BUGS and R with NIMBLE

(and automatic C++ compilation)

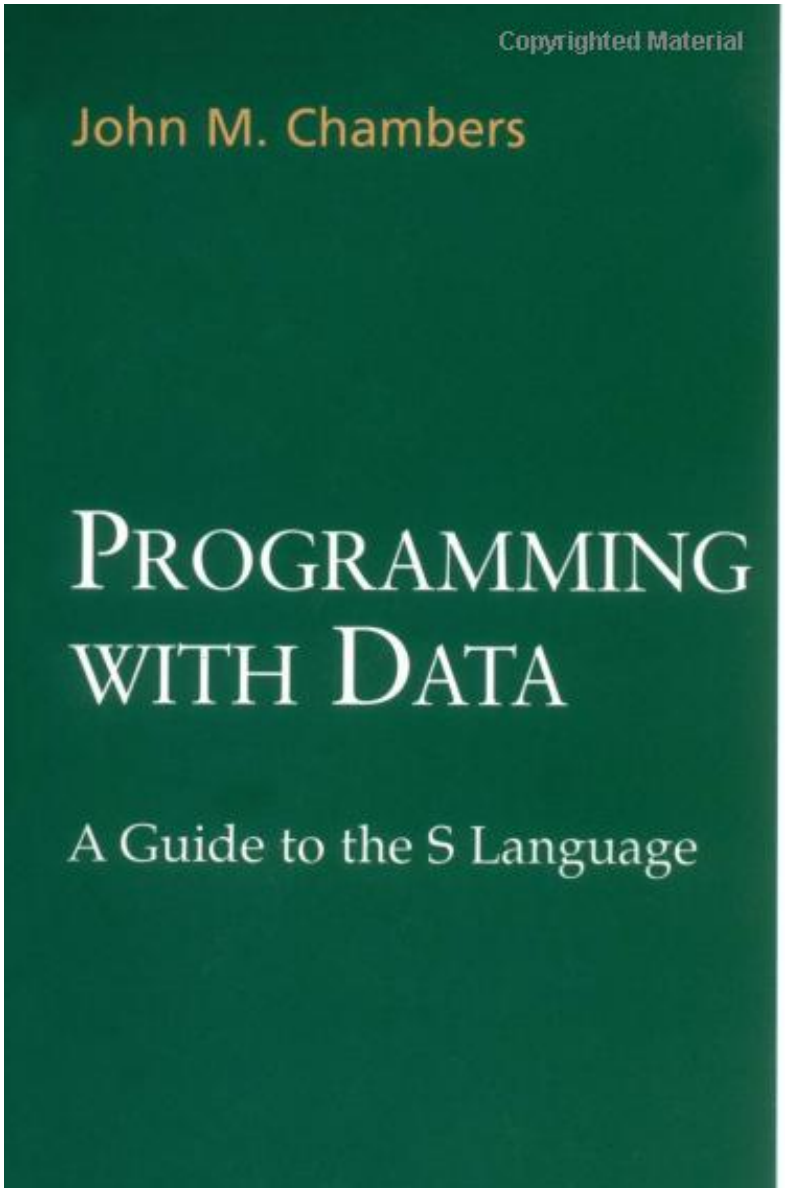
Perry de Valpine*
Daniel Turek
Chris Paciorek
Cliff Anderson-Bergman
Ras Bodik
Duncan Temple Lang

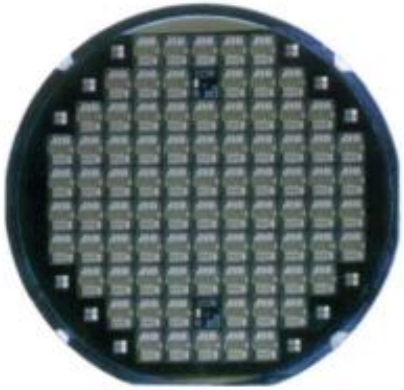
University of California
Berkeley
Davis

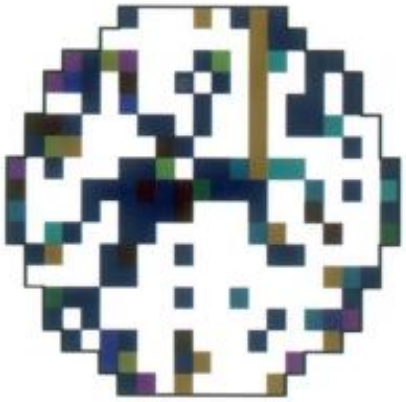
Funded by National Science Foundation
(Advances in Biological Informatics)

*Sabbatical host: Center for Math Research (CIMAT), Guanajuato, México

The core concept of R







(A lot of) Statistics = Models + Data

Packages reinvent the model part:

- Linear models: `lm(Y ~ A + B*X, …)`
- Generalized linear mixed models: `glmer(Y~A+(1+X|B), …)`
- Generalized additive models: `gam(Y ~ A + s(X), …)`
- Time-series models
- Spatial models
- Survival models
- On and on…
- …

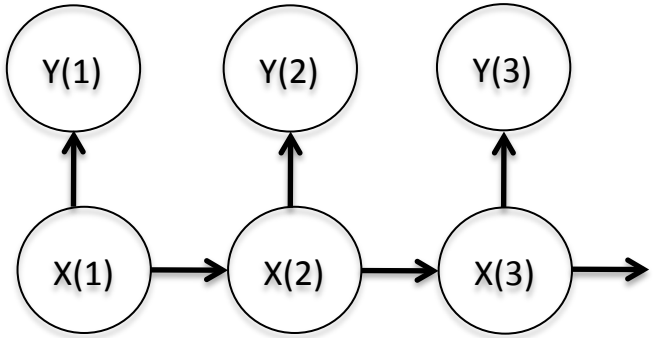
How about a language for models?

