Model-Based PRA: MCMC via Nimble

Mark J Brewer

Mark.Brewer@bioss.ac.uk

@markjbrewer.bsky.social

Director, Biomathematics and Statistics Scotland
http://www.bioss.ac.uk

Lübeck, 24–26 September 2025

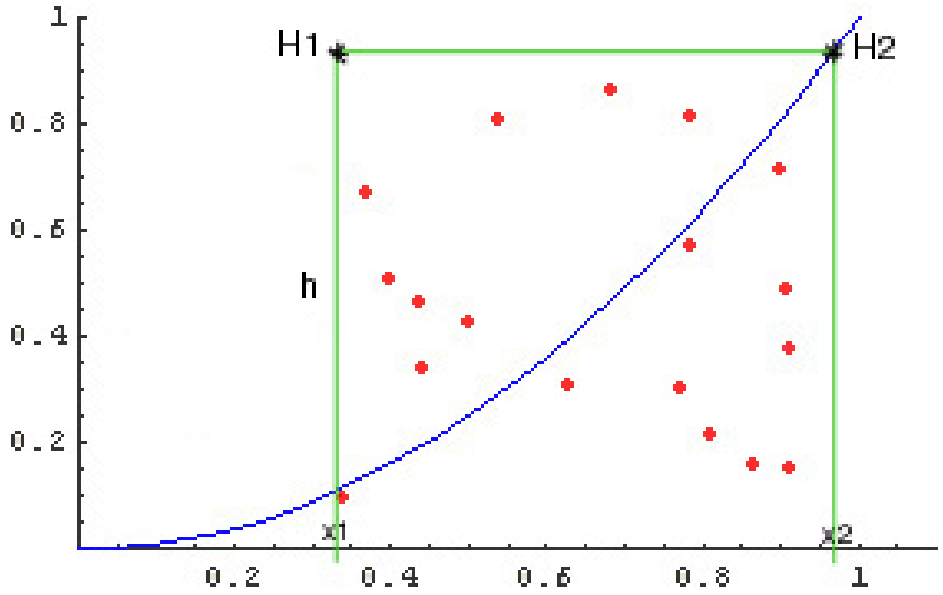
Model-Based PRA: MCMC via Nimble

- Preamble: MCMC, sampling-based inference
- Simple example: rainfall, tree rings
- Model-based PRA
- Brief introduction to Nimble
- Simple example: analysis

Sampling-Based Inference

- Bayesian computation requires integration — to obtain parameter estimates, posterior distributions etc
- Complex models — many parameters, non-conjugate priors — analytical integration not possible
- Instead, can integrate using simulation — so-called Monte Carlo methods

Monte Carlo Integration



Integral calculated by proportion of points beneath curve in box. *No use for complex posterior distributions. . .*

Markov Chain Monte Carlo

Monte Carlo integration not suitable for many dimensions, i.e. many parameters:

- as dimension increases, proportions of points “under the curve” tends to zero.

Markov Chain Monte Carlo

Monte Carlo integration not suitable for many dimensions, i.e. many parameters:

- as dimension increases, proportions of points “under the curve” tends to zero.

Trick: generate **correlated** points and update using simpler conditional distributions.

Markov Chain Monte Carlo

Monte Carlo integration not suitable for many dimensions, i.e. many parameters:

- as dimension increases, proportions of points “under the curve” tends to zero.

Trick: generate **correlated** points and update using simpler conditional distributions.

This is MCMC — generating a “chain” of points.

Markov Chain Monte Carlo

Can even update **one parameter at a time**.

