# 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 UCSF/Gladstone Institutes (alumnus)
Duncan Temple Lang UC Davis Statistics

[http://r-nimble.org](http://r-nimble.org)

U. Minnesota Biostatistics seminar
April, 2015

Funded by NSF DBI-1147230

# What do we want to do with hierarchical models?

1. Core algorithms

- MCMC
- Sequential Monte Carlo
- Laplace approximation
- Importance sampling

# What do we want to do with hierarchical models?

1. Core algorithms

- MCMC
- Sequential Monte Carlo
- Laplace approximation
- Importance sampling

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

# What do we want to do with hierarchical models?

1. Core algorithms

- MCMC
- Sequential Monte Carlo
- Laplace approximation
- Importance sampling

3. Idea combinations

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

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

# What can a practitioner do with hierarchical models?

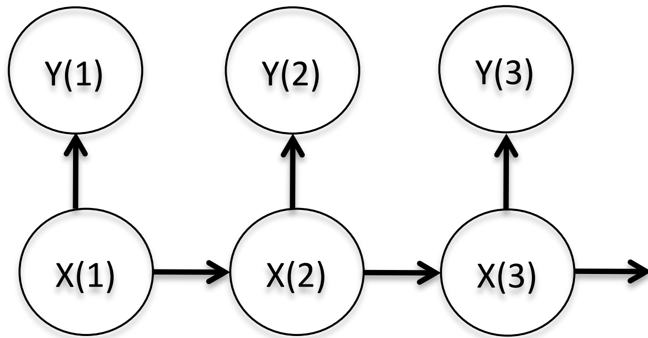
Two basic software designs:

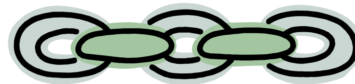
1. Typical R package = Model family + 1 or more algorithms
   - GLMMs: lme4, MCMCglmm
   - GAMMs: mgcv
   - spatial models: spBayes, INLA

