Beyond the black box: Using, programming, and sharing hierarchical modeling algorithms such as MCMC and Sequential MC 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, Mathematics and Statistics
Claudia Wehrhahn	UC Santa Cruz, Applied Math and Statistics
Abel Rodriguez	UC Santa Cruz, Applied Math and Statistics
Nick Michaud	UC Berkeley Statistics and ESPM

https://r-nimble.org

ISBA Edinburgh World Meeting

June 2018

Funded by the US National Science Foundation

What do we want to do with hierarchical models?

1. More and better MCMC
- Many different samplers
- Better adaptive algorithms

2. Numerical integration
- Laplace approximation
- Adaptive Gaussian quadrature
- Hidden Markov models

3. Maximum likelihood estimation
- Monte Carlo EM
- Data cloning
- Monte Carlo Newton-Raphson

4. Sequential Monte Carlo
- Auxiliary Particle Filter
- Ensemble Kalman Filter
- Unscented Kalman Filter

5. Normalizing constants (AIC or Bayes Factors)
- Importance sampling
- Bridge sampling
- Others

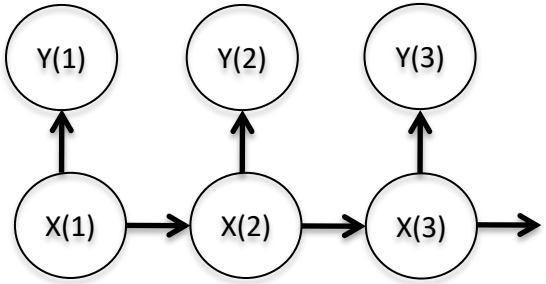
6. Model assessment
- Bootstrapping
- Calibrated posterior predictive checks
- Cross-validation
- Posterior re-weighting

7. Idea cominbations
- PF + MCMC
- Resample-move
- MCMC + Laplace/quadrature

These are just some ideas from a vast literature.

NIMBLE

Model language (BUGS/JAGS)



+

Algorithm Language

NIMBLE makes BUGS extensible from R:

- Add new functions
- Add new distributions
- Call external code

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

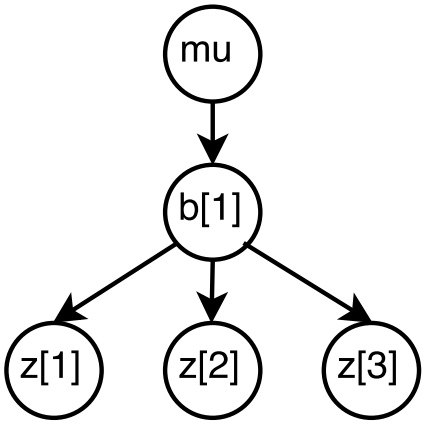
NIMBLE: Programming with models

NIMBLE Model Object

Model API (interface)

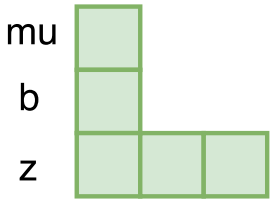
Graph queries:

- Dependencies
- Topological order
- Node traits




```
> model$getDependencies('b[1]')
> model$getDistribution('mu')
```

Data structures




```
> model$mu
> values(model, c('mu', 'z[1]')
```

Node Functions

mu: calculate, simulate, getLogProb, getParam

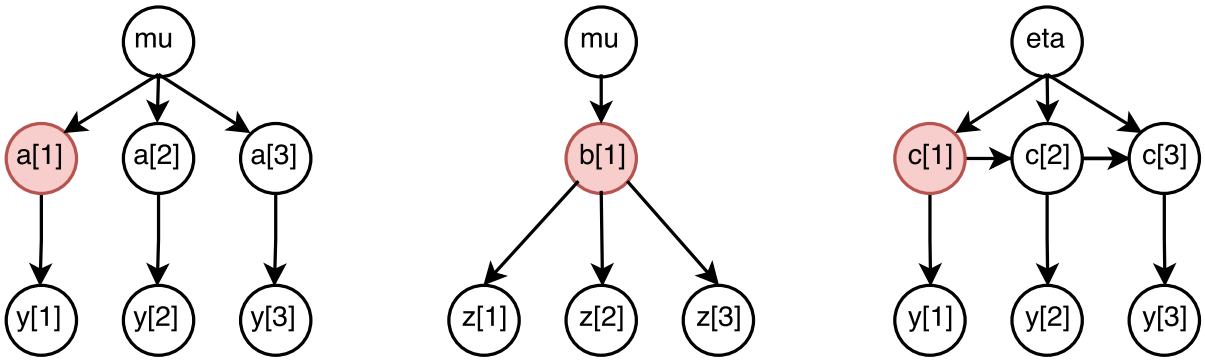
b: calculate, simulate, getLogProb, getParam

z: calculate, simulate, getLogProb, getParam

```
> model$simulate('b[1]')
> model$getParam('mu', 'sd')
```