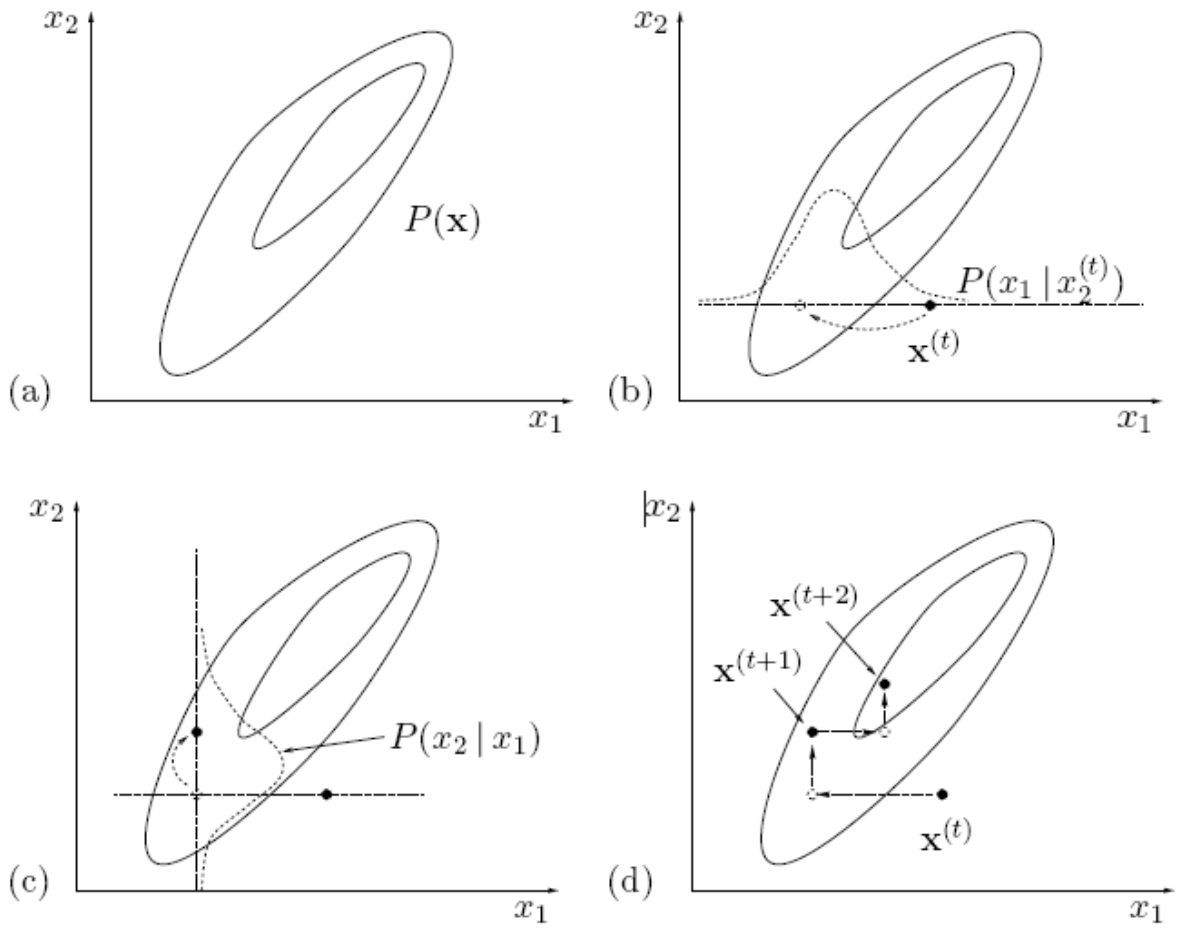
Markov Chain Monte Carlo

Can even update **one parameter at a time**.

Many different algorithms, examples include:

- Gibbs Sampling (sampling from known “full” conditionals)
- Metropolis-Hastings (more general, doesn’t require conjugacy)
- Hamiltonian Monte Carlo, HMC (reduces autocorrelation, used by Stan)