Languages for models

- BUGS
  - Bayesian inference Using Gibbs Sampling
  - WinBUGS, OpenBUGS, JAGS
  - motivated by MCMC
  - Most widely adopted
- Others: ADMB, PyMC, Stan, BLOG, Church
- Generally motivated by hierarchical models
  - Random effects and/or Bayesian analysis

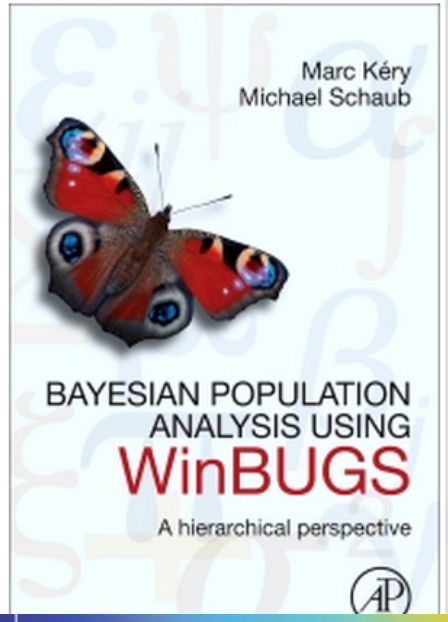
Model languages have been tied to algorithms:

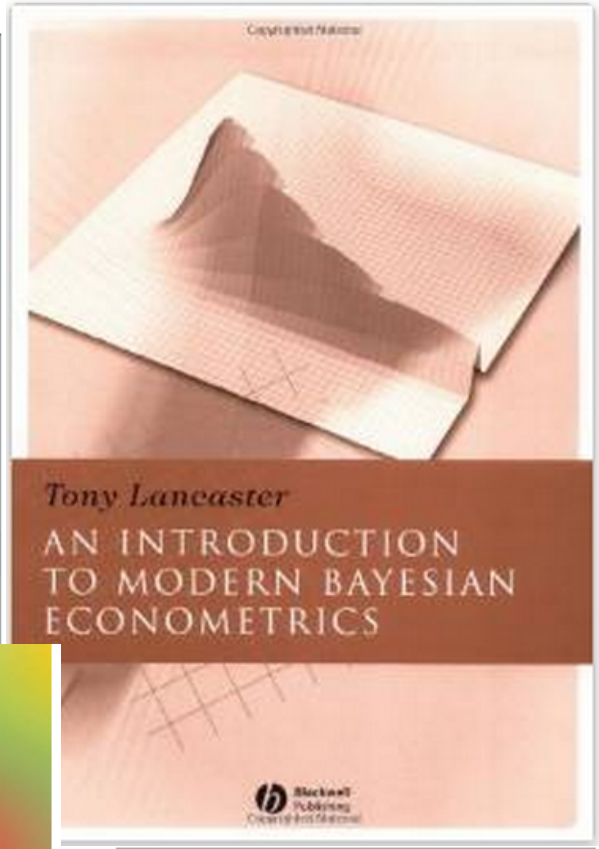
Model

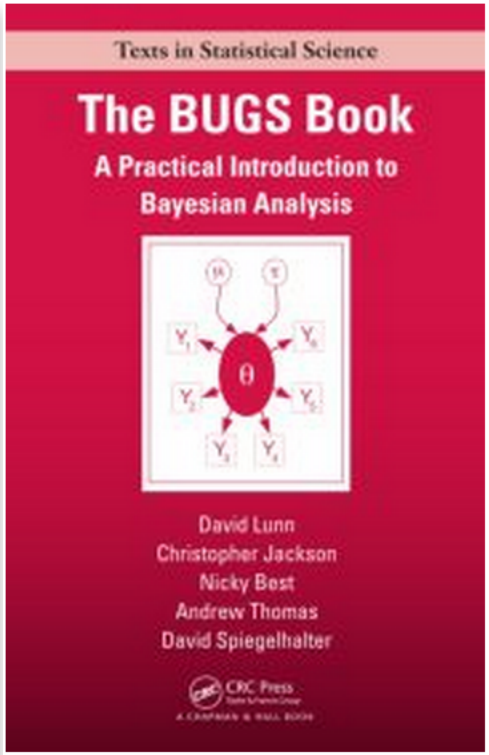


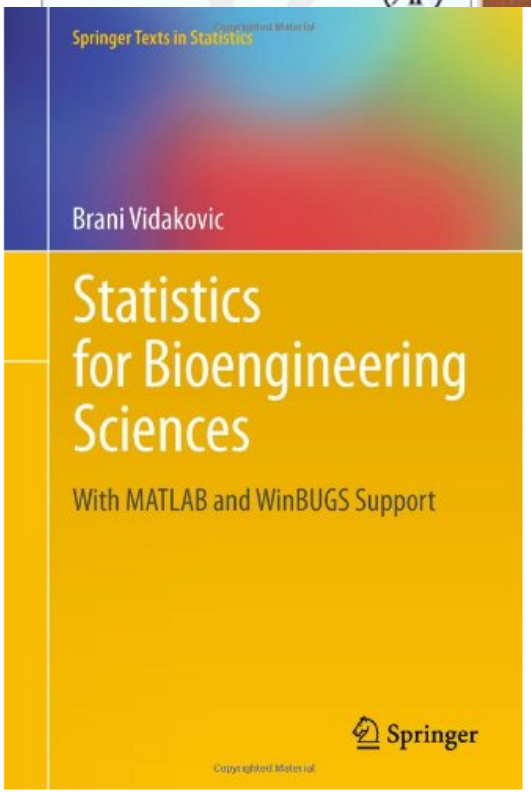
Algorithm

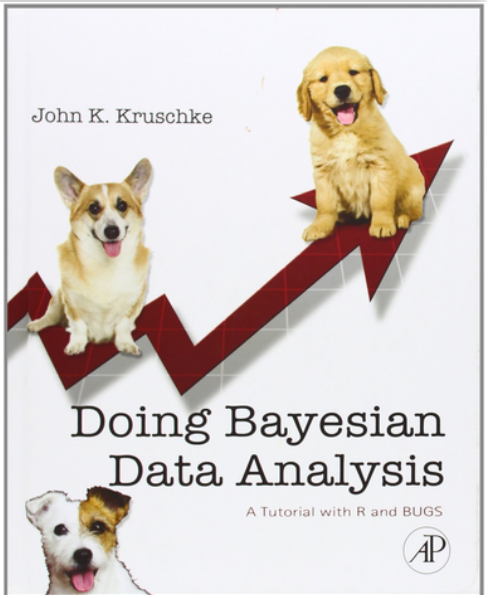
e.g. BUGS (WinBUGS, OpenBUGS, JAGS), ADMB, TMB, Stan

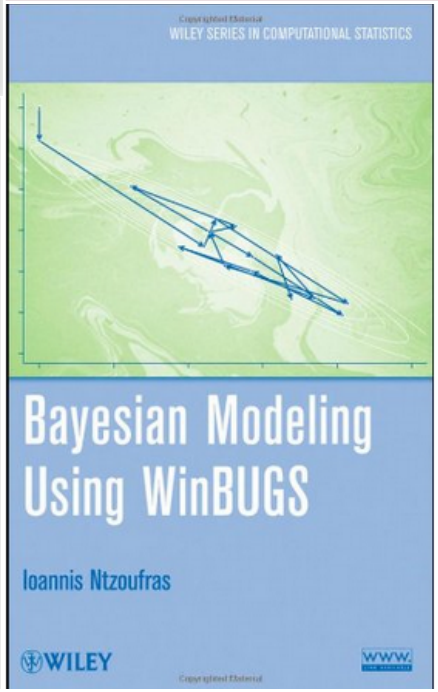

Marc Kéry
Michael Schaub
BAYESIAN POPULATION ANALYSIS USING
WinBUGS
A hierarchical perspective


