Enabling efficient MCMC sampling with user-chosen samplers and automated parameter blocking in NIMBLE (r-nimble.org)

Daniel Turek[1,2], Perry de Valpine[2], Christopher Paciorek[1]

[1]Department of Statistics, UC Berkeley; [2]Department of Environmental Science Policy and Management, UC Berkeley

The NIMBLE system

NIMBLE is

- A system for using algorithms on hierarchical statistical models (defined by BUGS code),
- A system for programming algorithms to be used on hierarchical models,
- A partial compiler for math in R, and
- A flexible extension of the BUGS and JAGS systems.

Why use NIMBLE rather than other packages?

- Customize your MCMC: choose samplers and blocking arbitrarily;
- Define your own distributions and functions for use in BUGS code;
- Apply algorithms other than MCMC to a BUGS-defined model: SMC/particle filter, MCEM, etc.;
- Write your own algorithms (including MCMC samplers) that can be used on any BUGS model; and
- Disseminate algorithms via an R package containing specific algorithm code and a dependency on the NIMBLE package.

Outline of the poster

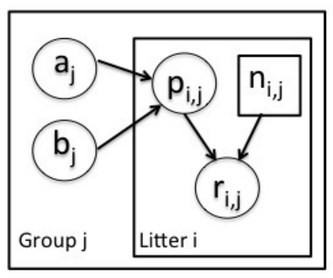
Here we'll focus on using the system to improve MCMC performance, considering three strategies:

- Customizing your MCMC with user-chosen samplers and blocking (left column)
- Automating blocking of parameters (center column)
- Writing your own MCMC sampler, which can then be used on any model (right column)

Using NIMBLE for tuning an MCMC

Example model: *litters*

There are $G = 2$ groups of rat litters, with $N = 16$ litters (i.e., mothers) in each group, and a variable number of pups in each litter. Survival of the pups in a litter is governed by a survival probability for each litter, $p_{i,j}$, with an exchangeable beta prior within each group. The model shows poor MCMC performance because of (1) dependence between hyperparameters for a given group and (2) dependence betwen hyperparameters and associated random effects of a given group.



BUGS code as used in NIMBLE

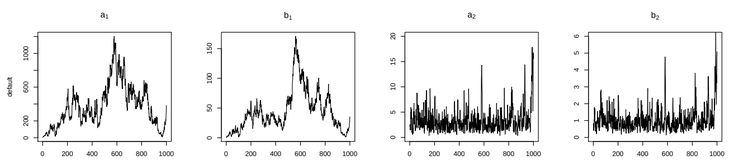
```
code <- nimbleCode({
 for (i in 1:G) {
  for (j in 1:N) {
    # likelihood (data model)
    r[i,j] ~ dbin(p[i,j], n[i,j])
    # latent process (random effects)
    p[i,j] ~ dbeta(a[i], b[i])
  }
  # prior for hyperparameters
  a[i] ~ dgamma(1, .001)
  b[i] ~ dgamma(1, .001)
 }
})
```

Define a default MCMC as in JAGS and BUGS

Set up the model (based on the BUGS code) and a default MCMC. This involves a few more steps than in BUGS or JAGS, but with the benefit that NIMBLE gives you greater control over how you use the model and set up an MCMC (or other algorithms).

```
model <- nimbleModel(code, constants = consts, data = data, inits = inits)
conf <- configureMCMC(model)
conf$addMonitors(c('a', 'b', 'p'))
mcmc <- buildMCMC(conf)
Cmodel <- compileNimble(model)
Cmcmc <- compileNimble(mcmc, project = model)
niter <- 10000
Cmcmc$run(niter)
smp <- as.matrix(Cmcmc$mvSamples)
```

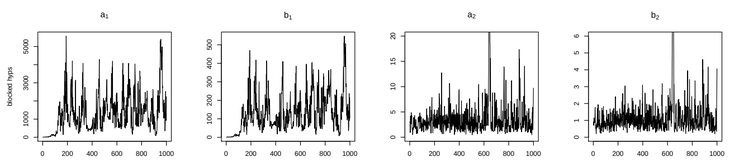
Here are trace plots for the default MCMC, showing poor mixing and strong dependence within pairs of hyperparameters.