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

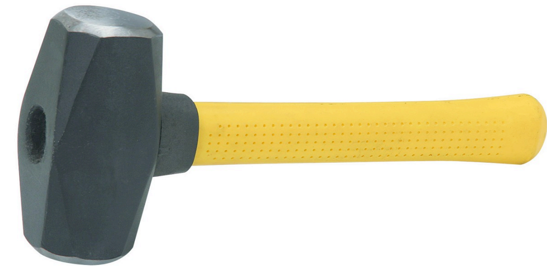
# Existing software

Model



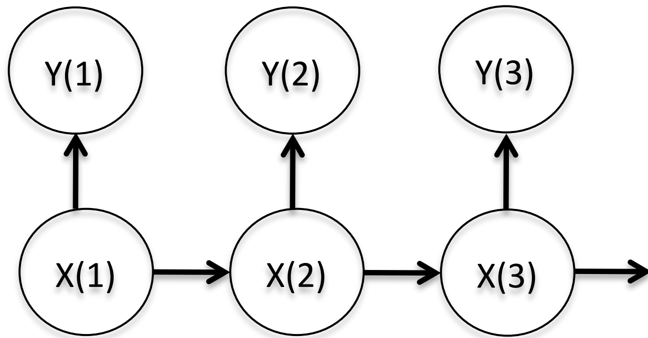
Algorithm



e.g., BUGS (WinBUGS, OpenBUGS, JAGS), INLA, Stan, various R packages

# NIMBLE: The Goal

Model

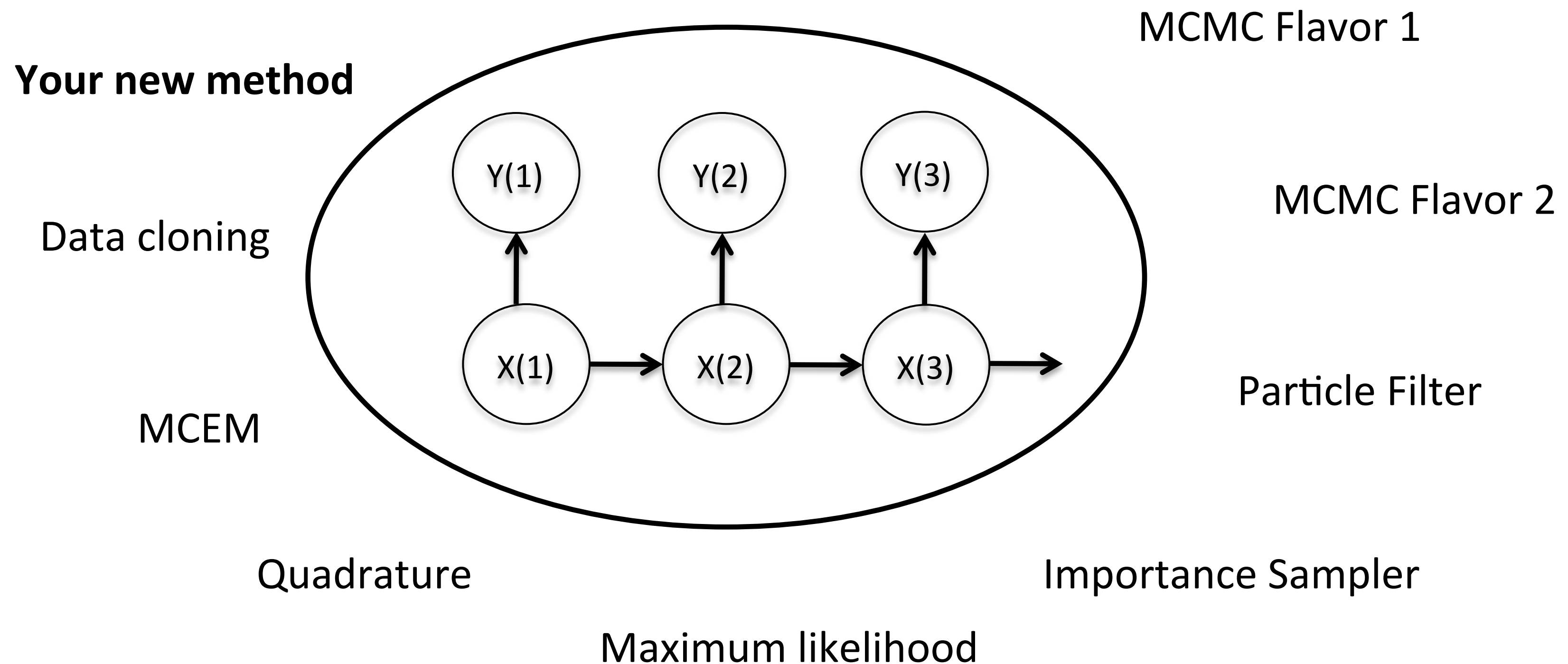



+

Algorithm language

=

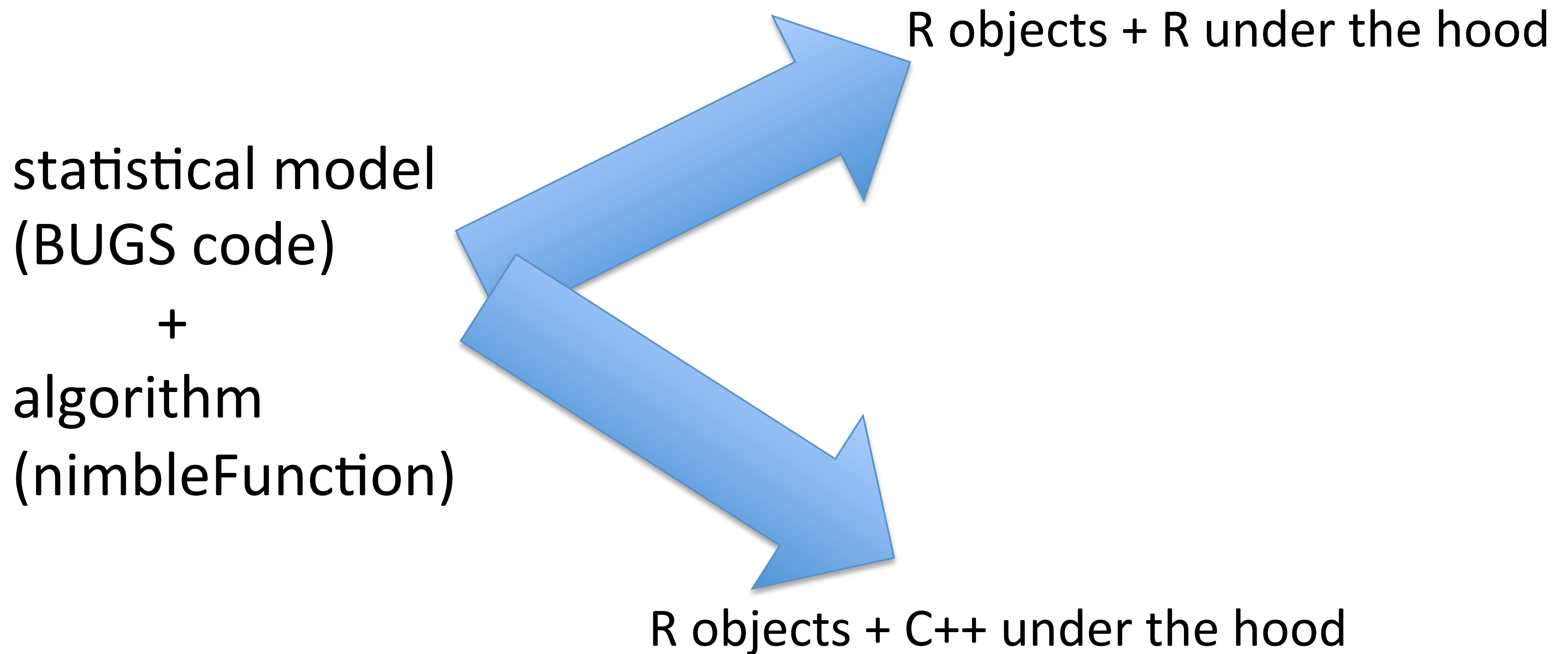
# Divorcing Model Specification from Algorithm



# 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 System Summary



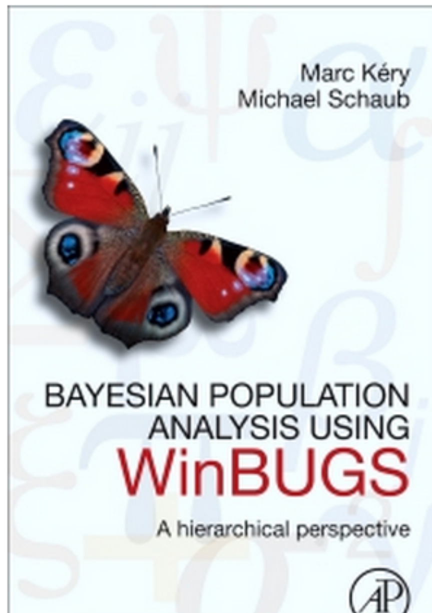

✧ We generate C++ code,
✧ compile and load it,
✧ provide interface object.

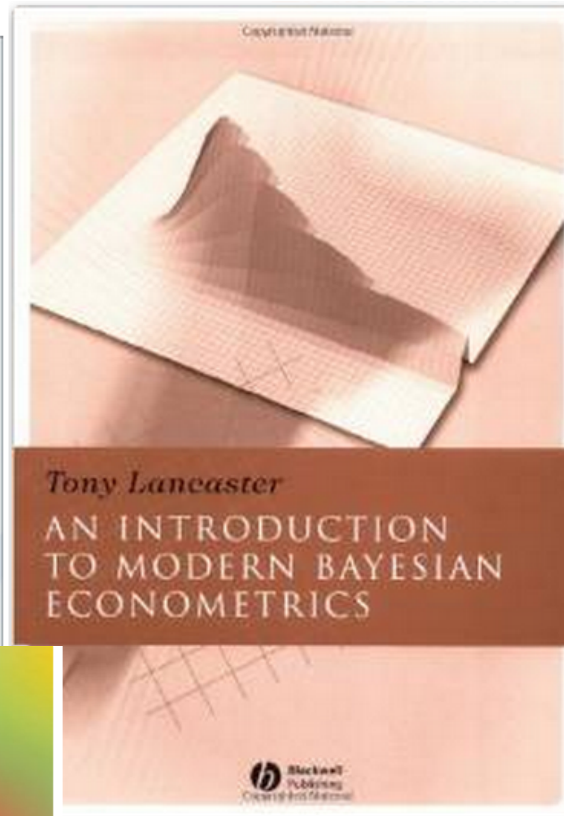
# NIMBLE

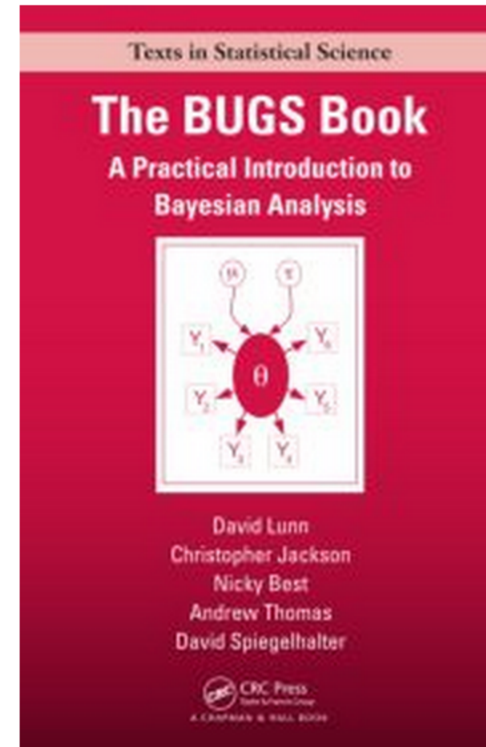
1. BUGS language ➔ R model object

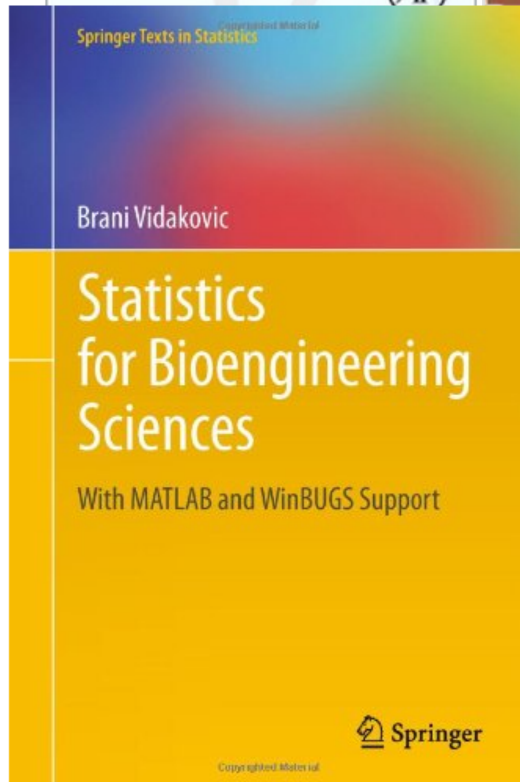
2. NIMBLE programming language within R
   - Compilation via C++

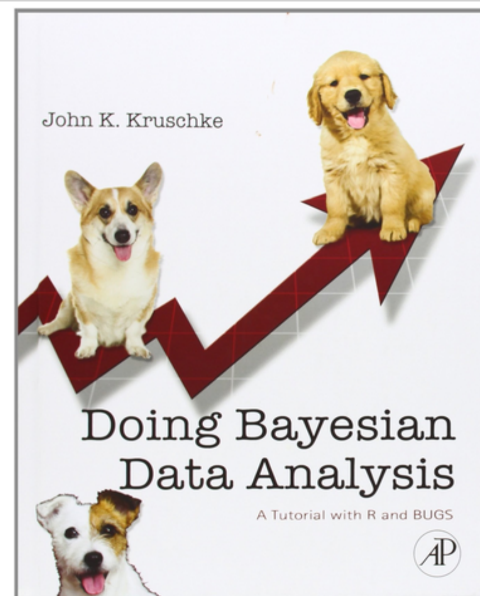
3. Algorithm library: MCMC, Particle Filter, etc.