Model-generic algorithm programming

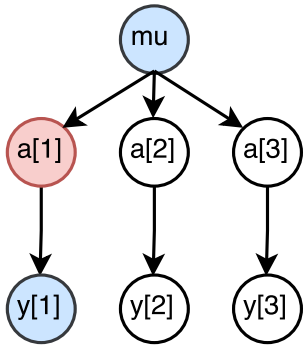
Wanted: a Metropolis-Hastings sampler with normal random-walk proposals.



Challenge: It should work for any node of any model.

Solution: Two-stage evaluation.

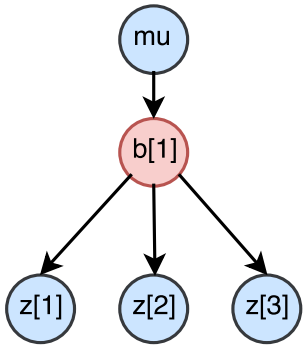
Model-generic algorithm programming



Model A

Target Node: "a[1]"

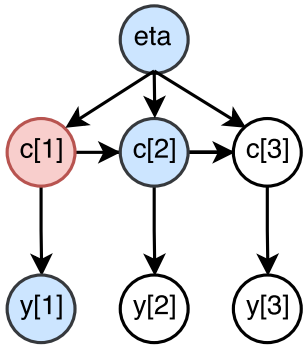
Calculation Nodes:
"mu", "a[1]", "y[1]"



Model B

Target Node: "b[1]"

Calculation Nodes:
"mu", "b[1]", "z[1]", "z[1]", "z[3]"



Model C

Target Node: "c[1]"

Calculation Nodes:
"mu", "c[1]", "c[2]", "y[1]"

NIMBLE: Model-generic programming

```
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 … #    })
```

2 kinds of functions

NIMBLE: Model-generic programming

```
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 … #    })
```

query model structure ONCE

NIMBLE: Model-generic programming

```
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 … #    })
```

