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

Perry de Valpine (PI)	UC Berkeley Environmental Science, Policy and Managem't
Daniel Turek	UC Berkeley Statistics and ESPM
Christopher Paciorek	**UC Berkeley Statistics**
Ras Bodík	UC Berkeley Electrical Engineering and Computer Science
Duncan Temple Lang	UC Davis Statistics

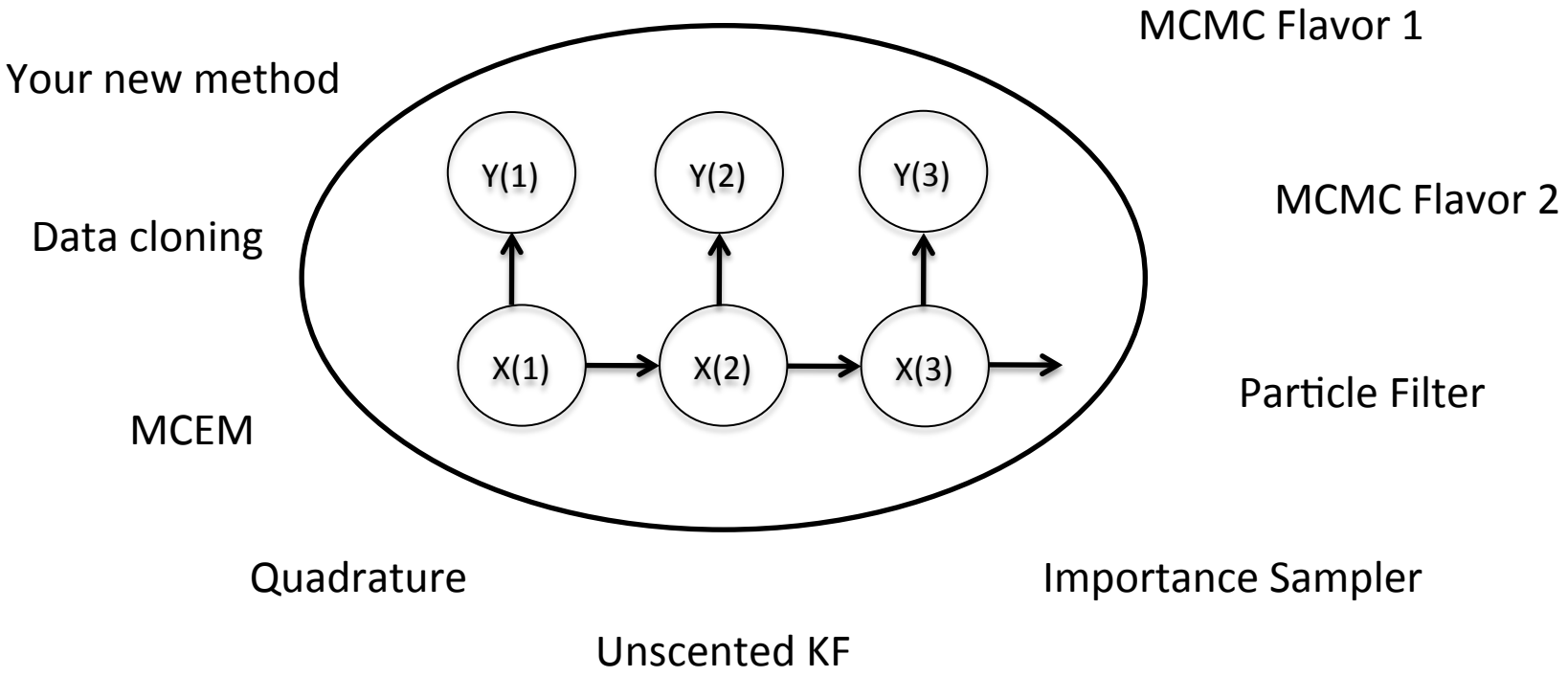
MCMSki 2014, Chamonix
January, 2014

Funded by NSF DBI-1147230

Background and Goals

- Software for fitting Bayesian models has opened their use to a wide variety of communities
- Most software for fitting hierarchical 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