Tony Lancaster
AN INTRODUCTION TO MODERN BAYESIAN ECONOMETRICS

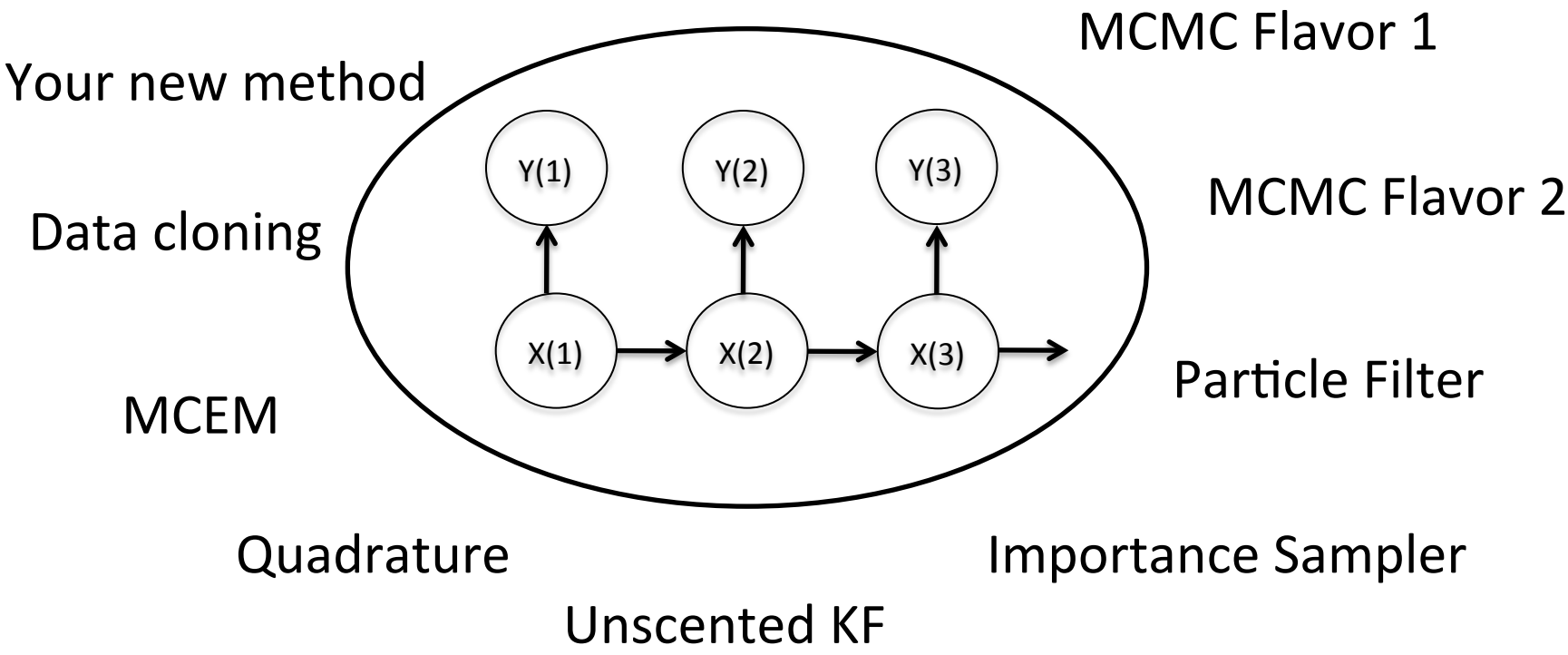

Texts in Statistical Science
The BUGS Book
A Practical Introduction to Bayesian Analysis
David Lunn
Christopher Jackson
Nicky Best
Andrew Thomas
David Spiegelhalter
CRC Press


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

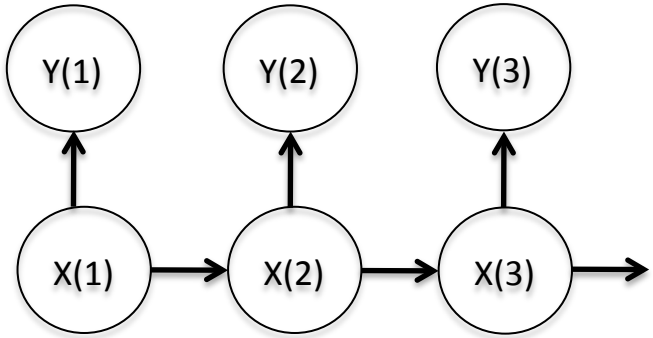

WILEY SERIES IN COMPUTATIONAL STATISTICS
Bayesian Modeling Using WinBUGS
Ioannis Ntzoufras
WILEY

What do we want to do with a hierarchical model?



Model languages have been tied to algorithms:

Model

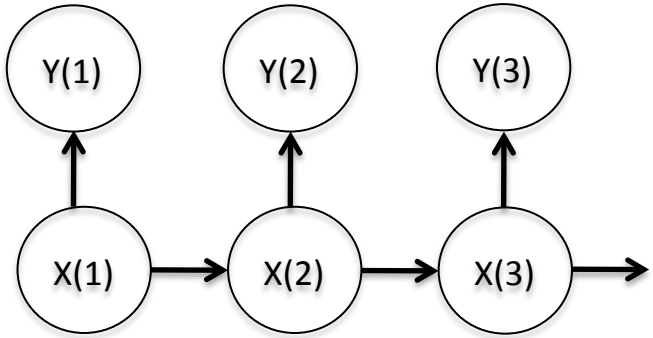


Algorithm

e.g. BUGS (WinBUGS, OpenBUGS, JAGS), ADMB, TMB, Stan

Goal of NIMBLE

Model (BUGS language)

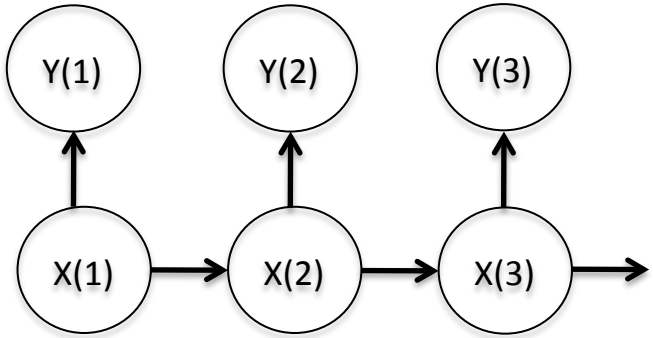


+

Algorithm Language

Goal of NIMBLE

Model (BUGS language)



+

Algorithm Language

MUST BE FAST = COMPILABLE

Numerical Inference for statistical Models using Bayesian and Likelihood Estimation

VERSION 0.3-1

NIMBLE

1. BUGS language → R model object

2. NIMBLE programming language
   - ✧ Domain-specific language (DSL) embedded in R
   - ✧ Compilation via C++
3. Algorithm library: MLE, MCMC etc.

NIMBLE

1. BUGS language → R model object

2. NIMBLE programming language
   - ✧ Domain-specific language (DSL) embedded in R
   - ✧ Compilation via C++
3. Algorithm library: MLE, MCMC etc.

What NIMBLE does with BUGS models?

```
model {
  for(i in 1:N) {
    theta[i] ~ dgamma(alpha, beta)
    lambda[i] <- theta[i] * tt[i]
    x[i] ~ dpois(lambda[i])
  }
  alpha ~ dexp(1.0)
  beta ~ dgamma(2, 2)
}
```

(pump model example from BUGS)