Choose the blocking scheme

NIMBLE allows users to swap out samplers and choose blocking schemes. Here we'll block the hyperparameters for each group.

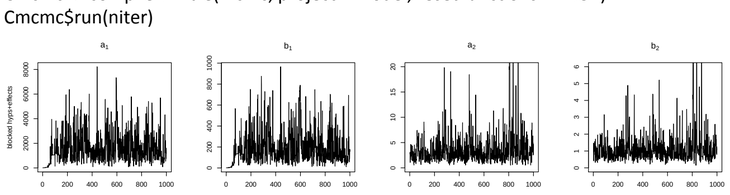
```
conf$removeSamplers(c('a', 'b'))
conf$addSampler(c('a[1]', 'b[1]'), 'RW_block')
conf$addSampler(c('a[2]', 'b[2]'), 'RW_block')
mcmc <- buildMCMC(conf)
Cmcmc <- compileNimble(mcmc, project = model, resetFunctions = TRUE)
Cmcmc$run(niter)
```



Use a specialized sampler

Blocking worked better but didn't account for dependence of the hyperparameters with their associated random effects (cross-level dependence). We'll replace the basic block sampler with a more sophisticated sampler that samples hyperparameters and associated random effects jointly, using random walk proposals for hyperparameters and (conditional) conjugate proposals for the random effects. It's equivalent to integrating over the random effects analytically and sampling from the marginal for the hyperparameters.

```
conf$removeSamplers(c('a', 'b', 'p'))
conf$addSampler(c('a[1]', 'b[1]'), 'crossLevel')
conf$addSampler(c('a[2]', 'b[2]'), 'crossLevel')
mcmc <- buildMCMC(conf)
Cmcmc <- compileNimble(mcmc, project = model, resetFunctions = TRUE)
Cmcmc$run(niter)
```



Automating MCMC sampler choices

Motivation

- Adaptive random walk block Metropolis has been successful in providing tuned proposals that account for dependence.
- But the choices of what samplers to use and what parameters to block are still done 'manually' by the analyst.
- Goal is to provide algorithms that search space of valid MCMC samplers for better MCMC performance.
- Key metric is effective sample size *per computing time*.

Trading off MCMC and computational efficiency

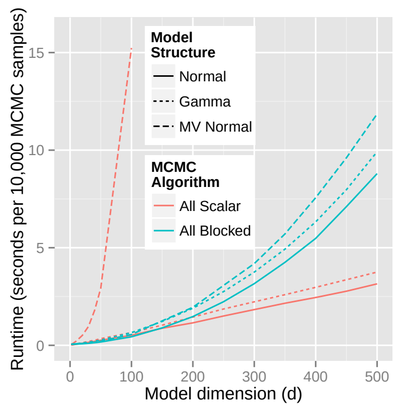
For a d-dimensional parameter, there are advantages and disadvantages in choosing between the use of d univariate samplers and a single d-dimensional block update.

Univariate random walk

- Computational: d univariate proposals: $d \times O(1) => O(d)$
- Computational: (only if have multivariate density) $d \times O(d^2) => O(d^3)$ density calculation
- Statistical: no accounting for dependence
- Statistical: Optimal proposal scales as $\sigma^2 \propto 1$

Multivariate random walk

- Computational: one multivariate proposal that is $O(d^2)$
- Computational: (only if have multivariate density) $O(d^2)$ density calculation
- Statistical: accounts for linear dependence
- Statistical: Optimal proposal scales as $\sigma^2 \propto \frac{1}{d}$, so multivariate sampling has a built-in disadvantage



Automated blocking algorithm

As an initial approach to automating MCMC sampler choice, we developed an automated blocking procedure to choose effective blocking for a given model and computation platform.