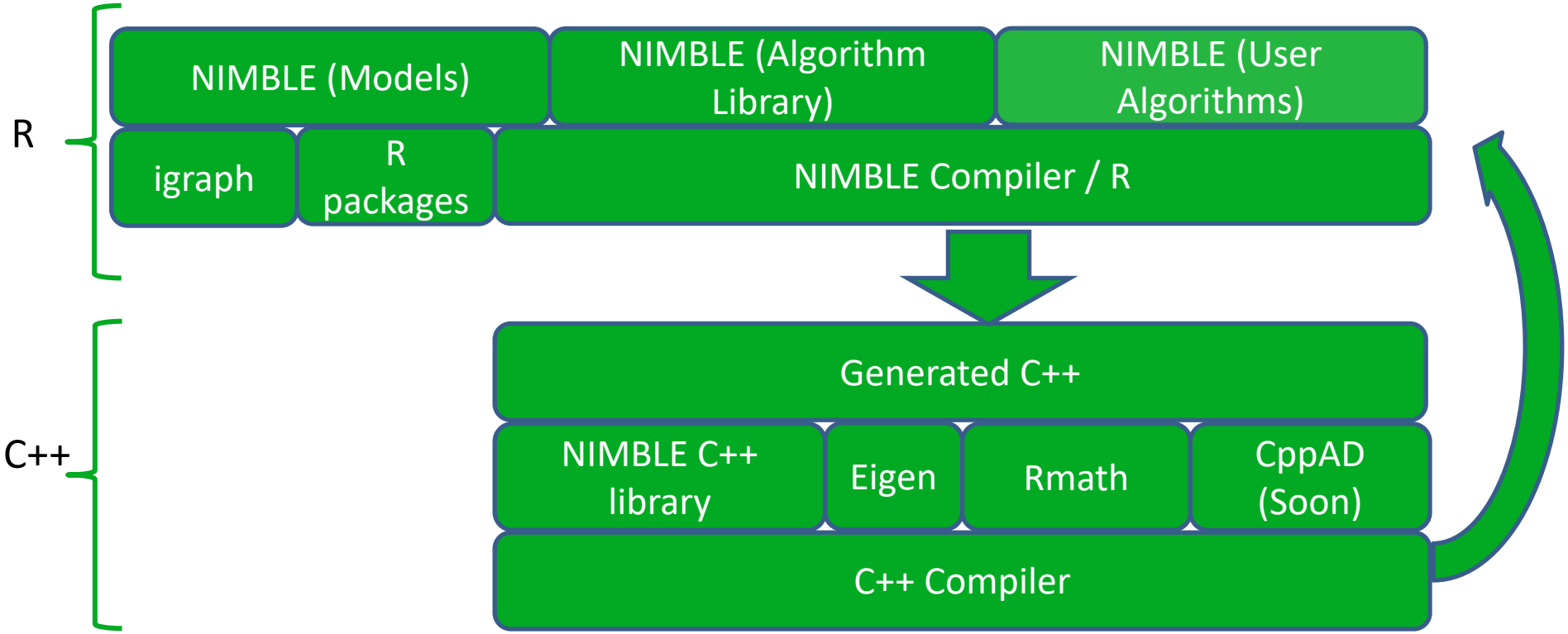
the actual (generic) algorithm

The NIMBLE compiler (run code)

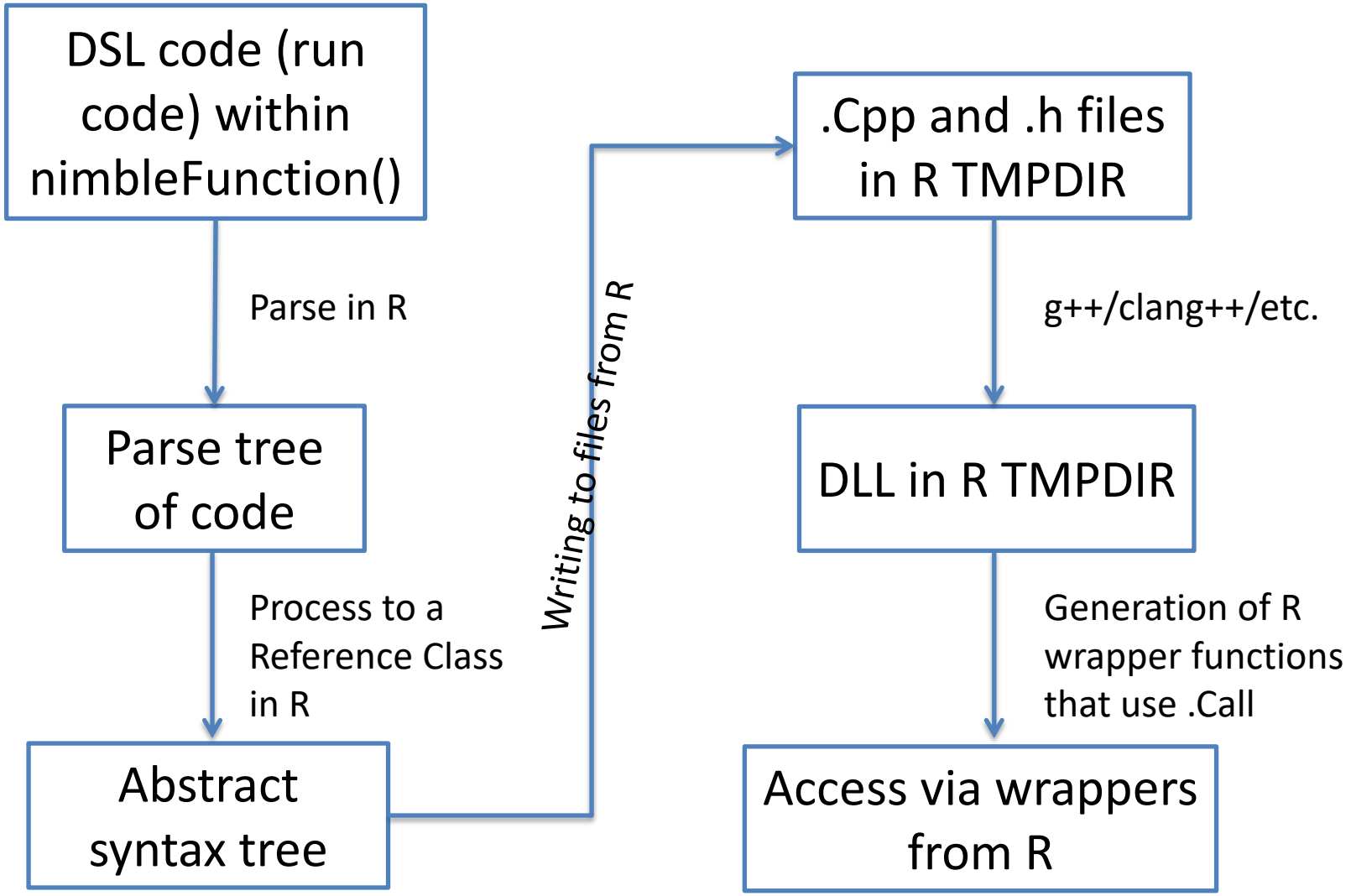
Feature summary:

- 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)
- Call out to your own C/C++ or back to R
- Many improvements / extensions planned
  - Derivatives (coming soon)

NIMBLE software stack



How an Algorithm is Processed in NIMBLE



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, Chinese Restaurant Process)
  - 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's MCMC engine

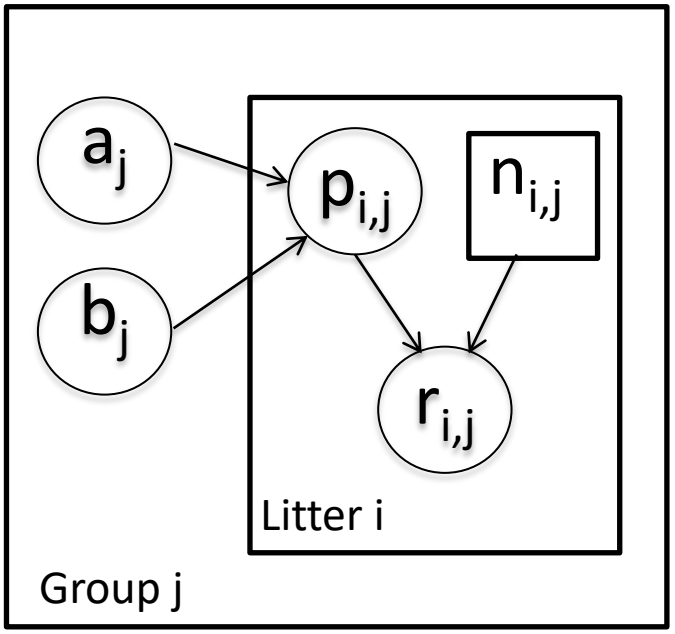
- One master nimbleFunction
  - initialize model values and sampler control parameters
  - iterate over MCMC iterations
    - cycle through samplers
    - save current parameter values
- Each kind of sampler is a nimbleFunction
- Default rules for "configuring" the MCMC (assigning samplers to the model nodes)
- Users can override the defaults
  - Choose different NIMBLE-provided samplers
  - Block parameters arbitrarily
  - Write their own samplers (a nimbleFunction!) and use them immediately