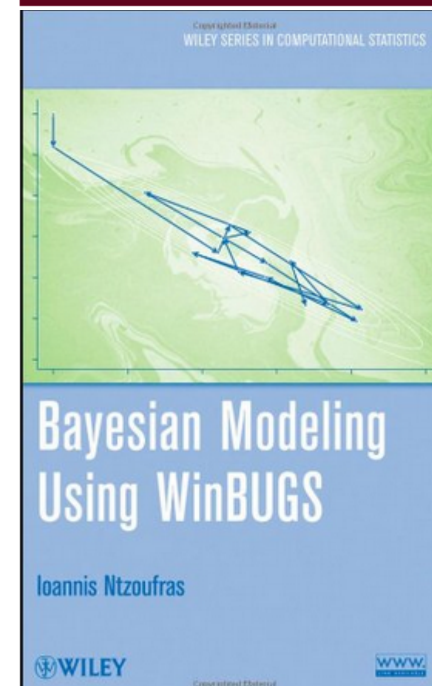

Marc Kéry
Michael Schaub
BAYESIAN POPULATION ANALYSIS USING
WinBUGS
A hierarchical perspective


Tony Lancaster
AN INTRODUCTION TO MODERN BAYESIAN ECONOMETRICS


Texts in Statistical Science
The BUGS Book
A Practical Introduction to Bayesian Analysis
David Lunn
Christopher Jackson
Nicky Best
Andrew Thomas
David Spiegelhalter
CRC Press

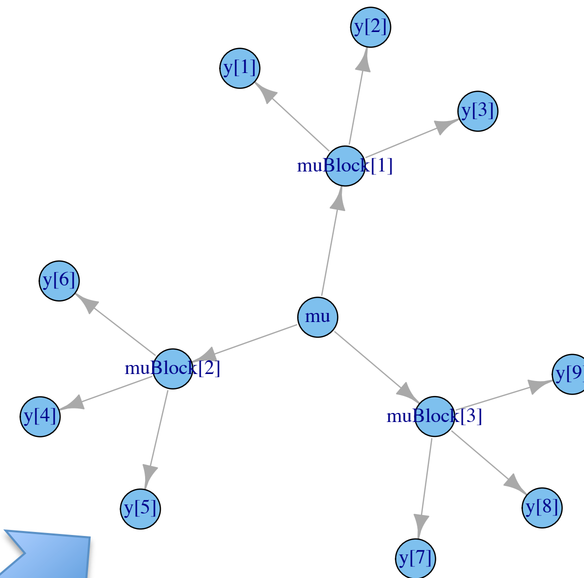

Springer Texts in Statistics
Brani Vidakovic
Statistics for Bioengineering Sciences
With MATLAB and WinBUGS Support
Springer


John K. Kruschke
Doing Bayesian Data Analysis
A Tutorial with R and BUGS


WILEY SERIES IN COMPUTATIONAL STATISTICS
Bayesian Modeling Using WinBUGS
Ioannis Ntzoufras
WILEY

# User Experience: Creating a Model from BUGS

```
littersCode <- 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.
Collect information in model object.

2

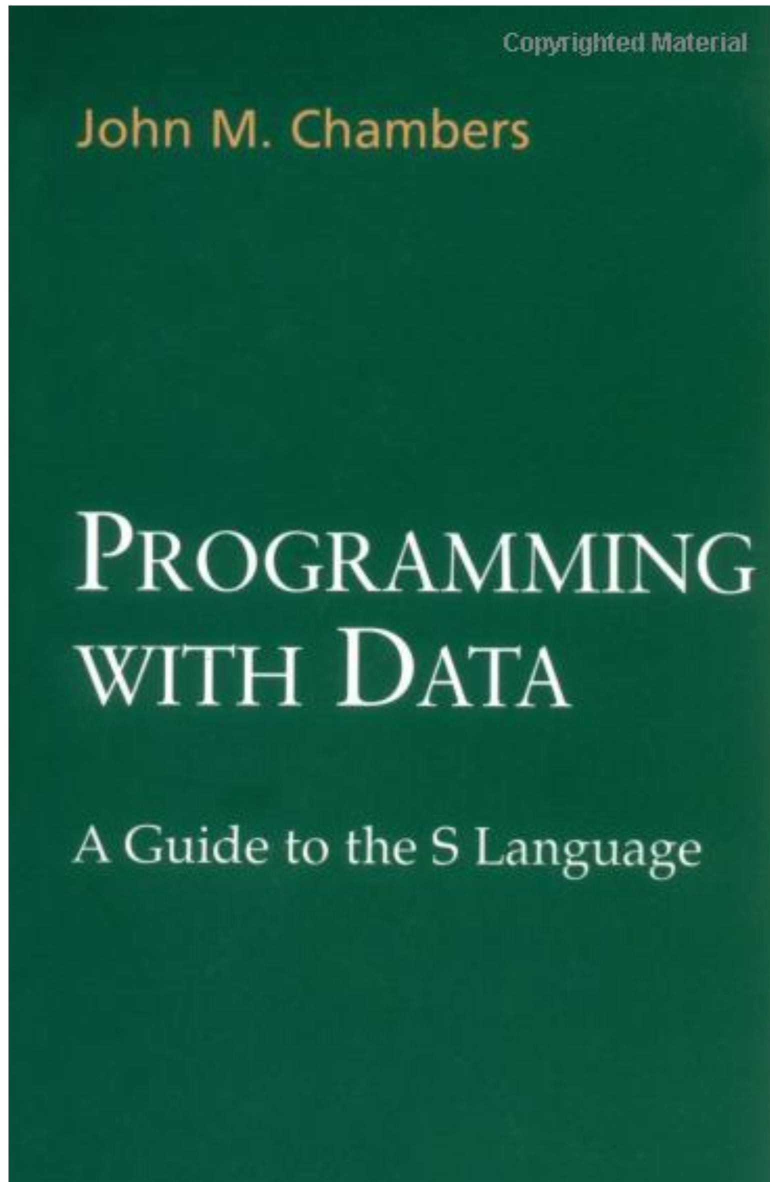
Use igraph plot method (we also use this to determine dependencies).

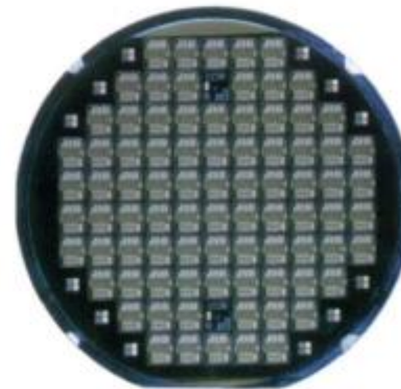
```
> littersModel <- nimbleModel(littersCode, constants = list(N = 16, G = 2), data = list(r = input$r))
> littersModel_cpp <- compileNimble(littersModel)
```

3

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

# The Success of R



# Programming with Models

You give NIMBLE:

```
littersCode <- 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);   } )
```

You get this:

```
> littersModel$a[1] <- 5
> simulate(littersModel, ‘p')
> p_deps <- littersModel$getDependencies(‘p’)
> calculate(littersModel, p_deps)
> getLogProb(pumpModel, ‘r')
```

NIMBLE also extends BUGS: multiple parameterizations, named parameters, expressions as arguments and (soon!) user-defined distributions and functions.

# User Experience: Specializing an Algorithm to a Model

```
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);
 })
```

```
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….]
…
```

```
> littersMCMCspec <- configureMCMC(littersModel)
> getUpdaters(littersMCMCspec)
[…snip…]
[3] RW sampler;   targetNode: b[1],  adaptive: TRUE,  adaptInterval: 200,  scale: 1
[4] RW sampler;   targetNode: b[2],  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$addSampler('slice', list(targetNode = 'a[1]', adaptInterval = 100))
> littersMCMCspec$addSampler('slice', list(targetNode = '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);
})
```

```
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), boxConstaints = 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. BUGS language ➔ R model object

2. NIMBLE programming language within R
   - Compilation via C++

3. Algorithm library: MCMC, Particle Filter, etc.

# NIMBLE: Programming With Models

We want:

- High-level processing (model structure) in R
- Low-level processing in C++

# NIMBLE: Programming With Models

```
objectiveFunction <- nimbleFunction (

 setup = function(model, nodes) {
  calcNodes <- model$getDependencies(nodes)
 },

run = function(P = double(1)) {
  values(model, nodes) <<- P
  sumLogProb <- calculate(model, calcNodes)
  return(sumLogProb)
  returnType(double())
 })
```

# NIMBLE: Programming With Models

```
objectiveFunction <- nimbleFunction (

 setup = function(model, nodes) {
  calcNodes <- model$getDependencies(nodes)
 },

 run = function(P = double(1)) {
  values(model, nodes) <<- P
  sumLogProb <- calculate(model, calcNodes)
  return(sumLogProb)
  returnType(double())
 })
```

query model structure ONCE.