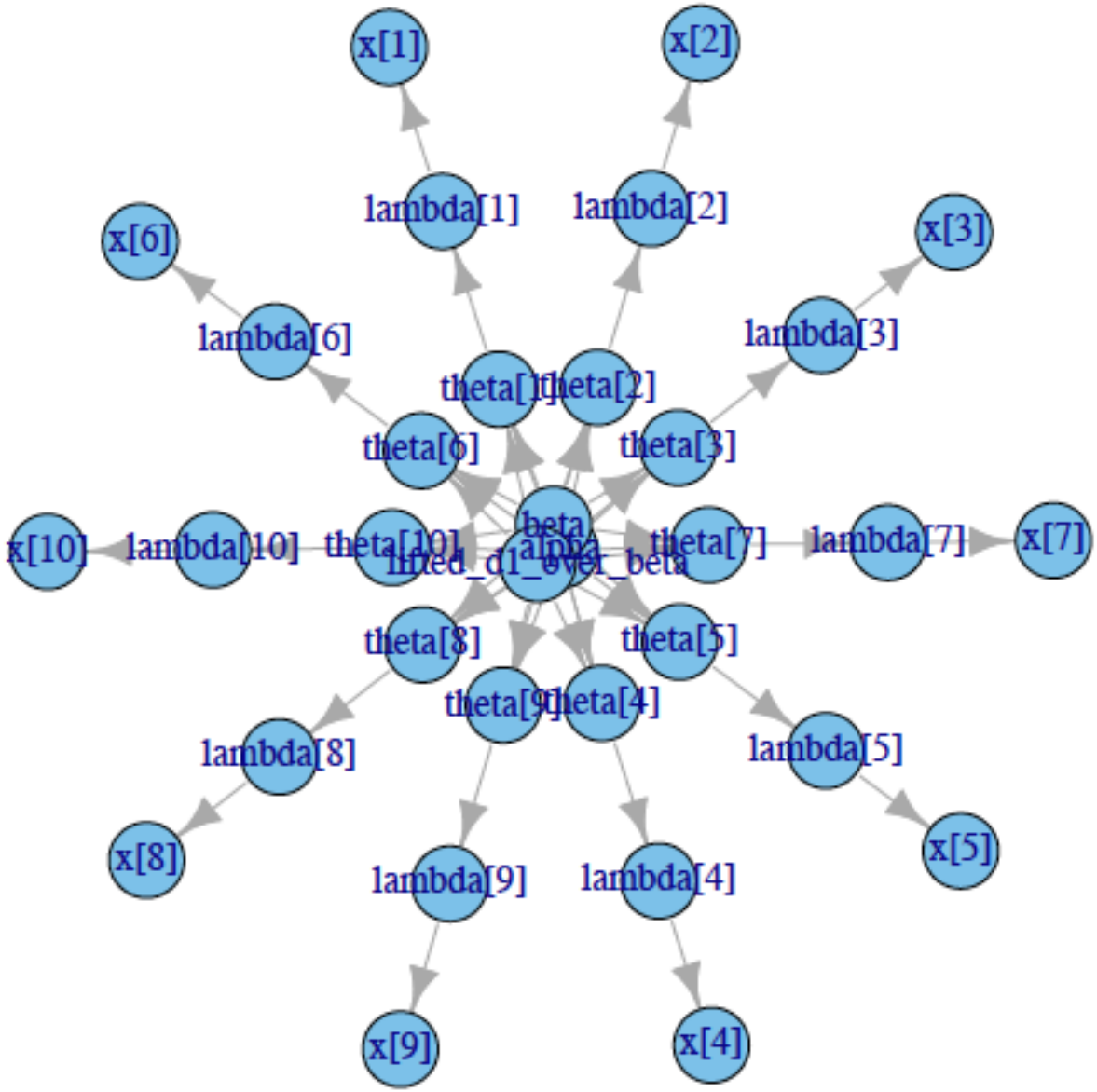


Figure due to igraph plot function

What NIMBLE does with BUGS models

We turn:

```
model {
  for(i in 1:N) {
    theta[i] ~ dgamma(alpha, beta)
    lambda[i] <- theta[i] * tt[i]
    x[i] ~ dpois(lambda[i])
  }
  alpha ~ dexp(1.0)
  beta ~ dgamma(2, 2)
}
```

into

```
> pumpModel$alpha <- 5

> simulate(pumpModel, 'beta')

> calculate(pumpModel,
      c('lambda[1:5]', 'x[1:5]'))

> calculate(pumpModel,
    pumpModel$getDependencies( 'beta'))

> getLogProb(pumpModel, 'x')
```

What NIMBLE does with BUGS models

We turn:

```
model {
  for(i in 1:N) {
    theta[i] ~ dgamma(alpha, beta)
    lambda[i] <- theta[i] * tt[i]
    x[i] ~ dpois(lambda[i])
  }
  alpha ~ dexp(1.0)
  beta ~ dgamma(2, 2)
}
```

into

```
> pumpModel$alpha <- 5

> simulate(pumpModel, 'beta')

> calculate(pumpModel,
      c('lambda[1:5]', 'x[1:5]'))

> calculate(pumpModel,
    pumpModel$getDependencies( 'beta'))

> getLogProb(pumpModel, 'x')
```

Use model variables

What NIMBLE does with BUGS models

We turn:

```
model {
  for(i in 1:N) {
    theta[i] ~ dgamma(alpha, beta)
    lambda[i] <- theta[i] * tt[i]
    x[i] ~ dpois(lambda[i])
  }
  alpha ~ dexp(1.0)
  beta ~ dgamma(2, 2)
}
```

into

```
> pumpModel$alpha <- 5

> simulate(pumpModel, 'beta')

> calculate(pumpModel,
      c('lambda[1:5]', 'x[1:5]'))

> calculate(pumpModel,
    pumpModel$getDependencies( 'beta'))

> getLogProb(pumpModel, 'x')
```

Simulate part of the model

What NIMBLE does with BUGS models

We turn:

```
model {
  for(i in 1:N) {
    theta[i] ~ dgamma(alpha, beta)
    lambda[i] <- theta[i] * tt[i]
    x[i] ~ dpois(lambda[i])
  }
  alpha ~ dexp(1.0)
  beta ~ dgamma(2, 2)
}
```

into

```
> pumpModel$alpha <- 5

> simulate(pumpModel, 'beta')

> calculate(pumpModel,
      c('lambda[1:5]', 'x[1:5]'))

> calculate(pumpModel,
    pumpModel$getDependencies( 'beta'))

> getLogProb(pumpModel, 'x')
```

