text{CRP}(\alpha), \qquad \alpha \sim \text{Gamma}(1, 1),$$

$$\tilde{\theta}_m \overset{iid}{\sim} N(0, 100), \;\; m = 1, \dots, M, \qquad \sigma^2 \sim \text{InvGamma}(1, 1).$$

The corresponding NIMBLE code is

```
code <- nimbleCode({
  z[1:N] ~ dCRP(alpha, size = N)
  alpha ~ dgamma(1, 1)
  sigma2 ~ dinvgamma(1, 1)
  for(i in 1:M) {
    thetatilde[i] ~ dnorm(0, var = 100)
  }
  for(i in 1:N) {
    for(j in 1:J) {
      y[i, j] ~ dnorm(thetatilde[z[i]], var = sigma2)
    }
  }
})

set.seed(1)
constants <- list(N = 10, J = 5, M = 5)
data <- list(y = matrix(c(rnorm(25, -25, 1), rnorm(25, 25, 1)), ncol=constants$J,
                        nrow=constants$N, byrow=TRUE))
inits <- list(thetatilde = rnorm(constants$M, 0, 10),
              z = sample(1:5, size = constants$N, replace = TRUE),
              alpha  = 1,
              sigma2 = 1)
modelRandEff <- nimbleModel(code, constants, data, inits)
```

Alternatively, each subject could have a vector of parameters being clustered. For example in the model above one could instead specify a vector of means, such as `thetaTilde[z[i], j]`, instead of a single mean. This allows group-specific parameters to also vary across the repeated mesasurements.

As before, the model can be fitted through MCMC sampling. NIMBLE will assign a specialized sampler to update `z` and `alpha`. See Chapter 7 for information about NIMBLE's MCMC engine, and Section 10.4.1 for details on MCMC sampling of the CRP.

10.3 Stick-breaking model

In NIMBLE, weights defined by sequentially breaking a stick, as in the stick-breaking process, are implemented as the `stick_breaking` link function. More details for using this function are available using `help(stick_breaking)`.

10.3.1 Specification and function

NIMBLE parametrizes the `stick_breaking` function by vector of values in $(0, 1)$.

10.3.1.1 Function

The weights $(w_1, \dots, w_L)$ follow a finite stick-breaking construction if

$$
\begin{aligned}
w_1 &= v_1, \\
w_l &= v_l \prod_{m<l} (1 - v_m), l = 2, \dots, L-1 \\
w_L &= \prod_{m<L} (1 - v_m).
\end{aligned}
$$

for $v_l \in [0, 1], l = 1, \dots, L-1$.

10.3.1.2 Specification

The `stick_breaking` function is specified in NIMBLE for a vector `w` of probabilities as

`w[1:L] <- stick_breaking(v[1:(L-1)])`

The argument `v` is a vector of values between 0 and 1 defining the sequential breaking points of the stick after removing the previous portions already broken off. It is of length $L-1$, implicitly assuming that its last component is equal to 1.

In order to complete the definition of the weights in the stick-breaking representation of G, a prior distribution on $(0,1)$ should to be assumed for v_l, $l = 1, \ldots, L-1$, for instance a beta prior.

10.3.2 Example

Here we illustrate how to use NIMBLE for the example described in section 10.2.2, but considering a stick-breaking representation for G. The model is given by

$$
\begin{gathered}
y_i \mid \theta^{\star}, {\sigma^{\star}}^{2}, z_i \overset{ind}{\sim} N({\theta^{\star}}_{z_i}, {\sigma^{2}}^{\star}_{z_i}), \qquad i = 1, \ldots, N, \\
z_i \sim \text{Categorical}(w), \qquad i = 1, \ldots, N, \\
v_l \overset{iid}{\sim} \text{Beta}(1, \alpha), \quad l = 1, \ldots, L-1, \\
\alpha \sim \text{Gamma}(1, 1), \\
\theta^{\star}_l \overset{iid}{\sim} N(0, 100), \qquad {\sigma^{2}}^{\star}_l \overset{iid}{\sim} \text{InvGamma}(1, 1), \quad l = 1, \ldots, L.
\end{gathered}
$$

where $w_1 = v_1$, $w_l = v_l \prod_{m<l}(1 - v_m)$, for $l = 1, \ldots, L-1$, and $w_L = \prod_{m<L}(1 - v_m)$.

```
code <- nimbleCode({
  for(i in 1:(L-1)){
    v[i] ~ dbeta(1, alpha)
  }
  alpha ~ dgamma(1, 1)
  w[1:L] <- stick_breaking(v[1:(L-1)])
  for(i in 1:L) {
    thetastar[i] ~ dnorm(0, var = 100)
    s2star[i] ~ dinvgamma(1, 1)
  }
  for(i in 1:N) {
    z[i] ~ dcat(w[1:L])
    y[i] ~ dnorm(thetastar[z[i]], var = s2star[z[i]])
  }
})

set.seed(1)
constants <- list(N = 100, L=50)
data <- list(y = c(rnorm(50, -5, sqrt(3)), rnorm(50, 5, sqrt(4))))
inits <- list(thetastar = rnorm(constants$L, 0, 100),
              s2star = rinvgamma(constants$L, 1, 1),
              z = sample(1:10, size = constants$N, replace = TRUE),
              v  = rbeta(constants$L, 1, 1),
              alpha = 1)
modelSB <- nimbleModel(code, constants, data, inits)
```

The resulting model may be carried through to MCMC sampling. NIMBLE will assign a specialized sampler to update `v`. See Chapter 7 for information about NIMBLE's MCMC engine, and Section 10.4.2 for details on MCMC sampling of the stick-breaking weights.

10.4 MCMC sampling of BNP models

BNP models can be specified in different, yet equivalent, manners. Examples 10.2.2 and 10.3.2 are examples of density estimation for real-valued data, and are specified through the CRP and the stick-breaking process, respectively. Different specifications lead NIMBLE to assign different sampling algorithms for the model. When the model is specified through a CRP, a collapsed sampler (Neal, 2000) is assigned. Under this specification, the random measure G is integrated out from the model. When a stick-breaking representation is used, a blocked Gibbs sampler is assigned, see Ishwaran and James (2001) and Ishwaran and James (2002).

10.4.1 Sampling CRP models

NIMBLE's MCMC engine provides specialized samplers for the `dCRP` distribution, updating each component of the membership vector sequentially. Internally, the sampler is assigned based on inspection of the model structure, evaluating conjugacy between the mixture kernel and the baseline distribution, as follows:

1. A conjugate sampler in the case of the baseline distribution being conjugate for the mixture kernel.
2. A non-conjugate sampler in the case of the baseline distribution not being conjugate for the mixture kernel.

Note that both samplers are specialized versions that operate on a vector having a CRP distribution. Details of these assignments are strictly internal to the CRP samplers. The current release of NIMBLE supports conjugate sampling for the `dCRP` distribution for the relationships listed in Table 10.1. Additional relationships are provided in Table 10.2 for normal mixture kernels when both mean and variance are unknown.

Table 10.1: Conjugate relationships for the `dCRP` distribution supported by NIMBLE's MCMC engine.

Baseline Distribution	Mixture (Dependent Node) Distribution	Parameter
Beta	Bernoulli	`prob`
	Binomial	`prob`
	Negative Binomial	`prob`
Dirichlet	Multinomial	`prob`
Gamma	Poisson	`lambda`
	Normal	`tau`
	Gamma	`rate`
	Inverse Gamma	`scale`

Baseline Distribution	Mixture (Dependent Node) Distribution	Parameter
	Exponential	`rate`
	Weibull	`lambda`
Inverse Gamma	Normal	`var`
Normal	Normal	`mean`
Multivariate Normal	Multivariate Normal	`mean`
Wishart	Multivariate Normal	`prec`
Inverse Wishart	Multivariate Normal	`cov`

Table 10.2: Additional conjugate relationships for the `dCRP` distribution supported by NIMBLE's MCMC engine.

Baseline Distribution	Mixture (Dependent Node) Distribution	Parameter
Normal-Gamma	Normal	`mean`, `tau`
Normal-Inverse Gamma	Normal	`mean`, `var`
Multivariate Normal-Wishart	Multivariate Normal	`mean`, `prec`
Multivariate Normal-Inverse Wishart	Multivariate Normal	`mean`, `cov`

To reduce computation and improve mixing, we only sample the parameters of the clusters (e.g., $\tilde{\theta}_j$ in 10.2.2 when the associated cluster is occupied, using the `CRP_cluster_wrapper` sampler, which wraps around an underlying actual sampler. In addition, this approach requires that any sampler assigned to parameters of the base measure, G_0, (i.e., unknown parameters in the prior for $\tilde{\theta}_j$) ignore cluster parameters associated with clusters that are not occupied, since their current values are meaningless. We assign a special slice sampler that determines the occupied clusters in any given iteration, called `slice_CRP_base_param`. Note that if you choose to use a different sampler for the base measure parameters, you should also avoid using the `CRP_cluster_wrapper` sampler.

Finally, a specialized sampler is assigned to the `conc` hyper parameter when a gamma hyper prior is assigned, see section 6 in Escobar and West (1995) for more details. Otherwise, a random walk Metropolis-Hastings sampler is assigned.

10.4.1.1 Initial values

Valid initial values should be provided for all elements of the process specified by a CRP structure before running the MCMC. A simple and safe choice for `z` is to provide a sample of size `N`, the same as its length, of values between 1 and some reasonable number of clusters (less than or equal to the length of `z`), with replacement, as done in the preceding CRP example. For the concentration parameter, a safe initial value is 1.

10.4.1.2 Sampling the random measure

In BNP models, it is oftenly of interest to make inference about the unknown measure G. NIMBLE provides the `getSamplesDPmeasure` sampler to generate posterior samples of the random measure G when a CRP structure is involved in the model.

The `getSamplesDPmeasure` function has two arguments: `MCMC` and `epsilon`.

The `MCMC` argument is a compiled or uncompiled MCMC object. The MCMC object should monitor the membership (or clustering) variable, the cluster parameters, all stochastic nodes of the cluster parameters, and the concentration parameter, if it is random. Use the `monitors` argument when configuring the MCMC to ensure these variables are monitored.

The `epsilon` argument is used to determine the truncation level of the representation of the random measure. Its default value is $1e-04$.

The sampler is used only after the MCMC for the model has been run; more details are available from `help(getsamplesDPmeasure)`.

Posterior samples of the random measure G are (almost surely) of the form $\sum_{l=1}^{\infty} w_l \delta_{\theta_l}$, where δ_θ is the Dirac measure at θ, w_l are stick-breaking weights, and θ_l are atoms (or point masses). The variables that define the stick-breaking weights are iid $Beta(1, \alpha + N)$ distributed, where α is the concentration parameter of the CRP distribution and N is the sample size. Independently, the atoms, θ_l, are iid and follow a distribution of the form

$$(\alpha + N)^{-1} \sum_{k=1}^{N^\star(z)} m_k(z) \delta_{\{\tilde{\theta}_k\}} + \alpha(\alpha + N)^{-1} G_0,$$

where G_0 is the prior baseline distribution, and z and $\tilde{\theta}_k$ are posterior samples of the labeling vector and cluster parameters, respectively. Their values, together with the values of α (if random) are obtained from the MCMC's output. Expressions $m_k(z)$ and $N^\star(z)$ are defined as in section 10.2.

The `getsamplesDPmeasure` function provides samples of a truncated version of the infinite mixture to a level L. The truncation level L is such that the tail probability left from the approximation is at most `epsilon`, denoted ϵ. The following relationship determines the truncation level: $L = \log(\epsilon)/\log[(\alpha + N)/(\alpha + N + 1)]$. The value of L varies at each iteration of the MCMC's output when α is random, while it is the same at each iteration when α is fixed. For more details about sampling the random measure and its truncation level see Gelfand and Kottas (2002).

Because of the discrete nature of the atom's distribution, the stick-breaking representation truncated to a level L will generally have repeated values. Therefore, in order to reduce the output's dimensionality, the stick-breaking weights of identical atoms are added up. This results in samples of G that have only the 'unique' atoms in $\{\theta_l\}_{l=1}^L$.

The following code exemplifies how to generate samples from G after defining the model as in Section 10.2.2.

```r
cmodel <- compileNimble(model)

monitors <- c('z', 'thetatilde', 's2tilde' , 'alpha')
modelConf <- configureMCMC(model, monitors = monitors)
modelMCMC <- buildMCMC(modelConf)

cmodelMCMC <- compileNimble(modelMCMC, project = model)
cmodelMCMC$run(1000)

samplesG <- getSamplesDPmeasure(cmodelMCMC)
```

10.4.2 Sampling stick-breaking models

NIMBLE's MCMC engine provides specialized samplers for the beta-distributed random variables that are the arguments to the stick-breaking function, updating each component of the weight vector sequentially. The sampler is assigned based on inspection of the model structure. Specifically, the specialized sampler is assigned when the membership vector has a categorical distribution, its weights are defined by a stick-breaking function, and the vector defining the weights follows a beta distribution.

10.4.2.1 Initial values

Valid initial values should be provided for all elements of the stick-breaking function and membership variable before running the MCMC. A simple and safe choice for z is to provide a sample of size N, of values between 1 and some value less than L, with replacement, as done in the preceding stick-breaking example. For the stick variables, safe initial values can be simulated from a beta distribution.

Part IV

Programming with NIMBLE

Overview

Part IV is the programmer's guide to NIMBLE. At the heart of programming in NIMBLE are nimbleFunctions. These support two principal features: (1) a *setup* function that is run once for each model, nodes, or other setup arguments, and (2) *run* functions that will be compiled to C++ and are written in a subset of R enhanced with features to operate models. Formally, what can be compiled comprises the NIMBLE language, which is designed to be R-like.

This part of the manual is organized as follows:

- Chapter 11 describes how to write simple nimbleFunctions, which have no setup code and hence don't interact with models, to compile parts of R for fast calculations. This covers the subset of R that is compilable, how to declare argument types and return types, and other information.
- Chapter 12 explains how to write nimbleFunctions that can be included in BUGS code as user-defined distributions or user-defined functions.
- Chapter 13 introduces more features of NIMBLE models that are useful for writing nimbleFunctions to use models, focusing on how to query model structure and carry out model calculations.

- Chapter 14 introduces two kinds of data structures: `modelValues` are used for holding multiple sets of values of model variables; `nimbleList` data structures are similar to R lists but require fixed element names and types, allowing the NIMBLE compiler to use them.
- Chapter 15 draws on the previous chapters to show how to write nimbleFunctions that work with models, or more generally that have a setup function for any purpose. Typically a setup function queries model structure (Chapter 13) and may establish some `modelValues` or `nimbleList` data structures or configurations (Chapter 14). Then *run* functions written in the same way as simple nimbleFunctions (Chapter 11) along with model operations (Chapter 13) define algorithm computations that can be compiled via C++.

Chapter 11

Writing simple nimbleFunctions

11.1 Introduction to simple nimbleFunctions

nimbleFunctions are the heart of programming in NIMBLE. In this chapter, we introduce simple nimbleFunctions that contain only one function to be executed, in either compiled or uncompiled form, but no setup function or additional methods.

Defining a simple nimbleFunction is like defining an R function: `nimbleFunction` returns a function that can be executed, and it can also be compiled. Simple nimbleFunctions are useful for doing math or the other kinds of processing available in NIMBLE when no model or modelValues is needed. These can be used for any purpose in R programming. They can also be used as new functions and distributions in NIMBLE's extension of BUGS (Chapter 12).

Here's a basic example implementing the textbook calculation of least squares estimation of linear regression parameters[1]:

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

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

```
##              [,1]
## [1,]  0.088547575
## [2,]  0.002025635
## [3,] -0.090387283
## [4,] -0.072660391
```

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

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

```
##              [,1]
## [1,]  0.088547575
## [2,]  0.002025635
## [3,] -0.090387283
## [4,] -0.072660391
```

In this example, we fit a linear model for 100 random response values (`y`) to four columns of randomly generated explanatory variables (`X`). We ran the nimbleFunction `solveLeastSquares` uncompiled, natively in R, allowing testing and debugging (Section 15.7). Then we compiled it and showed that the compiled version does the same thing, but faster[2]. NIMBLE's compiler creates C++ that uses the Eigen (http://eigen.tuxfamily.org) library for linear algebra.

Notice that the actual NIMBLE code is written as an R function definition that is passed to `nimbleFunction` as the `run` argument. Hence we call it the *run* code. run code is written in the NIMBLE language. This is similar to a narrow subset of R with some additional features. Formally, we view it as a distinct language that encompasses what can be compiled from a nimbleFunction.

To write nimbleFunctions, you will need to learn:

- what R functions are supported for NIMBLE compilation and any ways they differ from their regular R counterparts;
- how NIMBLE handles types of variables;
- how to declare types of `nimbleFunction` arguments and return values;
- that compiled nimbleFunctions always pass arguments to each other by reference.

The next sections cover each of these topics in turn.

11.2 R functions (or variants) implemented in NIMBLE

11.2.1 Finding help for NIMBLE's versions of R functions

Often, R help pages are available for NIMBLE's versions of R functions using the prefix 'nim' and capitalizing the next letter. For example, help on NIMBLE's version of `numeric` can be found by `help(nimNumeric)`. In some cases help is found directly using the name of the function as it appears in R.

11.2.2 Basic operations

Basic R operations supported for NIMBLE compilation are listed in Table 11.1.

[2] 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.

Table 11.1: Basic R manipulation functions in NIMBLE. For some of these functions, in addition to the usual use of `help()`, you can find help in R for NIMBLE's version of a function by using the "nim" prefix and capitalizing the next letter. E.g. `help(nimC)` for help with `c()`.

Function	Comments (differences from R)
`c()`	No `recursive` argument.
`rep()`	No `rep.int` or `rep_len` arguments.
`seq()` and ':'	Negative integer sequences from ':', e.g. , `2:1` do not work.
`which()`	No `arr.ind` or `useNames` arguments.
`diag()`	Works like R in three ways: `diag(vector)` returns a matrix with `vector` on the diagonal; `diag(matrix)` returns the diagonal vector of `matrix`; `diag(n)` returns an $n \times n$ identity matrix. No `nrow` or `ncol` arguments.
`diag()<-`	Works for assigning the diagonal vector of a matrix.
`dim()`	Works on a vector as well as higher-dimensional arguments.
`length()`	
`is.na()`	Does not correctly handle NAs from R that are type `'logical'`, so convert these using `as.numeric()` before passing from R to NIMBLE.
`is.nan()`	Treats `NA` of type 'double' as being `NaN` and `NA` of type 'logical' as not being `NaN`.
`any()`	One argument only; NAs treated as FALSE.
`all()`	One argument only; NAs treated as FALSE.
`numeric()`	Allows additional arguments to control initialization.
`logical()`	Allows additional arguments to control initialization.
`integer()`	Allows additional arguments to control initialization.
`matrix()`	Allows additional arguments to control initialization.
`array()`	Allows additional arguments to control initialization.
indexing	Arbitrary integer and logical indexing is supported for objects of one or two dimensions. For higher-dimensional objects, only `:` indexing works and then only to create an object of at most two dimensions.

In addition to the above functions, we provide functions `any_na()` and `any_nan()` for finding if there are any `NA` or `NaN` values in a vector. These are equivalent to using `any(is.na())` and `any(is.nan())` in R.

Other R functions with numeric arguments and return value can be called during compiled execution by wrapping them as a `nimbleRcall` (see Section 11.7).

Next we cover some of these functions in more detail.