# NIMBLE: Programming With Models

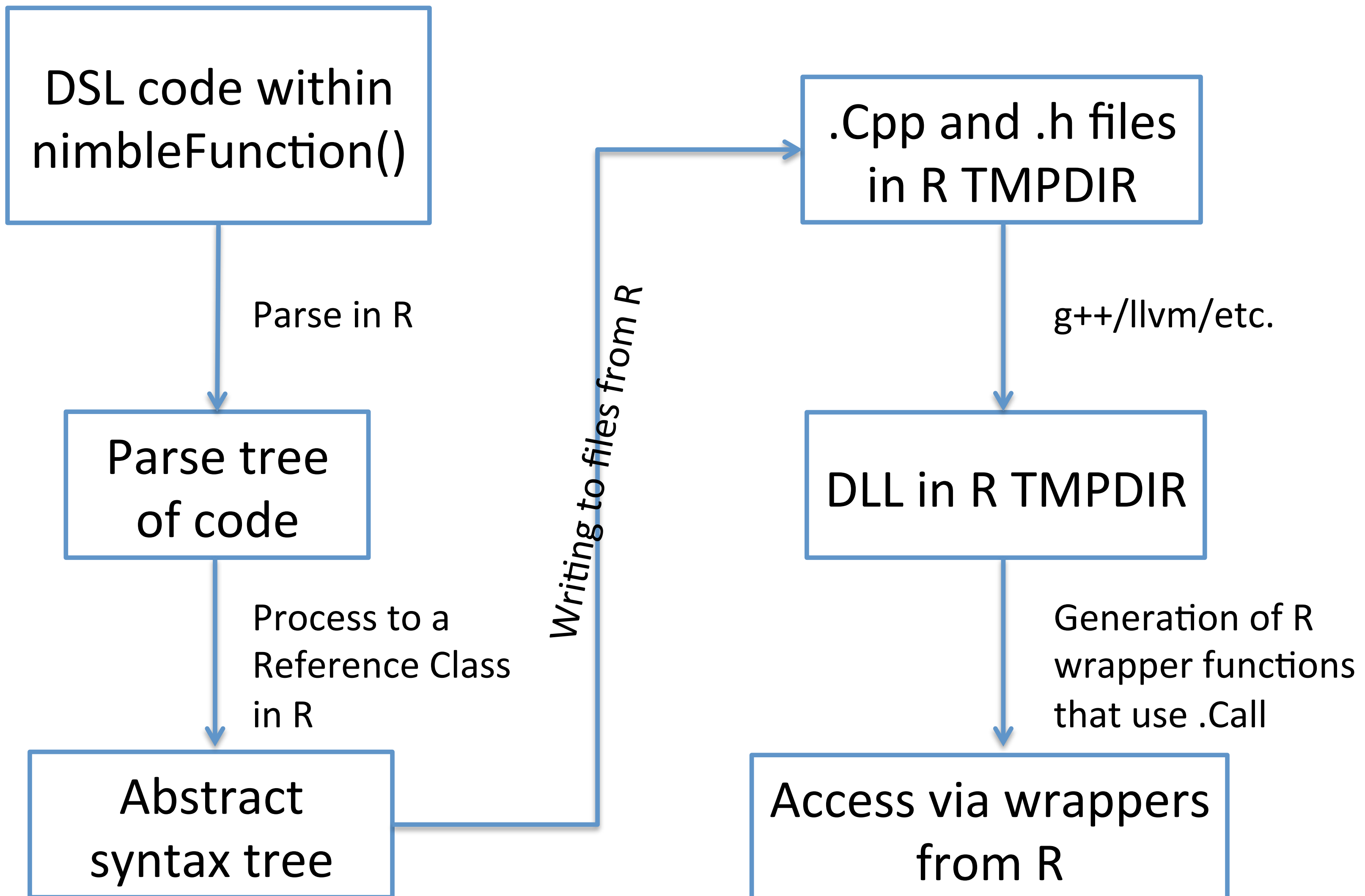
```
objectiveFunction <- nimbleFunction (

 setup = function(model, nodes) {
  calcNodes <- model$getDependencies(nodes)
 },

 run = function(P = double(1)) {
  values(model, nodes) <<- P
  sumLogProb <- calculate(model, calcNodes)
  return(sumLogProb)
  returnType(double())
 })
```

# The NIMBLE compiler

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
- Declare input & output types only
- Access to much of Rmath.h (e.g. distributions)
- Automatic R interface / wrapper
- Many improvements / extensions planned

# How an Algorithm is Processed in NIMBLE



# Programmer experience: Random walk updater

```
sampler_myRW <- nimbleFunction(contains = sampler_BASE,

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

run = function() {
   model_lp_initial <- getLogProb(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

    if(jump) {
      copy(from = model, to = mvSaved, row = 1,  nodes = calcNodes, logProb = TRUE)
     } else  copy(from = mvSaved, to = model, row = 1, nodes = calcNodes, logProb = TRUE)
   })
```

# NIMBLE

1. BUGS language ➔ R model object

2. NIMBLE programming language within R
   - Compilation via C++

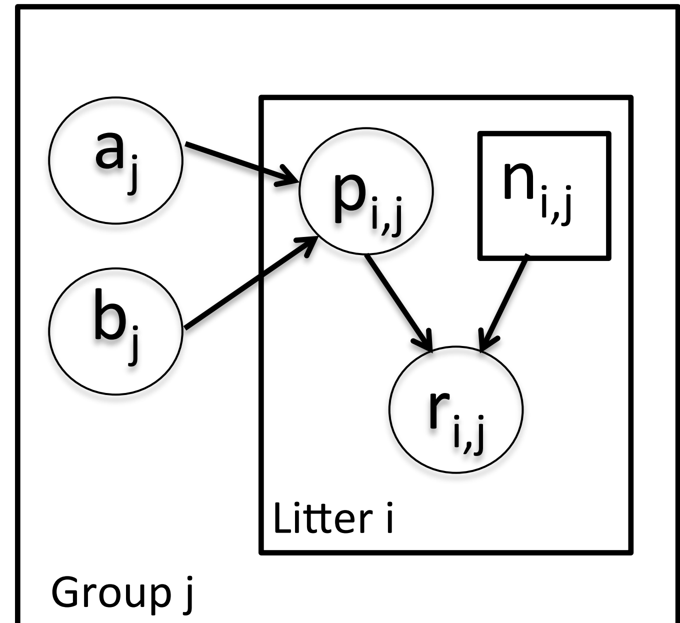
3. Algorithm library: MCMC, Particle Filter, etc.

# NIMBLE in Action: the Litters Example

Beta-binomial GLMM for clustered binary response data

Survival in two sets of 16 litters of pigs

```
littersModelCode <- quote({
  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);
 })
```

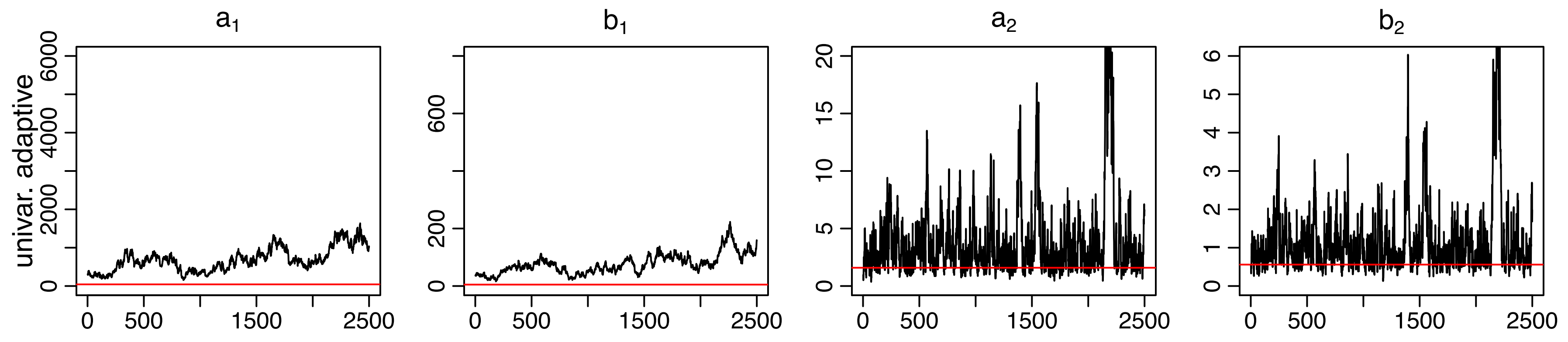



Challenges of the toy example:

- 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

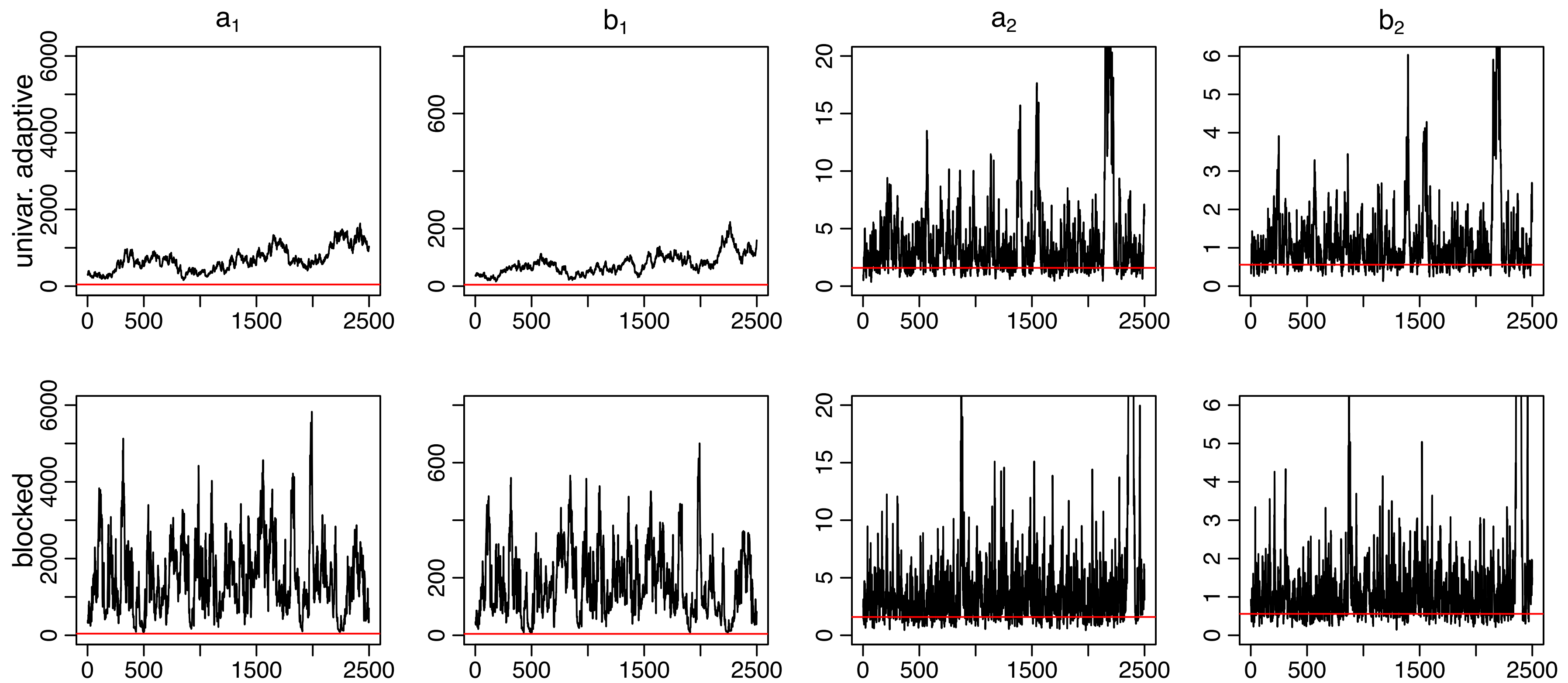
```
> littersMCMCspec <- configureMCMC(littersModel, list(adaptInterval = 100))
> littersMCMC <- buildMCMC(littersMCMCspec)
> littersMCMC_cpp <- compileNIMBLE(littersModel, project = littersModel)
> littersMCMC_cpp(10000)
```




Red line is MLE

# Blocked MCMC: Gibbs + Blocked Metropolis

```
> littersMCMCspec2 <- configureMCMC(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)
```

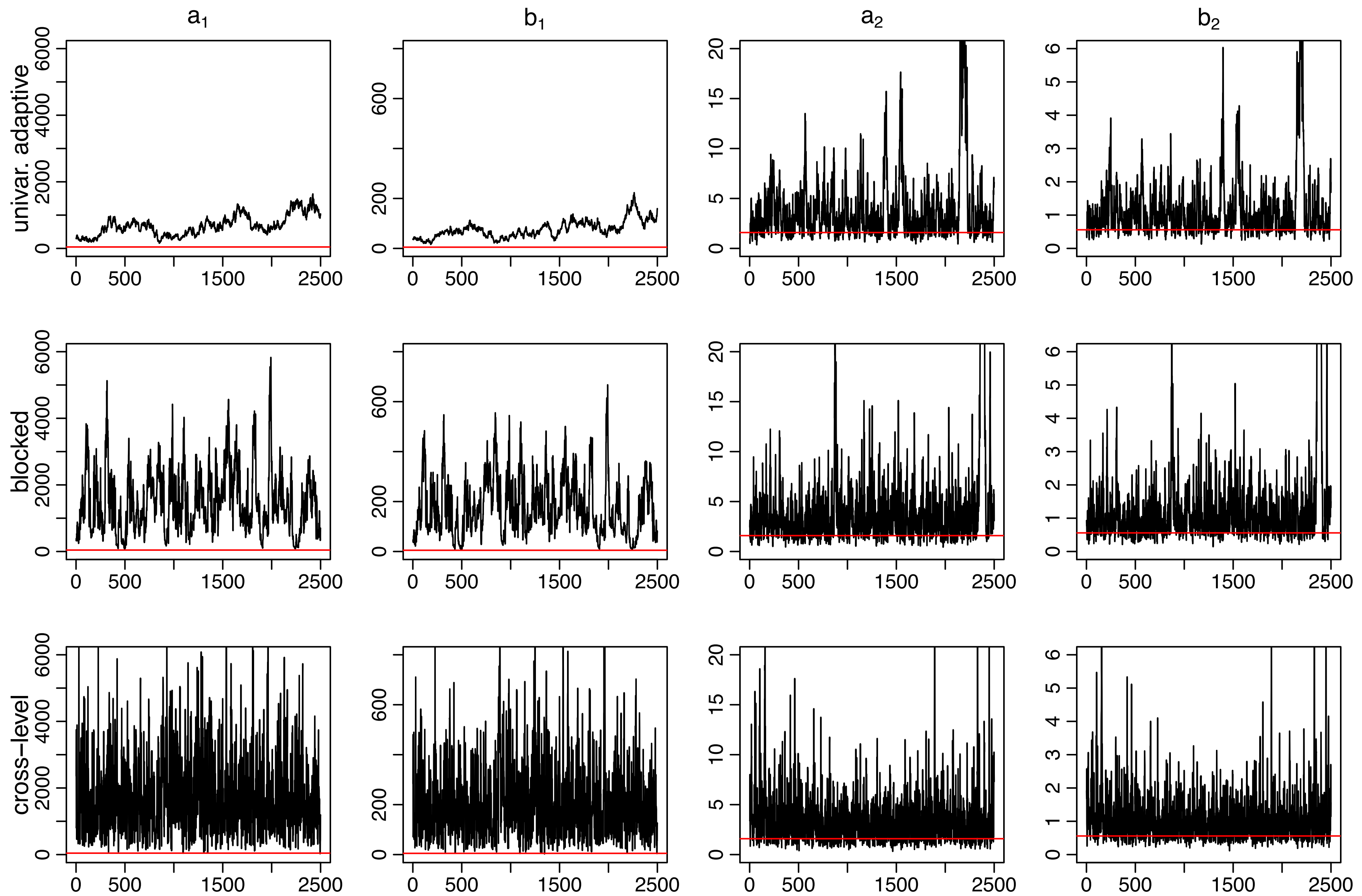


a1
b1
a2
b2
univar. adaptive
blocked
0
2000
4000
6000
200
600
5
10
15
20
1
2
3
4
5
6
500
1500
2500

# 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.
  - 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.

```
> littersMCMCspec3 <- configureMCMC(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)
```


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

# Litters MCMC: BUGS and JAGS

- BUGS gives similar performance to the default NIMBLE MCMC
  - Be careful – values of $sim.list and $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 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.