Divorcing Model Specification from Algorithm



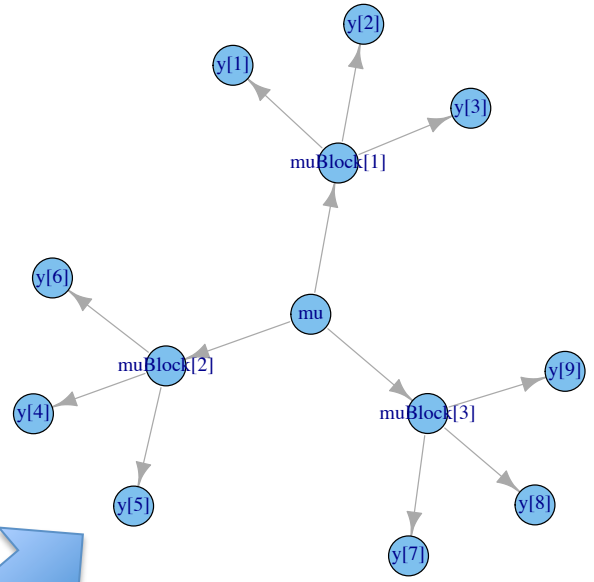
NIMBLE Design

- 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: Processing a BUGS Model

```
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);
  })
```



1

Parse and process BUGS code (R parse()).
Collect information in model object.

2

Use igraph plot method.

```
> littersModel <- BUGSmodel(littersModelCode, setupData = list(N = 16, G = 2, n = data))
```

3

Provides variables and functions for algorithms to use.

User Experience: Specializing an Algorithm to a Model

```
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);
 })
```

```
updater.RW.Normal <- nimbleFunction(
  ….
    origValue <- model[[targetNode]]
    propValue <- rnorm(1, mean = origValue, sd = scale)
    logProbCurrent <- getLogProb(model, calcNodes)
   model[[targetNode]] <- probValue
   logProbProposed <- calculate(model, calcNodes)
   logProbProposal <- dnorm(propValue, mean = origValue, sd = scale, log
= TRUE)
    …
```

```
> littersMCMCspec <- MCMCspec(littersModel, adaptInterval = 100)
> getUpdaters(littersMCMCspec)
Updater for nodes: beta
type: RW
rwInfo (list):
 --> 'scale' (numeric): 0.1
 --> 'adapt' (logical): TRUE
 --> 'propCov' (character): identity
 [...snip...]
> addUpdater(littersMCMCspec, updater(c('a', 'b'), 'Rwblock', rwInfo = list(scale = 0.1))
> addMonitor(littersMCMCspec, 'a'); addMonitor(littersMCMCspec, 'b')
> littersMCMC <- buildMCMC(littersMCMCspec)
> littersMCMC_Cpp <- compileToCpp(littersModel, littersMCMC)

> littersMCMC_Cpp$littersMCMC(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);
 })
```

```
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, paramNodes = c('a', 'b'), latentNodes = 'p')

> littersMCEM_Cpp <- compileNIMBLE(littersModel, littersMCEM)

> set.seed(0)
> littersMCEM_Cpp$littersMCEM(init = c(1000, 10, 100, 1), mcmcIts = 1000, tol = 1e-6)
```

Modularity:

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

- 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, boolean)
    - Basic (vectorized) math and distribution/probability calculations
    - Basic data storage classes ("modelValues")
    - Control structures – for loops and if-then-else
    - Functions
    - Linear algebra (via the Eigen package)
- Function definitions in the DSL include code for two steps:
    - A general function is written for any model structure
    - When a model is provided, a set of one-time (compile-time) processing is executed based on the model structure
    - Run-time code can use information determined from the compile-time processing
    - Compile-time processing is executed in R. Run-time processing can be compiled to C++

Programmer Experience: Creating an Algorithm

```
myAlgorithmGenerator <- nimbleFunction (

 compileArgs = list(model, …),

 runTimeArgs = list(…),

 setupCode = {

  # code that does the specialization of algorithm to model

 },

 runTimeCode = {

  # code that carries out the generic algorithm

 },

 returnType = double()
)
```

5 sections to a NIMBLE function.