11.2.2.1 *numeric*, *integer*, *logical*, *matrix* and *array*

`numeric`, `integer`, or `logical` will create a 1-dimensional vector of floating-point (or 'double' [precision]), integer, or logical values, respectively. The `length` argument specifies the vector length (default 0), and the `value` argument specifies the initial value either as a scalar (used for all vector elements, with default 0) or a vector. If a vector and its length is not equal to `length`, the remaining values will be zero, but we plan to implement R-style recycling in the next version of NIMBLE. The `init` argument specifies whether or not to initialize the elements in compiled code (default `TRUE`).

If first use of the variable does not rely on initial values, using `init = FALSE` will yield slightly more efficient performance.

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

`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 best way to create an identity matrix is with `diag(n)`, which returns an $n \times n$ identity matrix. NIMBLE also provides a deprecated nimbleFunction `identityMatrix` that does the same thing.

Examples of these functions, and the related function \`*setSize*\` for changing the size of a numeric object, are given in Section 11.3.2.4.

11.2.2.2 *length* and *dim*

`length` behaves like R's `length` function. It returns the *entire* length of X. That means if `X` is multivariate, `length(X)` returns the product of the sizes of the dimensions. NIMBLE's version of `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.

11.2.2.3 Deprecated creation of non-scalar objects using *declare*

Previous versions of NIMBLE provided a function `declare` for declaring variables. The more R-like functions `numeric`, `integer`, `logical`, `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.

11.2.3 Math and linear algebra

Numeric scalar and matrix mathematical operations are listed in Tables 11.2-11.3.

As in R, many scalar operations in NIMBLE will work component-wise on vectors or higher dimensional objects. For example if B and C are vectors, `A     = B + C` will add them and create vector C by component-wise addition of B and C. In the current version of NIMBLE, component-wise operations generally only work for vectors and matrices, not arrays with more than two dimensions. The only exception is assignment: `A = B` will work up to NIMBLE's current limit of four dimensions.

Table 11.2: 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. Vector input column indicates if the function can take a vector as an argument (i.e., if the function is vectorized).

Usage	Description	Comments	Status	Vector input
`x | y`, `x & y`	logical OR (|) and AND(&)		yes	yes
`!x`	logical not		yes	yes
`x > y, x >= y`	greater than (and or equal to)		yes	yes
`x < y, x <= y`	less than (and or equal to)		yes	yes
`x != y, x == y`	(not) equals		yes	yes
`x + y, x - y, x * y`	component-wise operators	mix of scalar and vector	yes	yes
`x / y`	component-wise division	vector x and scalar y	yes	yes
`x^y, pow(x, y)`	power	x^y; vector x,scalar y	yes	yes
`x %% y`	modulo (remainder)		yes	no
`min(x1, x2), max(x1, x2)`	min. (max.) of two scalars		yes	See `pmin`, `pmax`
`exp(x)`	exponential		yes	yes
`log(x)`	natural logarithm		yes	yes
`sqrt(x)`	square root		yes	yes
`abs(x)`	absolute value		yes	yes
`step(x)`	step function at 0	0 if $x < 0$, 1 if $x >= 0$	yes	yes
`equals(x, y)`	equality of two scalars	1 if $x == y$, 0 if $x! = y$	yes	yes
`cube(x)`	third power	x^3	yes	yes
`sin(x), cos(x), tan(x)`	trigonometric functions		yes	yes
`asin(x), acos(x), atan(x)`	inverse trigonometric functions		yes	yes
`asinh(x), acosh(x), atanh(x)`	inv. hyperbolic trig. functions		yes	yes
`logit(x)`	logit	$\log(x/(1-x))$	yes	yes
`ilogit(x), expit(x)`	inverse logit	$\exp(x)/(1+\exp(x))$	yes	yes
`probit(x)`	probit (Gaussian quantile)	$\Phi^{-1}(x)$	yes	yes
`iprobit(x), phi(x)`	inverse probit (Gaussian CDF)	$\Phi(x)$	yes	yes
`cloglog(x)`	complementary log log	$\log(-\log(1-x))$	yes	yes
`icloglog(x)`	inverse complementary log log	$1-\exp(-\exp(x))$	yes	yes
`ceiling(x)`	ceiling function	$\lceil(x)\rceil$	yes	yes
`floor(x)`	floor function	$\lfloor(x)\rfloor$	yes	yes
`round(x)`	round to integer		yes	yes
`trunc(x)`	truncation to integer		yes	yes
`lgamma(x), loggam(x)`	log gamma function	$\log\Gamma(x)$	yes	yes
`besselK(k, nu, ...expon.scaled)`	modified bessel function of the second kind	returns vector even if k a matrix/array	yes	yes
`log1p(x)`	log of 1 + x	$\log(1+x)$	yes	yes
`lfactorial(x),`	log factorial	$\log x!$	yes	yes

Usage	Description	Comments	Status	Vector input
`logfact(x)`				
`qDIST(x, PARAMS)`	"q" distribution functions	canonical parameterization	yes	yes
`pDIST(x, PARAMS)`	"p" distribution functions	canonical parameterization	yes	yes
`rDIST(x, PARAMS)`	"r" distribution functions	canonical parameterization	yes	yes
`dDIST(x, PARAMS)`	"d" distribution functions	canonical parameterization	yes	yes
`sort(x)`			no	
`rank(x, s)`			no	
`ranked(x, s)`			no	
`order(x)`			no	

Table 11.3: Functions operating on vectors and matrices. Status column indicates if the function is currently provided in NIMBLE.

Usage	Description	Comments	Status
`inverse(x)`	matrix inverse	x symmetric, positive def.	yes
`chol(x)`	matrix Cholesky factorization	x symmetric, positive def. returns upper triang. matrix	yes
`t(x)`	matrix transpose	$x^\top$	yes
`x%*%y`	matrix multiply	xy; x, y conformant	yes
`inprod(x, y)`	dot product	$x^\top y$; x and y vectors	yes
`solve(x,y)`	solve system of equations	$x^{-1}y$; y matrix or vector	yes
`forwardsolve(x, y)`	solve lower-triangular system of equations	$x^{-1}y$; x lower-triangular	yes
`backsolve(x, y)`	solve upper-triangular system of equations	$x^{-1}y$; x upper-triangular	yes
`logdet(x)`	log matrix determinant	$\log \lvert x \rvert$	yes
`asRow(x)`	convert vector to 1-row matrix	sometimes automatic	yes
`asCol(x)`	convert vector to 1-column matrix	sometimes automatic	yes
`sum(x)`	sum of elements of `x`		yes
`mean(x)`	mean of elements of `x`		yes
`sd(x)`	standard deviation of elements of `x`		yes
`prod(x)`	product of elements of `x`		yes
`min(x), max(x)`	min. (max.) of elements of `x`		yes
`pmin(x, y), pmax(x,y)`	vector of mins (maxs) of elements of `x` and `y`		yes
`interp.lin(x, v1, v2)`	linear interpolation		no
`eigen(x)`	matrix eigendecomposition; returns a nimbleList of type `eigenNimbleList`	x symmetric	yes
`svd(x)`	matrix singular value decomposition; returns a nimbleList of type `svdNimbleList`		yes

More information on the nimbleLists returned by the `eigen` and `svd` functions in NIMBLE can be found in Section 14.2.2.

11.2.4 Distribution functions

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

- Names of the distributions generally (but not always) match those of R, which sometimes differ from BUGS. See the list below.
- Supported parameterizations are also indicated in the list below.
- For multivariate distributions (multivariate normal, Dirichlet, and Wishart), 'r' functions only return one random draw at a time, and the first argument must always be 1.
- R's recycling rule (re-use of an argument as needed to accommodate longer values of other arguments) is generally followed, but the returned object is always a scalar or a vector, not a matrix or array.

As in R (and nimbleFunctions), arguments are matched by order or by name (if given). Standard arguments to distribution functions in R (`log`, `log.p`, `lower.tail`) can be used and have the same defaults. User-defined distributions for BUGS (Chapter 12) can also be used from nimbleFunctions.

For standard distributions, we rely on R's regular help pages (e.g., `help(dgamma)`. For distributions unique to NIMBLE (e.g., `dexp_nimble`, `ddirch`), we provide help pages.

Supported distributions, listed by their 'd' function, include:

- `dbinom(x, size, prob, log)`
- `dcat(x, prob, log)`
- `dmulti(x, size, prob, log)`
- `dnbinom(x, size, prob, log)`
- `dpois(x, lambda, log)`
- `dbeta(x, shape1, shape2, log)`
- `dchisq(x, df, log)`
- `ddexp(x, location, rate, log)`
- `ddexp(x, location, scale, log)`

- `dexp(x, rate, log)`
- `dexp_nimble(x, rate, log)`
- `dexp_nimble(x, scale, log)`
- `dgamma(x, shape, rate, log)`
- `dgamma(x, shape, scale, log)`
- `dinvgamma(x, shape, rate, log)`
- `dinvgamma(x, shape, scale, log)`
- `dlnorm(x, meanlog, sdlog, log)`
- `dlogis(x, location, scale, log)`
- `dnorm(x, mean, sd, log)`
- `dt_nonstandard(x, df, mu, sigma, log)`
- `dt(x, df, log)`
- `dunif(x, min, max, log)`
- `dweibull(x, shape, scale, log)`
- `ddirch(x, alpha, log)`
- `dlkj_corr_cholesky(x, shape, size, log)`
- `dmnorm_chol(x, mean, cholesky, prec_param, log)`

- `dmvt_chol(x, mu, cholesky, df, prec_param, log)`
- `dwish_chol(x, cholesky, df, scale_param, log)`
- `dinvwish_chol(x, cholesky, df, scale_param, log)`

In the last three, `cholesky` stands for Cholesky decomposition of the relevant matrix; `prec_param` is a logical indicating whether the Cholesky is of a precision matrix (TRUE) or covariance matrix (FALSE)[3]; and `scale_param` is a logical indicating whether the Cholesky is of a scale matrix (TRUE) or an inverse scale matrix (FALSE).

11.2.5 Flow control: *if-then-else*, *for*, *while*, and *stop*

These basic flow-control 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. Unlike in R, `if` is not itself a function that returns a value.

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 in Section 15.4.8.

`stop`, or equivalently `nimStop`, throws control to R's error-handling system and can take a character argument that will be displayed in an error message.

11.2.6 *print* and *cat*

`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.

11.2.7 Checking for user interrupts: *checkInterrupt*

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 (i.e., by pressing Control-C). Simply include `checkInterrupt` in run code at places where a check should be done. If there has been an interrupt waiting to be handled, the process will stop and return control to R.

11.2.8 Optimization: *optim* and *nimOptim*

NIMBLE provides a `nimOptim` function that partially implement's R's `optim` function with some minor differences. `nimOptim` supports optimization methods 'Nelder-Mead', 'BFGS', 'CG', 'L-BFGS-B', but does not support methods 'SANN' and 'Brent'. NIMBLE's `nimOptim` allows gradients to be supplied using user-provided functions if available or finite differences otherwise. Currently `nimOptim` does not support extra parameters to the function being optimized (via `\dots`), but a work-around is to create a new `nimbleFunction` that calls another one with the additional parameters. Finally, `nimOptim` requires a `nimbleList` datatype for the `control` parameter, whereas R's `optim` uses a simple R `list`. To define the `control` parameter, create a default value with the `nimOptimDefaultControl` function, and set any desired fields.

See `help(nimOptim)` for details, including example usage.

[3] For the multivariate t, these are more properly termed the 'inverse scale' and 'scale' matrices

11.2.9 'nim' synonyms 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 R functions with quite different functionality. 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.

NIMBLE tries to avoid conflicts by replacing some keywords immediately upon creating a nimbleFunction. These replacements include

- `c` → `nimC`
- `copy` → `nimCopy`
- `dim` → `nimDim`
- `print` → `nimPrint`
- `cat` → `nimCat`
- `step` → `nimStep`
- `equals` → `nimEquals`
- `rep` → `nimRep`
- `round` → `nimRound`
- `seq` → `nimSeq`
- `stop` → `nimStop`
- `switch` → `nimSwitch`
- `numeric, integer, logical` → `nimNumeric, nimInteger, nimLogical`
- `matrix, array` → `nimMatrix, nimArray`

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.

11.3 How NIMBLE handles types of variables

Variables in the NIMBLE language are statically typed. Once a variable is used for one type, it can't subsequently be used for a different type. This rule facilitates NIMBLE's compilation to C++. The NIMBLE compiler often determines types automatically, but sometimes the programmer needs to explicitly provide them.

The elemental types supported by NIMBLE include *double* (floating-point), *integer*, *logical*, and *character*. The *type* of a numeric or logical object refers to the number of dimensions and the elemental type of the elements. Hence if `x` is created as a double matrix, it can only be used subsequently for a double matrix. The size of each dimension is not part of its type and thus can be changed. Up to four dimensions are supported for double, integer, and logical. Only vectors (one dimension) are supported for character. Unlike R, NIMBLE supports true scalars, which have 0 dimensions.

11.3.1 nimbleList data structures

A `nimbleList` is a data structure that can contain arbitrary other NIMBLE objects, including other `nimbleLists`. Like other NIMBLE types, nimbleLists are strongly typed: each `nimbleList` is created from a configuration that declares what types of objects it will hold. `nimbleLists` are covered in Chapter 14.2.

11.3.2 How numeric types work

R's dynamic types support easy programming because one type can sometimes be transformed to another type automatically when an expression is evaluated. NIMBLE's static types makes it stricter than R.

11.3.2.1 When NIMBLE can automatically set a numeric type

When a variable is first created by assignment, its type is determined automatically by that assignment. For example, if `x` has not appeared before, then

```
x <- A %*% B # assume A and B are double matrices or vectors
```

will create `x` to be a double matrix of the correct size (determined during execution).

11.3.2.1.1 Avoid changing types of a variable within a nimbleFunction Because NIMBLE is statically typed, you cannot use the same variable name for two objects of different types (including objects of different dimensions).

Suppose we have (implicitly) created `x` as a double matrix. If `x` is used subsequently, it can only be used as a double matrix. This is true even if it is assigned a new value, which will again set its size automatically but cannot change its type.

```
x <- A %*% B # assume A and B are double matrices or vectors
x <- nimMatrix(0, nrow = 5, ncol = 2)  # OK: 'x' is still a double matrix
x <- rnorm(10)  # NOT OK: 'x' is a double vector
```

11.3.2.2 When a numeric object needs to be created before being used

If the contents of a variable are to be populated by assignment into some indices in steps, the variable must be created first. Further, it must be large enough for its eventual contents; it will not be automatically resized if assignments are made beyond its current size. For example, in the following code, `x` must be created before being filled with contents for specific indices.

```
x <- numeric(10)

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

11.3.2.3 How NIMBLE handles `NA`

NIMBLE supports use of `NA` with some caveats. In the compiled version of a nimbleFunction:

- `NA` values in a logical scalar or vector will not work;
- Assigning `NA` to a new variable will make that variable have type double (in R, this would be a logical!);
- Non-trivial use of `NA` in integer variables may fail by having the value become a large (negative) number;

- Use of `NA` in doubles should generally work.

These issues arise because NIMBLE uses R's encoding of NA values in C(++) but uses native compiled math and type-casting in C++, which do not always preserve R's NA encodings. In summary, NA should generally work for doubles and should be used cautiously (i.e. after you test that what you need to work actually works) for integers. For some needs, NaN is a suitable alternative to NA.

11.3.2.4 Changing the sizes of existing objects: *setSize*

`setSize` changes the size of an object, preserving its contents in column-major order.

```
# 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; last 10 elements will be 0
setSize(myNumericVector, 20)
```

```
# Example of creating a 1-by-10 matrix with values 1:10 and resizing it
myMatrix <- matrix(1:10, nrow = 1, ncol = 10)
# resize this matrix to be a 10-by-10 matrix
setSize(myMatrix, c(10, 10))
# The first column will have the 1:10
```

11.3.2.5 Confusions between scalars and length-one vectors

In R, there is no such thing is a true scalar; scalars can always be treated as vectors of length one. NIMBLE allows true scalars, which can create confusions. For example, consider the following code:

```
myfun <- nimbleFunction(
    run = function(i = integer()) { # i is an integer scalar
        randomValues <- rnorm(10)   # double vector
        a <- randomValues[i]        # double scalar
        b <- randomValues[i:i]      # double vector
        d <- a + b                  # double vector
        f <- c(i)                   # integer vector
    })
```

In the line that creates `b`, the index range `i:i` is not evaluated until run time. Even though `i:i` will always evaluate to simpy `i`, the compiler does not determine that. Since there is a vector index range provided, the result of `randomValues[i:i]` is determined to be a vector. The following line then creates `d` as a vector, because a vector plus a scalar returns a vector. Another way to create a vector from a scalar is to use `c`, as illustrated in the last line.

11.3.2.6 Confusions between vectors and one-column or one-row matrices

Consider the following code:

```
myfun <- nimbleFunction(
    run = function() {
        A <- matrix(value = rnorm(9), nrow = 3)
        B <- rnorm(3)
        Cmatrix <- A %*% B                # double matrix, one column
        Cvector <- (A %*% B)[,1]          # double vector
        Cmatrix <- (A %*% B)[,1]          # error, vector assigned to matrix
        Cmatrix[,1] <- (A %*% B)[,1]      # ok, if Cmatrix is large enough
    })
```

This creates a matrix `A`, a vector `B`, and matrix-multiplies them. The vector `B` is automatically treated as a one-column matrix in matrix algebra computations. The result of matrix multiplication is always a matrix, but a programmer may expect a vector, since they know the result will have one column. To make it a vector, simply extract the first column. More information about such handling is provided in the next section.

11.3.2.7 Understanding dimensions and sizes from linear algebra

As much as possible, NIMBLE behaves like R when determining types and sizes returned from linear algebra expressions, but in some cases this is not possible because R uses run-time information while NIMBLE must determine dimensions at compile time. For example, when matrix multiplying a matrix by a vector, R treats the vector as a one-column matrix unless treating it as a one-row matrix is the only way to make the expression valid, as determined at run time. NIMBLE usually must assume during compilation that it should be a one-column matrix, unless it can determine not just the number of dimensions but the actual sizes during compilation. When needed `asRow` and `asCol` can control how a vector will be treated as a matrix.

Here is a guide to such issues. 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] <- 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 11.3.2.4 for more details. Note that non-indexed assignment to `Y`, such as `Y <- 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-dimesional, and it will be automatically re-sized for the result.
- `Y <- v1 + (M2 %*% v2)[,1]` will return a vector. Y will either be created as a vector or must already exist as a vector and will be re-sized for the result.

11.3.2.8 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 1.0.0, NIMBLE attempts to issue a 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] <- x` without ensuring Y is at least 10x30. In the future we hope to prevent crashes, but in Version 1.0.0 we limit ourselves to trying to provide useful information.

11.4 Declaring argument and return types

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

As illustrated in the example in Section 11.1, the syntax for a type declaration is:

`type(nDim, sizes)`

where `type` is `double`, `integer`, `logical` or `character`. (In more general nimbleFunction programming, a type can also be a nimbleList type, discussed in Section 14.2.)

For example `run = function(x = double(1)) { ...}` sets the single argument of the run function to be a vector of numeric values of unknown size.

For `type(nDim, sizes)`, `nDim` is the number of dimensions, with 0 indicating scalar and omission of `nDim` defaulting to a scalar. `sizes` is an optional vector of fixed, known sizes.
For example, `double(2, c(4, 5))` declares a 4 × 5 matrix. If sizes are omitted, they will be set either by assignment or by `setSize`.

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)) {
})
```

Functions with return values must have their return type explicitly declared using `returnType`, which can occur anywhere in the run code. For example `returnType(integer(2))` declares the return type to be a matrix of integers. A return type of `void()` means there is no return value, which is the default if no `returnType` statement is included.

