NIMBLE User Manual

NIMBLE Development Team

Version 0.6-10

http://R-nimble.org
https://github.com/nimble-dev/nimble-docs

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 0.6-10, 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 the wiki of 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.

[1]See Chapter **??** for information about NIMBLE's version of BUGS.

3. A language, called NIMBLE, embedded within and similar in style to R, for writing algorithms that operate on models written in BUGS.
4. A compiler that generates C++ for your models and algorithms, compiles that C++, and lets you use it seamlessly from R without knowing anything about C++.

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 **??**.

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++.

Part II

Models in NIMBLE

Part III

Algorithms in NIMBLE

Chapter 2

Bayesian nonparametric models

As of version 0.6-11, NIMBLE provides initial support for Bayesian nonparametric (BNP) mixture modeling. These features are currently considered EXPERIMENTAL – please let us know (via the NIMBLE user mailing list or by emailing us) if you have any problems or have suggestions about functionality you would like NIMBLE to support or the NIMBLE interface for using BNP.

2.1 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 [4, 5, 8, 2, 3]. More specifically, a DP mixture model takes the form

$$y_i \mid G \overset{iid}{\sim} \int h(y_i \mid \theta) G(d\theta), (1)$$

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

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 model (1) (I would like to add this a a reference to (1)).

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 [1], discussed in Section 2.2. The model takes the form

$$y_i \mid \tilde{\boldsymbol{\theta}}, z_i \overset{ind}{\sim} h(\cdot \mid \tilde{\theta}_{z_i}), (2)$$

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

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

If a stick-breaking representation [10], discussed in section 2.3, is assumed for the random measure G, then the model takes the form

$$y_i \mid \boldsymbol{\theta}^\star, \boldsymbol{v} \overset{ind}{\sim} \sum_{l=1}^{\infty} \left\{ v_l \prod_{m<l} (1 - v_m) \right\} h(\cdot \mid \theta_l^\star), (3)$$

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

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 in the stick-breaking representation. NIMBLE includes definitions of structures required to implement the CRP and stick-breaking distributions, and the associated MCMC algorithms.

2.2 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 to a restaurant, potentially with an infinite number of tables. Each customer sits on 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.

2.2.1 Specification and Density

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

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 integers set to be partitioned. See `help('dCRP')` for details of these parameters.

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 2.4.1.

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.

Density

The CRP distribution partitions the set of positive integers $\{1, \dots, N\}$, 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 $\boldsymbol{z}$ 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

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

where $m_k(\boldsymbol{z})$ denotes the number of elements in $\boldsymbol{z}$ that are equal to k. 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 $\boldsymbol{z}$ 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.

2.2.2 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. The model is given by

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

```
code <- nimbleCode({
  z[1:N] ~ dCRP(alpha, size = N)
  alpha ~ dgamma(1, 1)
  for(i in 1:M) {
    thetatilde[i] ~ dnorm(0, 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$N, 0, var=100),
              s2tilde = rinvgamma(constants$N, 1, 1),
              xi = sample(1:10, size = constants$N, replace = TRUE),
              alpha  = 1)

## Error in rnorm(constants$N, 0, var = 100):  unused argument (var = 100)

Rmodel <- nimbleModel(code, constants, data, inits)

## Error in assignDimensions(dimensions, inits, data):  object 'inits' not found
```

The resulting model may be fitted through MCMC sampling. NIMBLE will assign a specialized sampler to update `z` and `alpha`. See chapter **??** for information about NIMBLE's MCMC engine, and section 2.4.1 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 an effort to mitigate this inefficiencies, we allow the user to set $N^\star = M$, with $M < N$. However, if this number is too small and is exceeded in any iteration a warning is issued.

2.3 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')`.

2.3.1 Specification and Function

NIMBLE parametrizes the `stick_breaking` function by a $(0, 1)-$valued vector.

Function

The weights $(w_1, \ldots, w_L)$ follow a finite stick-breaking construction if $w_1 = v_1$, $w_l = v_l \prod_{m<l}(1-v_m)$, and $w_L = \prod_{m<L}(1-v_m)$.

Specification

The `stick_breaking` function is specified in NIMBLE for a probabilities vector `w` as `w[1:L] <- stick_breaking(v[1:(L-1)])`

Parameter `v` is a vector of values between 0 and 1 defining the breaking length of the stick. 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.

