Beyond the black box: Flexible programming of hierarchical modeling algorithms for BUGS-based models using NIMBLE

Christopher Paciorek  UC Berkeley Statistics

Joint work with:
Perry de Valpine (PI)  UC Berkeley Environmental Science, Policy and Management
Daniel Turek  Williams College
Nick Michaud  UC Berkeley Statistics and ESPM
Duncan Temple Lang  UC Davis Statistics

http://r-nimble.org
Imperial College seminar
July 2018

Funded by NSF DBI-1147230 and NSF-ACI-1550488
What do we want to do with hierarchical models?

1. Core algorithms
   • MCMC
   • Sequential Monte Carlo
   • Laplace approximation
   • Importance sampling
   • Variational Bayes

NIMBLE: extensible software for hierarchical models (r-nimble.org)
What do we want to do with hierarchical models?

1. Core algorithms
   - MCMC
   - Sequential Monte Carlo
   - Laplace approximation
   - Importance sampling
   - Variational Bayes

2. Different flavors of algorithms
   - Many flavors of MCMC
   - Gaussian quadrature
   - Monte Carlo expectation maximization (MCEM)
   - Kalman Filter
   - Auxiliary particle filter
   - Posterior predictive simulation
   - Posterior re-weighting
   - Data cloning
   - Bridge sampling (normalizing constants)
   - YOUR FAVORITE HERE
   - YOUR NEW IDEA HERE

NIMBLE: extensible software for hierarchical models (r-nimble.org)
What do we want to do with hierarchical models?

1. Core algorithms
   - MCMC
   - Sequential Monte Carlo
   - Laplace approximation
   - Importance sampling
   - Variational Bayes

2. Different flavors of algorithms
   - Many flavors of MCMC
   - Gaussian quadrature
   - Monte Carlo expectation maximization (MCEM)
   - Kalman Filter
   - Auxiliary particle filter
   - Posterior predictive simulation
   - Posterior re-weighting
   - Data cloning
   - Bridge sampling (normalizing constants)
   - YOUR FAVORITE HERE
   - YOUR NEW IDEA HERE

3. Idea combinations
   - Particle MCMC
   - Particle Filter with replenishment
   - MCMC/Laplace approximation
   - Dozens of ideas in recent JRSSB/JCGS issues

NIMBLE: extensible software for hierarchical models (r-nimble.org)
What can a practitioner do with hierarchical models?

Two basic software designs:

1. Typical R package = Model family + 1 or more algorithms
   - GLMMs: lme4, MCMCglmm
   - GAMMs: mgcv
   - spatial models: spBayes, INLA
What can a practitioner do with hierarchical models?

Two basic software designs:

1. Typical R package = Model family + 1 or more algorithms
   - GLMMs: lme4, MCMCglmm
   - GAMMs: mgcv
   - spatial models: spBayes, INLA

2. Flexible model + black box algorithm
   - BUGS: WinBUGS, OpenBUGS, JAGS
   - PyMC
   - INLA
   - Stan
Existing software

e.g., BUGS (WinBUGS, OpenBUGS, JAGS), INLA, Stan, various R packages
NIMBLE: The Goal

Model

\[ X(1) \rightarrow X(2) \rightarrow X(3) \]

\[ Y(1) \rightarrow Y(2) \rightarrow Y(3) \]

Algorithm language

NIMBLE: extensible software for hierarchical models (r-nimble.org)
Divorcing Model Specification from Algorithm

Your new method
Data cloning
MCEM
Quadrature
Maximum likelihood

MCMC Flavor 1
MCMC Flavor 2
Particle Filter
Importance Sampler

NIMBLE: extensible software for hierarchical models (r-nimble.org)
Goals

– Retaining BUGS compatibility
– Providing a variety of standard algorithms
– Allowing users to easily modify those algorithms
– Allowing developers to add new algorithms (including modular combination of algorithms)
– Allowing users to operate within R
– Providing speed via compilation to C++, with R wrappers
NIMBLE System Summary

- statistical model (BUGS code) + algorithm (nimbleFunction)

- R objects + R under the hood

- R objects + C++ under the hood

- We generate C++ code,
- compile and load it,
- provide interface object.
1. Model specification

BUGS language $\rightarrow$ R/C++ model object

2. Algorithm library

MCMC, Particle Filter/Sequential MC, MCEM, etc.