Calculate log probabilities from part of the model

What NIMBLE does with BUGS models

We turn:

```
model {
  for(i in 1:N) {
    theta[i] ~ dgamma(alpha, beta)
    lambda[i] <- theta[i] * tt[i]
    x[i] ~ dpois(lambda[i])
  }
  alpha ~ dexp(1.0)
  beta ~ dgamma(2, 2)
}
```

into

```
> pumpModel$alpha <- 5

> simulate(pumpModel, 'beta')

> calculate(pumpModel,
      c('lambda[1:5]', 'x[1:5]'))

> calculate(pumpModel,
    pumpModel$getDependencies( 'beta'))

> getLogProb(pumpModel, 'x')
```

Query the model's structure:
How are variables connected?

What NIMBLE does with BUGS models

We turn:

```
model {
  for(i in 1:N) {
    theta[i] ~ dgamma(alpha, beta)
    lambda[i] <- theta[i] * tt[i]
    x[i] ~ dpois(lambda[i])
  }
  alpha ~ dexp(1.0)
  beta ~ dgamma(2, 2)
}
```

into

```
> pumpModel$alpha <- 5

> simulate(pumpModel, 'beta')

> calculate(pumpModel,
      c('lambda[1:5]', 'x[1:5]'))

> calculate(pumpModel,
    pumpModel$getDependencies( 'beta'))

> getLogProb(pumpModel, 'x')
```

Query previously calculated log probabilities

NIMBLE

1. BUGS language → R model object

2. NIMBLE programming language
   - ✧ Domain-specific language (DSL) embedded in R
   - ✧ Compilation via C++
3. Algorithm library: MLE, MCMC etc.

NIMBLE: Programming With Models

We want:

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

NIMBLE: Programming With Models

- Say we want an objective function
  - Input: Values for (alpha, beta)
  - Output: Log probability density of part of the model that depends on (alpha, beta)
    - The "Markov blanket"
  - The we can optimize for:
    - Maximum likelihood estimation
    - 1st step of Laplace approximation / Quadrature
    - MCMC proposal density

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())
  })
```

2 kinds of functions

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())
  })
```

pumpModel, c('alpha', 'beta')

2 kinds of functions

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 actual algorithm

Using a NIMBLE function

```
> objFun1 <- objectiveFunction(pumpModel,
                               c('alpha', 'beta'))
> objFun1$run(c(.9, 1.2))
-16.1

> CobjFun1 <- compileNimble(objFun1,
                            project = pumpModel)
> CobjFun1$run(c(0.9, 1.2))
-16.1
```

Using a NIMBLE function

```
> objFun1 <- objectiveFunction(pumpModel,
                               c('alpha', 'beta'))
> objFun1$run(c(.9, 1.2))
-16.1

> CobjFun1 <- compileNimble(objFun1,
                            project = pumpModel)
> CobjFun1$run(c(0.9, 1.2))
-16.1
```

Using a NIMBLE function

```
> objFun1 <- objectiveFunction(pumpModel,
                               c('alpha', 'beta'))
> objFun1$run(c(.9, 1.2))
-16.1

> CobjFun1 <- compileNimble(objFun1,
                            project = pumpModel)
> CobjFun1$run(c(0.9, 1.2))
-16.1
```

The NIMBLE compiler

One-line example:

nimbleFunction run code:

```
Y <- t(X) %*% X
```

The NIMBLE compiler

One-line example:

nimbleFunction run code:

```
Y <- t(X) %*% X
```

Compiler

The NIMBLE compiler

One-line example:

nimbleFunction run code:

```
Y <- t(X) %*% X
```

Compiler

C++:

type declaration for `Y, EigY, EigX`

set size of `Y`

Point `EigY` at `Y` and `EigX` at `X`

```
EigY = EigX.transpose() * EigX;
```

Uses Eigen C++ library

The NIMBLE compiler

<u>One-line example:</u>

nimbleFunction run code:

```
Y <- t(X) %*% X
```

Compiler

C++:

```
type declaration for Y, EigY, EigX
set size of Y
Point EigY at Y and EigX at X
EigY = EigX.transpose() * EigX;
```

Uses Eigen C++ library

R: compile the C++

load the shared object

generate an R interface object (function or class)

NIMBLE compiler writes C++ classes

```
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())
  })
```

Member data

Member function(s)

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
- 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
- Please contribute!

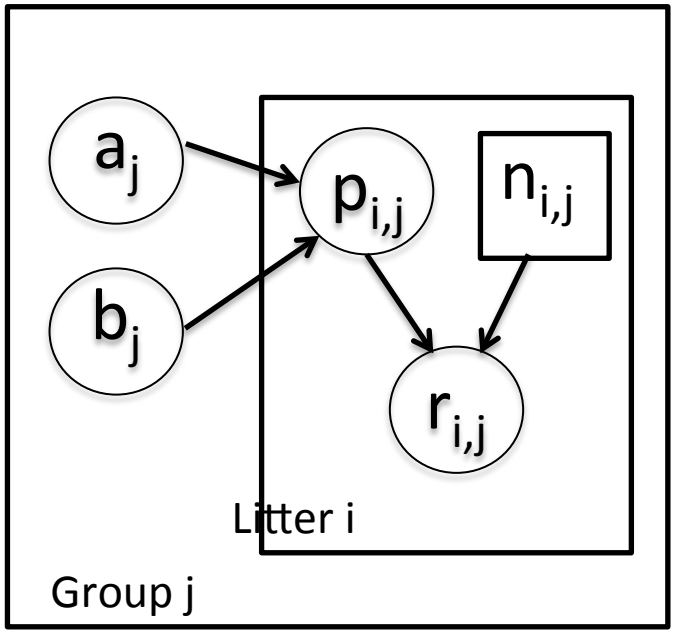
Keeping track of sets of model variables: modelValues

```
> mv <- modelValues(pumpModel, 1000)

> mv['alpha', 3] <- 5

> copy(model, mv, rowTo = 10)
```