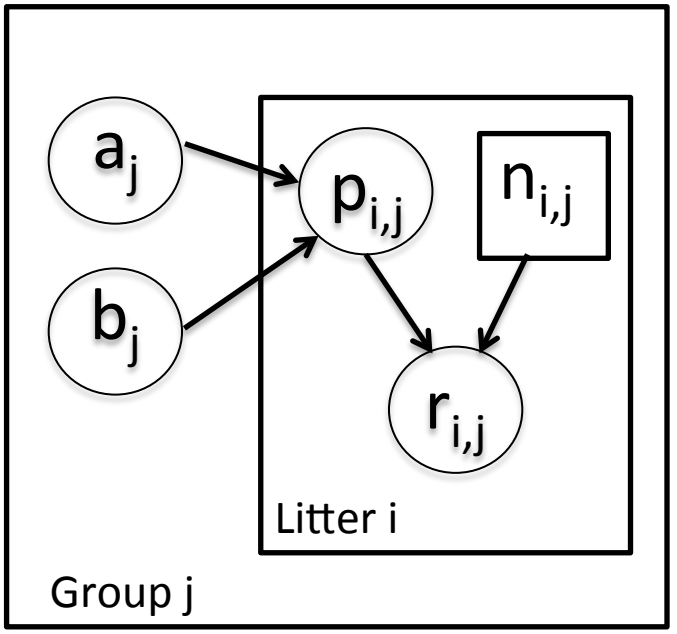
Programmer Experience: Metropolis Updater Example

```
updater.RW.Normal <- nimbleFunction(
  compileArgs = list(model, savedValues, targetNode),
  runTimeArgs = list(scale = double(default=0.1)),
  setupCode = {
        calcNodes <- getDependencies(model, targetNode) },
  runTimeCode = {
    origValue <- double(); propValue <- double(); logProbs  <- double(2); jump <- int()

    logProbs[2] <- getLogProb(model, calcNodes)       # original value model logProb
    propValue <- rnorm(1, mean = model[[targetNode]], sd = scale)
    model[[targetNode]] <- propValue
    logProbs[1] <- calculate(model, calcNodes)        # proposal value model logProb

    jump <- decide(logProbs[1] - logProbs[2])
    if(runtime(jump)) {
       copy(model,        savedValues[[1]], calcNodes, logProb = TRUE)
    }  else          {
       copy(savedValues[[1]], model,        calcNodes, logProb = TRUE)
     }
    return(jump)
  },
   returnType = int(),
)
```

NIMBLE in Action: the Litters Example

Beta-binomial for clustered binary response data

```
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);
 })
```

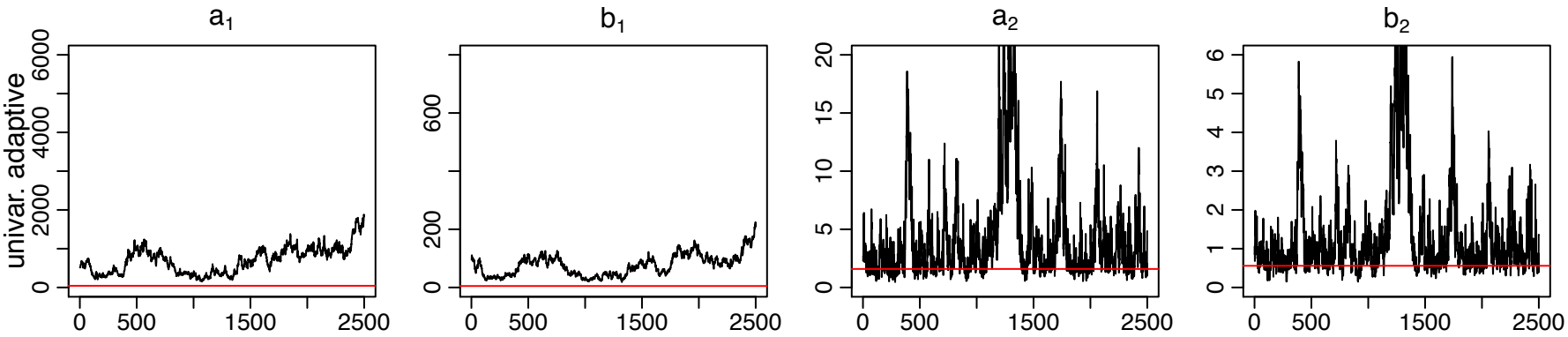


Challenges of the toy example:

- 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.