3. Algorithm specification

NIMBLE programming language within R $\rightarrow$ R/C++ algorithm object
User Experience: Creating a Model from BUGS

littersCode <- nimbleCode({
  for(j in 1:G) {
    for(l in 1:N) {
      r[i, j] ~ dbin(p[i, j], n[i, j]);
      p[i, j] ~ dbeta(a[j], b[j]);
    }
    mu[j] <- a[j]/(a[j] + b[j]);
    theta[j] <- 1.0/(a[j] + b[j]);
    a[j] ~ dgamma(1, 0.001);
    b[j] ~ dgamma(1, 0.001);
  }
})

Parse and process BUGS code. Collect information in model object.

> littersModel <- nimbleModel(littersCode, constants = list(N = 16, G = 2), data = list(r = input$r))
> littersModel_cpp <- compileNimble(littersModel)

Use igraph plot method (we also use this to determine dependencies).

Provides variables and functions (calculate, simulate) for algorithms to use.
The Success of R
Programming with Models

You give NIMBLE:

```r
littersCode <- nimbleCode( {
  for(j in 1:G) {
    for(l in 1:N) {
      r[i, j] ~ dbin(p[i, j], n[i, j]);
      p[i, j] ~ dbeta(a[j], b[j]);
    }
    mu[j] <- a[j]/(a[j] + b[j]);
    theta[j] <- 1.0/(a[j] + b[j]);
    a[j] ~ dgamma(1, 0.001);
    b[j] ~ dgamma(1, 0.001);  
  }
})
```

You get this:

```r
> littersModel$a[1] <- 5  # set values in model
> simulate(littersModel, 'p')  # simulate from prior
> p_deps <- littersModel$getDependencies('p')  # model structure
> calculate(littersModel, p_deps)  # calculate probability density
> getLogProb(pumpModel, 'r')
```

NIMBLE also extends BUGS: multiple parameterizations, named parameters, and user-defined distributions and functions.
User Experience: Specializing an Algorithm to a Model

```r
littersModelCode <- modelCode({
  for(j in 1:G) {
    for(l in 1:N) {
      r[i, j] ~ dbin(p[i, j], n[i, j]);
      p[i, j] ~ dbeta(a[j], b[j]);
    }
    mu[j] <- a[j]/(a[j] + b[j]);
    theta[j] <- 1.0/(a[j] + b[j]);
    a[j] ~ dgamma(1, 0.001);
    b[j] ~ dgamma(1, 0.001);
  }
})
```

```r
sampler_slice <- nimbleFunction(
  setup = function((model, mvSaved, control) {
    calcNodes <- model$getDependencies(control$targetNode)
    discrete <- model$getNodeInfo()[control$targetNode]$isDiscrete()
    [...snip...]
    run = function() {
      u <- getLogProb(model, calcNodes) - rexp(1, 1)
      x0 <- model[[targetNode]]
      L <- x0 - runif(1, 0, 1) * width
      [...snip....]
    }
  })
```

```r
> littersMCMCconf <- configureMCMC(littersModel)
> littersMCMCconf$printSamplers()
[...snip...]
[3] RW sampler; targetNode: b[1], adaptive: TRUE, adaptInterval: 200, scale: 1
[5] conjugate_beta sampler; targetNode: p[1, 1], dependents_dbin: r[1, 1]
[6] conjugate_beta sampler; targetNode: p[1, 2], dependents_dbin: r[1, 2]
[...snip...]
> littersMCMCconf$addSampler(‘a[1]’, ‘slice’, list(adaptInterval = 100))
> littersMCMCconf$addSampler(‘a[2]’, ‘slice’, list(adaptInterval = 100))
> littersMCMCconf$addMonitors(‘theta’)
> littersMCMC <- buildMCMC(littersMCMCspec)
> littersMCMC_Cpp <- compileNimble(littersMCMC, project = littersModel)
> runMCMC(littersMCMC_Cpp, niter = 20000)
```
littersModelCode <- quote{
  for(j in 1:G) {
    for(l in 1:N) {
      r[i, j] ~ dbin(p[i, j], n[i, j]);
      p[i, j] ~ dbeta(a[j], b[j]);
    }
    mu[j] <- a[j]/(a[j] + b[j]);
    theta[j] <- 1.0/(a[j] + b[j]);
    a[j] ~ dgamma(1, 0.001);
    b[j] ~ dgamma(1, 0.001);
  }
}