Note that because all values in models are stored as doubles and because of some limitations in NIMBLE's automatic casting, non-scalar return values of user-defined distributions must be doubles.

11.5 Compiled nimbleFunctions pass arguments by reference

Uncompiled nimbleFunctions pass arguments like R does, by copy. If `x` is passed as an argument to function `foo`, and `foo` modifies `x` internally, it is modifying its copy of `x`, not the original `x` that was passed to it.

Compiled nimbleFunctions pass arguments to other compiled nimbleFunctions by reference (or pointer). This is very different. Now if `foo` modifies `x` internally, it is modifying the same `x` that was passed to it. This allows much faster execution but is obviously a fundamentally different behavior.

Uncompiled execution of nimbleFunctions is primarily intended for debugging. However, debugging of how nimbleFunctions interact via arguments requires testing the compiled versions.

11.6 Calling external compiled code

If you have a function in your own compiled C or C++ code and an appropriate header file, you can generate a nimbleFunction that wraps access to that function, which can then be used in other nimbleFunctions. See `help(nimbleExternalCall)` for an example. This also contains an example of using an externally compiled function in the BUGS code of a model.

11.7 Calling uncompiled R functions from compiled nimbleFunctions

Sometimes one may want to combine R functions with compiled nimbleFunctions. Obviously a compiled nimbleFunction can be called from R. An R function with numeric inputs and output can be called from compiled nimbleFunctions. The call to the R function is wrapped in a nimbleFunction returned by `nimbleRcall`. See `help(nimbleRcall)` for an example, including an example of using the resulting function in the BUGS code of a model.

Chapter 12

Creating user-defined BUGS distributions and functions

NIMBLE allows you to define your own functions and distributions as *nimbleFunction*s for use in BUGS code. As a result, NIMBLE frees you from being constrained to the functions and distributions discussed in Chapter 5. 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 for the data and provide that as a user-defined distribution instead.

Since NIMBLE allows you to wrap calls to external compiled code or arbitrary R functions as nimbleFunctions, and since you can define model functions and distributions as nimbleFunctions, you can combine these features to build external compiled code or arbitrary R functions into a model. See Sections 11.6-11.7.

Note that NIMBLE generally expects user-defined distributions or functions to be defined in the global environment. If you define them in a function (which would generally be the case if you are using them in the context of parallelization), one approach would be to assign them to the global environment in your function:

```r
## for a user-defined function
assign('myfun', myfun, envir = .GlobalEnv)
## for a user-defined distribution
assign('dfoo', dfoo, envir = .GlobalEnv)
assign('rfoo', rfoo, envir = .GlobalEnv)
## similarly for 'p' and 'q' functions if you define them
```

12.1 User-defined functions

To provide a new function for use in BUGS code, simply create a nimbleFunction that has no *setup* code as discussed in Chapter 11. 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 11.4). 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 zero-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 in compiled use of a model, you cannot provide a scalar argument where the user-defined nimbleFunction expects a vector; unlike in R, scalars are not simply vectors of length 1.

12.2 User-defined distributions

To provide a user-defined distribution, you need to define density (‘d’) and (optionally) simulation (‘r’) `nimbleFunctions`, without setup code, for your distribution. In many cases you can then simply use your distribution in BUGS code as you would any distribution already provided by

NIMBLE, while in a few special cases[1] you need to explicitly register your distribution as described in Section 12.2.1.

You need to provide the simulation ('r') function if any algorithm used with a model that uses the distribution needs to simulate from the distribution. This is not the case for NIMBLE's built-in MCMC sampler functions, and therefore the simulation function is not generally required for standard MCMC in NIMBLE. However, the 'r' function is necessary for initialization of nodes that are assigned the user-defined distribution if no initial value is provided, and for sampling posterior predictive nodes (those nodes with no downstream data dependencies) that are assigned the user-defined distribution.

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 (solely for illustration as 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)
```

[1] These include providing alternative parameterizations, specifying the range of the distribution, or specifying that the distribution is a discrete distribution.

```
            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+1) - sum(lgamma(x+1)) + 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))
```

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. In fact, non-scalars must be declared as doubles.
- 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[2].

12.2.1 Using *registerDistributions* for alternative parameterizations and providing other information

Behind the scenes, NIMBLE uses the function `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 automatically 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 below. 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 (scale instead of rate) and to provide the range for the user-defined exponential distribution. We can then use the alternative parameterization in our BUGS code.

```
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)
        )
```

[2] 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.

```
    ))

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)))
```

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 names 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 R help on `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 or as the R list `nimble:::distributionsInputList`.

Chapter 13

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, its primary use occurs when writing nimbleFunctions (see Chapter 15). Information about node types, distributions, and dimensions can be used to determine algorithm behavior in *setup* code of nimbleFunctions. Information about node or variable values or the parameter and bound values of a node would generally be used for algorithm calculations in *run* code of nimbleFunctions. Similarly, carrying out numerical operations on a model, including setting node or variable values, would generally be done in run code.

13.1 The variables and nodes in a NIMBLE model

Section 6.2 defines what we mean by variables and nodes in a NIMBLE model and discusses how to determine and access the nodes in a model and their dependency relationships. Here we'll review and go into more detail on the topics of determining the nodes and node dependencies in a model.

13.1.1 Determining the nodes in a model

One can determine the variables in a model using `getVarNames` and the nodes in a model using `getNodeNames`, with optional arguments allowing you to select only certain types of nodes. We illustrate here with the pump model from Chapter 2.

```
pump$getVarNames()
```

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

```
pump$getNodeNames()
```

```
##  [1] "alpha"               "beta"                "lifted_d1_over_beta"
##  [4] "theta[1]"            "theta[2]"            "theta[3]"
##  [7] "theta[4]"            "theta[5]"            "theta[6]"
## [10] "theta[7]"            "theta[8]"            "theta[9]"
## [13] "theta[10]"           "lambda[1]"           "lambda[2]"
## [16] "lambda[3]"           "lambda[4]"           "lambda[5]"
## [19] "lambda[6]"           "lambda[7]"           "lambda[8]"
## [22] "lambda[9]"           "lambda[10]"          "x[1]"
## [25] "x[2]"                "x[3]"                "x[4]"
## [28] "x[5]"                "x[6]"                "x[7]"
## [31] "x[8]"                "x[9]"                "x[10]"
```

```
pump$getNodeNames(determOnly = TRUE)
```

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

```
pump$getNodeNames(stochOnly = TRUE)
```

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

```
pump$getNodeNames(dataOnly = TRUE)
```

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

You can see one lifted node (see next section), `lifted_d1_over_beta`, involved in a reparameterization to NIMBLE's canonical parameterization of the gamma distribution for the `theta` nodes.

We can determine the set of nodes contained in one or more nodes or variables using `expandNodeNames`, illustrated here for an example with multivariate nodes. The `returnScalarComponents` argument also allows us to return all of the scalar elements of multivariate nodes.

```
multiVarCode2 <- nimbleCode({
    X[1, 1:5] ~ dmnorm(mu[], cov[,])
    X[6:10, 3] ~ dmnorm(mu[], cov[,])
    for(i in 1:4)
        Y[i] ~ dnorm(mn, 1)
})

multiVarModel2 <- nimbleModel(multiVarCode2,
                              dimensions = list(mu = 5, cov = c(5,5)),
                             calculate = FALSE)



multiVarModel2$expandNodeNames("Y")
```

```
## [1] "Y[1]" "Y[2]" "Y[3]" "Y[4]"
```

```
multiVarModel2$expandNodeNames(c("X", "Y"), returnScalarComponents = TRUE)
```

```
##  [1] "X[1, 1]"  "X[1, 2]"  "X[1, 3]"  "X[6, 3]"  "X[7, 3]"  "X[8, 3]"
##  [7] "X[9, 3]"  "X[10, 3]" "X[1, 4]"  "X[1, 5]"  "Y[1]"     "Y[2]"
## [13] "Y[3]"     "Y[4]"
```

As discussed in Section 6.2.6, you can determine whether a node is flagged as data using `isData`.

13.1.2 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:

1. 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.
2. 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.
3. Use of alternative parameterizations of distributions, described in Section 5.2.4 causes new nodes to be introduced. 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.

13.1.3 Determining dependencies in a model

Next we'll see how to determine the node dependencies (or 'descendants' or child nodes) 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.

```
pump$getDependencies("alpha")
```

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

```
pump$getDependencies(c("alpha", "beta"))
```

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

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

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

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

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

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

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

```
pump$getDependencies("alpha", downstream = TRUE, dataOnly = TRUE)
```

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

In addition, one can determine parent nodes using `getParents`.

```
pump$getParents("alpha")
```

```
## character(0)
```

13.2 Accessing information about nodes and variables

13.2.1 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')
```

```
## y[1] y[2] y[3] y[4]
## TRUE TRUE TRUE TRUE
```

```
m$getDistribution('sigma')
```

```
##   sigma
## "dunif"
```

```
m$isDiscrete(c('y', 'mu', 'sigma'))
```

```
##  y[1]  y[2]  y[3]  y[4]    mu sigma
## FALSE FALSE FALSE FALSE FALSE FALSE
```

```
m$isDeterm('mu')
```

```
##    mu
## FALSE
```

```
m$getDimension('mu')
```

```
## value
##     0
```

```
m$getDimension('mu', includeParams = TRUE)
```

```
## value  mean    sd   tau   var
##     0     0     0     0     0
```

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.

13.2.2 Getting information about a distribution

One can also get generic information about a distribution based on the name of the distribution using the function `getDistributionInfo`. 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`.

13.2.3 Getting distribution parameter values for a node

The function `getParam` provides access to values of the parameters of a node's distribution. The 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 5.2.4.1 for available parameterizations. (These can also be seen in `nimble:::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))
gammaModel$getParam('x', 'scale')
```

```
## [1] 1.2
```

```
gammaModel$getParam('x', 'rate')
```

```
## [1] 0.8333333
```

getParam is part of the NIMBLE language, so it can be used in run code of nimbleFunctions.

13.2.4 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 is used (Section 5.2.7), 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')
```

```
## [1] -0.5
```

```
getBound(exampleModel, 'y', 'upper')
```

```
## [1] Inf
```

```
exampleModel$b <- 3
exampleModel$calculate(exampleModel$getDependencies('b'))
```

```
## [1] -4.386294
```

```
getBound(exampleModel, 'a', 'upper')
```

```
## [1] 3
```

```
exampleModel$getBound('b','lower')
```

```
## [1] 0
```

getBound is part of the NIMBLE language, so it can be used in run code of 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.

13.3 Carrying out model calculations

13.3.1 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[1]. 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`.

13.3.1.1 Example: simulating arbitrary collections of nodes

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

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

model$a <- 1
model$y
```

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

```
model$simulate("y[1:3]")
# simulate(model, "y[1:3]")
```

[1] Standard usage is `model$calculate(nodes)` but `calculate(model, nodes)` is synonymous.

```
model$y
```

```
## [1] -4.3723097  1.7331821  0.6234022         NA         NA
```

```
model$simulate("y")
model$y
```

```
## [1]  6.6051146  2.0859962 -0.9702564 -0.3898915 -0.5978887
```

```
model$z
```

```
##            [,1]       [,2]        [,3]
## [1,] -1.0314207 -0.9537793 -0.79453166
## [2,] -2.3710229 -0.3980044 -0.30881797
## [3,] -0.3245763 -0.3112171  0.36144477
## [4,] -0.9442988  0.7960927  1.39879110
## [5,] -0.7658900  0.9864283 -0.05607042
```

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

```
## [1]  1.5882948  1.5578821  3.8961663 -0.3898915 -0.5978887
```

```
model$z
```

```
##            [,1]       [,2]        [,3]
## [1,] -1.0314207 -0.9537793 -0.79453166
## [2,] -2.3710229 -0.3980044 -0.30881797
## [3,] -0.3245763 -0.3112171  0.36144477
## [4,] -0.9442988  0.7960927  1.39879110
## [5,] -0.7658900  0.9864283 -0.05607042
```

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

```
##            [,1]       [,2]       [,3]
## [1,]  1.6203124  1.5516260  1.4942336
## [2,]  1.6213524  1.5516332  1.5761658
## [3,]  4.0065304  4.0713699  3.8007847
## [4,] -0.2254835 -0.4765649 -0.3632563
## [5,] -0.5756517 -0.6255796 -0.4584634
```

The example illustrates a number of features:

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 inthe 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 such as `getDependencies`. These return nodes in topologically-sorted order, which means no node is manipulated 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.

13.3.2 Pre-defined nimbleFunctions for operating on model nodes: *simNodes*, *calcNodes*, and *getLogProbNodes*

`simNodes`, `calcNodes` and `getLogProbNodes` 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 `getLogProbNodes`, 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 `getLogProbNodes` 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
```

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

```
cSimpleModel$y
```

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

```
# simulating x and y
cSimXY$run()
cSimpleModel$x
```

```
## [1] -0.6589120  0.6605313 -0.0132557 -0.9314803
```

```
cSimpleModel$y
```

```
## [1]  0.5557771 -1.4278091 -0.5394082 -2.4728829
```

```
cCalcXDep$run()
```

```
## [1] -12.46535
```

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

```
## [1] -12.46535
```

```
cSimXY$run()

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

```
## [1] -12.46535
```

```
cCalcXDep$run()
```

```
## [1] -9.277808
```

13.3.3 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_’. In general users will not need to access these logProb variables directly but rather will use `getLogProb`. However, knowing they exist can be useful, in part because these variables can be monitored in an MCMC.

When the stochastic node is scalar, the `logProb` variable will have the same size. For example:

```
model$logProb_y
```

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

```
model$calculate("y")
```

```
## [1] -11.02766
```

```
model$logProb_y
```

```
## [1] -2.087536 -2.085793 -2.489620 -2.166821 -2.197894
```

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.

Chapter 14

Data structures in NIMBLE

NIMBLE provides several data structures useful for programming.

We'll first describe *modelValues*, which are containers designed for storing values for models. Then in Section 14.2 we'll describe *nimbleLists*, which have a similar purpose to lists in R, allowing you to store heterogeneous information in a single object.

modelValues can be created in either R or in nimbleFunction setup code. nimbleLists can be created in R code, in nimbleFunction setup code, and in nimbleFunction run code, from a nimbleList definition created in R or setup code. Once created, modelValues and `nimbleLists` can then be used either in R or in nimbleFunction setup or run code. If used in run code, they will be compiled along with the nimbleFunction.

14.1 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 15)[1]. 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 used in run code of nimbleFunctions.

14.1.1 Creating modelValues objects

Here is a simple example of creating a modelValues object:

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

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

[1] One may want to read this section after an initial reading of Chapter 15.

```
pumpModelValues$x
```

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

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

Alternatively, one can define a modelValues object manually by first defining a modelValues *configuration* via the `modelValuesConf` function, and then creating an instance from that configuration, 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
```

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

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 object. When a nimbleFunction is compiled, any modelValues objects it uses are also compiled. A NIMBLE model always contains a modelValues object that it uses as a default location to store the values of 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 modelValues is then accessed with `$`.

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

rResize <- resizeMV(customMV)
cResize <- compileNimble(rResize)
cResize$run(5)
cCustomMV <- cResize$mv
```

```
# cCustomMV is a compiled modelValues object
cCustomMV[['a']]
```

```
## [[1]]
## [1] NA NA
##
## [[2]]
## [1] NA NA
##
## [[3]]
## [1] 0 0
##
## [[4]]
## [1] 0 0
##
## [[5]]
## [1] 0 0
```

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.

In the example above a modelValues object is passed to setup code, but a modelValues configuration can also be passed, with creation of modelValues object(s) from the configuration done in setup code.

14.1.2 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
customMV['a', 2][2] <- 4

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

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

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

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

# make sure the size of inputs is correct
# customMV['a', 1] <- 1:10  "
# problem: size 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]]`.

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

```
getsize(customMV)
```

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

```
resize(customMV, 3)
getsize(customMV)
```

```
## [1] 3
```

```
customMV$a
```

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

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

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

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

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

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

`as.list` will return a list with an element for each variable. Each element will be an array of the same size and shape as the variable with an additional index for the iteration (e.g., MCMC iteration when used for MCMC output). By default iteration is the first index, but it can be the last if needed.

```
as.list(customMV, 'a')    # get only 'a'
```

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

```
as.list(customMV)         # get all variables
```

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

```
as.list(customMV, 'a', iterationAsLastIndex = TRUE) # put iteration in last index
```

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

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

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

```
## [1] 1 2 3
```

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

```
customMV$varNames
```

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

```
customMV$sizes
```

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

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

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

```
## [1] 0 1
```

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

```
## [1] 1 1
```

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.

14.1.2.1 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 row. `calcNodesMV` will save the log probabilities back into the modelValues object if `saveLP = TRUE`, a run-time argument.

Here are some examples:

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

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

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

```
## [1] -18.52723 -14.93433 -23.21395 -22.64131 -12.31624
```

```
cGetLogProbMany$run() #
```

```
## [1] -18.52723 -14.93433 -23.21395 -22.64131 -12.31624
```

```
result <- as.matrix(cSimManyXY$mv) # extract simulated values
```

14.2 The nimbleList data structure

nimbleLists provide a container for storing different types of objects in NIMBLE, similar to the list data structure in R. Before a nimbleList can be created and used, a *definition*[2] for that nimbleList must be created that provides the names, types, and dimensions of the elements in the nimbleList. nimbleList definitions must be created in R (either in R's global environment or in setup code), but the nimbleList instances can be created in run code.

Unlike lists in R, nimbleLists must have the names and types of all list elements provided by a definition before the list can be used. A nimbleList definition can be made by using the `nimbleList` function in one of two manners. The first manner is to provide the `nimbleList` function with a series of expressions of the form `name = type(nDim)`, similar to the specification of run-time arguments to nimbleFunctions. The types allowed for a nimbleList are the same as those allowed as run-time arguments to a nimbleFunction, described in Section 11.4. For example, the following line of code creates a nimbleList definition with two elements: `x`, which is a scalar integer, and `Y`, which is a matrix of doubles.

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

[2] The *configuration* for a modelValues object is the same concept as a *definition* here; in a future release of NIMBLE we may make the usage more consistent between modelValues and nimbleLists.

The second method of creating a nimbleList definition is by providing an R list of *nimbleType* objects to the `nimbleList()` function. A nimbleType object can be created using the `nimbleType` function, which must be provided with three arguments: the `name` of the element being created, the `type` of the element being created, and the `dim` of the element being created. For example, the following code creates a list with two nimbleType objects and uses these objects to create a nimbleList definition.

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

 # this nimbleList definition is identical to the one created above
 exampleNimListDef2 <- nimbleList(nimbleListTypes)
```

Creating definitions using a list of `nimbleType`s can be useful, as it allows for programmatic generation of nimbleList elements.