NIMBLE MCMC: the Litters Example

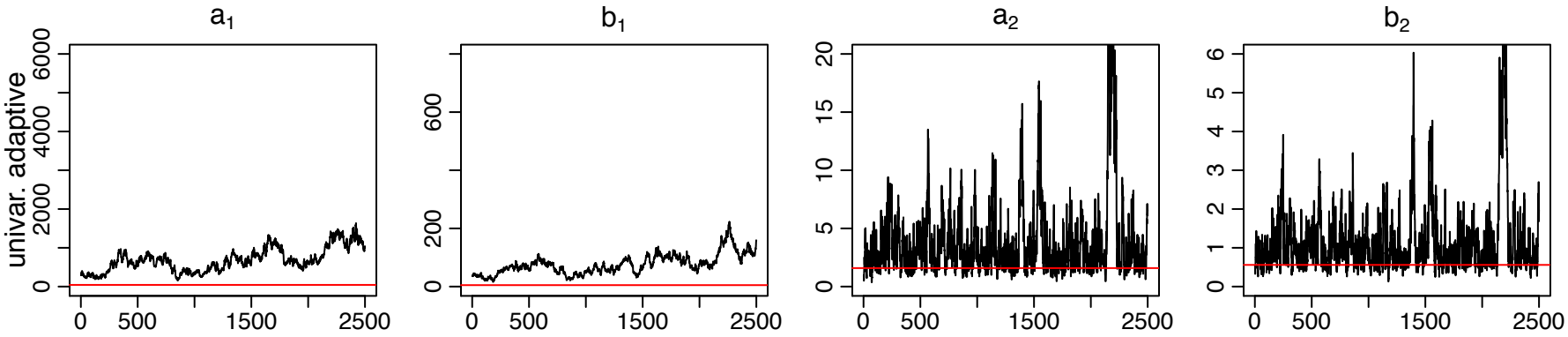
Beta-binomial for clustered binary response data

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



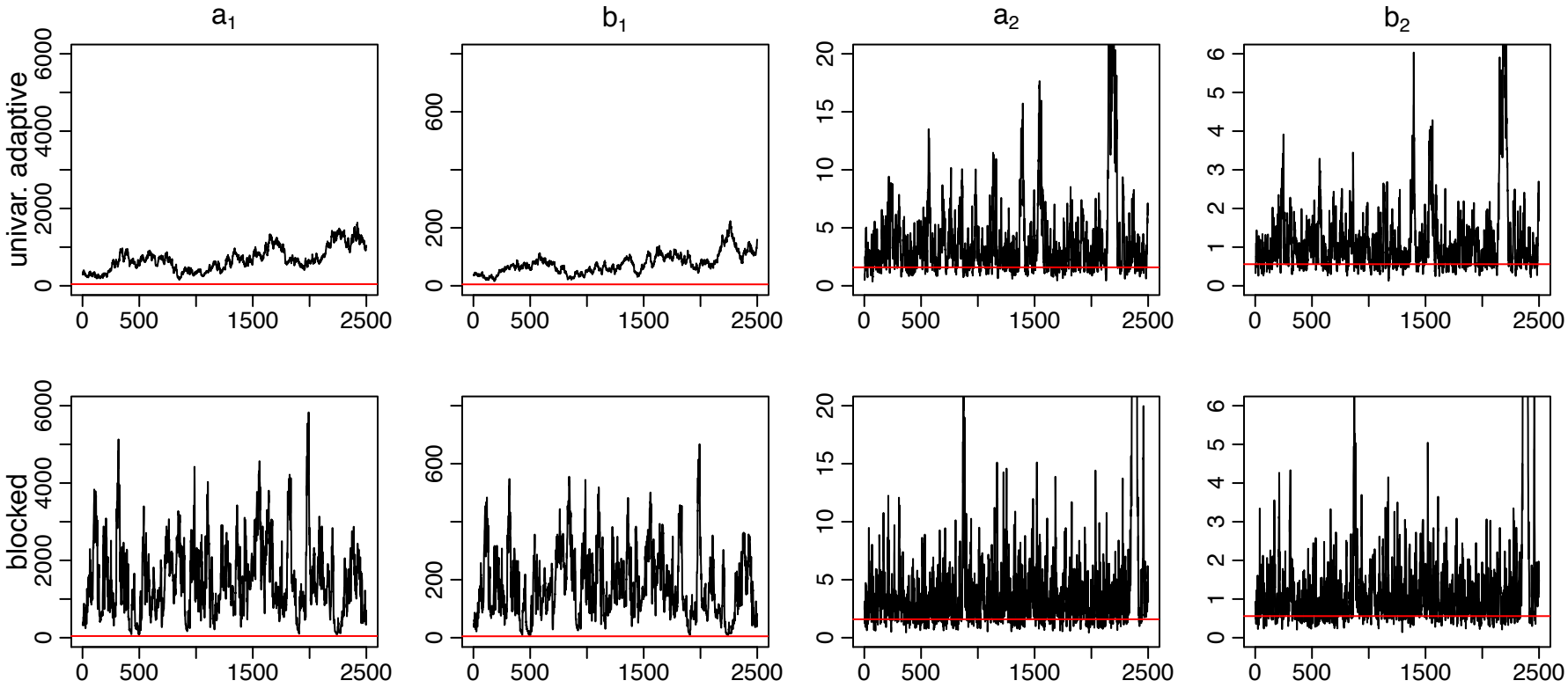
Default MCMC: Gibbs + Metropolis

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

> littersMCMC <- buildMCMC(littersMCMCspec)