buildMCEM <- nimbleFunction(
  while(runtime(converged == 0)) {
    ...
    calculate(model, paramDepDetermNodes)
    mcmcFun(mcmc.its, initialize = FALSE)
    currentParamVals[1:nParamNodes] <- getValues(model, paramNodes)
    op <- optim(currentParamVals, objFun, maximum = TRUE)
    newParamVals <- op$maximum
    ...

> littersMCEM <- buildMCEM(littersModel, latentNodes = ‘p’, mcmcControl = list(adaptInterval = 50), boxConstraints = list( list(‘a’, ‘b’), limits = c(0, Inf))), buffer = 1e-6)
> set.seed(0)
> littersMCEM(maxit = 50, m1 = 500, m2 = 5000)

Modularity:

One can plug any MCMC sampler into the MCEM, with user control of the sampling strategy, in place of the default MCMC.
NIMBLE

1. Model specification

    BUGS language ➜ R/C++ model object

2. Algorithm library

    MCMC, Particle Filter/Sequential MC, MCEM, etc.

3. Algorithm specification

    NIMBLE programming language within R ➜ R/C++ algorithm object

NIMBLE: extensible software for hierarchical models (r-nimble.org)
NIMBLE’s algorithm library

– MCMC samplers:
  • Conjugate, adaptive Metropolis, adaptive blocked Metropolis, slice, elliptical slice sampler, particle MCMC, specialized samplers for particular distributions (Dirichlet, CAR)
  • Flexible choice of sampler for each parameter
  • User-specified blocks of parameters
  • Cross-validation, WAIC

– Sequential Monte Carlo (particle filters)
  • Various flavors

– MCEM

– Write your own
NIMBLE in Action: the Litters Example

Beta-binomial GLMM for clustered binary response data
Survival in two sets of 16 litters of pigs

littersModelCode <- nimbleCode({
  for(j in 1:2) {
    for(l in 1:16) {
      r[i, j] ~ dbin(p[i, j], n[i, j]);
      p[i, j] ~ dbeta(a[j], b[j]);
    }
    mu[j] <- a[j]/(a[j] + b[j]);
    theta[j] <- 1.0/(a[j] + b[j]);
    a[j] ~ dgamma(1, 0.001);
    b[j] ~ dgamma(1, 0.001);
  }
})

Challenges of the toy example:

- BUGS manual: “The estimates, particularly $a_1$, $a_2$ suffer from extremely poor convergence, limited agreement with m.l.e.’s and considerable prior sensitivity. This appears to be due primarily to the parameterisation in terms of the highly related $a_j$ and $b_j$, whereas direct sampling of $mu_j$ and $theta_j$ would be strongly preferable.”
- But that’s not all that’s going on. Consider the dependence between the p’s and their $a_j$, $b_j$ hyperparameters.
- And perhaps we want to do something other than MCMC.
Default MCMC: Gibbs + Metropolis

> littersMCMCspec <- configureMCMC(littersModel, list(adaptInterval = 100))
> littersMCMC <- buildMCMC(littersMCMCspec)
> littersMCMC_cpp <- compileNIMBLE(littersModel, project = littersModel)
> runMCMC(littersMCMC_cpp, niter = 10000)
Red line is MLE
Blocked MCMC: Gibbs + Blocked Metropolis

```r
> littersMCMCspec2 <- configureMCMC(littersModel, list(adaptInterval = 100))
> littersMCMCspec2$addSampler(c('a[1]', 'b[1]'), 'RW_block', list(adaptInterval = 100))
> littersMCMCspec2$addSampler(c('a[2]', 'b[2]'), 'RW_block', list(adaptInterval = 100))
> littersMCMC2 <- buildMCMC(littersMCMCspec2)
> littersMCMC2_cpp <- compileNIMBLE(littersMCMC2, project = littersModel)
> runMCMC(littersMCMC2_cpp, niter = 10000)
```
Blocked MCMC: Gibbs + Cross-level Updaters