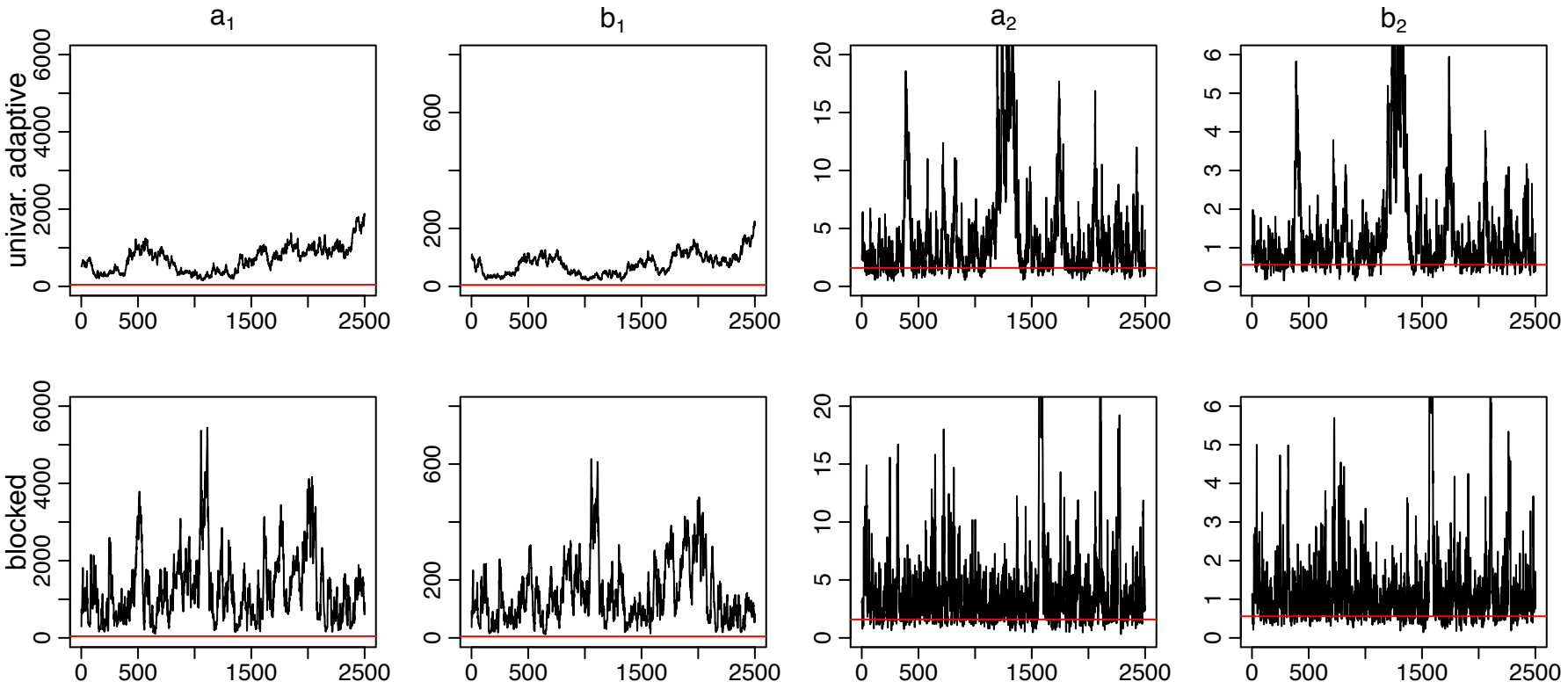
Default MCMC: Gibbs + Metropolis

```
> littersMCMCspec <- MCMCspec(littersModel, adaptInterval = 100)
> littersMCMC <- buildMCMC(littersMCMCspec)
> littersMCMC_Cpp <- compileNIMBLE(littersModel, littersMCMC)
> littersMCMC_Cpp$littersMCMC(10000)
```