Markov Chain Monte Carlo



Markov Chain Monte Carlo

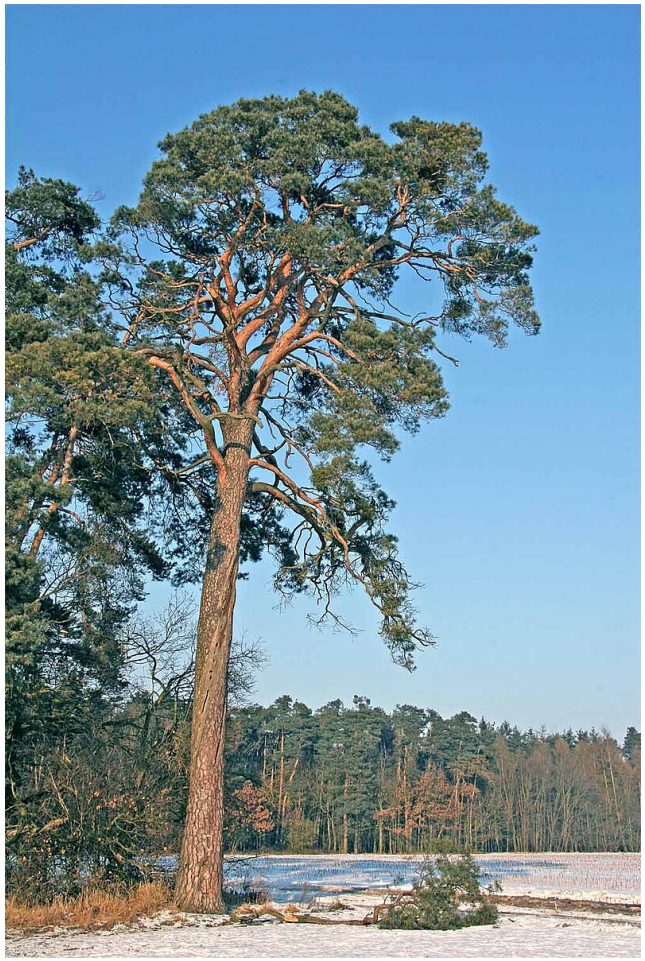
We obtain a *sample of points* for each parameter:

- these are **random samples from the posterior distribution**.

Take means to get point estimates, use percentiles to get interval estimates, etc.

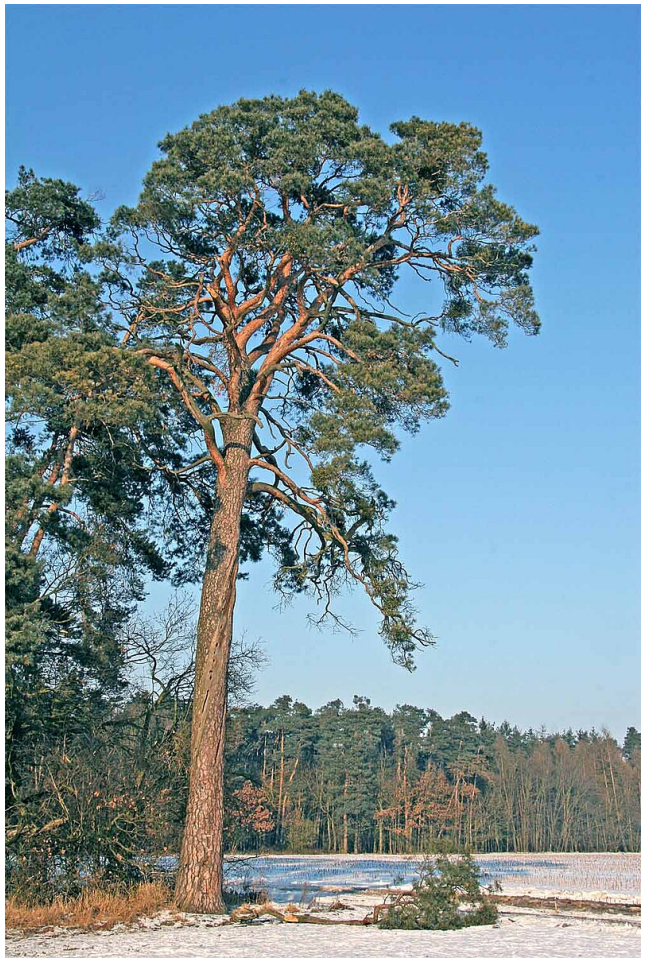
Histograms or KDEs can illustrate the posterior distribution.