# 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
- Current implementation is basic; more sophisticated treatments should help

---

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 June 2014 with regular releases since. Lots to do:
  - Improve the user interface and speed up compilation
  - Refinement/extension of the DSL for algorithms
  - Enhance current algorithms provided (e.g., extend basic MCEM and particle filter)
  - 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

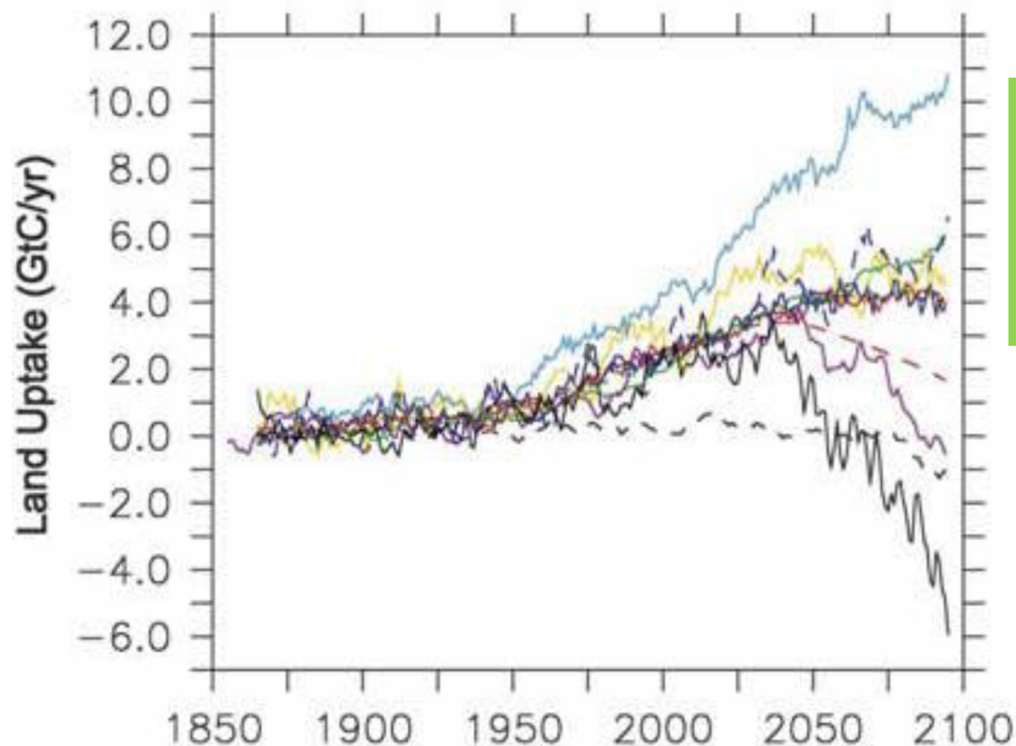
# Using NIMBLE in Applications

- PalEON Project (www3.nd.edu/~paleolab/paleonproject)
- How can we assess and improve deterministic ecological models that forecast changes in ecosystems?
  - The models are parameterized based on short-term forest growth/respiration data
  - But we want to make predictions over decades
  - Our approach: use historical and fossil data to estimate past vegetation and climate and use this information for model initialization, assessment, and improvement

# PalEON 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*

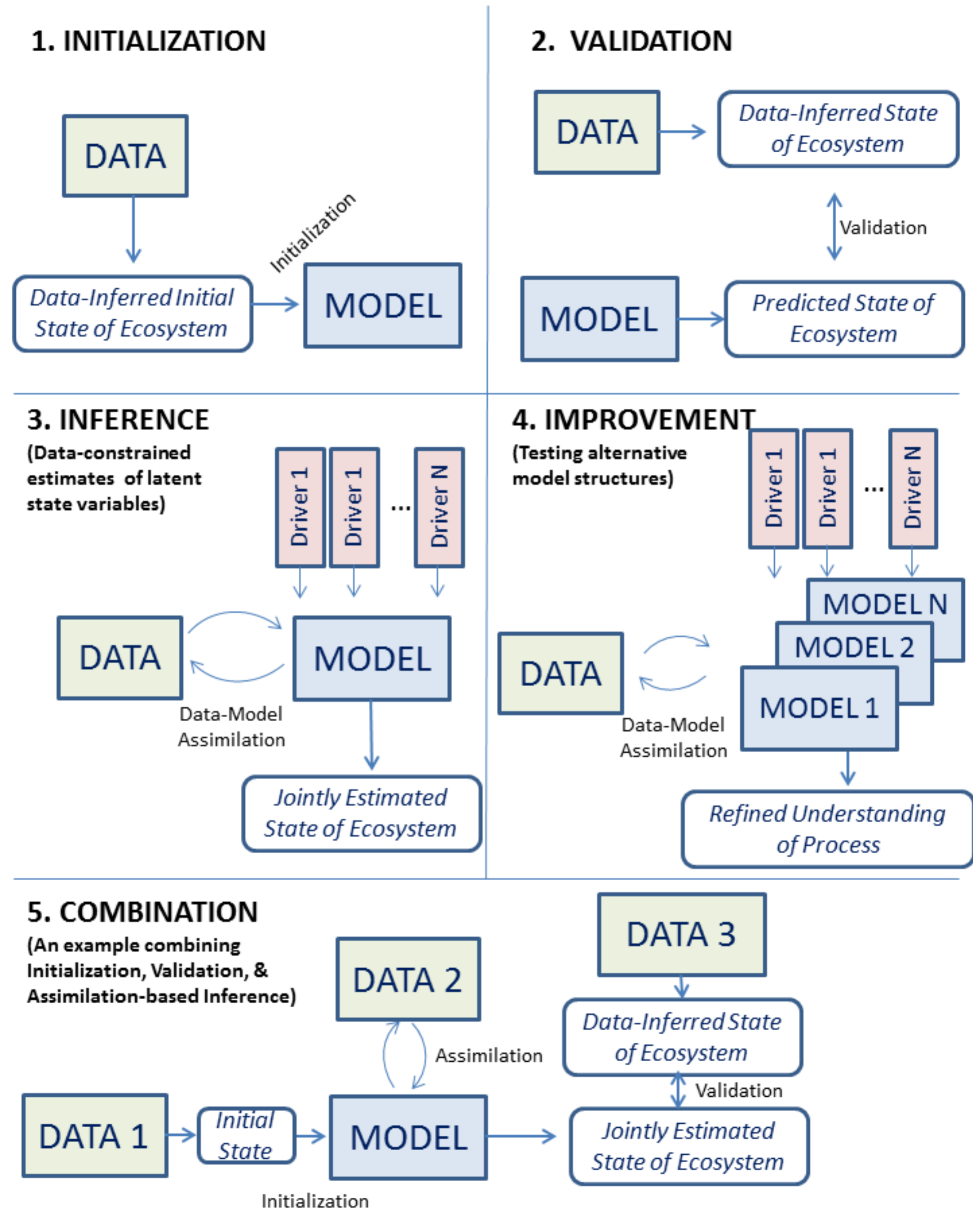



Critical issue: model parameterization and representation of decadal- to centennial-scale processes are poorly constrained by data

