Extensible software for hierarchical modeling:
using the NIMBLE platform to explore models and algorithms

Christopher Paciorek    UC Berkeley Statistics

Joint work with:
Perry de Valpine (PI)    UC Berkeley Environmental Science, Policy and Managem’t
Daniel Turek            UC Berkeley Statistics and ESPM
Cliff Anderson-Bergman  UC Berkeley Statistics and ESPM
Duncan Temple Lang       UC Davis Statistics

http://r-nimble.org

useR! 2014, Los Angeles
July, 2014

Funded by NSF DBI-1147230
Background and Goals

• Software for fitting hierarchical models has opened their use to a wide variety of communities
• Most software for fitting such models is either model-specific or algorithm-specific
• Software is often a black box and hard to extend
• Our goal is to divorce model specification from algorithm, while
  – Retaining BUGS compatibility
  – Providing a variety of standard 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: extensible software for hierarchical models (r-nimble.org)
Divorcing Model Specification from Algorithm

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

• High-level processing in R (as much as possible)
  • Process BUGS language for declaring models (with some extensions)
  • Process model structure (node dependencies, conjugate relationships, etc.)
  • Generate and customize algorithm specifications
  • Generate model-specific C++ code to be compiled on the fly
  • Provide matching implementation in R for prototyping / debugging / testing
  • Some high-level algorithm control possible in R (adapting tuning parameters, monitoring convergence, high levels of iteration)

• Low-level processing in C++
  • Model and algorithm computations
  • “Run-time” parameters allow some modification of behavior without recompiling
User Experience: Creating a Model from BUGS

```r
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);
  }
})
```

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

2. Use igraph plot method.

3. Provides variables and functions (calculate, simulate) for algorithms to use.

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

NIMBLE: extensible software for hierarchical models (r-nimble.org)
User Experience: Specializing an Algorithm to a Model

```r
littersModelCode <- modelCode({
  for(j in 1:G) {
    for(I 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);
  }
})

> littersMCMCspec <- MCMCspec(littersModel)
> getUpdaters(littersMCMCspec)
[...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...]
> littersMCMCspec$addSampler(‘slice’, list(targetNodes = c(‘a[1]’, ‘a[2]’), adaptInterval = 100))
> littersMCMCspec$addMonitor(‘theta’)
> littersMCMC <- buildMCMC(littersMCMCspec)
> littersMCMC_Cpp <- compileNimble(littersMCMC, project = littersModel)

> littersMCMC_Cpp (20000)
```
User Experience: Specializing an Algorithm to a Model (2)

```
littersModelCode <- quote({
  for(j in 1:G) {
    for(I 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
buildMCEM <- nimbleFunction(
  while(runNme(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), boxConstrains = list( list(‘a’, ‘b’), limits = c(0, Inf)), buffer = 1e-6)
> set.seed(0)
> littersMCEM(maxit = 50, m1 = 500, m2 = 5000)

Modularity (UNDER CONSTRUCTION):

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

• Analogy: BUGS is a Domain-Specific Language (DSL) for models
• NIMBLE provides a DSL for algorithms
  • The DSL is a modified subset of R.
• We provide
  • Basic types (double, logical)
  • Basic (vectorized) math and distribution/probability calculations
  • Basic data storage classes (“modelValues”)
  • Control structures – for loops and if-then-else
  • Ability to define functions
  • Linear algebra (via the Eigen package)
  • Specific functions for a model: calculate, simulate

• Function definitions in the DSL include code for two steps:
  • A generic run-time function is written in the DSL for any model structure
  • When a model is provided, a set of one-time setup processing is executed in R based on the model structure to “specialize” algorithm to model
  • Run-time code can use information determined from the setup processing
myAlgorithmGenerator <- nimbleFunction(

  setup = function(model, <otherSetupArguments>) {

    # code that does the specialization of algorithm to model
    # e.g., determine nodes to sample,
    # initialize storage

  },

  run = function(<runtimeArguments>) {

    # code that carries out the generic algorithm
    # for example, iterations of an algorithm
    # simulate into nodes, calculate log probability values
    returnType(double())
    return(x)
  }
)

Usage:
specializedAlgo <- myAlgorithmGenerator(myModel, <setupArgs>)
specializedAlgo(<runtimeArguments>)
sampler_slice <- nimbleFunction( contains = sampler_BASE, 
  setup = function(model, mvSaved, control) { 
    targetNode <- control$targetNode 
    adaptive <- control$adaptive 
    .... 
    calcNodes <- model$getDependencies(targetNode) 
    .... 
    discrete <- model$getNodeInfo()[[targetNode]]$isDiscrete() 
  }, 
  run = function() { 
    u <- getLogProb(model, calcNodes) - rexp(1, 1) 
    x0 <- model[[targetNode]] 
    L <- x0 - runif(1, 0, 1) * width 
    R <- L + width 
    maxStepsL <- floor(runif(1, 0, 1) * maxSteps) 
    maxStepsR <- maxSteps - 1 - maxStepsL 
    lp <- setAndCalculateTarget(L) 
    while(maxStepsL > 0 & !is.nan(lp) & lp >= u) { 
      L <- L - width 
      lp <- setAndCalculateTarget(L) 
      maxStepsL <- maxStepsL - 1 
    } ....
NIMBLE in Action: the Litters Example

Beta-binomial for clustered binary response data