Once a nimbleList definition has been created, new instances of nimbleLists can be made from that definition using the `new` member function. The `new` function can optionally take initial values for the list elements as arguments. Below, we create a new nimbleList from our `exampleNimListDef` and specify values for the two elements of our list:

```
 exampleNimList <- exampleNimListDef$new(x = 1, Y = diag(2))
```

Once created, nimbleList elements can be accessed using the `$` operator, just as with lists in R. For example, the value of the `x` element of our `exampleNimbleList` can be set to `7` using

```
 exampleNimList$x <- 7
```

nimbleList definitions can be created either in R's global environment or in `setup` code of a nimbleFunction. Once a nimbleList definition has been made, new instances of nimbleLists can be created using the `new` function in R's global environment, in setup code, or in run code of a nimbleFunction.

nimbleLists can also be passed as arguments to run code of nimbleFunctions and returned from nimbleFunctions. To use a nimbleList as a run function argument, the name of the nimbleList definition should be provided as the argument type, with a set of parentheses following. To return a nimbleList from the run code of a nimbleFunction, the `returnType` of that function should be the name of the nimbleList definition, again using a following set of parentheses.

Below, we demonstrate a function that takes the `exampleNimList` as an argument, modifies its `Y` element, and returns the nimbleList.

```
mynf <- nimbleFunction(
  run = function(vals = exampleNimListDef()){
    onesMatrix <- matrix(value = 1, nrow = 2, ncol = 2)
    vals$Y  <- onesMatrix
    returnType(exampleNimListDef())
    return(vals)
  })

# pass exampleNimList as argument to mynf
mynf(exampleNimList)
```

```
## nimbleList object of type nfRefClass_59
## Field "x":
## [1] 7
## Field "Y":
##      [,1] [,2]
## [1,]    1    1
## [2,]    1    1
```

nimbleList arguments to run functions are passed by reference – this means that if an element of a nimbleList argument is modified within a function, that element will remain modified when the function has finished running. To see this, we can inspect the value of the `Y` element of the `exampleNimList` object and see that it has been modified.

```
exampleNimList$Y
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    1    1
```

In addition to storing basic data types, nimbleLists can also store other nimbleLists. To achieve this, we must create a nimbleList definition that declares the types of nested nimbleLists a nimbleList will store. Below, we create two types of nimbleLists: the first, named `innerNimList`, will be stored inside the second, named `outerNimList`:

```
# first, create definitions for both inner and outer nimbleLists
innerNimListDef <- nimbleList(someText = character(0))
outerNimListDef <- nimbleList(xList = innerNimListDef(),
                              z = double(0))

# then, create outer nimbleList
outerNimList <- outerNimListDef$new(z = 3.14)

# access element of inner nimbleList
outerNimList$xList$someText <- "hello, world"
```

Note that definitions for inner, or nested, nimbleLists must be created before the definition for an outer nimbleList.

14.2.1 Pre-defined nimbleList types

Several types of nimbleLists are predefined in NIMBLE for use with particular kinds of nimbleFunctions.

- `eigenNimbleList` and `svdNimbleList` are the return types of the `eigen` and `svd` functions (Section @ref(sec:eigen-nimFunctions}).
- `waicNimbleList` is the type provided for WAIC output from an MCMC.
- `waicDetailsNimbleList` is the type provided for detailed WAIC output from an MCMC.
- `optimResultNimbleList` is the return type of `nimOptim`.
- `optimControlNimbleList` is the type for the control list input to `nimOptim`.

14.2.2 Using *eigen* and *svd* in nimbleFunctions

NIMBLE has two linear algebra functions that return nimbleLists. The `eigen` function takes a symmetic matrix, `x`, as an argument and returns a nimbleList of type `eigenNimbleList`. nimbleLists of type `eigenNimbleList` have two elements: `values`, a vector with the eigenvalues of `x`, and `vectors`, a square matrix with the same dimension as `x` whose columns are the eigenvectors of `x`. The `eigen` function has two additional arguments: `symmetric` and `only.values`. The `symmetric` argument can be used to specify if `x` is a symmetric matrix or not. If `symmetric = FALSE` (the default value), `x` will be checked for symmetry. Eigendecompositions in NIMBLE for symmetric matrices are both faster and more accurate. Additionally, eigendecompostions of non-symmetric matrices can have complex entries, which are not supported by NIMBLE. If a complex entry is detected, NIMBLE will issue a warning and that entry will be set to `NaN`. The `only.values` arument defaults to `FALSE`. If `only.values = TRUE`, the `eigen` function will not calculate the eigenvectors of `x`, leaving the `vectors` nimbleList element empty. This can reduce calculation time if only the eigenvalues of `x` are needed.

The `svd` function takes an $n \times p$ matrix `x` as an argument, and returns a nimbleList of type `svdNimbleList`. nimbleLists of type `svdNimbleList` have three elements: `d`, a vector with the singular values of `x`, `u` a matrix with the left singular vectors of `x`, and `v`, a matrix with the right singular vectors of `x`. The `svd` function has an optional argument `vectors` which defaults to a value of `"full"`. The `vectors` argument can be used to specify the number of singular vectors that are returned. If `vectors = "full"`, `v` will be an $n \times n$ matrix and `u` will be an $p \times p$ matrix. If `vectors = "thin"`, `v` will be an$n \times m$ matrix, where $m = \min(n, p)$, and `u` will be an $m \times p$ matrix. If `vectors = "none"`, the `u` and `v` elements of the returned nimbleList will not be populated.

nimbleLists created by either `eigen` or `svd` can be returned from a nimbleFunction, using `returnType(eigenNimbleList())` or `returnType(svdNimbleList())` respectively. nimbleLists created by `eigen` and `svd` can also be used within other nimbleLists by specifying the nimbleList element types as `eigenNimbleList()` and `svdNimbleList()`. The below example demonstrates the use of `eigen` and `svd` within a nimbleFunction.

```r
eigenListFunctionGenerator <- nimbleFunction(
  setup = function(){
    demoMatrix <- diag(4) + 2
    eigenAndSvdListDef <- nimbleList(demoEigenList = eigenNimbleList(),
                                     demoSvdList = svdNimbleList())
    eigenAndSvdList <- eigenAndSvdListDef$new()
  },
  run = function(){
    # we will take the eigendecomposition and svd of a symmetric matrix
    eigenAndSvdList$demoEigenList <<- eigen(demoMatrix, symmetric = TRUE,
                                            only.values = TRUE)
    eigenAndSvdList$demoSvdList <<- svd(demoMatrix, vectors = 'none')
    returnType(eigenAndSvdListDef())
    return(eigenAndSvdList)
  })
eigenListFunction <- eigenListFunctionGenerator()

outputList <-  eigenListFunction$run()
```

```
outputList$demoEigenList$values
```

```
## [1] 9 1 1 1
```

```
outputList$demoSvdList$d
```

```
## [1] 9 1 1 1
```

The eigenvalues and singular values returned from the above function are the same since the matrix being decomposed was symmetric. However, note that both eigendecompositions and singular value decompositions are numerical procedures, and computed solutions may have slight differences even for a symmetric input matrix.

Chapter 15

Writing nimbleFunctions to interact with models

15.1 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 value to be returned[1]. Using the `function` keyword in R triggers the operation of creating an object that is the function.

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

1. *Setup* code is provided as a regular R function, but the programmer does not control what it returns. Typically the inputs to *setup* code are objects like a model, a vector of nodes, a modelValues object or a modelValues configuration, 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, which was introduced in Chapter 11. 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 value, and in some circumstances for local variables. There are two kinds of *run* code:

    a. There is always a primary function, given as the argument `run`[2].
    b. There can optionally be other functions, or 'methods' in the language of object-oriented programming, that share the same objects created by the *setup* function.

Here is a small example to fix ideas:

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

[1] Normally this is the value of the last evaluated code, or the argument to `return`.

[2] This can be omitted if you don't need it.

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

code <- nimbleCode({
    a ~ dnorm(0, 1)
    b ~ dnorm(a, 1)
})
testModel <- nimbleModel(code, check = FALSE)
logProbCalcPlusA <- logProbCalcPlus(testModel, "a")
testModel$b <- 1.5
logProbCalcPlusA$run(0.25)
```

```
## [1] -2.650377
```

```
dnorm(1.25,0,1,TRUE)+dnorm(1.5,1.25,1,TRUE) # direct validation
```

```
## [1] -2.650377
```

```
testModel$a  # "a" was set to 0.5 + valueToAdd
```

```
## [1] 1.25
```

The call to the R function called `nimbleFunction` returns a function, similarly to defining a function in R. That function, `logProbCalcPlus`, takes arguments for its `setup` function, executes it, and returns an object, `logProbCalcPlusA`, that has a *run* member function (method) accessed by `$run`. In this case, the `setup` function obtains the stochastic dependencies of the `node` using the `getDependencies` member function of the model (see Section 13.1.3) and stores them in `dependentNodes`. In this way, `logProbCalcPlus` can adapt to any model. It also creates a variable, `valueToAdd`, that can be used by the nimbleFunction.

The object `logProbCalcPlusA`, returned by `logProbCalcPlus`, is permanently bound to the results of the processed `setup` function. In this case, `logProbCalcPlusA$run` takes a scalar input value, `P`, assigns `P + valueToAdd` to the given node in the model, and returns the sum of the log probabilities of that node and its stochastic dependencies[3]. We say `logProbCalcPlusA` is 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.

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

15.2 Using and compiling nimbleFunctions

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

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

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

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

```
## [1] 1.25
```

```
CtestModel$b
```

```
## [1] 1.5
```

```
lpA <- ClogProbCalcPlusA$run(1.5)
lpA
```

```
## [1] -5.462877
```

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

```
## [1] -5.462877
```

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

```
## [1] 2.5
```

```
testModel$a        # the uncompiled model was not modified
```

```
## [1] 1.25
```

15.3 Writing setup code

15.3.1 Useful tools for setup functions

The setup function is typically used to determine information on nodes in a model, set up modelValues or nimbleList 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 code 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 from a model** Often these are obtained from the `getNodeNames`, `getDependencies`, and other methods of a model, described in Sections 13.1-13.2.
- **modelValues objects** These are discussed in Sections 14.1 and 15.4.4.

- **nimbleList objects** New instances of `nimbleLists` can be created from a nimbleList definition in either setup or run code. See Section 14.2 for more information.
- **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 (Section 15.4.7).
- **lists of other nimbleFunctions** In addition to containing single other nimbleFunctions, a nimbleFunction can contain a list of other nimbleFunctions (Section 15.4.8).

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

15.3.2 Accessing and modifying numeric values from setup

While models and nodes created during setup cannot be modified[4], numeric values and modelValues can be, as illustrated by extending the example from above.

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

```
## [1] 1
```

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

```
## [1] 1
```

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

```
## [1] -16.46288
```

```
CtestModel$a     # a == 1.5 + 3
```

```
## [1] 4.5
```

15.3.3 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 (two-dimensional double) in one specialization but as a vector (one-dimensional double) in another, there will be a problem during compilation. The sizes may differ in each specialization.

Treatment of vectors of length one 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 one in the setup code and then use `setSize` in the run-time code.

15.3.4 Control of setup outputs

Sometimes setup code may create variables that are not used in run code. By default, NIMBLE inspects run code and omits variables from setup that do not appear in run code from compila-

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

tion. However, sometimes a programmer may want to force a numeric or character variable to be included in compilation, even if it is not used directly in run 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 during compilation, for example `X` and `Y`, simply include

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

anywhere in the setup code.

15.4 Writing run code

In Chapter 11 we described the functionality of the NIMBLE language that could be used in run code without setup code (typically in cases where no models or modelValues are needed). Next we explain the additional features that allow use of models and modelValues in the run code.

15.4.1 Driving models: *calculate*, *calculateDiff*, *simulate*, *getLogProb*

These four functions are the primary ways to operate a model. Their syntax was explained in Section 13.3. Except for `getLogProb`, it is usually important for the `nodes` vector to be sorted in topological order. Model member functions such as `getDependencies` and `expandNodeNames` will always return topoligically sorted node names.

Most R-like indexing of a node vector is allowed within the argument to `calculate`, `calculateDiff`, `simulate`, and `getLogProb`. For example, all of the following are allowed:

```
myModel$calculate(nodes)
myModel$calculate(nodes[i])
myModel$calculate(nodes[1:3])
myModel$calculate(nodes[c(1,3)])
myModel$calculate(nodes[2:i])
myModel$calculate(nodes[ values(model, nodes) + 0.1 < x ])
```

Note that in the last line of code, one must have that the length of `nodes` is equal to that of `values(model, nodes)`, which means that all the nodes in `nodes` must be scalar nodes.

Also note that one cannot create new vectors of nodes in run code. They can only be indexed within a call to `calculate`, `calculateDiff`, `simulate` or `getLogProb`.

15.4.2 Getting and setting variable and node values

15.4.2.1 Using indexing with nodes

Here is an example that illustrates getting and setting of nodes, subsets of nodes, or variables. Note the following:

- In `model[[v]]`, `v` can only be a single node or variable name, not a vector of multiple nodes nor an element of such a vector (`model[[ nodes[i] ]]` does not work). The node itself may be a vector, matrix or array node.
- In fact, `v` can be a node-name-like character string, even if it is not actually a node in the model. See example 4 in the code below.

- One can also use `model$varName`, with the caveat that `varName` must be a variable name. This usage would only make sense for a nimbleFunction written for models known to have a specific variable. (Note that if `a` is a scalar node in `model`, then `model[['a']]` will be a scalar but `model$a` will be a vector of length -)
- one should use the `<<-` global assignment operator to assign into model nodes.

Note that NIMBLE does not allow variables to change dimensions. Model nodes are the same, and indeed are more restricted because they can't change sizes. In addition, NIMBLE distinguishes between scalars and vectors of length 1. These rules, and ways to handle them correctly, are illustrated in the following code as well as in Section 11.3.

```
code <- nimbleCode({
    z ~ dnorm(0, sd = sigma)
    sigma ~ dunif(0, 10)
    y[1:n] ~ dmnorm(zeroes[1:n], cov = C[1:5, 1:5])
})
n <- 5
m <- nimbleModel(code, constants = list(n = n, zeroes = rep(0, n),
                                        C = diag(n)))
cm <- compileNimble(m)

nfGen <- nimbleFunction(
    setup = function(model) {
    # node1 and node2 would typically be setup arguments, so they could
    # have different values for different models.  We are assigning values
    # here so the example is clearer.
        node1 <- 'sigma'            # a scalar node
        node2 <- 'y[1:5]'           # a vector node
        notReallyANode <- 'y[2:4]'  # y[2:4] allowed even though not a node!
    },
    run = function(vals = double(1)) {
        tmp0 <- model[[node1]]           # 1. tmp0 will be a scalar
        tmp1 <- model[[node2]]           # 2. tmp1 will be a vector
        tmp2 <- model[[node2]][1]        # 3. tmp2 will be a scalar
        tmp3 <- model[[notReallyANode]]  # 4. tmp3 will be a vector
        tmp4 <- model$y[3:4]      # 5. hard-coded access to a model variable
        # 6. node1 is scalar so can be assigned a scalar:
        model[[node1]] <<- runif(1)
        model[[node2]][1] <<- runif(1)
                 # 7. an element of node2 can be assigned a scalar
        model[[node2]] <<- runif(length(model[[node2]]))
                 # 8. a vector can be assigned to the vector node2
        model[[node2]][1:3] <<- vals[1:3]
             # elements of node2 can be indexed as needed
        returnType(double(1))
        out <- model[[node2]] # we can return a vector
        return(out)
    }
    )

Rnf <- nfGen(m)
Cnf <- compileNimble(Rnf)
Cnf$run(rnorm(10))
```

```
## [1]  0.474092599  1.193691666 -0.116395472  0.627634118  0.005961335
```

Use of `[[ ]]` allows one to programmatically access a node based on a character variable containing the node name; this character variable would generally be set in setup code. In contrast, use of `$` hard codes the variable name and would not generally be suitable for nimbleFunctions intended for use with arbitrary models.

15.4.2.2 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 do the inverse. In both cases, values from nodes with two 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
```

One can also index elements of nodes in the argument to values, in the same manner as discussed for `calculate` and related functions in Section 15.4.1.

Note again the potential for confusion between scalars and vectors of length 1. `values` returns a vector and expects a vector when used on the left-hand side of an assignment. If only a single value is being assigned, it must be a vector of length 1, not a scalar. This can be achieved by wrapping a scalar in `c()` when necessary. For example:

```
# c(rnorm(1)) creates vector of length one:
values(model, nodes[1]) <- c(rnorm(1))
# won't compile because rnorm(1) is a scalar
# values(model, nodes[1]) <- rnorm(1)

out <- values(model, nodes[1]) # out is a vector
out2 <- values(model, nodes[1])[1] # out2 is a scalar
```

15.4.3 Getting parameter values and node bounds

Sections 13.2.3-13.2.4 describe how to get the parameter values for a node and the range (bounds) of possible values for the node using getParam and getBound. Both of these can be used in run code.

15.4.4 Using modelValues objects

The modelValues structure was introduced in Section 14.1. Inside nimbleFunctions, modelValues are designed to easily save values from a model object during the running of a nimbleFunction. A modelValues object used in run code must always exist in the setup code, either by passing it in as a setup argument or creating it in the setup code.

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

```
# Accepting modelValues as a setup argument
swConf <- modelValuesConf(vars = "w",
                                    types = "double",
                                    sizes = 1)
setup = function(propModelValues, model, savedWeightsConf){
    # Building a modelValues in the setup function
    savedWeights <- modelValues(conf = savedWeightsConf)
    # List of nodes to be used in run function
    modelNodes <- model$getNodeNames(stochOnly = TRUE,
                                     includeData = FALSE)
}
```

The simplest way to pass values back and forth between models and modelValues inside of a nimbleFunction is with copy, which has the synonym nimCopy. 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 in Section 14.1. However the function as.matrix does not work in run code.

Here is a run function to use these modelValues:

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

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

Once the nimbleFunction is built, the modelValues object can be accessed using $, which is shown in more detail below. In fact, since modelValues, like most NIMBLE objects, are reference class objects, one can get a reference to them before the function is executed and then use that reference afterwards.

```
# simple model and modelValues for example use with code above
targetModelCode <- nimbleCode({
    x ~ dnorm(0,1)
    for(i in 1:4)
        y[i] ~ dnorm(0,1)
})

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

# creating the models
targetModel = nimbleModel(targetModelCode, check = FALSE)
propModel = nimbleModel(propModelCode, check = FALSE)
cTargetModel = compileNimble(targetModel)
cPropModel = compileNimble(propModel)


sampleMVConf = modelValuesConf(vars = c("x", "y", "propLL"),
    types = c("double", "double", "double"),
    sizes = list(x = 1, y = 4, propLL = 1) )

sampleMV <- modelValues(sampleMVConf)

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

# nimbleFunction for calculating importance weights
# uses setup and run functions as defined in previous code chunk
impWeights_Gen <- nimbleFunction(setup = setup,
                                 run = run)


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

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

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

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

# retrieving the modelValues objects
# extracted objects are C-based modelValues objects

savedPropSamp_1 = CImpWeights$propModelValues
savedPropSamp_2 = CPropSamp$mv

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

savedPropSamp_1["x",1]
```

```
## [1] -0.3998956
```

```
savedPropSamp_2["x",1]
```

```
## [1] -0.3998956
```

```
savedPropSamp_1["x",1] <- 0 # example of directly setting a value
savedPropSamp_2["x",1]
```

```
## [1] 0
```

```
# viewing the saved importance weights
savedWeights <- CImpWeights$savedWeights
unlist(savedWeights[["w"]])
```

```
##  [1] 0.2673310 1.3081190 0.9378951 0.5024644 1.7765357 3.5155732 0.3893390
##  [8] 0.5530012 0.5085069 2.3890876
```

```
# viewing first 3 rows -- note that savedPropSsamp_1["x", 1] was altered
as.matrix(savedPropSamp_1)[1:3, ]
```

```
##      propLL[1]      x[1]       y[1]        y[2]       y[3]        y[4]
## [1,] -3.689026  0.000000  0.4941960  0.62275962 -0.1610471  0.09637918
## [2,] -6.864741  1.303486 -0.9937620 -0.03470697  0.5615520 -0.99986562
## [3,] -6.199326 -1.265670  0.9577035 -0.52875091 -0.7335897 -0.02557756
```

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.