1. Initialize by running MCMC using univariate updaters.
2. Compute empirical correlations $\rho_{j,k}$ from current algorithm (blocking scheme).
3. Construct *hierarchical clustering tree* via `hclust()` based on distance metric: $d_{j,k} = 1 - |\rho_{j,k}|$.
4. Run multiple chains, each with a different degree of blocking determined by cutting the tree from (3) at various heights.
5. Choose the cut point (blocking) with highest effective sample size as the best blocking scheme.
6. If best blocking scheme is still in flux, go to (2) with blocking scheme from (5) as current algorithm; otherwise choose the current blocking scheme as the algorithm to use to generate final posterior samples.

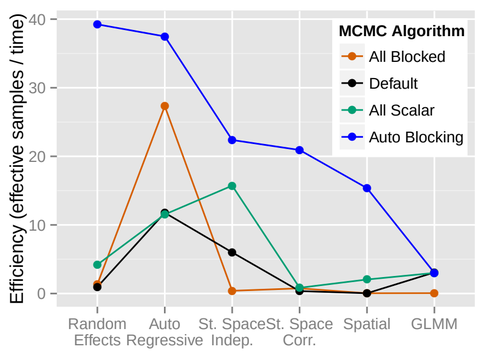
The automated blocking procedure is available within NIMBLE by calling `configureMCMC` like this: `conf <- configureMCMC(model, autoBlock = TRUE)` before building and running the MCMC.

Automated blocking results

MCMC performance results for suite of example models

Model	MCMC Scheme	ESS	Runtime	Efficiency
Random Effects	All Blocked	0.4	0.29	1.3
	Default	1.1	1.19	1.0
	All Scalar	2.1	0.51	4.2
	Informed Blocking	19.0	0.50	38.2
	Informed Cross-Level	101.3	2.64	38.5
	Auto Blocking	19.0	0.48	39.2
Auto-Regressive	All Blocked	8.9	0.3	27.3
	All Scalar	6.5	0.6	11.5
	Auto Blocking	12.7	0.3	37.5
State Space Independent	All Blocked	0.3	0.8	0.4
	Default	27.6	4.6	6.0
	All Scalar	20.2	1.3	15.7
	Auto Blocking	29.1	1.3	22.4
State Space Correlated	All Blocked	0.6	0.7	0.8
	Default	1.7	4.9	0.4
	All Scalar	1.1	1.3	0.8
	Informed Blocking	18.4	1.2	15.6
	Auto Blocking	26.1	1.2	20.9
Spatial	All Blocked	0.2	5.71	0.04
	Default	0.4	10.86	0.04
	All Scalar	171.3	83.87	2.0
	Auto Blocking	1208.0	78.62	15.4
GLMM	All Blocked	2.2	44.3	0.05
	All Scalar	60.9	22.6	3.0
	Auto Blocking	60.9	22.6	3.0

Table : Effective sample size (ESS) is measured in effective samples per 10,000 iterations, Runtime is presented as seconds per 10,000 iterations, and Efficiency is effective samples produced per second of runtime.



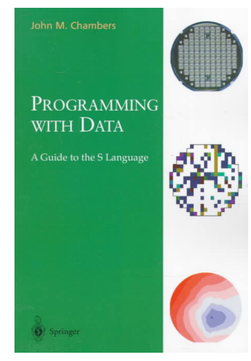
Next steps

- Explore theory to justify retaining samples during adaptation phase.
- Automate exploration of sampler choices (including reparameterization), not just blocks.
- In exploration phase, use multiple updaters on a single parameter or multiple blocking schemes within an iteration to enable more efficient optimization amongst the updaters.

Programming with models

Background and motivation

- S (and R) revolutionized statistics and data analysis by putting data front and center.
- Existing software provides great power for fitting models via the algorithms chosen by the developers.
- We want to enable analysts and developers to more easily program algorithms to operate on models.
- Developers can then easily disseminate algorithms to users.