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

Blocked MCMC: Gibbs + Blocked Metropolis

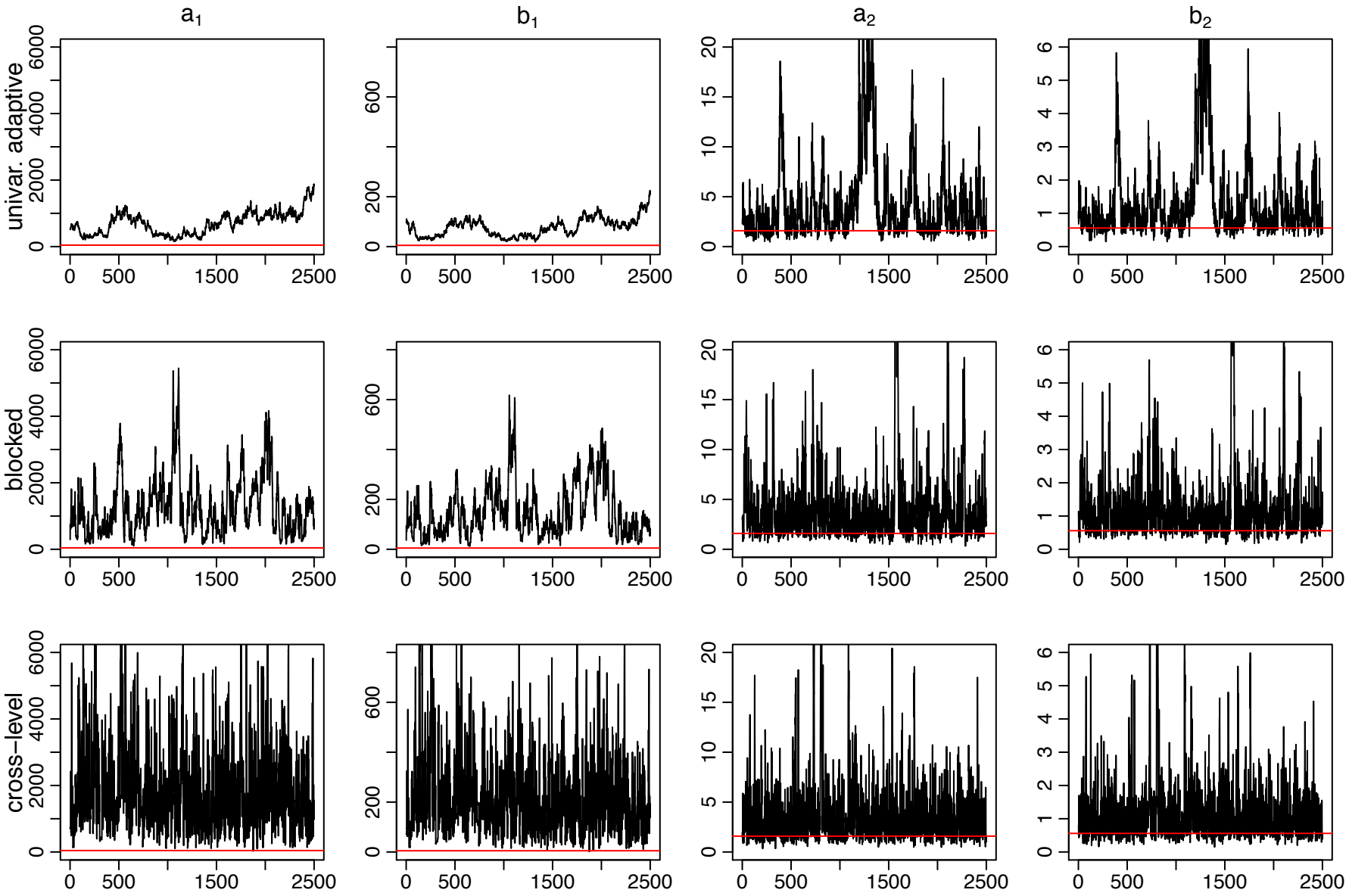
```
> littersMCMCspec2 <- MCMCspec(littersModel, adaptInterval = 100)
> addUpdater(littersMCMspec2, updater(c('a[1]', 'b[1]'), 'Rwblock', rwInfo = list(scale = 0.1))
> addUpdater(littersMCMspec2, updater(c('a[2]', 'b[2]'), 'Rwblock', rwInfo = list(scale = 0.1))
> littersMCMC2 <- buildMCMC(littersMCMCspec2)
> littersMCMC2_Cpp <- compileNIMBLE(littersModel, littersMCMC2)
> littersMCMC2_Cpp$littersMCMC2(10000)
```