15.4.5 Using model variables and modelValues in expressions

Each way of accessing a variable, node, or modelValues can be used amidst 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$z` or `model[["z"]]`, NIMBLE will try to do the right thing by treating this as a scalar. In the event of problems[5], a more explicit way to access `z` is `model$z[1]` or `model[["z"]][1]`.

15.4.6 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.

[5] Please tell us!

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

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

```
## sharedValues = 1
##
## sharedValues = 2
```

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

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

```
## sharedValues = 1
##
## sharedValues = 2
```

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

15.4.7 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 $.

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

usePreviousDemo1 <- usePreviousDemo(2)
usePreviousDemo1$run(1:10)
```

```
## 2
## sharedValues = 2
##
## sharedValues = 3
##
## 15 30 45 60 75
```

```
## [1] 30
```

```
CusePreviousDemo1 <- compileNimble(usePreviousDemo1)
CusePreviousDemo1$run(1:10)
```

```
## 2
## sharedValues = 2
##
## sharedValues = 3
##
## 15
## 30
## 45
## 60
## 75
```

```
## [1] 30
```

15.4.8 Virtual nimbleFunctions and nimbleFunctionLists

Often it is useful for one nimbleFunction to have a list of other nimbleFunctions, all of whose methods have the same arguments and return types. For example, NIMBLE's MCMC engine 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 1.0.0 it is limited to simple, single-level inheritance.

Here is how it works:

```r
baseClass <- nimbleFunctionVirtual(
    run = function(x = double(1)) {returnType(double())},
    methods = list(
        foo = function() { returnType(double()) }
    ))

derived1 <- nimbleFunction(
    contains = baseClass,
    setup = function(){},
    run = function(x = double(1)) {
        print("run 1")
        return(sum(x))
        returnType(double())
    },
    methods = list(
        foo = function() {
            print("foo 1")
            return(rnorm(1, 0, 1))
            returnType(double())
        }
    )
)

derived2 <- nimbleFunction(
    contains = baseClass,
    setup = function(){},
    run = function(x = double(1)) {
        print("run 2")
        return(prod(x))
        returnType(double())
    },
    methods = list(
        foo = function() {
            print("foo 2")
            return(runif(1, 100, 200))
            returnType(double())
        }
    )
)

useThem <- nimbleFunction(
    setup = function() {
        nfl <- nimbleFunctionList(baseClass)
        nfl[[1]] <- derived1()
        nfl[[2]] <- derived2()
    },
    run = function(x = double(1)) {
        for(i in seq_along(nfl)) {
            print( nfl[[i]]$run(x) )
            print( nfl[[i]]$foo() )
        }
    }
    )

useThem1 <- useThem()
set.seed(1)
useThem1$run(1:5)
```

```
## run 1
## 15
## foo 1
## -0.6264538
## run 2
## 120
## foo 2
## 157.2853
```

```
CuseThem1 <- compileNimble(useThem1)
set.seed(1)
CuseThem1$run(1:5)
```

```
## run 1
## 15
## foo 1
## -0.626454
## run 2
## 120
## foo 2
## 157.285
```

One can also use `seq_along` with nimbleFunctionLists (and only with nimbleFunctionLists). As in R, `seq_along(myFunList)` is equivalent to `1:length(myFunList)` if the length of `myFunList` is greater than zero. It is an empty sequence if the length is zero.

Virtual nimbleFunctions cannot define setup values to be inherited.

15.4.9 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.

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++.

15.4.10 User-defined data structures

Before the introduction of nimbleLists in Version 0.6-4, NIMBLE did not explicitly have user-defined data structures. An alternative way to create a data structure in NIMBLE is to 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:

```
dataNF <- nimbleFunction(
    setup = function() {
        X <- 1
        Y <- as.numeric(c(1, 2))
        Z <- matrix(as.numeric(1:4), nrow = 2)
        setupOutputs(X, Y, Z)
    })

useDataNF <- nimbleFunction(
    setup = function(myDataNF) {},
    run = function(newX = double(), newY = double(1), newZ = double(2)) {
        myDataNF$X <<- newX
        myDataNF$Y <<- newY
        myDataNF$Z <<- newZ
    })

myDataNF <- dataNF()
myUseDataNF <- useDataNF(myDataNF)
myUseDataNF$run(as.numeric(100), as.numeric(100:110),
                matrix(as.numeric(101:120), nrow = 2))
myDataNF$X
```

```
## [1] 100
```

```
myDataNF$Y
```

```
##  [1] 100 101 102 103 104 105 106 107 108 109 110
```

```
myDataNF$Z
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]  101  103  105  107  109  111  113  115  117   119
## [2,]  102  104  106  108  110  112  114  116  118   120
```

```
myUseDataNF$myDataNF$X
```

```
## [1] 100
```

```
nimbleOptions(buildInterfacesForCompiledNestedNimbleFunctions = TRUE)
CmyUseDataNF <- compileNimble(myUseDataNF)
CmyUseDataNF$run(-100, -(100:110), matrix(-(101:120), nrow = 2))
CmyDataNF <- CmyUseDataNF$myDataNF
CmyDataNF$X
```

```
## [1] -100
```

```
CmyDataNF$Y
```

```
##  [1] -100 -101 -102 -103 -104 -105 -106 -107 -108 -109 -110
```

```
CmyDataNF$Z
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,] -101 -103 -105 -107 -109 -111 -113 -115 -117  -119
## [2,] -102 -104 -106 -108 -110 -112 -114 -116 -118  -120
```

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.

15.5 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) {
        # proposal standard deviation
        scale <- if(!is.null(control$scale)) control$scale else 1
        calcNodes <- model$getDependencies(target)
    },

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

        # 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 exisiting 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:

- The nimbleFunction must include the contains statement `contains = sampler_BASE`.
- The `setup` function must have the 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.
- The nimbleFunction must 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`. It is good practice to specify default values for any control parameters that are not provided in the `control` argument, as done in the example. 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)`.

The purpose of the `mvSaved` modelValues object is to store the state of the model, including both node values and log probability values, as it exists before any changes are made by the sampler. This allows restoration of the state when a proposal is rejected, as can be seen in the example above. When a proposal is accepted, one should copy from the model into the `mvSaved` object. NIMBLE's MCMC engine expects that `mvSaved` contains an up-to-date copy of model values and logProb values at the end of the run code of a sampler.

Note that NIMBLE generally expects the user-defined sampler to be defined in the global environment. If you define it in a function (which would generally be the case if you are using it in the context of parallelization), one approach would be to assign the user-defined sampler to the global environment in your function:

```
assign('sampler_yourSampler', sampler_yourSampler, envir = .GlobalEnv)
```

15.5.1 User-defined samplers and posterior predictive nodes

As of version 0.13.0, NIMBLE's handling of posterior predictive nodes in MCMC sampling has changed in order to improve MCMC mixing. Samplers for nodes that are not posterior predictive nodes no longer condition on the values of the posterior predictive nodes. This is done by two changes:

- turning off posterior predictive nodes as dependencies in the use of `getDependencies` when building an MCMC, and
- moving all posterior predictive samplers to be last in the order of samplers used in an MCMC iteration.

The first change calls for careful consideration when writing new samplers. It is done by making `buildMCMC` set a NIMBLE system option (described below) that tells `getDependencies` to ignore posterior predictive nodes (defined as nodes that are themselves not data and have no data nodes in their entire downstream (descendant) dependency network) before it builds the samplers (i.e., runs the setup code of each sampler). That way, the setup code of all samplers that use `getDependencies` will automatically comply with the new system.

User-defined samplers that determine dependencies using `getDependencies` should automatically work in the new system (although it is worth checking). However if a user-defined sampler manually specifies dependencies, and if those dependencies include posterior predictive nodes, the MCMC results could be incorrect. Whether incorrect results occur could depend on other parts of the model structure connected to node(s) sampled by the user-defined sampler (i.e., the target node(s)) and/or posterior predictive nodes. Therefore, it is strongly recommended that all user-defined samplers rely on `getDependencies` to determine which nodes depend on the target node(s).

There are several new NIMBLE system options available to take control of the behavior just described.

- `getDependenciesIncludesPredictiveNodes` determines whether posterior predictive nodes are included in results of `getDependencies`. This defaults to `TRUE`, so that outside of building samplers, the behavior of `getDependencies` should be identical to previous versions of NIMBLE.
- `MCMCusePredictiveDependenciesInCalculations` gives the value to which `getDependenciesIncludesPredictiveNodes` will be set while building samplers. This defaults to `FALSE`.
- `MCMCorderPosteriorPredictiveSamplersLast` determines whether posterior predictive samplers are moved to the end of the sampler list. This default to `TRUE`. Behavior prior to version 0.13.0 corresponds to `FALSE`.

Here are some examples of how to use these options:

- If you want to have MCMC samplers condition on posterior predictive nodes, do `nimbleOptions(MCMCusePredictiveDependenciesInCalculations = TRUE)`.
- If you want to get identical MCMC samplers, including their ordering, as in versions prior to 0.13.0, do `nimbleOptions(MCMCusePredictiveDependenciesInCalculations = TRUE)` and `nimbleOptions(MCMCorderPosteriorPredictiveSamplersLast = FALSE)`.
- If you want to experiment with the behavior of `getDependencies` when ignoring posterior predictive nodes, do `nimbleOptions(getDependenciesIncludesPredictiveNodes = FALSE)`.

15.6 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 models 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`.

15.7 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, e.g., `debug(mynf$run)` 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. We often resort to using `print` statements for debugging compiled code. Expert users fluent in C++ may also try setting `nimbleOptions(pauseAfterWritingFiles = TRUE)` and adding debugging code into the generated C++ files.

15.8 Timing nimbleFunctions with *run.time*

If your nimbleFunctions are correct but slow to run, you can use benchmarking tools to look for bottlenecks and to compare different implementations. If your functions are very long-running (say 1ms or more), then standard R benchmarking tools may suffice, e.g. the `microbenchmark` package

```
library(microbenchmark)
microbenchmark(myCompiledFunVersion1(1.234),
               myCompiledFunVersion2(1.234))  # Beware R <--> C++ overhead!
```

If your nimbleFunctions are very fast, say under 1ms, then `microbenchmark` will be inaccurate due to R-to-C++ conversion overhead (that won't happen in your actual functions). To get timing information in C++, NIMBLE provides a `run.time` function that avoids the R-to-C++ overhead.

```
myMicrobenchmark <- compileNimble(nimbleFunction(
    run = function(iters = integer(0)){
        time1 <- run.time({
            for (t in 1:iters) myCompiledFunVersion1(1.234)
        })
        time2 <- run.time({
            for (t in 1:iters) myCompiledFunVersion2(1.234)
        })
        return(c(time1, time2))
        returnType(double(1))
    }))
print(myMicroBenchmark(100000))
```

15.9 Clearing and unloading compiled objects

Sometimes it is useful to clear all the compiled objects from a project and unload the shared library produced by your C++ compiler. To do so, you can use `clearCompiled(obj)` where `obj` is a compiled object such as a compiled model or nimbleFunction (e.g., a compiled MCMC algorithm). This will clear *all* compiled objects associated with your NIMBLE project. For example, if `cModel` is a compiled model, `clearCompiled(cModel)` will clear both the model and all associated nimbleFunctions such as compiled MCMCs that use that model. Be careful, use of `clearCompiled` can be dangerous. There is some risk that if you have copies of the R objects that interfaced to compiled C++ objects that have been removed, and you attempt to use those R objects after clearing their compiled counterparts, you will crash R. We have tried to minimize that risk, but we can't guarantee safe behavior.

15.10 Reducing memory usage

NIMBLE can create a lot of objects in its processing, and some of them use R features such as reference classes that are heavy 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 ony appear within other nimbleFunctions. If you want access to all such nimbleFunctions, use the option `buildInterfacesForCompiledNestedNimbleFunctions = TRUE`. This will use more memory but can be useful for debugging.

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.

Part V

Automatic Derivatives in NIMBLE

Chapter 16

Automatic Derivatives

As of version 1.0.0, NIMBLE can automatically provide numerically accurate derivatives of potentially arbitrary order for most calculations in models and/or nimbleFunctions. This feature enables methods such as Hamiltonian Monte Carlo (HMC, see package `nimbleHMC`), Laplace approximation, and fast optimization with methods that use function gradients.

Automatic (or algorithmic) differentiation (AD) refers to the method of carrying derivative information through a set of mathematical operations. When done this way, derivatives are numerically accurate to the precision of the computer. This is distinct from finite difference methods, used by R packages such as `numDeriv` (Gilbert and Varadhan, 2019) and `pracma` (Borchers, 2022), which approximate derivatives by calculating function values at extremely nearby points. Finite difference methods are slower and less accurate than AD. It is also distinct from writing separate functions for each derivative, which can become very complicated and sometimes slower than AD. NIMBLE uses the CppAD package (Bell, 2022) as its AD engine, following TMB's success in doing so (Kristensen et al., 2016). A general reference on AD is Griewank and Walther (2008).

Using a packaged AD algorithm should be as simple as setting `buildDerivs=TRUE` in the model. On the other hand, writing new algorithms (as nimbleFunctions) that use AD requires understanding how the AD system works internally, including what can go wrong. Calls to `compileNimble` that include AD features will result in slower C++ compilation.

We will introduce NIMBLE's AD features step by step, from simply turning them on for a model to using them in your own nimbleFunctions. We will show you:

1. how to turn on derivatives in a model and use them in Laplace approximation (for Hamiltonian Monte Carlo (HMC), see 7.11.2).
2. how to modify user-defined functions and distributions to support derivatives.
3. what functions are supported and not supported for derivatives.
4. basics of obtaining derivatives in your own algorithms written as `nimbleFunctions`.
5. advanced methods of obtaining derivatives in `nimbleFunctions`, including *double-taping*.
6. how to get derivatives involving model calculations.
7. automatic parameter transformations to give any model an unconstrained parameter space for algorithms to work in.
8. an example showing use of nimble's derivatives for maximum likelihood estimation.

First, make sure to set the option to enable derivative features. This should be `TRUE` by default, but just in case:

```
nimbleOptions(enableDerivs = TRUE)
```

16.1 How to turn on derivatives in a model

To allow algorithms to use automatic derivatives for a model, include `buildDerivs=TRUE` in the call to `nimbleModel`. If you want derivatives to be set up for all models, you can run `nimbleOptions(buildModelDerivs = TRUE)` and omit the `buildDerivs` argument.

We'll re-introduce the simple Poisson Generalized Linear Mixed Model (GLMM) example model from 7.11.2.1 use Laplace approximation on it. There will be 10 groups (`i`) of 5 observations (`j`) each. Each observation has a covariate, `X`, and each group has a random effect `ran_eff`. Here is the model code:

```
model_code <- nimbleCode({
  # priors
  intercept ~ dnorm(0, sd = 100)
  beta ~ dnorm(0, sd = 100)
  sigma ~ dunif(0, 10)
  # random effects and data
  for(i in 1:10) {
    # random effects
    ran_eff[i] ~ dnorm(0, sd = sigma)
    for(j in 1:5) {
      # data
      y[i,j] ~ dpois(exp(intercept + beta*X[i,j] + ran_eff[i]))
    }
  }
})
```

We'll simulate some values for `X`.

```
set.seed(123)
X <- matrix(rnorm(50), nrow = 10)
```

Next, we build the model, including `buildDerivs=TRUE`.

```
model <- nimbleModel(model_code, constants = list(X = X), calculate = FALSE,
                     buildDerivs = TRUE) # Here is the argument needed for AD.
```

16.1.1 Finish setting up the GLMM example

As preparation for the Laplace examples below, we need to finish setting up the GLMM. We could have provided data in the call to `nimbleModel`, but instead we will simulate it using the model itself. Specifically, we will set parameter values, simulate data values, and then set those as the data to use.

```
model$intercept <- 0
model$beta <- 0.2
```

```
model$sigma <- 0.5
model$calculate() # This will return NA because the model is not fully initialized.
```

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

```
model$simulate(model$getDependencies('ran_eff'))
model$calculate() # Now the model is fully initialized: all nodes have valid values.
```

```
## [1] -80.74344
```

```
model$setData('y') # Now the model has y marked as data, with values from simulation.
```

If you are not very familiar with using a `nimble` model, that might have been confusing, but it was just for setting up the example.

Finally, we will make a compiled version of the model.

```
Cmodel <- compileNimble(model)
```

16.2 How to use Laplace approximation

Next we will show how to use nimble's Laplace approximation, which uses derivatives internally, to get maximum (approximate) likelihood estimates for the GLMM model above. What Laplace approximation approximates in this context is the integral over continuous random effects needed to calculate the likelihood. Hence, it gives an approximate likelihood (often quite accurate) that can be used for maximum likelihood estimation. Note that the Laplace approximation uses second derivatives, and the gradient of the Laplace approximation (used for finding the MLE efficiently) uses third derivatves. These are described in detail by Skaug and Fournier (2006) and Fournier et al. (2012).

To create a Laplace approximation specialized to the parameters of interest for this model, we use the nimbleFunction `buildLaplace`. For many models, the setup code in `buildLaplace` will automatically determine the random effects to be integrated over and the associated nodes to calculate. In fact, if you omit the parameter nodes, it will assume that all top-level nodes in the model should be treated as parameters. If fine-grained control is needed, these various sets of nodes can be input directly into `buildLaplace`. To see what default handling of nodes is being done for your model, use `setupMargNodes` with the same node inputs as `buildLaplace`.

```
glmm_laplace <- buildLaplace(model, c('intercept','beta','sigma'))
Cglmm_laplace <- compileNimble(glmm_laplace, project = model)
```

Now we are ready to use some of the methods provided by `buildLaplace`. These include calculating the Laplace approximation for some input parameter values, calculating its gradient, and maximizing the Laplace-approximated likelihood.

```
# Get the Laplace approximation for one set of parameter values.
Cglmm_laplace$calcLaplace(c(0, 0, 1))
```

```
## [1] -65.57246
```

```
Cglmm_laplace$gr_Laplace(c(0, 0, 1)) # Get the corresponding gradient.
```

```
## [1] -1.866842  8.001648 -4.059556
```

```r
MLE <- Cglmm_laplace$findMLE(c(0, 0, 1)) # Find the (approximate) MLE.
MLE$par     # MLE parameter values
```

```
## [1] -0.1492313  0.1934101  0.5703648
```

```r
MLE$value   # MLE log likelihood value
```

```
## [1] -63.44875
```

The final outputs show the MLE for `intercept`, `beta`, and `sigma`, followed by the maximum (approximate) likelihood.

More information about the MLE can be obtained in two ways. The `summary` method can give estimated and standard errors as well as the variance-covariance matrix for the parameters and/or the random effects. The `summaryLaplace` function returns similar information but with names included in a more useful way. For example:

```r
Cglmm_laplace$summary(MLE)$randomEffects$estimates
```

```
##  [1] -0.33711223 -0.02963214  0.40581611  1.04780122 -0.36729920  0.26915416
##  [7] -0.54949741 -0.11866631  0.10009139 -0.04408944
```

```r
summaryLaplace(Cglmm_laplace, MLE)$params
```

```
##             estimate        se
## intercept -0.1492313 0.2465005
## beta       0.1934101 0.1467229
## sigma      0.5703648 0.2066583
```

`buildLaplace` actually offers several choices in how computations are done, differing in how they use *double taping* for derivatives or not. In some cases one or another choice might be more efficient. See `help(buildLaplace)` if you want to explore it further.

Finally, let's confirm that it worked by comparing to results from package `lme4`. In this case, `lme4` will be faster because it is highly specialized to GLMMs. Here our interest is in checking that nimble's Laplace approximation worked correctly in a case where we have an established tool such as `lme4`.