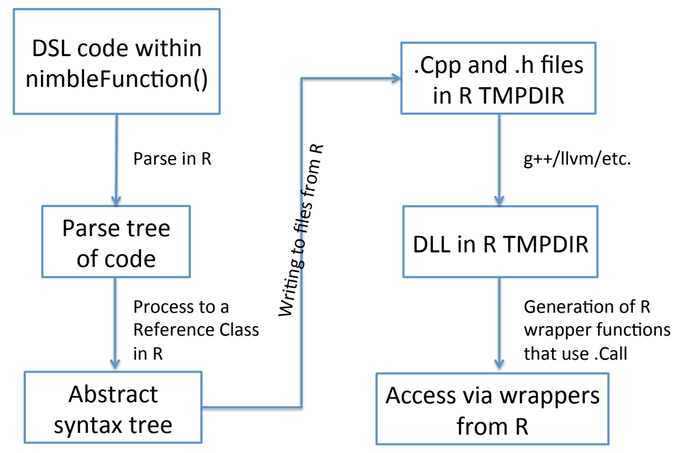


The NIMBLE language

NIMBLE provides a *domain specific language (DSL)* for writing model generic algorithms via *nimbleFunctions*, which contain two components:

- setup code: *specializes* algorithm to model by querying model relationships and setting up data structures
- run code: model-generic code to execute the algorithm that can use information determined in setup code

NIMBLE processing flow



Writing an algorithm: a simple example

Here's how to write a basic objective function (which could be passed to an optimization function such as R's optim).

- setup code: needs to determine which nodes are dependencies so the run code knows which quantities to calculate.
- run code: computes log probability density for all relevant nodes for any model

```
objectiveFunction <- nimbleFunction (
 setup = function(model, nodes) {
  calcNodes <- model$getDependencies(nodes)
 },
 run = function(vals = double(1)) {
  values(model, nodes) <<- vals
  sumLogProb <- calculate(model, calcNodes)
  return(sumLogProb)
  returnType(double())
 })
```

query model structure ONCE

the actual algorithm

Writing a user-defined MCMC sampler

We use *nimbleFunctions* to define new samplers, including the *cross-level* sampler in column 1.

Here's an example of how a user could readily implement use of reflection in a random walk sampler.

```
sampler_RW_reflect <- nimbleFunction(
  contains = sampler_BASE,
  setup = function(model, mvSaved, target, control) {
    dist <- model$getNodeDistribution(target)
     rg <- getDistribution(dist)$range
    if(rg[1] > -Inf || rg[2] < Inf)
          reflect <- TRUE else reflect <- FALSE
    calcNodes  <- model$getDependencies(target)
  },
  run = function() {
    propValue <- rnorm(1, mean = model[[target]], sd = scale)
    if(reflect) {
      if(propValue < rg[1]) propValue <- 2*rg[1] - propValue
      if(propValue > rg[2]) propValue <- 2*rg[2] - propValue
    }
    model[[target]] <<- propValue
    logMHR <- calculateDiff(model, calcNodes)
    jump <- decide(logMHR)
    if(jump)
      nimCopy(from = model, to = mvSaved, row = 1, nodes = calcNodes)
    else
      nimCopy(from = mvSaved, to = model, row = 1, nodes = calcNodes)
  })
```

We can then directly use the new sampler for a given node in an MCMC on a model:

```
 conf <- configureMCMC(model)
conf$addSampler('tau', type = 'RW_reflect')
mcmc <- buildMCMC(conf)
```

References and acknowledgements

The NIMBLE system:

- de Valpine, P., D. Turek, C.J. Paciorek, C. Anderson-Bergman, D. Temple Lang, and R. Bodik. 2016. Programming with models: writing statistical algorithms for general model structures with NIMBLE. Journal of Computational and Graphical Statistics, in press. doi: 10.1080/10618600.2016.1172487.

The automated blocking algorithm:

- Turek, D., P. de Valpine, C.J. Paciorek and C. Anderson-Bergman. Automated parameter blocking for efficient Markov chain Monte Carlo sampling. Bayesian Analysis, in press.

This work was supported by grant DBI-1147230 from the US National Science Foundation and by support to DT from the Berkeley Institute for Data Science.

NIMBLE