- Cross-level dependence is a key barrier in this and many other models.
- We wrote a new “cross-level” updater function using the NIMBLE DSL.
  - Blocked Metropolis random walk on a set of hyperparameters with conditional Gibbs updates on dependent nodes (provided they are in a conjugate relationship).
  - Equivalent to (analytically) integrating the dependent (latent) nodes out of the model.

```r
> littersMCMCspec3 <- configureMCMC(littersModel, adaptInterval = 100)
> topNodes1 <- c('a[1]', 'b[1]')
> littersMCMCspec3$addSampler(topNodes1, 'crossLevel', list(adaptInterval = 100)
> topNodes2 <- c('a[2]', 'b[2]')
> littersMCMCspec3$addSampler(topNodes2, 'crossLevel', list(adaptInterval = 100)
> littersMCMC3 <- buildMCMC(littersMCMCspec3)
> littersMCMC3_cpp <- compileNIMBLE(littersMCMC3, project = littersModel)
> runMCMC(littersMCMC3_cpp, niter = 10000)
```
Litters MCMC: BUGS and JAGS

• Customized sampling possible in NIMBLE greatly improves performance.
  • BUGS gives similar performance to the default NIMBLE MCMC
    • Be careful – values of $\text{sim.list}$ and $\text{sims.matrix}$ in R2WinBUGS output are randomly permuted
    • Mixing for $a_2$ and $b_2$ modestly better than default NIMBLE MCMC
  • JAGS slice sampler gives similar performance as BUGS, but fails for some starting values with this (troublesome) parameterization
• NIMBLE provides user control and transparency.
  • NIMBLE is faster than JAGS on this example (if one ignores the compilation time), though not always.
  • Note: we’re not out to build the best MCMC but rather a flexible framework for algorithms – we’d love to have someone else build a better default MCMC and distribute for use in our system.
NIMBLE

1. Model specification

   BUGS language $\Rightarrow$ R/C++ model object

2. Algorithm library

   MCMC, Particle Filter/Sequential MC, MCEM, etc.

3. Algorithm specification

   NIMBLE programming language within R $\Rightarrow$ R/C++ algorithm object
We want:

• High-level processing (model structure) in R

• Low-level processing in C++
sampler_myRW <- nimbleFunction(

setup = function(model, mvSaved, targetNode, scale) {
  calcNodes <- model$getDependencies(targetNode)
},
run = function() {
  model_lp_initial <- calculate(model, calcNodes)
  proposal <- rnorm(1, model[[targetNode]], scale)
  model[[targetNode]] <<- proposal
  model_lp_proposed <- calculate(model, calcNodes)
  log_MH_ratio <- model_lp_proposed - model_lp_initial
  if(decide(log_MH_ratio)) jump <- TRUE
  else jump <- FALSE
  # .... Various bookkeeping operations ... #
})
sampler_myRW <- nimbleFunction(

setup = function(model, mvSaved, targetNode, scale) {
  calcNodes <- model$getDependencies(targetNode)
},

run = function() {
  model_lp_initial <- calculate(model, calcNodes)
  proposal <- rnorm(1, model[[targetNode]], scale)
  model[[targetNode]] <- proposal
  model_lp_proposed <- calculate(model, calcNodes)
  log_MH_ratio <- model_lp_proposed - model_lp_initial

  if(decide(log_MH_ratio)) jump <- TRUE
  else jump <- FALSE

  # .... Various bookkeeping operations ...
})
sampler_myRW <- nimbleFunction(

setup = function(model, mvSaved, targetNode, scale) {
  calcNodes <- model$getDependencies(targetNode)
},
run = function() {
  model_lp_initial <- calculate(model, calcNodes)
  proposal <- rnorm(1, model[[targetNode]], scale)
  model[[targetNode]] <- proposal
  model_lp_proposed <- calculate(model, calcNodes)
  log_MH_ratio <- model_lp_proposed - model_lp_initial

  if(decide(log_MH_ratio)) jump <- TRUE
  else jump <- FALSE

  # .... Various bookkeeping operations ... #
})
The NIMBLE compiler (run code)

Feature summary:

• R-like matrix algebra (using Eigen library)
• R-like indexing (e.g. X[1:5,])
• Use of model variables and nodes
• Model calculate (logProb) and simulate functions
• Sequential integer iteration
• If-then-else, do-while
• Access to much of Rmath.h (e.g. distributions)
• Automatic R interface / wrapper
• Call out to your own C/C++ or back to R
• Many improvements / extensions planned

NIMBLE: extensible software for hierarchical models (r-nimble.org)
How an Algorithm is Processed in NIMBLE

- **DSL code (run code) within nimbleFunction()**
- **Parse in R**
- **Parse tree of code**
- **Process to a Reference Class in R**
- **Abstract syntax tree**

- **.Cpp and .h files in R TMPDIR**
- **Writing to files from R**
- **DLL in R TMPDIR**
- **g++/llvm/etc.**

- **Generation of R wrapper functions that use .Call**
- **Access via wrappers from R**

*NIMBLE: extensible software for hierarchical models (r-nimble.org) 35*
Modular algorithms: particle MCMC

- Particle filter (SMC) approximates a posterior for latent states using a sample.
- Traditionally used in state space models where the sample particles are propagated in time to approximate: $p(x_t | y_{1:t}, \theta)$.

Figure 2: Particle filtering (from (Lehmann 2003))

- Weights from ‘correction’ step can be used to estimate $p(y_{1:t} | \theta)$.
- Embed in MCMC to do approximate marginalization over $x_{1:t}$. 

Particle MCMC in NIMBLE

sampler_PMCMC <- nimbleFunction(

setup = function(model, mvSaved, target, control) {
    ....
    my_particleFilter <- buildAuxiliaryFilter(model, control$latents, control = list(saveAll = TRUE, smoothing = TRUE, lookahead = lookahead))
    ....
},
run = function() {
    ....
    modelLP0 <- modelLL0 + calculate(model, target)
    propValue <- rnorm(1, mean = model[[target]], sd = scale)
    model[[target]] <<- propValue
    modelLL1 <- my_particleFilter$run(m)
    modelLP1 <- modelLL1 + calculate(model, target)
    jump <- my_decideAndJump$run(modelLP1, modelLP0, 0, 0)
    ....
})
Status of NIMBLE and Next Steps

• First release was June 2014 with regular releases since. Lots to do:
  – Improve the user interface and speed up compilation (in progress)
  – Scalability for large models (in progress)
  – Bayesian nonparametrics with Claudia Wehrhahn & Abel Rodriguez (UCSC) (first release a few weeks ago and more work in progress)
  – Refinement/extension of the DSL for algorithms (in progress)
    • e.g., automatic differentiation, parallelization
  – Additional algorithms written in NIMBLE DSL
    • e.g., normalizing constant calculation, Laplace approximations

• Interested?
  – We have funding for a postdoc/programmer!
  – We have funding to bring users/developers to Berkeley.
  – Announcements: nimble-announce Google site
  – User support/discussion: nimble-users Google site
  – Write an algorithm using NIMBLE!
  – Help with development of NIMBLE: email nimble.stats@gmail.com or see github.com/nimble-dev
NIMBLE: What can I program?