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

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 integrating the dependent (latent) nodes out of the model.
- We can then add this updater to an MCMC for a given model……

```
> littersMCMCspec3 <- MCMCspec(littersModel, adaptInterval = 100)

> topNodes1 <- c('a[1]', 'b[1]')
> addUpdater(littersMCMCspec3, updater(nodes = topNodes1, type='crossLevel',   auxInfo=list(lowerNodes =
        getDependencies(littersModel, topNodes1, self = FALSE)

> topNodes2 <- c('a[2]', 'b[2]')
> addUpdater(littersMCMCspec3, updater(nodes = topNodes2, type='crossLevel',   auxInfo=list(lowerNodes =
        getDependencies(littersModel, topNodes2, self = FALSE)

> littersMCMC3 <- buildMCMC(littersMCMCspec3)
> littersMCMC3_Cpp <- compileNIMBLE(littersModel, littersMCMC3)
> littersMCMC3_Cpp$littersMCM3(10000)
```


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

Litters MCMC: BUGS and JAGS

- BUGS gives results as good or better than our cross-level MCMC.
  - I believe that BUGS must be, in essence, integrating over the latent nodes to achieve this.
  - However, without examining the source code, it's unclear what is going on.
- JAGS seems to perform well for the identifiable quantities.
  - But different runs give different posterior estimates for the poorly-identified a_j and b_j parameters.
  - Again, without examining the source code, it's unclear what is going on.
- Erratum: BUGS and JAGS give similar performance to the default NIMBLE MCMC; notes above based on permuted samples
- 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 tool – someone else could build a better default MCMC and distribute for use in our system.
- Cautionary note: NIMBLE results are based on code under development.

Stepping outside the MCMC box: maximum likelihood/empirical Bayes via MCEM

```
> littersMCEM    <- buildMCEM(littersModel, paramNodes = c('a', 'b'), latentNodes = 'p')
> littersMCEM_Cpp <- compileToCpp(littersModel, littersMCEM)

littersMCEM_Cpp$littersMCEM(init = c(getValues(littersModel, 'a'), getValues(littersModel, 'b')), mcmc.its =
1E3, tol = 1E-3)
```

- Gives estimates consistent with direct ML estimation 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.