```r
library(lme4)
y <- as.numeric(model$y) # Rearrage inputs for call to glmer
X <- as.numeric(X)
group <- rep(1:10, 5)
lme4_fit <- glmer(y ~ X + (1 | group), family = poisson)
```

```
## Warning in checkConv(attr(opt, "derivs"), opt$par, ctrl = control$checkConv, :
## Model failed to converge with max|grad| = 0.0178573 (tol = 0.002, component 1)
```

```r
# The warning above seems to be innocuous.
summary(lme4_fit)
```

```
## Generalized linear mixed model fit by maximum likelihood (Laplace
##   Approximation) [glmerMod]
##  Family: poisson  ( log )
## Formula: y ~ X + (1 | group)
## 
##      AIC      BIC   logLik deviance df.resid 
##    132.9    138.6    -63.4    126.9       47 
## 
## Scaled residuals: 
##      Min       1Q   Median       3Q      Max 
## -1.02386 -0.74101 -0.03475  0.33823  2.46453 
## 
## Random effects:
##  Groups Name        Variance Std.Dev.
##  group  (Intercept) 0.3285   0.5731  
## Number of obs: 50, groups:  group, 10
## 
## Fixed effects:
##              Estimate Std. Error z value Pr(>|z|)    
## (Intercept) -0.153720   0.003913 -39.289   <2e-16 ***
## X            0.193464   0.143528   1.348    0.178    
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Correlation of Fixed Effects:
##   (Intr)
## X -0.002
## optimizer (Nelder_Mead) convergence code: 0 (OK)
## Model failed to converge with max|grad| = 0.0178573 (tol = 0.002, component 1)
```

```
logLik(lme4_fit)
```

```
## 'log Lik.' -63.44896 (df=3)
```

The results match within numerical tolerance typical of optimization problems. Specifically, the coefficients for `(Intercept)` and `X` match nimble's `Intercept` and `beta`, the random effects standard deviation for `group` matches nimble's `sigma`, and the maximum likelihood values match.

16.3 How to support derivatives in user-defined functions and distributions

It is possible to activate derivative support for a user-defined function or distribution used in a model. Simply set `buildDerivs=TRUE` in `nimbleFunction`.

Here is an extremely toy example. Let's say we want a model with one node that follows a user-defined distribution, which will happen to be the same as `dnorm` (a normal distribution) for illustration.

The model code is:

```
toyCode <- nimbleCode({
  x ~ d_my_norm(mean = mu, sd = sigma)
  mu ~ dunif(-10, 10)
  sigma ~ dunif(0, 5)
})
```

The user-defined distribution is:

```
d_my_norm <- nimbleFunction(
  run = function(x = double(), mean = double(), sd = double(),
                 log = integer(0, default = 0)) {
    ans <- -log(sqrt(2*pi*sd)) - 0.5*((x-mean)/sd)^2
    if(log) return(ans)
    return(exp(ans))
    returnType(double())
  },
  buildDerivs=TRUE
)
```

Now we can build the model with `buildDerivs=TRUE` and compile it:

```
# Don't worry about the warnings from nimbleModel in this case.
toyModel <- nimbleModel(toyCode, inits = list(mu = 0, sigma = 1, x = 0),
                        buildDerivs = TRUE)
```

Now `toyModel` can be used in algorithms such as HMC and Laplace approximation, as above, or new ones you might write, as below.

16.4 What operations are and aren't supported for AD

Much of the math supported by NIMBLE will work for AD. Here are some details on what is and isn't supported, as well as on some options to control how linear algebra operations are handled.

Features that are not supported for AD include:

- stochastic indexing (i.e. the index itself in a latent state) in models.
- cumulative ("p") and quantile ("q") functions for distributions.
- truncated distributions in models (because they use cumulative distribution functions).
- random number generation.
- specific distributions including `dcat`, `dcar_normal`, `dcar_proper`, and `dinterval`.
- some specific functions including `step`, `%%` (mod), `eigen`, `svd`, and `bessel_k`.
- specific model operations including `model$calculateDiff`, `model$simulate`, `model$getBound`, and `model$getParam`.

Some details on what is supported include:

- `round`, `floor`, `ceil`, and `trunc` are all supported with derivatives defined to be zero everywhere.
- `pow(a, b)` requires positive `a` and `b`.
- A new function `pow_int(a, b)` returns `pow(a, round(b))` and thus sets all derivatives with respect to `b` to zero. This allows valid derivatives with respect to `a` even if it takes a negative value.

For the linear algebra functions %*%, chol, forwardsolve, backsolve, and inverse, there are special extensions provided by NIMBLE for CppAD called (in CppAD terms) "atomics". By default, these atomics will be used and often improve efficiency. There may be cases where they decrease efficiency, which might include when the matrix operands are small or contain many zeros. To compare results with and without use of the atomics, they can be turned off with a set of options:

```
nimbleOptions(useADmatMultAtomic = FALSE) # for %*%
nimbleOptions(useADcholAtomic = FALSE)    # for chol
nimbleOptions(useADmatInverseAtomic = FALSE) # for inverse
nimbleOptions(useADsolveAtomic = FALSE)   # for forwardsolve and backsolve
```

When a linear algebra atomic is turned off, the AD system simply uses all the scalar operations that compose the linear algebra operation.

Another important feature of AD is that sometimes values get "baked in" to AD calculations, meaning they are used *and permanently retained* from the first set of calculations and then can't have their value changed later (unless an algorithm does a "reset", described below). For people writing user-defined distributions and functions, a brief summary of what can get baked in includes:

- the extent of any for-loops.
- values of any arguments that are not of type 'double', including e.g. the 'log' argument in d_my_norm. ('d' functions called from models always have log = TRUE, so in that case it is not a problem.)
- the evaluation path followed by any if-then-else calls.
- values of any indices.

See below for more thorough explanations.

16.5 Basics of obtaining derivatives in nimbleFunctions

Now that we have seen a derivative-enabled algorithm work, let's see how to obtain derivatives in methods you write. From here on, this part of the manual is oriented towards algorithm developers. We'll start by showing how derivatives work in nimbleFunctions *without* using a model. The AD system allows you to obtain derivatives of one function or method from another.

Let's get derivatives of the function y = exp(-d * x) where x is a vector, d is a scalar, and y is a vector.

```
derivs_demo <- nimbleFunction(
  setup = function() {},
  run = function(d = double(), x = double(1)) {
    return(exp(-d*x))
    returnType(double(1))
  },
  methods = list(
    derivsRun = function(d = double(), x = double(1)) {
      wrt <- 1:(1 + length(x)) # total length of d and x
      return(derivs(run(d, x), wrt = wrt, order = 0:2))
      returnType(ADNimbleList())
    }
  ),
  buildDerivs = 'run'
)
```

Here are some things to notice:

- Having `setup` code allows the nimbleFunction to have multiple methods (i.e. to behave like a class definition in object-oriented programming). Some nimbleFunctions don't have `setup` code, but `setup` code is required when there will be a call to `derivs`. If, as here, you don't need the `setup` to do anything, you can simply use `setup=TRUE`, which is equivalent to `setup=function(){}`.
- Any arguments to `run` that are real numbers (i.e. regular double precision numbers, but not integers or logicals) will have derivatives tracked when called through `derivs`.
- The "with-respect-to" (`wrt`) argument gives indices of the arguments for which you want derivatives. In this case, we're including all elements of `d` and `X`. The indices form one sequence over all arguments.
- `order` is a vector of derivative orders to return, which can include any of `0` (the function value, not a derivative!), `1` (1st order), or `2` (2nd order). Higher-order derivatives can be obtained by double-taping, described below.
- Basics of `nimbleFunction` programming are covered in Chapters 11-15. These include type declarations (`double()` for scalar, `double(1)` for vector), the distinction between `setup` code and `run` code, and how to write more methods (of which `run` is simply a default name when there is only one method).
- `derivs` can alternatively be called `nimDerivs`. In fact the former will be converted to the latter internally.

Let's see this work.

```
my_derivs_demo <- derivs_demo()
C_my_derivs_demo <- compileNimble(my_derivs_demo)
d <- 1.2
x <- c(2.1, 2.2)
C_my_derivs_demo$derivsRun(d, x)
```

```
## nimbleList object of type NIMBLE_ADCLASS
## Field "value":
## [1] 0.08045961 0.07136127
## Field "jacobian":
##            [,1]        [,2]        [,3]
## [1,] -0.1689652 -0.09655153  0.00000000
## [2,] -0.1569948  0.00000000 -0.08563352
## Field "hessian":
## , , 1
##
##           [,1]      [,2] [,3]
## [1,] 0.3548269 0.1222986    0
## [2,] 0.1222986 0.1158618    0
## [3,] 0.0000000 0.0000000    0
##
```

```
## , , 2
##
##           [,1] [,2]      [,3]
## [1,] 0.3453885    0 0.1170325
## [2,] 0.0000000    0 0.0000000
## [3,] 0.1170325    0 0.1027602
```

We can see that using `order = 0:2` results in the the value ("0th" order result, i.e. the value returned by `run(d, x)`), the Jacobian (matrix of first order derivatives), and the Hessian (array of second order derivatives). The `run` function here has taken three inputs (in the order `d`, `x[1]`, `x[2]`) and returned two outputs (the first and second elements of `exp(-d*x)`).

The i-th Jacobian row contains the first derivatives of the i-th output with respect to `d`, `x[1]`, and `x[2]`, in that order. The first and second indices of the Hessian array follow the same ordering as the columns of the Jacobian. The third index of the Hessian array is for the output index. For example, `hessian[3,1,2]` is the second derivative of the second output value (third index = 2) with respect to `x[2]` (first index = 3) and `d` (second index = 1).

(Although it may seem inconsistent to have the output index be first for Jacobian and last for Hessian, it is consistent with some standards for how these objects are created in other packages and used mathematically.)

When a function being called for derivatives (`run` in this case) has non-scalar arguments (`x` in this case), the indexing of inputs goes in order of arguments, and in column-major order within arguments. For example, if we had arguments `x = double(1)` and `z = double(2)` (a matrix), the inputs would be ordered as `x[1]`, `x[2]`, …, `z[1, 1]`, `z[2, 1]`, …, `z[1, 2]`, `z[2, 2]`, …, etc.

16.5.1 Checking derivatives with uncompiled execution

Derivatives can also be calculated in uncompiled execution, but they will be much slower and less accurate: slower because they are run in R, and less accurate because they use finite element methods (from packages `pracma` and/or `numDeriv`). Uncompiled execution is mostly useful for checking that compiled derivatives are working correctly, because although they are slower and less accurate, they are also much simpler internally and thus provide good checks on compiled results. For example:

```
my_derivs_demo$derivsRun(d, x)
```

```
## nimbleList object of type NIMBLE_ADCLASS
## Field "value":
## [1] 0.08045961 0.07136127
## Field "jacobian":
##            [,1]        [,2]        [,3]
## [1,] -0.1689652 -0.09655153  0.00000000
## [2,] -0.1569948  0.00000000 -0.08563352
## Field "hessian":
## , , 1
##
##           [,1]      [,2] [,3]
## [1,] 0.3548269 0.1222986    0
## [2,] 0.1222986 0.1158618    0
## [3,] 0.0000000 0.0000000    0
##
## , , 2
##
##           [,1] [,2]      [,3]
## [1,] 0.3453885    0 0.1170325
## [2,] 0.0000000    0 0.0000000
## [3,] 0.1170325    0 0.1027602
```

We can see that the results are very close, but typically not identical, to those from the compiled version.

16.5.2 Holding some local variables out of derivative tracking

Sometimes one wants to omit tracking of certain variables in derivative calculations. Here is an example. Say that we write `exp(-d*x)` using a for-loop instead of a vectorized operation as above. It wouldn't make sense to track derivatives for the for-loop index (`i`), and indeed it would cause `compileNimble` to fail. The code to use a for-loop while telling `derivs` to ignore the for-loop index is:

```
derivs_demo2 <- nimbleFunction(
  setup = function() {},
  run = function(d = double(), x = double(1)) {
    ans <- numeric(length = length(x))
    for(i in 1:length(x))
      ans[i] <- exp(-d*x[i])
    return(ans)
    returnType(double(1))
  },
  methods = list(
    derivsRun = function(d = double(), x = double(1)) {
      wrt <- 1:(1 + length(x)) # total length of d and x
      return(derivs(run(d, x), wrt = wrt, order = 0:2))
      returnType(ADNimbleList())
    }
  ),
  buildDerivs = list(run = list(ignore = 'i'))
)
```

We can see that it gives identical results as above, looking at only the Jacobian to keep the output short.

```
my_derivs_demo2 <- derivs_demo2()
C_my_derivs_demo2 <- compileNimble(my_derivs_demo2)
d <- 1.2
x <- c(2.1, 2.2)
C_my_derivs_demo2$derivsRun(d, x)$jacobian
```

```
##            [,1]        [,2]       [,3]
## [1,] -0.1689652 -0.09655153 0.00000000
## [2,] -0.1569948  0.00000000 -0.08563352
```

One might think it should be obvious that `i` should not be involved in derivatives, but sometimes math is actually done with a for-loop index. Another way to ensure derivatives won't be tracked for `i` would be to write `i <- 1L`, before the loop. By assigning a definite integer to `i`, it will be established as definitely of integer type throughout the function and thus not have derivatives tracked. (If you are not familiar with the `L` in R, try `storage.mode(1L)` vs `storage.mode(1)`.)

Because derivatives are not tracked for integer or logical variables, arguments to `run` that have `integer` or `logical` types will not have derivatives tracked.

Note the more elaborate value for `buildDerivs`. Above, this was just a character vector. Here it is a named list, with each element itself being a list of control values. The control value `ignore` is a character vector of variable names to ignore in derivative tracking. `buildDerivs = 'run'` is equivalent to `buildDerivs = list(run = list())`.

16.5.3 Using AD with multiple nimbleFunctions

Derivatives will be tracked through whatever series of calculations occur in a method, possibly including calls to other methods or nimbleFunctions that have `buildDerivs` set. Let's look at an example where we have a separate function to return the element-wise square root of an input vector. The net calculation for derivatives will be `sqrt(exp(-d*x)))`.

```
nf_sqrt <- nimbleFunction(
  run = function(x = double(1)) {
    return(sqrt(x))
    returnType(double(1))
  },
  buildDerivs = TRUE
)

derivs_demo3 <- nimbleFunction(
  setup = function() {},
  run = function(d = double(), x = double(1)) {
    ans <- exp(-d*x)
    ans <- nf_sqrt(ans)
    return(ans)
    returnType(double(1))
  },
  methods = list(
    derivsRun = function(d = double(), x = double(1)) {
      wrt <- 1:(1 + length(x)) # total length of d and x
      return(derivs(run(d, x), wrt = wrt, order = 0:2))
      returnType(ADNimbleList())
    }
  ),
  buildDerivs = 'run'
)
```

And then let's see it work:

```
my_derivs_demo3 <- derivs_demo3()
C_my_derivs_demo3 <- compileNimble(my_derivs_demo3)
d <- 1.2
x <- c(2.1, 2.2)
C_my_derivs_demo3$derivsRun(d, x)$jacobian
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.2978367 -0.1701924  0.0000000
## [2,] -0.2938488  0.0000000 -0.1602812
```

Note that for a nimbleFunction without setup code, one can say `buildDerivs=TRUE`, `buildDerivs = 'run'`, or `buildDerivs = list(run = list())`. One can't take derivatives of `nf_sqrt` on its own, but it can be called by a method of a nimbleFunction that can have its derivatives taken (e.g. the `run` method of a `derivs_demo3` object).

16.5.4 Understanding more about how AD works: *taping* of operations

At this point, it will be helpful to grasp more of how AD works and its implementation in nimble via the CppAD library (Bell, 2022). AD methods work by following a set of calculations multiple times, sometimes in reverse order.

For example, the derivative of $y = \exp(x^2)$ can be calculated (by the chain rule of calculus) as the derivative of $y = \exp(z)$ (evaluated at $z = x^2$) times the derivative of $z = x^2$ (evaluated at x). This can be calculated by first going through the sequence of steps for the value, i.e. (i) $z = x^2$, (ii) $y = \exp(z)$, and then again for the derivatives, i.e. (i) $dz = 2x(dx)$, (ii) $dy = \exp(z)dz$. These steps determine the instantaneous change dy that results from an instantaneous change dx. (It may be helpful to think of both values as a function of a third variable such as t. Then we are determining dy/dt as a function of dx/dt.) In CppAD, the basic numeric object type (double) is replaced with a special type and corresponding versions of all the basic math operations so that those operations can track not just the values but also derivative information.

The derivative calculations typically need the values. For example, the derivative of $\exp(z)$ is (also) $\exp(z)$, for which the value of z is needed, which in this case is x^2, which is one of the steps in calculating the value of $\exp(x^2)$. (In fact, in this case the derivative of $\exp(z)$ can more simply use the value of y itself.) Hence, values are calculated before derivatives, which are calculated by stepping through the calculation sequence a *second* time

What was just described is called *forward* mode in AD. The derivatives can also be calculated by working through $\exp(x^2)$ in *reverse* mode. This is often less familiar to graduates of basic calculus courses. In reverse mode, one determines what instantaneous change dx would give an instantaneous change dy. For example, given a value of dy, we can calculate $1/dz = \exp(z)/dy$ and then $(1/dx) = (x^2)/dz$. Again, values are calculated first, followed by derivatives in a second pass through the calculation sequence, this time in reverse order.

In general, choice of forward mode versus reverse mode has to do with the lengths of inputs and outputs and possibly specific operations involved. In nimble, some reasonable choices are used.

In CppAD, the metaphor of pre-digital technology – magnetic tapes – is used for the set of operations in a calculation. When a line with `nimDerivs` is first run, the operations in the function it calls are *taped* and then re-used – *played* – in forward and/or reverse orders to obtain derivatives. Hence we will refer to the AD *tape* in explaining some features below. It is also possible to reset (i.e. re-record) the AD tape used for a particular `nimDerivs` call.

When taped operations are played, the resulting operations can themselves be taped. We call this *meta-taping* or *double-taping*. It is useful because it can sometimes boost efficiency and it can be used to get third and higher order derivatives.

16.5.5 Resetting a `nimDerivs` call

The first time a call to `nimDerivs` runs, it records a tape of the operations in its first argument (`run(d, x)` in the examples here). On subsequent calls, it re-uses that tape without re-recording it. That means subsequent calls are usually much faster. *It also means that subsequent calls **must** use the same sized arguments as the recorded call.* For example, we can use `C_my_derivs_demo$derivsRun` with new arguments of the same size(s):

```
C_my_derivs_demo$derivsRun(-0.4, c(3.2, 5.1))
```

```
## nimbleList object of type NIMBLE_ADCLASS
## Field "value":
## [1] 3.596640 7.690609
## Field "jacobian":
##           [,1]     [,2]     [,3]
## [1,] -11.50925 1.438656 0.000000
## [2,] -39.22211 0.000000 3.076244
## Field "hessian":
## , , 1
##
##           [,1]       [,2] [,3]
## [1,] 36.829591 -8.2003386    0
## [2,] -8.200339  0.5754624    0
## [3,]  0.000000  0.0000000    0
##
## , , 2
##
##           [,1] [,2]       [,3]
## [1,] 200.03275    0 -23.379452
## [2,]   0.00000    0   0.000000
## [3,] -23.37945    0   1.230497
```

However, if we call `C_my_derivs_demo$derivsRun` with `length(x)` different from 2, the result will be garbage. If we need to change the size of the arguments, we need to re-record the tape. This is done with the `reset` argument.

Here is a slightly more general version of the `derivs_demo` allowing a user to reset the tape. It also takes `order` and `wrt` as arguments instead of hard-coding them.