```r
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);
  }
}
```

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

```r
> littersMCMCspec2 <- MCMCspec(littersModel, list(adaptInterval = 100))
> littersMCMCspec2$addSampler('RW_block', list(targetNodes = c('a[1]', 'b[1]'), adaptInterval = 100))
> littersMCMCspec2$addSampler('RW_block', list(targetNodes = c('a[2]', 'b[2]'), adaptInterval = 100))
> littersMCMC2 <- buildMCMC(littersMCMCspec2)
> littersMCMC2_cpp <- compileNIMBLE(littersMCMC2, project = littersModel)
> littersMCMC2_cpp(10000)
```
NIMBLE: extensible software for hierarchical models (r-nimble.org)
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.
  • The updater is a blocked Metropolis random walk on a set of hyperparameters with conditional Gibbs updates on dependent nodes (provided they are in a conjugate relationship).
  • This is equivalent to (analytically) integrating the dependent (latent) nodes out of the model.
• We can then add this updater to an MCMC for a given model....

```r
> littersMCMCspec3 <- MCMCspec(littersModel, adaptInterval = 100)
> topNodes1 <- c('a[1]', 'b[1]')
> littersMCMCspec3$addSampler('crossLevel', list(topNodes = topNodes1, adaptInterval = 100))

> topNodes2 <- c('a[2]', 'b[2]')
> littersMCMCspec3$addSampler('crossLevel', list(topNodes = topNodes1, adaptInterval = 100))

> littersMCMC3 <- buildMCMC(littersMCMCspec3)
> littersMCMC3_cpp <- compileNIMBLE(littersMCMC3, project = littersModel)
> littersMCMC3_cpp (10000)
```
NIMBLE: extensible software for hierarchical models (r-nimble.org)
Litters MCMC: BUGS and JAGS

• 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 $a2$ and $b2$ modestly better than default NIMBLE MCMC
• JAGS slice sampler gives similar performance 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).
  • 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.
Stepping outside the MCMC box:
maximum likelihood/empirical Bayes via MCEM

> littersMCEM <- buildMCEM(littersModel, latentNodes = 'p')
> littersMCEM(maxit = 500, m1 = 500, m2 = 5000)

- Gives estimates consistent with direct ML estimation (possible in this simple model with conjugacy for ‘p’) to 2-3 digits
- VERY slow to converge, analogous to MCMC mixing issues
- Stochasticity in the embedded MCMC makes this basic MCEM unstable; a more sophisticated treatment should help here

Many algorithms are of a modular nature/combine other algorithms, e.g.
- particle MCMC
- normalizing constant algorithms
- many, many others in the literature in the last 15 years
Status of NIMBLE and Next Steps

• First release was last week; lots to do, including:
  – Improve the user interface and speed up compilation
  – Refinement/extension of the DSL for algorithms
  – Enhance current algorithms provided (e.g., add multivariate conjugate updates for MCMC)
  – Additional algorithms written in NIMBLE DSL (e.g., particle MCMC)
  – Advanced features (e.g., auto. differentiation, paralleliz’n)

• Interested?
  – 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