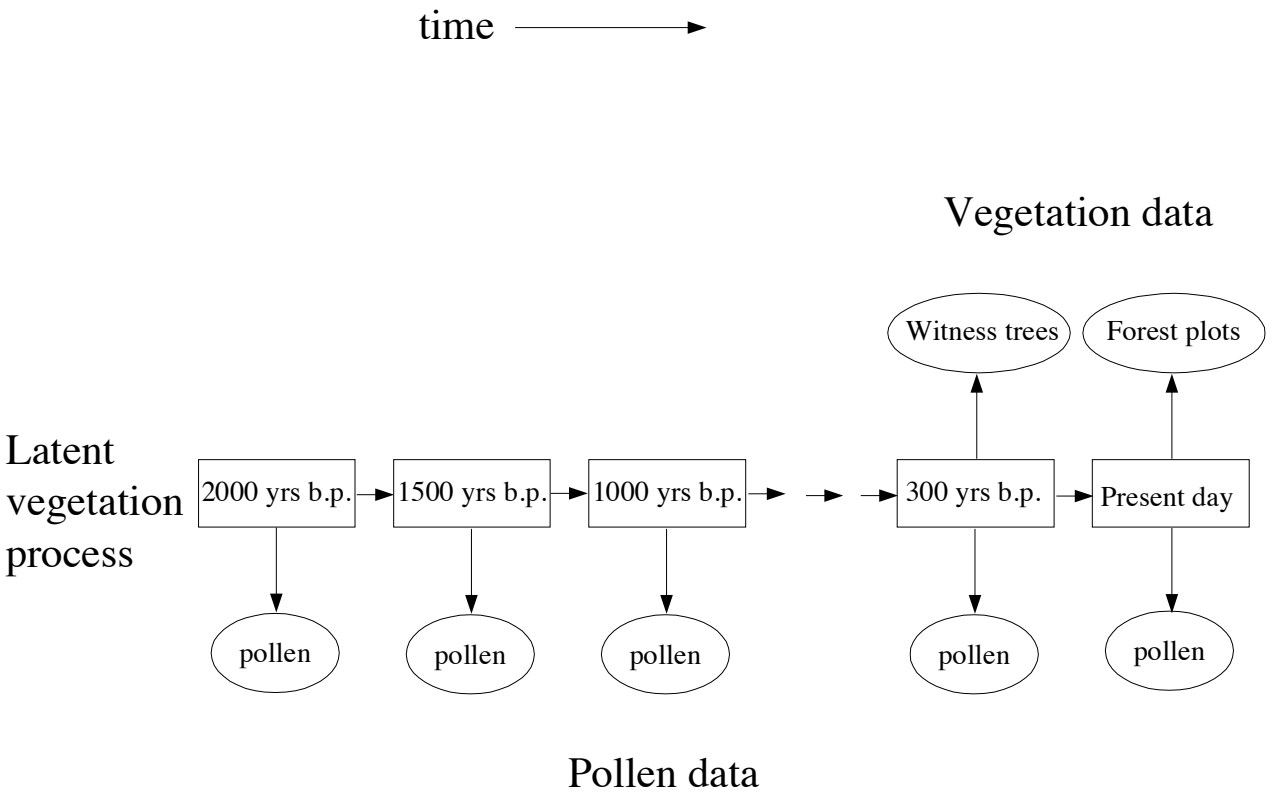
- particle MCMC
- normalizing constant algorithms
- posterior predictive simulations that are not just a drag on the MCMC

NIMBLE and modular modeling

- Modular modeling involves working with multiple submodels in an iterative and interactive workflow
  - Nodes might be fixed at constant values
  - Samples from one submodel may be used in another submodel
  - Subgraphs may be updated on their own
  - Simulation from the model may be useful
- The NIMBLE system provides the flexibility for these sorts of operations
  - Model is an R object you can query and manipulate
    - Functions to query the dependencies in a model
    - Simulate from model
    - Set values in the model
    - Calculate density values for nodes
  - Fixing nodes at constant values
  - Choosing to update only certain nodes
  - Cutting feedback
  - Combining algorithms in a modular fashion, with the components run as compiled C++ code