```
derivs_demo4 <- nimbleFunction(
  setup = function() {},
  run = function(d = double(), x = double(1)) {
    return(exp(-d*x))
    returnType(double(1))
  },
  methods = list(
    derivsRun = function(d = double(), x = double(1),
                         wrt = integer(1), order = integer(1),
                         reset = logical(0, default=FALSE)) {
      return(derivs(run(d, x), wrt = wrt, order = order, reset = reset))
      returnType(ADNimbleList())
    }
  ),
  buildDerivs = 'run'
)
```

Now we will illustrate the use of `reset`. To make shorter output, we'll request only the Jacobian.

```
my_derivs_demo4 <- derivs_demo4()
C_my_derivs_demo4 <- compileNimble(my_derivs_demo4)
d <- 1.2
x <- c(2.1, 2.2)
# On the first call, reset is ignored because the tape must be recorded.
C_my_derivs_demo4$derivsRun(d, x, 1:3, 1)
```

```
## nimbleList object of type NIMBLE_ADCLASS
## Field "value":
## numeric(0)
## Field "jacobian":
##            [,1]        [,2]        [,3]
## [1,] -0.1689652 -0.09655153  0.00000000
## [2,] -0.1569948  0.00000000 -0.08563352
## Field "hessian":
## <0 x 0 x 0 array of double>
##
```

```
# On the second call, reset=FALSE, so the tape is re-used.
C_my_derivs_demo4$derivsRun(-0.4, c(3.2, 5.1), 1:3, 1)
```

```
## nimbleList object of type NIMBLE_ADCLASS
## Field "value":
## numeric(0)
## Field "jacobian":
##           [,1]     [,2]     [,3]
## [1,] -11.50925 1.438656 0.000000
## [2,] -39.22211 0.000000 3.076244
## Field "hessian":
## <0 x 0 x 0 array of double>
##
```

```
# If we need a longer X, we need to say reset=TRUE
C_my_derivs_demo4$derivsRun(1.2, c(2.1, 2.2, 2.3), 1:4, 1, reset=TRUE)
```

```
## nimbleList object of type NIMBLE_ADCLASS
## Field "value":
## numeric(0)
## Field "jacobian":
##            [,1]        [,2]        [,3]        [,4]
## [1,] -0.1689652 -0.09655153  0.00000000  0.00000000
## [2,] -0.1569948  0.00000000 -0.08563352  0.00000000
## [3,] -0.1455711  0.00000000  0.00000000 -0.07595012
## Field "hessian":
## <0 x 0 x 0 array of double>
##
```

```
# Now we can use with reset=FALSE with X of length 3
C_my_derivs_demo4$derivsRun(-0.4, c(3.2, 5.1, 4.5), 1:4, 1)
```

```
## nimbleList object of type NIMBLE_ADCLASS
## Field "value":
## numeric(0)
## Field "jacobian":
##           [,1]     [,2]     [,3]     [,4]
## [1,] -11.50925 1.438656 0.000000 0.000000
## [2,] -39.22211 0.000000 3.076244 0.000000
## [3,] -27.22341 0.000000 0.000000 2.419859
## Field "hessian":
## <0 x 0 x 0 array of double>
##
```

You could also make multiple instances of `derivs_demo4` and keep track of the argument sizes currently recorded for each one.

Other important situations where you need to reset a tape are when values that are otherwise "baked in" to an AD tape need to be changed. By "baked in", we mean that some values are permanently part of a tape until it is reset.

16.5.5.1 What gets baked into AD tapes until `reset=TRUE`

Specifically, you need to reset a tape when:

- the extent of any for-loops change. For-loops are not natively recorded, but rather each operation done in execution of the for-loop is recorded. (This is sometimes called "loop unrolling".) Therefore, in `for(i in start:end)`, the values of `start` and `end` are in effect baked into a tape.
- the outcomes of any if-then-else conditions in a nimbleFunction are changed.
- values of any `integer` or `logical` inputs change.
- any other member data used in calculations changes. It is possible to create a variable in the `setup` function and use it in methods such as `run`. Such variables become member data in the terminology of object-oriented programming. Such variables will not have derivatives tracked if used in a function while it is being taped, so the values will be baked into the tape. This can be useful if you understand what is happening or confusing if not.

Uncompiled execution of `derivs` ignores the reset argument.

16.5.6 A note on performance benchmarking

If you are interested in measuring the performance of AD in nimble, please remember that the first call to `nimDerivs` and any subsequent call with `reset=TRUE` will be slower, often much slower, that subsequent calls.

16.6 Advanced uses: double taping

Suppose you are interested only in the Jacobian, not in the value, and/or want only some elements of the Jacobian. You might still need the value, but obtaining the value alone from an AD tape (i.e. from `derivs`) will be slower than obtaining it by simply calling the function. On the other hand, AD methods need to calculate the value before calculating first order derivatives, so the value will be calculated anyway. However, in some cases, some of the steps of value calculations aren't really needed if one only wants, say, first-order derivatives. In addition, if not all elements of the Jacobian are wanted, then some unnecessary calculations will be done internally that one might want to avoid.

A way to cut out unnecessary calculations is to record a tape of a tape, which we call double taping. Let's see an example before explaining further.

```
derivs_demo5 <- nimbleFunction(
  setup = function() {},
  run = function(d = double(), x = double(1)) {
    return(exp(-d*x))
    returnType(double(1))
  },
  methods = list(
    jacobian_run_wrt_d  = function(d = double(), x = double(1),
                         wrt = integer(1),
                         reset = logical(0, default=FALSE)) {
      ans <- derivs(run(d, x), wrt = wrt, order = 1, reset = reset)
      jac <- ans$jacobian[,1] # derivatives wrt 'd' only
      return(jac)
      returnType(double(1))
    },
    derivsJacobian = function(d = double(), x = double(1),
                              wrt = integer(1),
                              order = integer(1),
                              reset = logical(0, default = FALSE)) {
      innerWrt <- nimInteger(value = 1, length = 1)
      ans <- nimDerivs(jacobian_run_wrt_d(d, x, wrt = innerWrt, reset = reset),
                       wrt = wrt, order = order, reset = reset)
      return(ans)
      returnType(ADNimbleList())
    }
  ),
  buildDerivs = c('run', 'jacobian_run_wrt_d')
)
```

What is happening in this code? `jacobian_run_wrt_d` is a method that returns part of the Jacobian of `run(d, x)`, specifically the first column, which contains derivatives with respect to `d`. It happens to use AD to do it, but otherwise it is just some function with inputs `d` and `x` and a vector output. (Arguments that are integer or logical do not have derivatives tracked.) `derivsJacobian` calculates derivatives of `jacobian_run_wrt_d`. This means that the `value` returned by `derivsJacobian` will be the value of `jacobian_run_wrt_d`, which comprises the first derivatives of `run` with respect to `d`. The `jacobian` returned by `derivsJacobian` will contain second derivatives of `run` with respect to `d` and each of `d`, `x[1]`, and `x[2]`. And the `hessian` returned by `derivsJacobian` will contain some *third* derivatives of `run`. Notice that `buildDerivs` now includes `jacobian_run_wrt_d`.

Let's see this in use.

```
my_derivs_demo5 <- derivs_demo5()
C_my_derivs_demo5 <- compileNimble(my_derivs_demo5)
d <- 1.2
x <- c(2.1, 2.2)
C_my_derivs_demo5$derivsJacobian(d, x, wrt = 1:3, order = 0:2)
```

```
## nimbleList object of type NIMBLE_ADCLASS
## Field "value":
## [1] -0.1689652 -0.1569948
## Field "jacobian":
##           [,1]      [,2]      [,3]
## [1,] 0.3548269 0.1222986 0.0000000
## [2,] 0.3453885 0.0000000 0.1170325
## Field "hessian":
## , , 1
##
##             [,1]        [,2] [,3]
## [1,] -0.74513642 -0.08786189    0
## [2,] -0.08786189 -0.05020679    0
## [3,]  0.00000000  0.00000000    0
##
## , , 2
##
##            [,1] [,2]        [,3]
## [1,] -0.7598548    0 -0.10047667
## [2,]  0.0000000    0  0.00000000
## [3,] -0.1004767    0 -0.05480546
```

We can compare these results to those shown above from the same values of `d` and `x`. The `value` here is the same as `jacobian[1:2, 1]` above. The `jacobian` here is the same as `t(hessian[1, 1:3, 1:2])` or (by symmetry of second derivatives) `t(hessian[1:3, 1, 1:2])` above. And the `hessian[i,j,k]` here contains the third derivative of the k-th output of `run(d, x)` with respect to `d`, the i-th input, and the j-th input.

It is also possible to call `derivs` on (i.e. tape) a function containing a `derivs` call from which some 0-th and/or 2nd order derivatives are extracted. It is even possible to triple tape by calling `derivs` on a function calling `derivs` on a function calling `derivs`, and so on.

16.7 Derivatives involving model calculations

Obtaining derivatives involving model calculations takes some special considerations, and there are two ways to do it.

We will use the Poisson GLMM above as an example model for which we want derivatives. We can obtain derivatives with respect to any variables for all or any subset of model calculations.

16.7.1 Method 1: `nimDerivs` of `model$calculate`

Recall that `model$calculate(nodes)` returns the sum of the log probabilities of all stochastic nodes in `nodes`. Deterministic calculations are also executed; they contribute 0 to the sum of probabilities but may be needed for inputs to subsequent calculations. Calculations are done in the order of `nodes`, which should be a valid order for the model, often obtained from `model$getDependencies`.

The simplest way to get derivatives for model calculations is to use `model$calculate` as the function taped by `derivs` (here shown by its alternative name `nimDerivs` for illustration).

```
derivs_nf <- nimbleFunction(
  setup = function(model, with_respect_to_nodes, calc_nodes) {},
  run = function(order = integer(1),
                 reset = logical(0, default=FALSE)) {
    ans <- nimDerivs(model$calculate(calc_nodes), wrt = with_respect_to_nodes,
                     order = order, reset = reset)
    return(ans)
    returnType(ADNimbleList())
  }
)
```

In `derivs_nf`:

- The mere presence of `model`, `with_respect_to_nodes`, and `calc_nodes` as `setup` arguments makes them available as member data for `run` or other methods.
- `model` will be a model object returned from `nimbleModel`.
- `with_respect_to_nodes` will be the names of nodes we want derivatives with respect to.
- `calc_nodes` will be the nodes to be calculated, in the order given.
- `order` can contain any of `0` (value), `1` (1st order), or `2` (2nd order) derivatives requested, as above.
- `reset` should be TRUE if the AD tape should be reset (re-recorded), as above. There are additional situations when a tape for `model$calculate` should be reset, discussed below.

Thus, `nimDerivs(model$calculate(calc_nodes), wrt = with_respect_to_nodes, <other args>)` takes derivatives of a function whose inputs are the values of `with_respect_to_nodes` (using their values in the model object) and whose output is the summed log probability returned by `model$calculate(calc_nodes)`. The internal handling of this case is distinct from other calls to `nimDerivs`.

Now we can make an instance of `derivs_nf`, compile the model and nimbleFunction instance, and look at the results. We will assume the `model` was built as above in the Laplace example.

```
wrt_nodes <- c('intercept','beta', 'sigma')
calc_nodes <- model$getDependencies(wrt_nodes)
```

```
derivs_all <- derivs_nf(model, wrt_nodes, calc_nodes)
cModel <- compileNimble(model)
cDerivs_all <- compileNimble(derivs_all, project = model)
derivs_result <- cDerivs_all$run(order = 0:2)
derivs_result
```

```
## nimbleList object of type NIMBLE_ADCLASS
## Field "value":
## [1] -80.74344
## Field "jacobian":
##          [,1]       [,2]     [,3]
## [1,] -9.358104 -0.3637619 -5.81414
## Field "hessian":
## , , 1
##
##          [,1]      [,2]      [,3]
## [1,] -62.35820 -16.21705   0.00000
## [2,] -16.21705 -69.47074   0.00000
## [3,]   0.00000   0.00000 -45.11516
```

As above, using `order = 0:2` results in the the value (0th order), Jacobian (1st order), and Hessian (2nd order). The function `model$calculate` is organized here to have inputs that are the current values of `intercept`, `beta`, and `sigma` in the model. It has output that is the summed log probability of the `calc_nodes`. The Jacobian columns are first derivatives with respect to `intercept`, `beta` and `sigma`, respectively. The first and second indices of the Hessian array follow the same ordering. For example `derivs_result$hessian[2,1,1]` is the second derivative with respect to `beta` (first index = 2) and `intercept` (second index = 1).

In the case of `model$calculate`, the first index of the Jacobian and the last index of the Hessian are always 1 because derivatives are of the first (and only) output value.

The ordering of inputs is similar to that used above, such as for the arguments `d` and `x`, but in this case the inputs are model nodes. When non-scalar nodes such as matrix or array nodes are used as `with_respect_to_nodes`, the resulting elements of Jacobian columns and Hessian first and second indices will follow column-major order. For example, for a 2x2 matrix `m`, the element order would be `m[1, 1]`, `m[2, 1]`, `m[1, 2]`, `m[2, 2]`. This will usually be the same ordering as the names returned by `model$expandNodeNames("m", returnScalarElements=TRUE)`.

The actual values used as inputs are the current values in the compiled model object. These would typically be set by code like `values(model, with_respect_to_nodes) <<- my_values` before the call to `nimDerivs` in the above example.

As above, derivatives can also be calculated in uncompiled execution, but they will be much slower and less accurate. Uncompiled execution is mostly useful for checking that compiled derivatives are working correctly. This will use values in the *uncompiled* model object.

```
derivs_all$run(order = 0:2)
```

```
## nimbleList object of type NIMBLE_ADCLASS
## Field "value":
## [1] -80.74344
## Field "jacobian":
##           [,1]       [,2]     [,3]
## [1,] -9.358104 -0.3637619 -5.81414
## Field "hessian":
## , , 1
##
##           [,1]      [,2]      [,3]
## [1,] -62.35821 -16.21705   0.00000
## [2,] -16.21705 -69.47074   0.00000
## [3,]   0.00000   0.00000 -45.11517
```

We can see that all the results clearly match the compiled results to a reasonable numerical precision.

16.7.2 Method 2: `nimDerivs` of a method that calls `model$calculate`

Sometimes one needs derivatives of calculations done in a nimbleFunction as well as in a model. And sometimes one needs to change values in a model before and/or after doing model calculations and have those changes recorded in the derivative tape.

For these reasons, it is possible to take derivatives of a method that includes a call to `model$calculate`. Continuing the above example, say we want to take derivatives with respect to the log of `sigma`, so the input vector will be treated as `intercept`, `beta`, and `log(sigma)`, in that order. We will convert `log(sigma)` to `sigma` before using it in the model, and we want the derivative of that transformation included in the tape (i.e. using the chain rule).

(Using `log(sigma)` instead of `sigma` is useful so an algorithm such as optimization or HMC can use an unconstrained parameter space. A constraint such as `sigma > 0` can be difficult for many algorithms. See below for NIMBLE's automated parameter transformation system if you need to do this systematically.)

Here is a nimbleFunction to do that.

```
derivs_nf2 <- nimbleFunction(
  setup = function(model, wrt_nodes, calc_nodes) {
    derivsInfo <- makeModelDerivsInfo(model, wrt_nodes, calc_nodes)
    updateNodes <- derivsInfo$updateNodes
    constantNodes <- derivsInfo$constantNodes
    n_wrt <- length(wrt_nodes)
    # If wrt_nodes might contain non-scalar nodes, the more general way
    # to determine the length of all scalar elements is:
    # length(model$expandNodeNames(wrt_nodes, returnScalarComponents = TRUE))
  },
  run = function(x = double(1)) {
    x_trans <- x               # x[1:2] don't need transformation
    x_trans[3] <- exp(x[3])    # transformation of x[3]
    values(model, wrt_nodes) <<- x_trans # put inputs into model
    ans <- model$calculate(calc_nodes)   # calculate model
    return(ans)
    returnType(double(0))
  },
  methods = list(
    derivsRun = function(x = double(1),
                         order = integer(1),
                         reset = logical(0, default=FALSE)) {
      wrt <- 1:n_wrt
      ans <- nimDerivs(run(x), wrt = wrt, order = order, reset = reset,
                       model = model, updateNodes = updateNodes,
                       constantNodes = constantNodes)
      return(ans)
      returnType(ADNimbleList())
    }
  ),
  buildDerivs = list(run = list()) # or simply 'run' would work in this case
)
```

Let's see how this can be used.