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(I 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);
  })
```

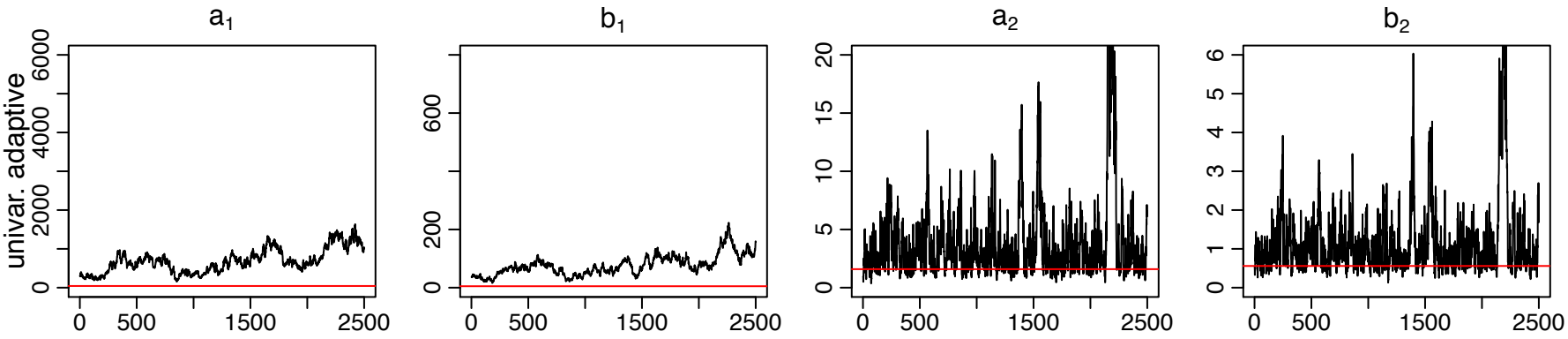


Challenges of the toy example:

- 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

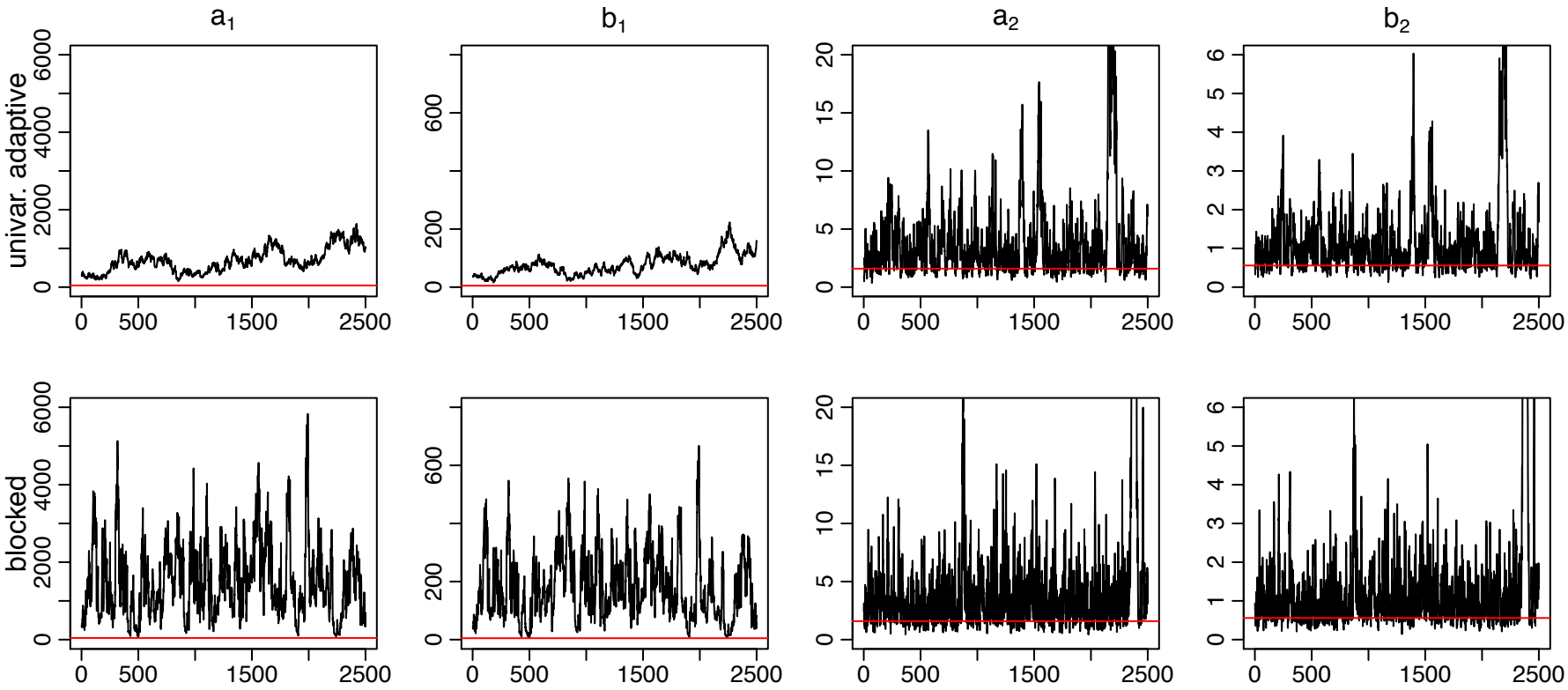
```
> littersConf <- configureMCMC(littersModel, control = list(adaptInterval = 100))
> littersMCMC <- buildMCMC(littersConf)
> littersMCMC_cpp <- compileNIMBLE(littersModel, project = littersModel)
> littersMCMC_cpp$run(10000)
```



Red line is MLE

Blocked MCMC: Gibbs + Blocked Metropolis

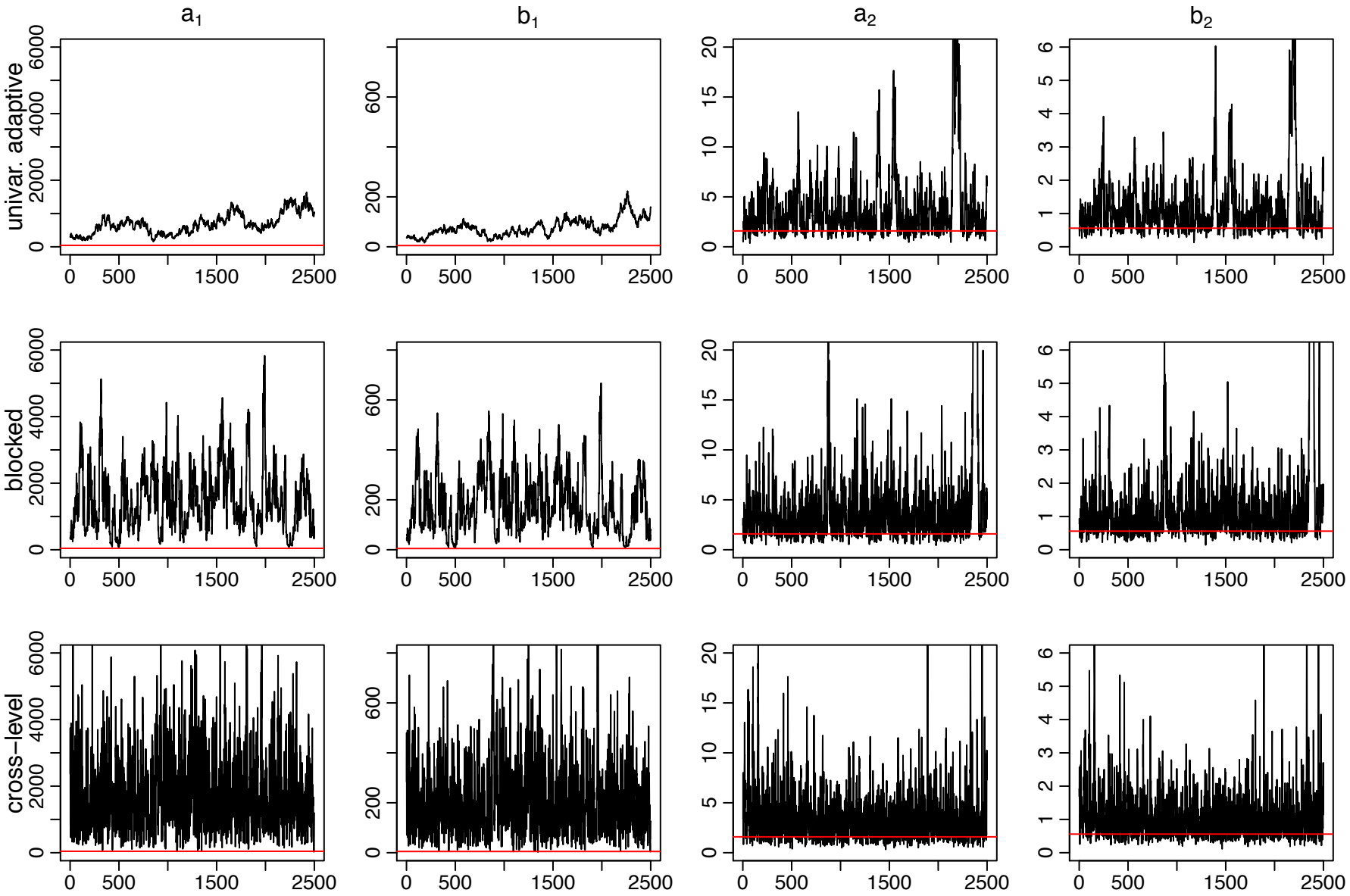
```
> littersConf2 <- configureMCMC(littersModel, control = list(adaptInterval = 100))
> littersConf2$addSampler(c('a[1]', 'b[1]'), 'RW_block', list(adaptInterval = 100)
> littersConf2$addSampler(c('a[2]', 'b[2]'), 'RW_block', list(adaptInterval = 100)
> littersMCMC2 <- buildMCMC(littersConf2)
> littersMCMC2_cpp <- compileNIMBLE(littersMCMC2, project = littersModel)
> littersMCMC2_cpp$run(10000)
```


a1
b1
a2
b2
univar. adaptive
blocked
0
500
1500
2500
2000
4000
6000
200
600
5
10
15
20
1
2
3
4
5
6

Blocked MCMC: 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.
  - Knorr-Held and Rue (2002)'s 'one-block' sampler