Paleoecology example

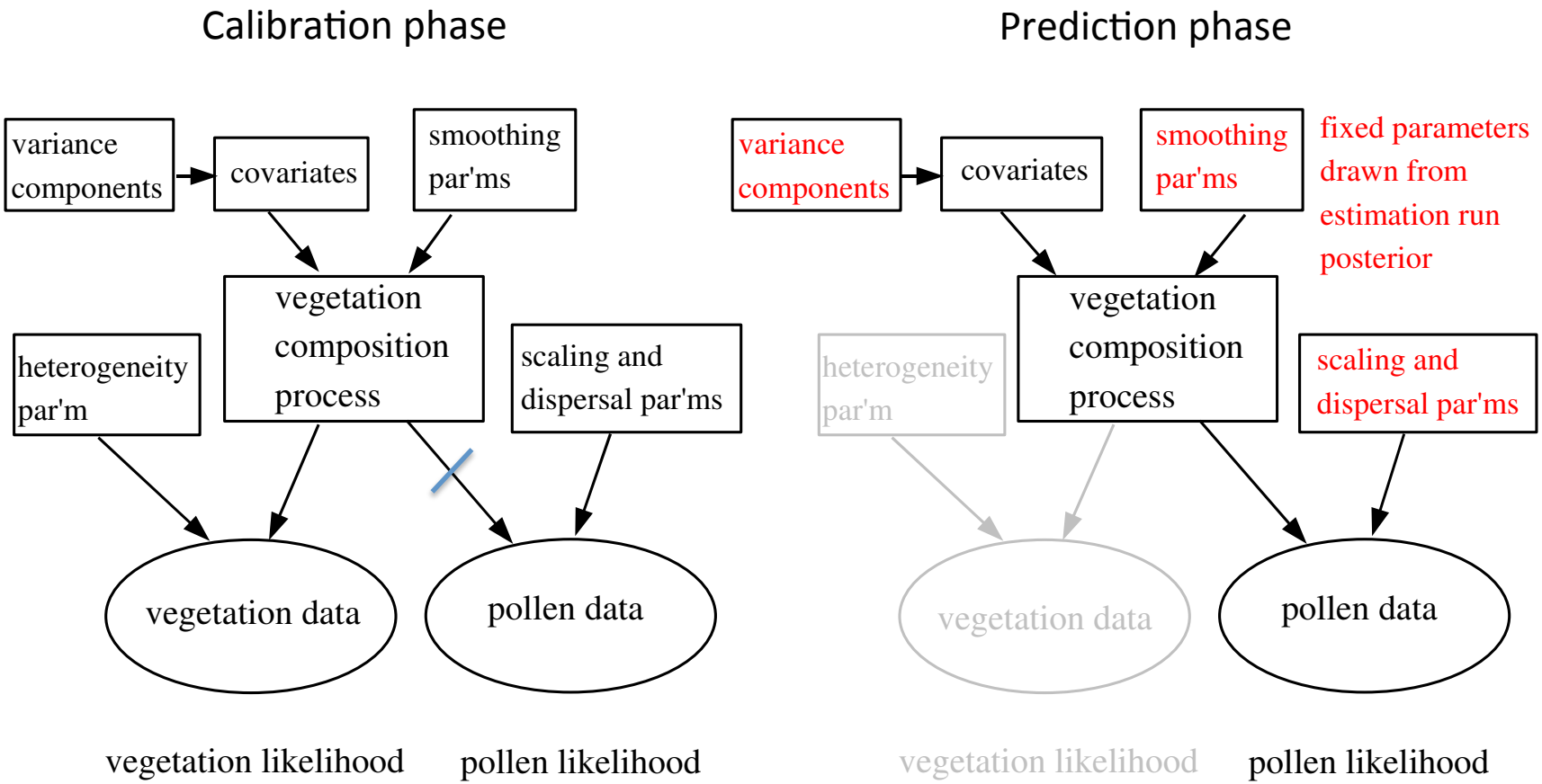
- Goal: predict vegetation composition from pollen deposits in lake sediments



Paleoecology example

- Goal: predict vegetation composition from pollen deposits in lake sediments
- Calibration phase: “regress” pollen composition on vegetation composition for time periods with vegetation data
- Prediction phase: predict vegetation in space-time from pollen composition over thousands of years
- Themes
  - Modular models
  - Cutting feedback
  - Running prediction model for multiple samples of calibration parameters
  - Flexible manipulation of MCMC sampling schemes, consideration of alternative algorithms

Paleoecology example



Status of NIMBLE and Next Steps

- Basic R package has been developed but lots to do, including:
  - Improved user interface
  - Refinement/extension of the DSL for algorithms
  - Extensions to the BUGS language
  - Additional algorithms written in NIMBLE DSL
- Interested? We're starting an email list [mailto: paciorek@berkeley.edu] and would like to
  - start broadening our group of developers and
  - initiate a group of users and algorithm programmers
- Initial release date targeted for late spring/early summer 2014.