- Your own distribution for use in a model
- Your own function for use in a model
- Your own MCMC sampler for a variable in a model
- A new MCMC sampling algorithm for general use
- A new algorithm for hierarchical models
- An algorithm that composes other existing algorithms (e.g., MCMC-SMC combinations)
PalEON Project
www3.nd.edu/~paleolab/paleonproject

Goal: Improve the predictive capacity of terrestrial ecosystem models

“This large variation among carbon-cycle models ... has been called ‘uncertainty’. I prefer to call it ‘ignorance’.”
- Prentice (2013) Grantham Institute

Critical issue: model parameterization and representation of decadal- to centennial-scale processes are poorly constrained by data

Approach: use historical and fossil data to estimate past vegetation and climate and use this information for model initialization, assessment, and improvement
PalEON Statistical Applications

- Estimate spatially-varying composition and biomass of tree species from count and zero-inflated size data in year 1850
- Estimate temporal variations in temperature and precipitation over 2000 years from tree rings and lake/bog records
- Estimate tree composition spatially over 2000 years from fossil pollen in lake sediment cores
- Estimate biomass over time at a site from fossil pollen in lake sediment cores
Inferring Biomass from Pollen

• Calibration with multiple spatial locations:
  – “Regress” multinomial counts on biomass
  – For each taxon, have proportion of the taxon be a smooth function of biomass using splines and parameters of Beta distributions:
    • \( \alpha_k = \exp(Z(b)\beta_k) \)
      – Estimate spline coefficients for each taxon

• Predict biomass over time at one location:
  – State space model for biomass over time
  – Fixed spline coefficients from calibration
  – Inverse problem (just Bayesian inference)
    • \( \alpha_k = \exp(Z(b_t)\beta_k) \)
Predicting biomass from compositional data

**Calibration:** at settlement time we have biomass estimates (based on survey data and a spatial model) and pollen composition (from sediment cores)

**Prediction:** based on calibration model and pollen composition over time, predict biomass

NIMBLE: extensible software for hierarchical models (r-nimble.org)
Calibration model

• Pollen proportion for each taxon determined by transformation of a flexible (spline) function of biomass
  • shape1 and shape2 parameters of beta distribution (stick-breaking prior for multinomial) are splines of biomass
  • Primary calibration parameters are spline coefficients
• Overdispersed multinomial likelihood for pollen counts given modeled proportions
• Fit in NIMBLE (could be fit in various other packages)
Mean and variability of modeled pollen proportions across ponds vary with biomass
Prediction Model

\[ \sigma \]

\[ b_t \]

\[ Z(b_t) \]

\[ \beta_{1k}, \beta_{2k} \]

\[ \alpha_{1tk}, \alpha_{2tk} \]

\[ p_{tk} \]

\[ \gamma_{tk} \]

NIMBLE: extensible software for hierarchical models (r-nimble.org)
Prediction Model

for(t in 1:nTimes)
  Y[t, 1] ~ dbetabin(shape1[t, 1], shape2[t, 1], n[t])
for(k in 2:(nTaxa-1)) {
  Y[t, k] ~ dbetabin(shape1[t, k], shape2[t, k], n[t]-sum(Y[t, 1:(k-1)]))
}
shape1[ , ] <- exp(Zb[ , ] %*% beta1[ ,])
shape2[ , ] <- exp(Zb[ , ] %*% beta2[ ,])
for( t in 1:nTimes)
  Zb[t, 1:nKnots] <- bspline(b[t], knots[1:nKnots])
for(t in 2:nTimes)
  b[t] ~ dGenPareto (3b[t-1]-3b[t-2]+b[t-3], sigma)
sigma ~ dunif(0, 10) # Gelman (2006)
b[1] ~ dunif(0, 400)
NIMBLE solution: customized MCMC sampler for \(\{b[t-3], b[t-2], b[t-1], b[t], b[t+1], b[t+2], b[t+3]\}\), with a normal approximation to likelihood to generate good (quasi-conjugate) proposals.
PalEON Acknowledgements

• Pollen-biomass Collaborators: Ann Raiho, Jason McLachlan (Notre Dame Biology)

• PalEON investigators: Jason McLachlan (Notre Dame, PI), Mike Dietze (BU), Andrew Finley (Michigan State), Amy Hessl (West Virginia), Phil Higuera (Idaho), Mevin Hooten (USGS/Colorado State), Steve Jackson (USGS/Arizona), Dave Moore (Arizona), Neil Pederson (Harvard Forest), Jack Williams (Wisconsin), Jun Zhu (Wisconsin)

• NSF Macrosystems Program