```
> littersConf3 <- configureMCMC(littersModel, control = list(adaptInterval = 100))
> topNodes1 <- c('a[1]', 'b[1]')
> littersConf3$addSampler(topNodes1, 'crossLevel', list(adaptInterval = 100)
> topNodes2 <- c('a[2]', 'b[2]')
> littersConf3$addSampler(topNodes2, 'crossLevel', list(adaptInterval = 100)
> littersMCMC3 <- buildMCMC(littersConf3)
> littersMCMC3_cpp <- compileNIMBLE(littersMCMC3, project = littersModel)
> littersMCMC3_cpp$run(10000)
```


a1
b1
a2
b2
univar. adaptive
blocked
cross−level
0
2000
4000
6000
200
600
5
10
15
20
1
2
3
4
5
6
0
500
1500
2500

User-defined sampler

```
sampler_awesome <- nimbleFunction(

setup = function(model, mvSaved, targetNode, control) {
   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 … #    })
```

Bayesian nonparametrics

Goal: provide Dirichlet process (DP) and related nonparametric density modeling in a general system.

Status:

- Two standard representations of Bayesian nonparametric mixture models based on DPs
  - Chinese restaurant process distribution for clustering observations
    - Collapsed Gibbs sampler
  - Stick-breaking finite-dimensional approximation
    - Blocked Gibbs sampler (conjugate beta-categorical samplers for stick-breaking parameters)
- Plans:
  - More efficient implementations
  - Dependent DP, hierarchical DP, etc.

DP Mixture example: Avandia meta-analysis

```
codeBNP <- nimbleCode({
    for(i in 1:nStudies) {
        y[i] ~ dbin(size = m[i], prob = q[i]) # avandia MIs
        x[i] ~ dbin(size = n[i], prob = p[i]) # control MIs
        q[i] <- expit(theta + gamma[i])       # Avandia log-odds
        p[i] <- expit(gamma[i])               # control log-odds
        gamma[i] ~ dnorm(muTilde[xi[i]],      # random effects (from mixture)
                         var = tauTilde[xi[i]])   # with mean/var from one component
    }
    # mixture component (cluster) parameters drawn from DP base measures
    for(i in 1:nStudies) {
        muTilde[i] ~ dnorm(mu0, sd = sd0)
        tauTilde[i] ~ dinvgamma(a0, b0)
    }
    # CRP for clustering studies to mixture components
    xi[1:nStudies] ~ dCRP(alpha, size = nStudies)
    # hyperparameters
    alpha ~ dgamma(1, 1)
    mu0 ~ dnorm(0, sd = 10)
    sd0 ~ dunif(0, 100)
    a0 ~ dunif(0, 100)
    b0 ~ dunif(0, 100)
    theta ~ dflat()          # effect of Avandia
})
```

DP Mixture example: Avandia meta-analysis

```
confBNP = configureMCMC(modelBNP, print = TRUE)

## [1] CRP_concentration sampler: alpha
## [2] conjugate_dflat_dnorm sampler: mu0
## [3] RW sampler: sd0
## [4] RW sampler: a0
## [5] RW sampler: b0
## [6] RW sampler: theta
## [7] CRP sampler: xi[1:48]
## [8] conjugate_dnorm_dnorm_dynamicDeps sampler: muTilde[1]
## [9] conjugate_dnorm_dnorm_dynamicDeps sampler: muTilde[2]
## [10] conjugate_dnorm_dnorm_dynamicDeps sampler: muTilde[3]
## ….
## [104] RW sampler: gamma[1]
## [105] RW sampler: gamma[2]
## [106] RW sampler: gamma[3]
## [107] RW sampler: gamma[4]
```

DP Mixture example: Meta-analysis with stick-breaking