Friedlingstein et al. 2006 J. Climate

# Using statistical model output with deterministic ecoysystem models

1. Initialization
2. Validation
3. Inference
4. Improvement
5. (all of the above)

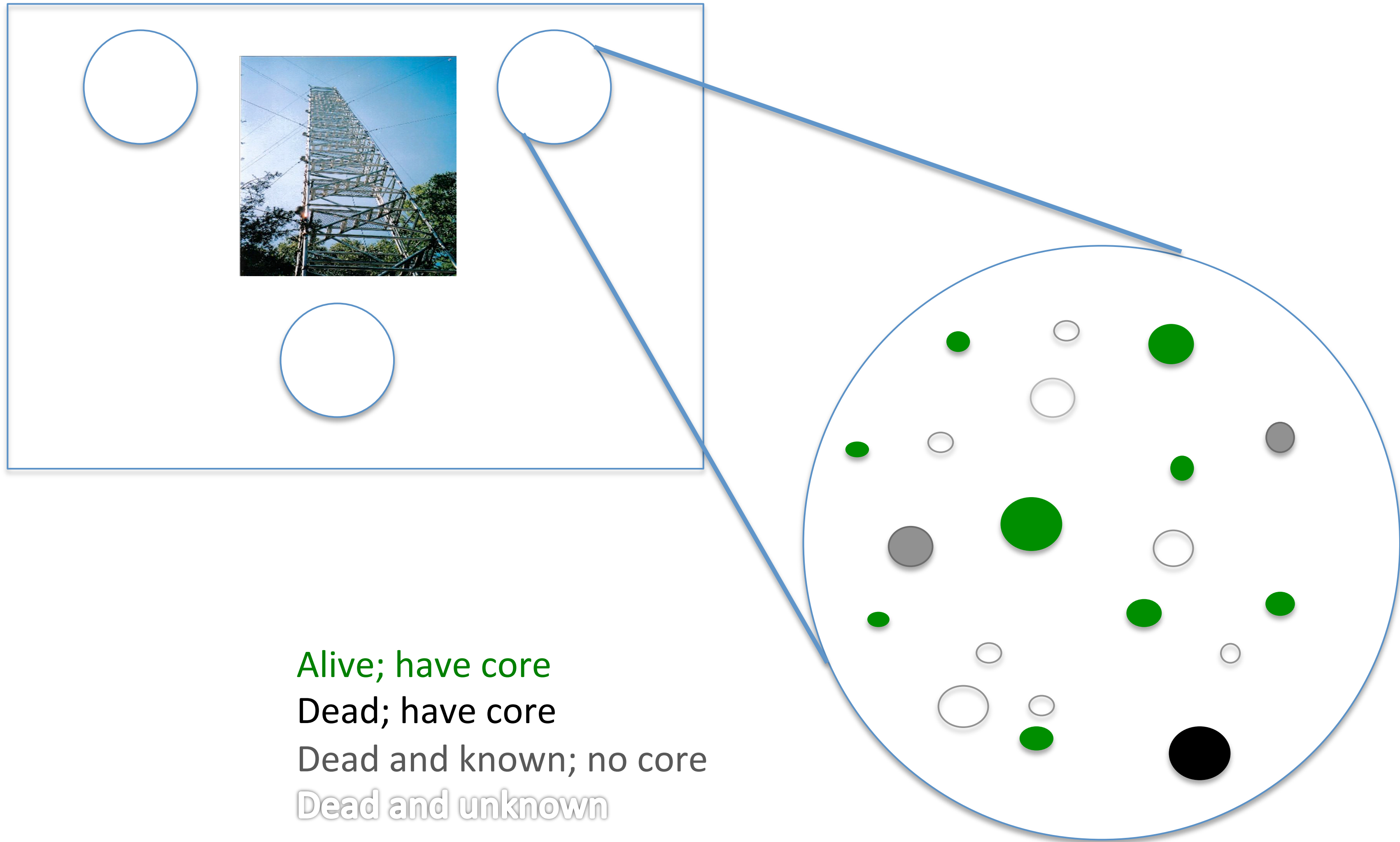


# PalEON Statistical Applications

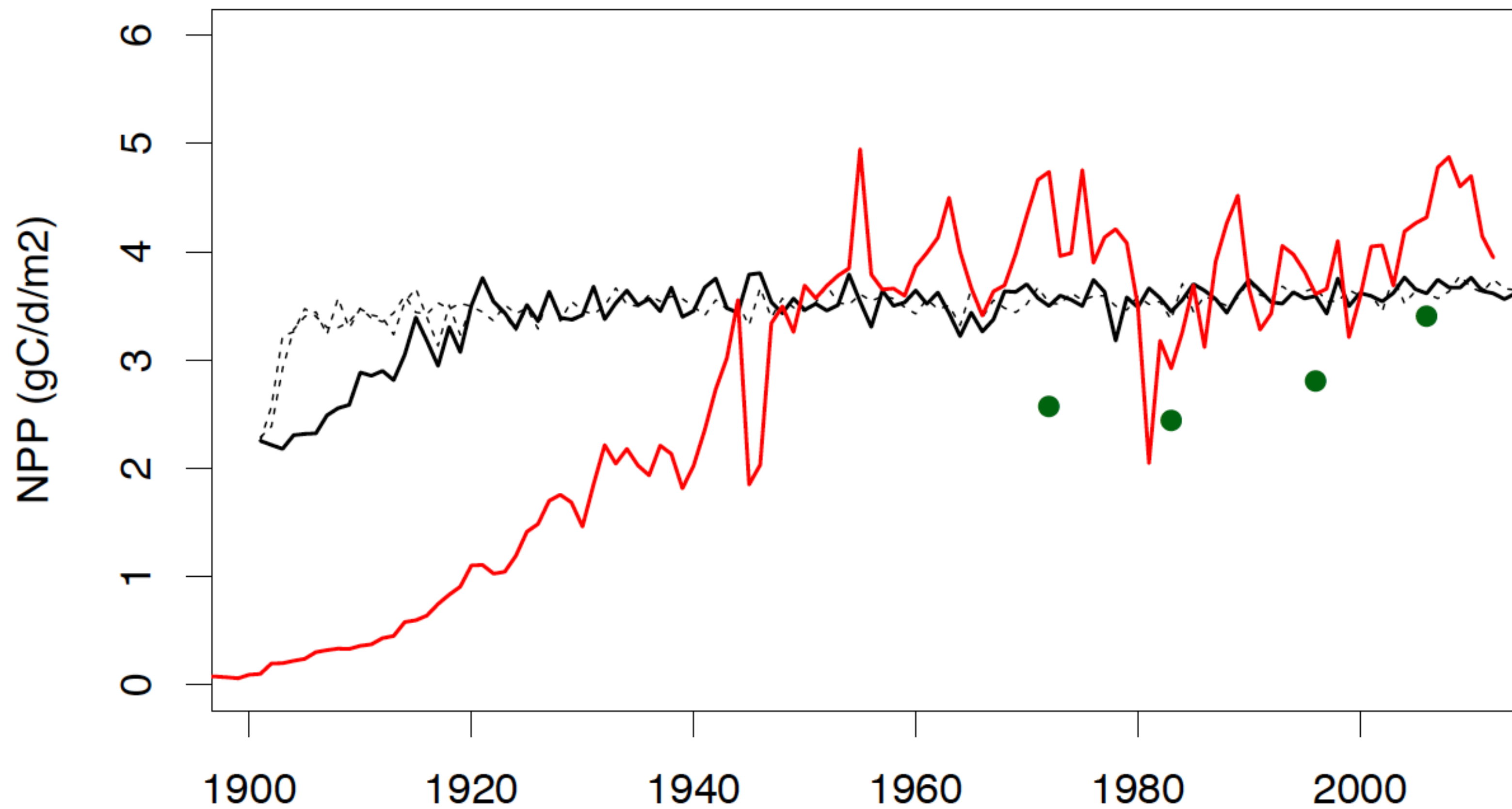
- Estimate spatially-varying composition and biomass of tree species from settlement era count and size data
- Estimate temporal variations in temperature and precipitation over 2000 years from tree rings and lake/bog records
- Estimate tree composition and biomass spatially over 2000 years from fossil pollen in lake sediment cores
- Estimate forest growth from tree rings and forest census data over 100 years

# Tree Ring Data for Forest Growth Inference

# Estimating forest growth



# Missing data problem




Using tree rings from current trees produces increasing bias the further back in time you go.