2.3.2 Example

Here we illustrate how to use NIMBLE for the example described in section 2.2.2, but considering a stick-breaking representation for G. The model is given by

$$y_i \mid \boldsymbol{\theta}^\star, \boldsymbol{\sigma}^{\star 2}, z_i \overset{ind}{\sim} N({\theta^\star}_{z_i}, {\sigma^{2\star}}_{z_i}), \quad i = 1, \ldots, N,$$

$$\boldsymbol{z} \sim Discrete(\boldsymbol{w}), \quad v_l \overset{iid}{\sim} Beta(1,1), \; l = 1, \ldots, L-1,$$

$$\theta^\star_l \overset{iid}{\sim} N(0, 100), \quad \sigma^{2\star}_l \overset{iid}{\sim} InvGamma(1,1), \; l = 1, \ldots, L.$$

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

```r
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, 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)
Rmodel <- 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 **??** for information about NIMBLE's MCMC engine, and section 2.4.2 for details on MCMC sampling of the stick-breaking weights.

2.4 MCMC sampling of BNP models

BNP models can be specified in different, yet equivalent, manners. Examples 2.2.2 and 2.3.2 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 [9] 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 [6] and [7].

2.4.1 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. Additionally, a specialized sampler is assigned to the `conc` hyper parameter when a gamma hyper prior is assigned, see section 6 in [3] for more details. Otherwise, a random walk Metropolis-Hastings sampler is assigned, case where `NaN` can be produced in the MCMC steps.

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 10, with replacement, as done in the preceding CRP example. For the concentration parameter, a safe initial value is 1.

Sampling the random measure

In BNP models, it is oftenly of interest to make inference about the unknown measure G. NIMBLE provides a sampler for this random measure when a CRP structure is involved in the model. This sampler is implemented as `getSamplesDPmeasure_samples`. More details for using this function are available using `help('getSamplesDPmeasure')`.

The arguments of this sampler are a CRP-based BNP model and an uncompiled model values object. The model values object should include the membership variable, the cluster parameter, and the concentration parameter, if it is random. Use the `monitors` argument when configuring the MCMC to ensure this variables are monitored in the MCMC.

The following code exemplifies how to generate samples from G after defining the model as in section 2.2.2.

```
cRmodel <- compileNimble(Rmodel)

monitors <- c('z', 'thetatilde', 's2tilde' , 'alpha')
RmodelConf <- configureMCMC(Rmodel, monitors = monitors, print = FALSE)

## Error in nl_checkVarNamesInModel(model, vars):  These variables are not in
model:  thetatilde,s2tilde.

RmodelMCMC <- buildMCMC(RmodelConf)

## Error in inherits(conf, "modelBaseClass"):  object 'RmodelConf' not found

cRmodelMCMC <- compileNimble(RmodelMCMC, project = Rmodel)

## Error in compileNimble(RmodelMCMC, project = Rmodel):  object 'RmodelMCMC'
not found

cRmodelMCMC$run(1000)

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

rdens <- nimble:::get_DP_measure_samples(Rmodel, mRmodelMCMC$mvSamples)

## Error in exists("dll", envir = mvSaved):  object 'mRmodelMCMC' not found

cdens <- compileNimble(rdens, project = Rmodel)

## Error in compileNimble(rdens, project = Rmodel):  object 'rdens' not found

cdens$run()

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

samplesG <- cdens$samples

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

2.4.2 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.

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 10, with replacement, $10 \leq L$, 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

Bibliography

[1] Blackwell, D. and J. MacQueen (1973). Ferguson distributions via Pólya urn schemes. *The Annals of Statistics 1*, 353–355.

[2] Escobar, M. D. (1994). Estimating normal means with a Dirichlet process prior. *Journal of the American Statistical Association 89*, 268–277.

[3] Escobar, M. D. and M. West (1995). Bayesian density estimation and inference using mixtures. *Journal of the American Statistical Association 90*, 577–588.

[4] Ferguson, T. S. (1973). A Bayesian analysis of some nonparametric problems. *Annals of Statistics 1*, 209–230.

[5] Ferguson, T. S. (1974). Prior distribution on the spaces of probability measures. *Annals of Statistics 2*, 615–629.

[6] Ishwaran, H. and L. F. James (2001). Gibbs sampling methods for stick-breaking priors. *Journal of the American Statistical Association 96*(453), 161–173.

[7] Ishwaran, H. and L. F. James (2002). Approximate Dirichlet process computing in finite normal mixtures: smoothing and prior information. *Journal of Computational and Graphical Statistics 11*, 508–532.

[8] Lo, A. Y. (1984). On a class of Bayesian nonparametric estimates I: Density estimates. *The Annals of Statistics 12*, 351–357.

[9] Neal, R. (2000). Markov chain sampling methods for Dirichlet process mixture models. *Journal of Computational and Graphical Statistics 9*, 249–265.

[10] Sethuraman, J. (1994). A constructive definition of Dirichlet prior. *Statistica Sinica 2*, 639–650.