> littersMCMC_cpp <- compileNIMBLE(littersModel,
                                   project = littersModel)

> littersMCMC_cpp(10000)
```


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

Blocked MCMC: Gibbs + Blocked Metropolis

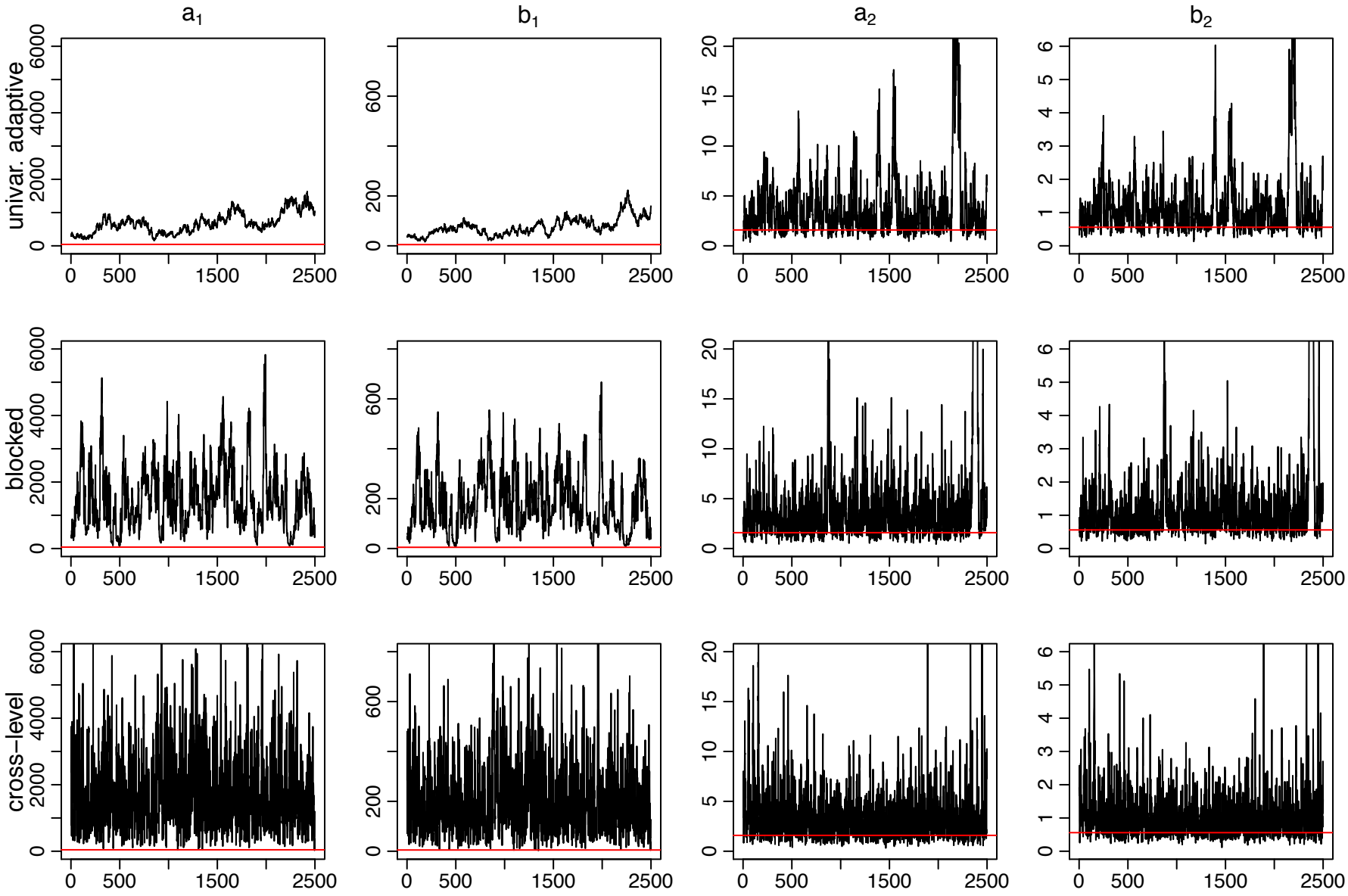
```
> littersMCMCspec2 <- MCMCspec(littersModel, list(adaptInterval = 100))

> 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)
```


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

Cross-level MCMC: Gibbs + Cross-level Updaters

- Problem: Correlation across levels of model
- New “cross-level” updater in NIMBLE language.
  - Random walk on one level
  - Joint conjugate proposal on dependent level

```
…
> littersMCMCspec3$addSampler(
                    ‘crossLevel’,
              list(topNodes = c('a[1]','b[1]'),
                   adaptInterval = 100) )
…
```


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

http://R-nimble.org