Solutions:

- Record dead trees; core if possible
- Use old census data if available
- Infer missing trees based on understanding of forest development

# Statistical Model

```
for(j in 1:nDBH) {
   logDobs[j] ~ dt(log(D[dbh_tree_id[j], dbh_time_id[j]]), sd = sig_d_obs, 3)
}
for(i in 1:nWidths) {
   logXobs[i] ~ dnorm(log(X[incr_tree_id[i], incr_time_id[i]]), sd = sig_x_obs)
}
for(i in 1:n) {
   for(j in 1:last_time[i]) {
      X[i, j] ~ dlnorm(beta[i] + beta_t[j], sd = sig_x)
   }
   for(j in (last_time[i]+1):nT) {
        X[i, j] <- 0
   }
   D0[i] ~ dunif(0, 300)
   D[i, 1] <- D0[i] + X[i, 1]/10
   for(j in 2:nT) {
      D[i, j] <- D[i, j-1] + X[i, j]/10
   }
   beta[i] ~ dnorm(beta0, sd = beta_sd)
}
for(j in 1:nT) {
   beta_t[j] ~ dnorm(0, sd = beta_t_sd)
}
beta0 ~ dnorm(0, .00001)
sig_d_obs ~ dunif(0, 1000)
sig_x_obs ~ dunif(0, 1000)
....
```

Tree census lik

Tree ring lik

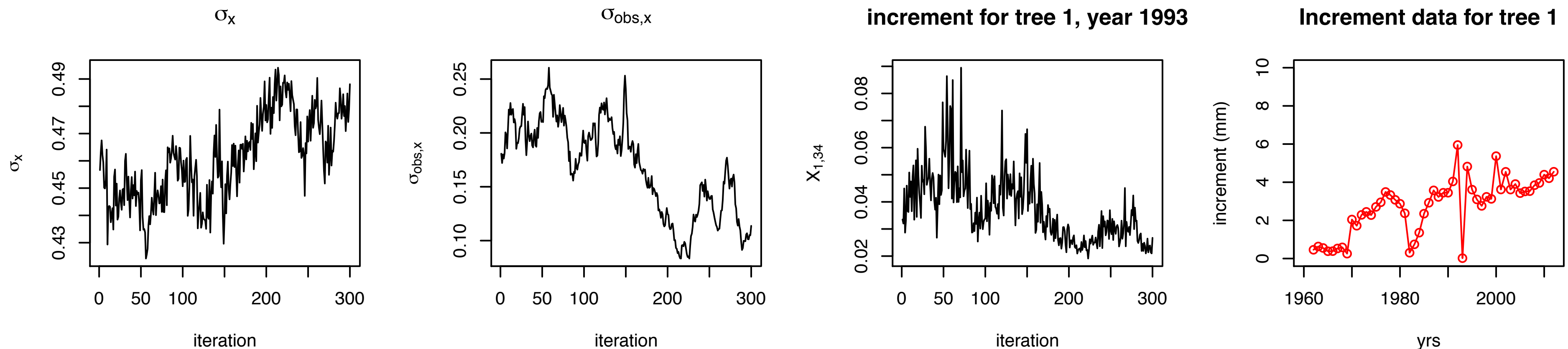
Increment process model

Tree growth

Hyperpriors

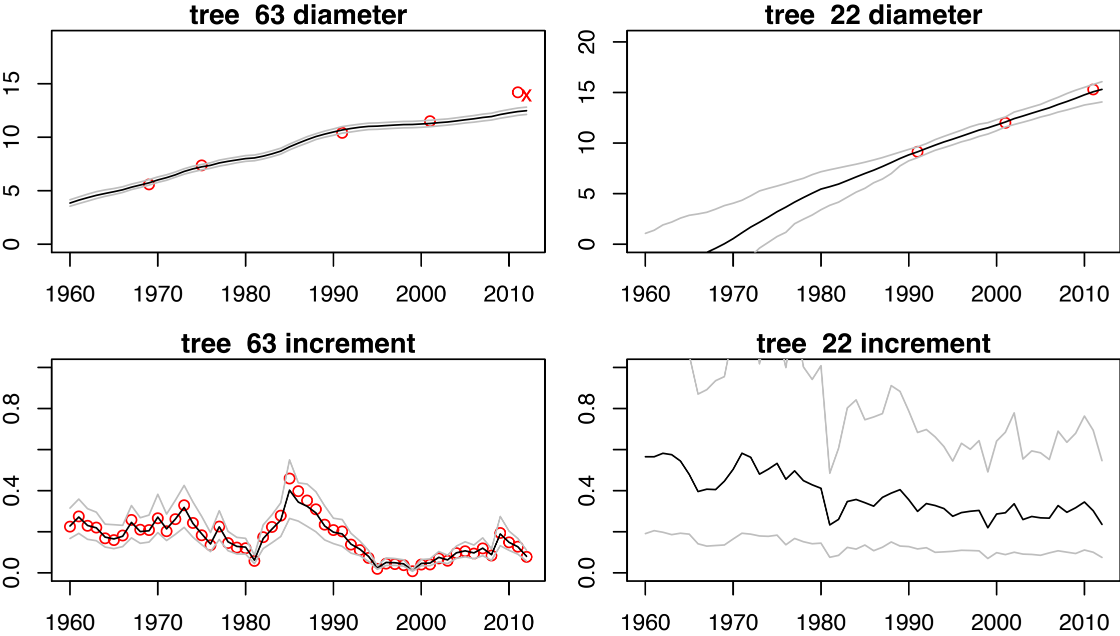
# MCMC: Timing and inference

- Creating of model and algorithm takes 25-30 minutes in both NIMBLE and JAGS (~10000 parameters)
  - For certain large models, NIMBLE is slower than JAGS to create model/algorithm; we're working on speeding up the processing
- 50000 MCMC iterations take 30-40 minutes in both NIMBLE and JAGS
- Difficult to identify increment measurement error from increment process variation:
  - Block sampling of the two parameters has limited impact; slice sampling does not help
  - Also constrained by certain individual increment values

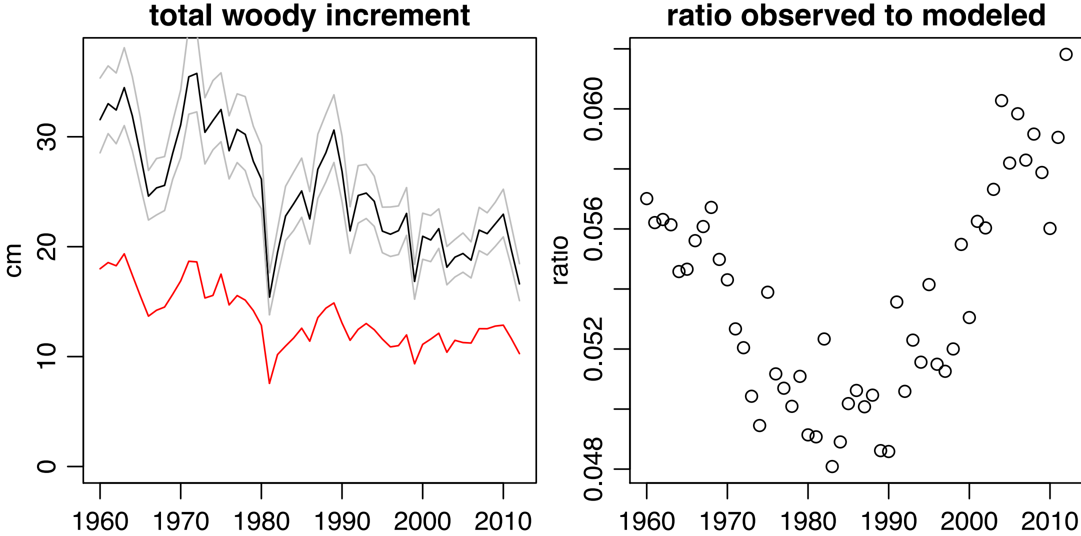


# Initial Results

Imputation




Woody increment per year



# How Can NIMBLE Help?

- ✓ User control over MCMC specification
- ✓ Transparency when an algorithm fails
- Provide non-MCMC-based estimation for: better inference, MCMC starting values, MCMC validation
- Provide algorithms for model comparison and model criticism
- Provide a broader range of possible model structures and/or improved computation for those models (e.g., Dirichlet-multinomial distribution, CAR/MRF models)

# PalEON Acknowledgements


- 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
- Neil Pederson, Jack Williams for slides