Simple Example: Tree Rings



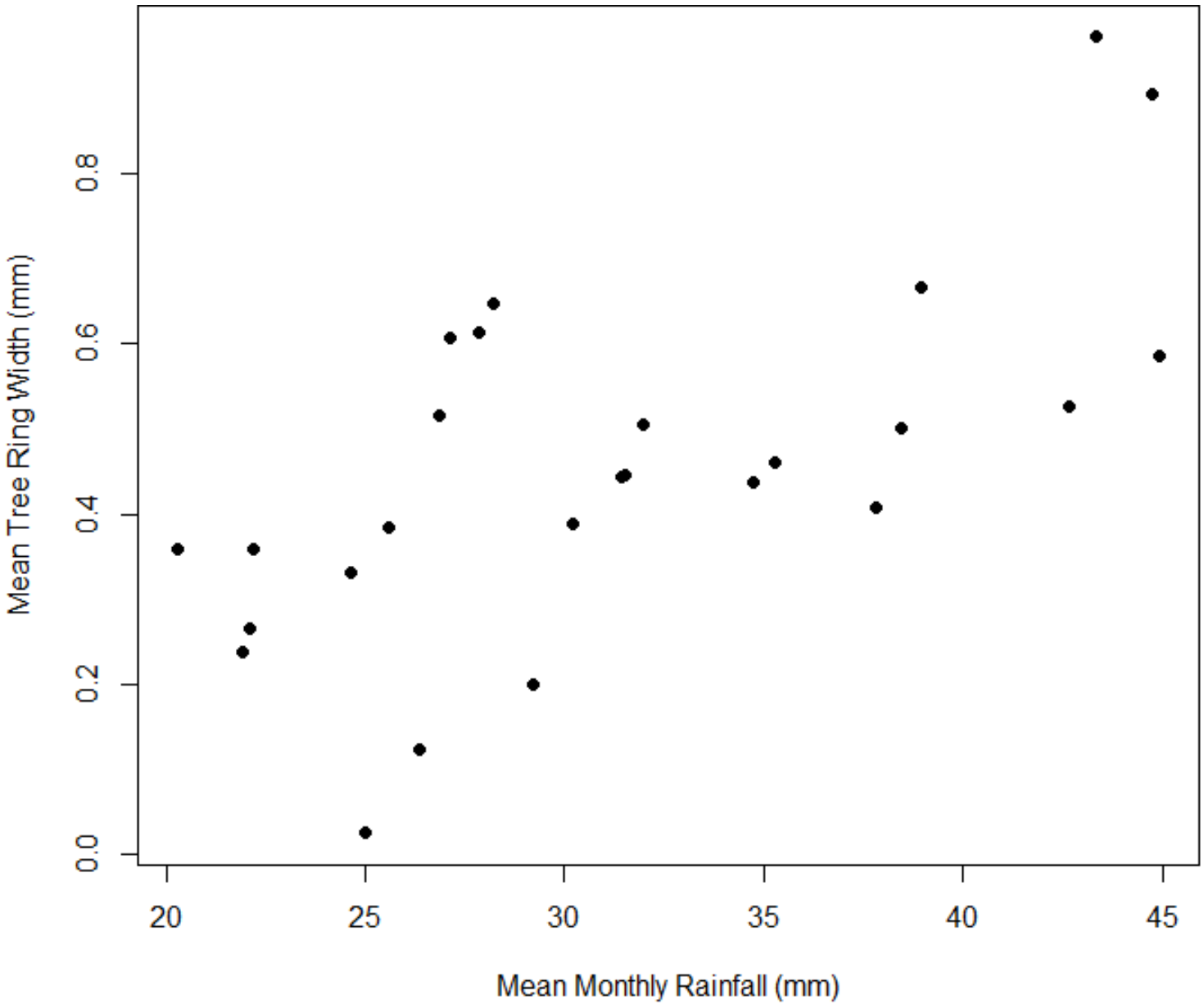
- Data from Antonio Gazol, now at the Instituto Pirenaico de Ecología in Zaragoza, Spain
- Tree ring data from *P. sylvestris* (Scots Pine) in Corbalán, Spain (Aragón)