```
codeBNP <- nimbleCode({
    for(i in 1:nStudies) {
        y[i] ~ dbin(size = m[i], prob = q[i]) # avandia MIs
        x[i] ~ dbin(size = n[i], prob = p[i]) # control MIs
        q[i] <- expit(theta + gamma[i])       # Avandia log-odds
        p[i] <- expit(gamma[i])               # control log-odds
        gamma[i] ~ dnorm(muTilde[xi[i]],      # random effects (from mixture)
                         var = tauTilde[xi[i]])   # with mean/var from one component
        xi[i] ~ dcat(prob[1:nSub])            # cluster assignment
    }
    # mixture component (cluster) parameters drawn from DP base measures
    for(i in 1:nStudies) {
        muTilde[i] ~ dnorm(mu0, sd = sd0)
        tauTilde[i] ~ dinvgamma(a0, b0)
    }
    prob[1:nSub] <- stick_breaking(z[1:(nSub-1)])
    for(i in 1:(nSub-1))
        z[i] ~ dbeta(1, alpha)
    # hyperparameters
    alpha ~ dgamma(1, 1)
    mu0 ~ dflat()
    sd0 ~ dunif(0, 100)
    a0 ~ dunif(0, 100)
    b0 ~ dunif(0, 100)
    theta ~ dflat()          # effect of Avandia
})
```

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: $p(x_t|y_{1:t}, \theta)$

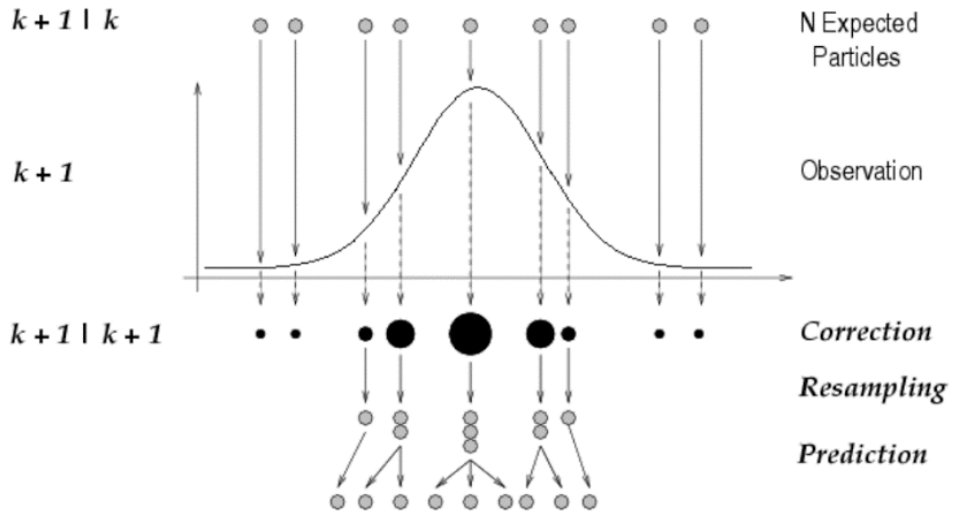


Figure 2: Particle filtering (from (Lehmann 2003))

- 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$latents, control =
list(saveAll = TRUE, smoothing = TRUE, lookahead = lookahead))
    ....
},
run = function() {
    ....
    modelLogProb0 <- modelLogLik0 + calculate(model, target)
    propValue <- rnorm(1, mean = model[[target]], sd = scale)
    model[[target]] <<- propValue
    modelLogLik1 <- my_particleFilter$run(m)
    modelLogProb1 <- modelLogLik1 + calculate(model, target)
    jump <- decide(modelLogProb1 - modelLogProb0)
    ....
})
```

nested nimbleFunction (composable algorithm)

Reproducible Research

Algorithm/methods developers tend to write their own algorithms in R, Python, MATLAB, or C/C++ because they are implementing new computations and need language flexibility.

Drawbacks:

- code may not be discoverable by a large audience
- users must become familiar with a new system
- parts of the code may duplicate other work

Applied statisticians tend to write their own model-specific code in R, Python, MATLAB, or C/C++ because available software is not flexible enough (e.g., unusual model structures, specialized MCMC samplers, specialized numerical implementation).

Drawbacks:

- users may not have the expertise to implement some methods
- parts of the code may duplicate other work, requiring additional work of the analyst
- code may not be well-tested

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)

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)
  - Ongoing Bayesian nonparametrics with Claudia Wehrhahn & Abel Rodriguez
  - 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 or programmer**
  - **We have funding to bring selected users to Berkeley for intensive collaboration**
  - 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