```
derivs_all2 <- derivs_nf2(model, wrt_nodes, calc_nodes)
cDerivs_all2 <- compileNimble(derivs_all2, project = model)
params <- values(model, wrt_nodes)
params[3] <- log(params[3])
cDerivs_all2$derivsRun(params, order = 0:2)
```

```
## nimbleList object of type NIMBLE_ADCLASS
## Field "value":
## [1] -80.74344
## Field "jacobian":
##           [,1]       [,2]     [,3]
## [1,] -9.358104 -0.3637619 -2.90707
## Field "hessian":
## , , 1
##
##           [,1]      [,2]      [,3]
## [1,] -62.35820 -16.21705   0.00000
## [2,] -16.21705 -69.47074   0.00000
## [3,]   0.00000   0.00000 -14.18586
```

Notice that these results are the same as we saw above except for third elements, which represent derivatives with respect to `sigma` above and to `log(sigma)` here.

There are several important new points here:

- The only valid way to get values into the model that will be recorded on the AD tape is with `values(model, nodes) <<- some_values` as shown. Other ways such as `nimCopy` or `model[[node]] <<- some_value` are not currently supported to work with AD.
- The call to `nimDerivs` of `run` must be told some information about the model calculations that will be done inside of `run`. `model` is of course the model that will be used. `constantNodes` is a vector of node names whose values are needed for the calculations but are not expected to change until you use `reset = TRUE`. Values of these nodes will be baked into the AD tape

until `reset = TRUE`. `updateNodes` is a vector of node names whose values are needed for the calculations, might change between calls even with `reset = FALSE`, and are neither part of `wrt_nodes` nor a deterministic part of `calc_nodes`. The function `makeModelDerivsInfo`, as shown in the `setup` code, determines what is usually needed for `constantNodes` and `updateNodes`.

- In Method 1 above, the NIMBLE compiler automatically determines `constantNodes` and `updateNodes` using `makeModelDerivsInfo` based on the inputs to `nimDerivs(model$calculate(...),...)`.
- One can use double-taping, but if so *both* calls to `nimDerivs` need the `model`, `updateNodes`, and `constantNodes` arguments, which should normally be identical.

16.7.2.1 Advanced topic: more about `constantNodes` and `updateNodes`

In most cases, you can obtain `constantNodes` and `updateNodes` from `makeModelDerivsInfo` without knowing exactly what they mean. But for advanced uses and possible debugging needs, let's explore these arguments in more detail. The purpose of `constantNodes` and `updateNodes` is to tell the AD system about all nodes that will be needed for taped model calculations. Specifically:

- `updateNodes` and `constantNodes` are vectors of nodes names that together comprise any nodes that are necessary for `model$calculate(calc_nodes)` but are:
  - not in the `wrt` argument to `nimDerivs` (only relevant for Method 1), and
  - not assigned into the model using `values(model, nodes) <<- some_values` prior to `model$calculate` (only relevant for Method 2), and
  - not a deterministic node in `calc_nodes`.
- `updateNodes` includes node names satisfying those conditions whose values might change between uses of the tape regardless of the `reset` argument.
- `constantNodes` includes node names satisfying those conditions whose values will be baked into the tape (will not change) until the next call with `reset=TRUE`.

To fix ideas, say that we want derivatives with respect to `ran_eff[1]` for calculations of it and the data that depend on it, including any deterministic nodes. In other words, `wrt` will be `ran_eff[1]`, and `calc_nodes` will be:

```
model$getDependencies('ran_eff[1]')
```

```
##  [1] "ran_eff[1]"
##  [2] "lifted_exp_oPintercept_plus_beta_times_X_oBi_comma_j_cB_plus_ran_eff_oBi_cB_cP_L7[1, 
##  [3] "lifted_exp_oPintercept_plus_beta_times_X_oBi_comma_j_cB_plus_ran_eff_oBi_cB_cP_L7[1, 
##  [4] "lifted_exp_oPintercept_plus_beta_times_X_oBi_comma_j_cB_plus_ran_eff_oBi_cB_cP_L7[1, 
##  [5] "lifted_exp_oPintercept_plus_beta_times_X_oBi_comma_j_cB_plus_ran_eff_oBi_cB_cP_L7[1, 
##  [6] "lifted_exp_oPintercept_plus_beta_times_X_oBi_comma_j_cB_plus_ran_eff_oBi_cB_cP_L7[1, 
##  [7] "y[1, 1]"
##  [8] "y[1, 2]"
##  [9] "y[1, 3]"
## [10] "y[1, 4]"
## [11] "y[1, 5]"
```

(The "lifted" nodes here are for the `exp(intercept + beta*X[i,j] + ran_eff[i])`, i.e. the inputs to `dpois`. See chapters @ref(#cha-lightning-intro) and @ref(#cha-using-models) to learn about lifted nodes.)

In this case, the log probability of `ran_eff[1]` itself requires `sigma`. Since `sigma` is not in `wrt_nodes`, it is not assigned into the model by the line `values(model, wrt_nodes) <<- x_trans`. It is also not a deterministic node (or any node) in `calc_nodes`, so it must be included in `updateNodes` or `constantNodes`. Since it might change between calls, it should be included in `updateNodes`.

Next, notice that the stochastic node `y[1, 1]` in `calc_nodes` means that the log probability of `y[1, 1]` will be calculated, and this requires the actual value of `y[1, 1]`. This node is part of `calc_nodes` but is not a deterministic part of it, so it must be provided in either `updateNodes` or `constantNodes`. When data values will not be changed often, it is better to put those nodes in `constantNodes`, because that will be more efficient than putting them in `updateNodes`. (If the data values are changed, use `reset=TRUE`, which will reset values of all `constantNodes` in the tape.)

The function `makeModelDerivsInfo` inspects the model and determines the usual needs for `updateNodes` and `constantNodes`. By default, data nodes are put in `constantNodes`. Use `dataAsConstantNodes = FALSE` in `makeModelDerivsInfo` if you want them put in `updateNodes`.

Note that a deterministic node in `calc_nodes` will have its value calculated as part of the operations recorded in the AD tape, so it does not need to be included in `updateNodes` or `constantNodes`.

As usual in model-generic programming in NIMBLE, be aware of lifted nodes and their implications. Suppose in the Poisson GLMM we had used a precision parameterization for the random effects, with the changes in this code snippet:

```
precision ~ dgamma(0.01, 0.01)
# <other lines>
  ran_eff[i] ~ dnorm(0, tau = precision)
```

This would result in a lifted node for the standard deviation, calculated as `1/sqrt(precision)`. That lifted node is what would actually be used in `dnorm` for each `ran_eff[i]`. Now if `ran_eff[1]` is in `wrt_nodes`, the *lifted node* (but *not* `precision`) will be in `updateNodes` (as determined by `makeModelDerivsInfo`). If you then change the value of `precision`, you must be sure to calculate the lifted node before obtaining derivatives. Otherwise the value of the lifted node will correspond to the old value of precision. These considerations are not unique to AD but rather are part of model-generic programming (see chapter 15).

An example of model-generic programming to update any lifted nodes that depend on `precision` would be:

```
model$calculate(model$getDependencies('precision', determOnly=TRUE))
```

In Method 1 above, NIMBLE automatically uses `makeModelDerivsInfo` based on the code `nimDerivs(model$calculate(<args>), <args>)`. However, in Method 2, when `model$calculate(<args>)` is used in a method such as `run`, then a call to `nimDerivs(run(<args>), <args>)` requires the `model`, `updateNodes` and `constantNodes` to be provided. Hence, the two functions (`run` and `derivsRun`) must be written in coordination.

16.7.2.2 When do you need to reset a tape with model calculations?

The additional rule for when tapes need to be reset based on model calculations is:

- when values of any `constantNodes` are changed. Usually these will only be data nodes.

16.8 Parameter transformations

Many algorithms that in some way explore a parameter space are best used in an unconstrained parameter space. For example, there are a bunch of optimization methods provided in R's `optim`, but only one (`L-BFGS-B`) allows constraints on the parameter space. Similarly, HMC is implemented to work in an unconstrained parameter space.

NIMBLE provides a nimbleFunction to automatically create a transformation from original parameters to unconstrained parameters and the inverse transformation back to the original parameters. Denoting `x` as an original parameter and `g(x)` as the transformed parameter, the cases handled include:

- If scalar $x \in (0, \infty)$, then $g(x) = \log(x)$. For example, `x ~ dweibull` in a model means $x \in (0, \infty)$.
- If scalar $x \in (0, 1)$, then $g(x) = \text{logit}(x)$. For example, `x ~ dbeta` in a model means $x \in (0, 1)$
- If scalar $x \in (a, \infty)$, then $g(x) = \log(x - a)$.
- If scalar $x \in (-\infty, b)$, then $g(x) = -\log(b - x)$.
- If scalar $x \in (a, b)$, then $g(x) = \text{logit}((x - a)/(b - a))$. For example `x ~ dunif(a, b)` in a model means $x \in (a, b)$.
- If matrix `x[1:n, 1:n] ~ dwishart` or `x[1:n, 1:n] ~ dinvwishart` in a model, then $g(x) = \text{chol}(x)$ for non-diagonal elements of $\text{chol}(x)$ and $g(x) = \log(\text{chol}(x))$ for diagonal elements of $\text{chol}(x)$, where $\text{chol}(x)$ is the Cholesky decomposition of x. Note that x in these cases follows a Wishart or inverse Wishart distribution, respectively, and thus is a random precision or covariance matrix. That means it must be positive definite and thus have positive diagonal elements of $\text{chol}(x)$.
- If vector `x[1:n] ~ ddirch` in a model, then $g(x[1]) = \text{logit}(x[1])$ and $g(x[i]) = \text{logit}(x[i]/(1 - \sum_{k=1}^{i-1} x[i]))$ for $i > 1$. In this case, `x` follows a Dirichlet distribution and thus has the simplex constraint that $\sum_{i=1}^{n} x[i] = 1$.
- If vector `x[1:n] ~ dlkj_corr_cholesky`, $g(x)$ is the LKJ Cholesky transformation. This provides the LKJ prior for a covariance matrix.

Note that the scalar transformations are all monotonic increasing functions of the original parameter.

The worked example next will illustrate use of the parameter transformation system.

Chapter 17

Example: maximum likelihood estimation using `optim` with gradients from `nimDerivs`.

In this example, we will obtain maximum likelihood estimates of a Poisson GLM. It will be similar to the GLMM above, but without the random effects. Maximization of a likelihood function can be done with methods that don't require gradients, but it is much faster to use methods that do use gradients.

To illustrate the parameter transformation system, we will give `p` a uniform prior, constrained to values between 0 and 1. The maximum likelihood estimation below will not use the prior probability, but nevertheless the prior will be interpreted for its constraints on valid values in this case. This need for parameter transformation is somewhat artifical in the interest of simplicity.

```
model_code <- nimbleCode({
  # priors (will be ignored for MLE)
  p ~ dunif(0, 1)
  log_p <- log(p)
  beta ~ dnorm(0, sd = 100)
  # random effects and data
  for(i in 1:50) {
    # data
    y[i] ~ dpois(exp(log_p + beta*X[i]))
  }
})

set.seed(123)
X <- rnorm(50) # Simulate values for X

model <- nimbleModel(model_code, constants = list(X = X), calculate = FALSE,
                     buildDerivs = TRUE) # Create nimble model object
```

As in the first example above, we will simulate data using the model itself. One can instead provide data as an argument to `nimbleModel`.

```
model$p <- 0.5      # Assign parameter values and simulate data (y).
model$beta <- 0.2
model$calculate()  # We expect this to be NA because there are no y values yet.
```

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

```
model$simulate(model$getDependencies(c('beta', 'p'), self = FALSE)) # Simulate y values.
model$calculate()  # Now we expect a valid result.
```

```
## [1] -53.33015
```

```
model$setData('y') # Now the model has y marked as data, with values from simulation.
Cmodel <- compileNimble(model) # Make compiled version.
```

Now that the example model is set up, we'll write a nimbleFunction to provide the negative log likelihood and its gradient using an unconstrained parameter space. The default behavior of `optim` below is to do a minimization problem, so returning the negative log likelihood means that `optim` will find the maximum likelihood parameters.

```
logLikelihood_nf <- nimbleFunction(
  setup = function(model, paramNodes) {
    # Determine nodes for calculating the log likelihood for parameters given by
    # paramNodes, ignoring any priors.
    calcNodes <- model$getDependencies(paramNodes, self = FALSE)
    # Set up the additional arguments for nimDerivs involving model$calculate
    derivsInfo <- makeModelDerivsInfo(model, paramNodes, calcNodes)
    updateNodes <- derivsInfo$updateNodes
    constantNodes <- derivsInfo$constantNodes
    # Create a parameter transformation between original and unconstrained
    # parameter spaces.
    transformer <- parameterTransform(model, paramNodes)
  },
  methods = list(
    neg_logLikelihood_p = function(p = double(1)) {
      # Put values in model and calculate negative log likelihood.
      values(model, paramNodes) <<- p
      return(-model$calculate(calcNodes))
      returnType(double())
    },
    neg_logLikelihood = function(ptrans = double(1)) {
      # Objective function for optim,
      # using transformed parameter space.
      p <- transformer$inverseTransform(ptrans)
      return(neg_logLikelihood_p(p))
      returnType(double())
    },
    gr_neg_logLikelihood = function(ptrans = double(1)) {
      # Gradient of neg log likelihood
      p <- transformer$inverseTransform(ptrans)
      d <- derivs(neg_logLikelihood_p(p), wrt = 1:length(p), order = 1,
                  model = model, updateNodes = updateNodes,
                  constantNodes = constantNodes)
      return(d$jacobian[1,])
      returnType(double(1))
    },
    transform = function(p = double(1)) {
      # Give user access to the transformation ...
      return(transformer$transform(p))
      returnType(double(1))
    },
    inverse = function(ptrans = double(1)) { # ... and its inverse.
      return(transformer$inverseTransform(ptrans))
      returnType(double(1))
    }
  ),
  buildDerivs = 'neg_logLikelihood_p'
)
```

Sometimes you might want a call to `transformer$transform` or `transformer$inverseTransform` to be included in a derivative calculation. In this case, we don't.

Next we build and compile an instance of `logLikelihood_nf` specialized (by the 'setup" function) to our model and its parameters.

```
ll_glm <- logLikelihood_nf(model, c("p", "beta"))
Cll_glm <- compileNimble(ll_glm, project = model)
```

Now let's see how the parameter transformation works for this model. For `beta`, it already has an unconstrained range of valid values, so it has no transformation (i.e. it has an indentity transformation). For `p`, it is constrained beween 0 and 1, so it will be logit-transformed.

```
Cll_glm$transform(c(0.2, 1))
```

```
## [1] -1.386294  1.000000
```

```
# The inverse transformation goes back to the original parameters, within rounding:
Cll_glm$inverse(c(-1.386294, 1))
```

```
## [1] 0.2000001 1.0000000
```

Now we are ready to use the methods provided in `Cll_glm` in a call to R's `optim` to find the MLE.

```
MLE <- optim(c(0,0), # initial values
         fn = Cll_glm$neg_logLikelihood, # function to be minimized
         gr = Cll_glm$gr_neg_logLikelihood, # gradient of function to be minimized
         method = "BFGS") # optimization method to use
MLE$par
```

```
## [1] 0.08257249 0.25577134
```

```
-MLE$value
```

```
## [1] -47.73243
```

Those outputs show the MLE in the parameters `logit(p)`, and `beta`, following by the maximum log likelihood value.

Now let's confirm that it's working correctly by comparing to results from `glm`.

```
glm_fit <- glm(I(model$y) ~ X, family = poisson)
coef(glm_fit)
```

```
## (Intercept)           X
##  -0.6527680   0.2557751
```

```
logLik(glm_fit)
```

```
## 'log Lik.' -47.73243 (df=2)
```

We see that the coefficient for `X` matches `beta` (within numerical tolerance), and the maximum log likelihoods match. To understand the intercept, we need to be careful about the parameter transformation used in nimble. The inverse transformation will give `p`, and the log of that should match the intercept estimated by `glm`.

```
Cll_glm$inverse(MLE$par)
```

```
## [1] 0.5206314 0.2557713
```

```
log(Cll_glm$inverse(MLE$par)[1]) # This should match glm's intercept
```

```
## [1] -0.652713
```

It looks like it worked.

Bibliography

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

Ariyo, O., Quintero, A., Muñoz, J., Verbeke, G., and Lesaffre, E. (2020). Bayesian model selection in linear mixed models for longitudinal data. *Journal of Applied Statistics*, 47(5):890–913.

Banerjee, S., Carlin, B., and Gelfand, A. (2015). *Hierarchical Modeling and Analysis for Spatial Data*. Chapman & Hall, Boca Raton, 2 edition.

Bell, B. (2022). CppAD: A Package for Differentiation of C++ Algorithms.

Blackwell, D. and MacQueen, J. (1973). Ferguson distributions via Pólya urn schemes. *The Annals of Statistics*, 1:353–355.

Borchers, H. W. (2022). *pracma: Practical Numerical Math Functions*. R package version 2.3.8.

Escobar, M. D. (1994). Estimating normal means with a Dirichlet process prior. *Journal of the American Statistical Association*, 89:268–277.

Escobar, M. D. and West, M. (1995). Bayesian density estimation and inference using mixtures. *Journal of the American Statistical Association*, 90:577–588.

Ferguson, T. S. (1973). A Bayesian analysis of some nonparametric problems. *Annals of Statistics*, 1:209–230.

Ferguson, T. S. (1974). Prior distribution on the spaces of probability measures. *Annals of Statistics*, 2:615–629.

Fournier, D. A., Skaug, H. J., Ancheta, J., Ianelli, J., Magnusson, A., Maunder, M. N., Nielsen, A., and Sibert, J. (2012). AD Model Builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models. *Optimization Methods and Software*, 27(2):233–249.

Gelfand, A. E. and Kottas, A. (2002). A computational approach for full nonparametric bayesian inference under dirichlet process mixture models. *Journal of Computational and Graphical Statistics*, 11(2):289–305.

Gelman, A., Hwang, J., and Vehtari, A. (2014). Understanding predictive information criteria for Bayesian model. *Statistics and Computing*, 24(6):997–1016.

George, E., Makov, U., and Smith, A. (1993). Conjugate likelihood distributions. *Scandinavian Journal of Statistics*, 20(2):147—156.

Gilbert, P. and Varadhan, R. (2019). *numDeriv: Accurate Numerical Derivatives.* R package version 2016.8-1.1.

Green, P. J. (1995). Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82(4):711–732.

Griewank, A. and Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition.* Society for Industrial and Applied Mathematic, Philadelphia, PA, second edition edition.

Hug, J. E. and Paciorek, C. J. (2021). A numerically stable online implementation and exploration of WAIC through variations of the predictive density, using NIMBLE. Technical report, arXiv preprint.

Ishwaran, H. and James, L. F. (2001). Gibbs sampling methods for stick-breaking priors. *Journal of the American Statistical Association*, 96(453):161–173.

Ishwaran, H. and James, L. F. (2002). Approximate Dirichlet process computing in finite normal mixtures: smoothing and prior information. *Journal of Computational and Graphical Statistics*, 11:508–532.

Kristensen, K., Nielsen, A., Berg, C. W., Skaug, H., and Bell, B. M. (2016). TMB: Automatic differentiation and Laplace approximation. *Journal of Statistical Software*, 70(5):1–21.

Lewandowski, D., Kurowicka, D., and Joe, H. (2009). Generating random correlation matrices based on vines and extended onion method. *Journal of multivariate analysis*, 100(9):1989–2001.

Lo, A. Y. (1984). On a class of Bayesian nonparametric estimates I: Density estimates. *The Annals of Statistics*, 12:351–357.

Lunn, D., Spiegelhalter, D., Thomas, A., and Best, N. (2009). The BUGS project: Evolution, critique and future directions. *Statistics in Medicine*, 28(25):3049—3067.

Neal, R. (2000). Markov chain sampling methods for Dirichlet process mixture models. *Journal of Computational and Graphical Statistics*, 9:249–265.

Neal, R. M. (2003). Slice sampling. *The Annals of Statistics*, 31(3):705–741.

Neal, R. M. (2011). MCMC Using Hamiltonian Dynamics. In *Handbook of Markov Chain Monte Carlo*, Chapman & Hall/CRC Handbooks of Modern Statistical Methods. Chapman and Hall/CRC.

Paciorek, C. (2009). Understanding intrinsic Gaussian Markov random field spatial models, including intrinsic conditional autoregressive models. Technical report, University of California, Berkeley.

Roberts, G. O. and Sahu, S. K. (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.

Rue, H. and Held, L. (2005). *Gaussian Markov Random Fields: Theory and Applications.* Chapman & Hall, Boca Raton.

Sethuraman, J. (1994). A constructive definition of Dirichlet prior. *Statistica Sinica*, 2:639–650.

Skaug, H. J. and Fournier, D. A. (2006). Automatic approximation of the marginal likelihood in non-Gaussian hierarchical models. *Computational Statistics & Data Analysis*, 51(2):699–709.

Stan Development Team (2021a). Stan language functions reference manual. Technical report.

Stan Development Team (2021b). Stan language reference manual. Technical report.

Vehtari, A., Gelman, A., and Gabry, J. (2017). Practical bayesian model evaluation using leave-one-out cross-validation and waic. *Statistics and Computing*, 27(5):1413–1432.

Watanabe, S. (2010). Asymptotic equivalence of Bayes cross validation and widely applicable information criterion in singular learning theory. *Journal of Machine Learning Research*, 11(Dec):3571–3594.