Simple Example: Tree Rings

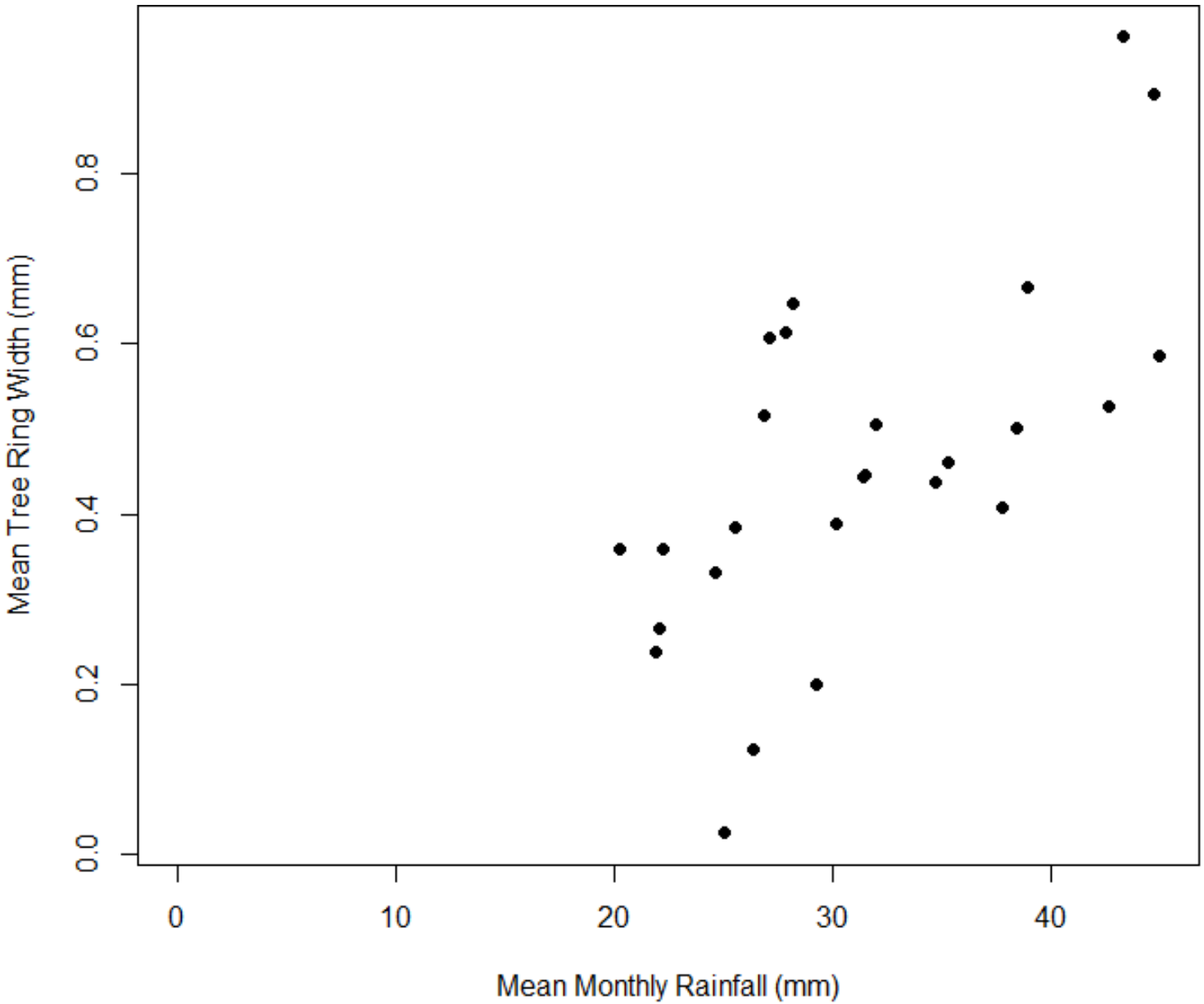


- Data for 26 years (1987 to 2012)
- Mean monthly rainfall (mm)
- Corresponding mean tree ring width (mm)

Simple Example: Tree Rings



Simple Example: Tree Rings



Model-Based PRA

- We can conduct PRA on models of our data
- Typically, this may involve regression-type models, with outcome/response variables modelled as dependent on explanatory variables
- If we take a fully Bayesian approach, this will account for uncertainty in the fitting of the model

Model-Based PRA

This requires:

- Finding a suitable model for the data
- Assigning prior distributions/probabilities to model parameters
- Obtaining posterior distribution estimates, typically via MCMC or similar computational tools

Model-Based PRA

- Assuming a simple linear regression model for simplicity
- Start by simulating values from the covariate distribution; with data, could assume normality or use a KDE etc
- We obtain the same number of samples as in the MCMC (so we have matched covariates and sampled parameters)

Model-Based PRA

- From these, we obtain new response data — using the regression line for that sample and the residual uncertainty
- Happens for every value of the threshold, and for hazard vs. non-hazard situations
- This fully Bayesian approach includes the uncertainty in the estimation of the model parameters
- Also accounts for correlation between model parameters

Model-Based PRA

- Relating back to the definitions of V and R
- $V = \mathrm{E}[z \mid \neg H] - \mathrm{E}[z \mid H]$
- $R = \mathrm{E}[z \mid \neg H] - \mathrm{E}[z]$
- We are effectively using MCMC to estimate the expectations above, accounting for modelling uncertainty

NIMBLE

- **NIMBLE**: **N**umerical **I**nference of statistical **M**odels for **B**ayesian and **L**ikelihood **E**stimation
- Has three main components:

NIMBLE

- **NIMBLE**: **N**umerical **I**nference of statistical **M**odels for **B**ayesian and **L**ikelihood **E**stimation
- Has three main components:
  1. A “BUGS”-style language for describing statistical models;

NIMBLE

- **NIMBLE**: **N**umerical **I**nference of statistical **M**odels for **B**ayesian and **L**ikelihood **E**stimation
- Has three main components:
  1. A “BUGS”-style language for describing statistical models;
  2. Algorithm library for NIMBLE models (MCMC, HMC, seqMC, quadrature)

NIMBLE

- **NIMBLE**: **N**umerical **I**nference of statistical **M**odels for **B**ayesian and **L**ikelihood **E**stimation
- Has three main components:
  1. A “BUGS”-style language for describing statistical models;
  2. Algorithm library for NIMBLE models (MCMC, HMC, seqMC, quadrature)
  3. A language in R for programming, which generates, compiles, and runs C++ code

NIMBLE

BioSS

`https://r-nimble.org`

- Extensive website, many examples and training materials freely available
- Several extension packages in R, e.g.:
  - **nimbleSMC**: for sequential Monte Carlo (particle filtering)
  - **nimbleEcology**: occupancy models etc
  - **nimbleSCR**: for capture-recapture models
  - **bayesNSGP**: Bayesian analysis of (non-stationary) Gaussian processes

NIMBLE

`https://r-nimble.org`

- NIMBLE allows you to:
  - define your own distributions and functions for use in model-definitions;
  - choose and customise your algorithms for MCMC etc;
  - write your own MCMC algorithms;
  - do everything in R, without needing to know or write C or C++;

NIMBLE

`https://r-nimble.org`

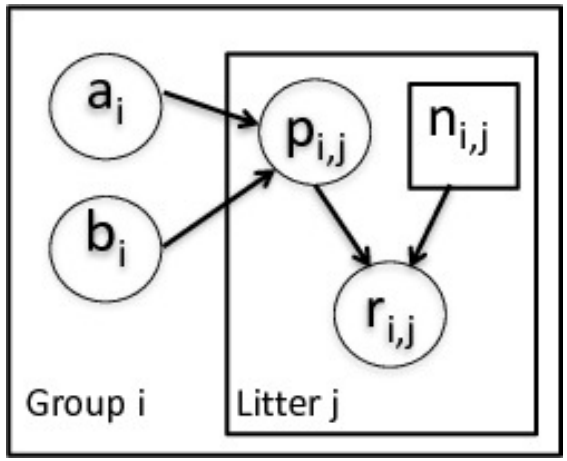
- NIMBLE is not necessarily optimal for:
  - standard Gibbs Sampling (JAGS more efficient?);
  - complex models which Stan can handle well;
  - very large models (tens of thousands of nodes) — can take a long time to compile, although subsequent run times should be OK.

Alternatives: JAGS, Stan, PyMC

NIMBLE: Defining Models

- **Stochastic declarations:**
  - `x ~ dgamma(shape,scale)`
- **Deterministic declarations:**
  - `y <- 2 * x`
- **Loops:** (over observations)
  - ```
    for(i in 1:10) {
      lambda[i] <- exp(mu[i])
      y[i] ~ dpois(lambda[i])
    }
    ```
    ```

NIMBLE: Litters Example



- Two groups of rat litters, `N=16` litters in each group, number of pups in each litter $n_{i,j}$
- Survival $r_{i,j}$ of pups in a litter governed by a survival probability for each litter, $p_{i,j}$
- Probabilities for litters within a group come from common distribution
- $p_{i,j} \sim \text{Beta}(a_i, b_i)$ for group i

NIMBLE: Model Code

```
littersCode <- nimbleCode({
    for (i in 1:G) {
        for (j in 1:N) {
            # likelihood (data model)
            r[i,j] ~ dbin(p[i,j], n[i,j])
            # latent process (random effects)
            p[i,j] ~ dbeta(a[i], b[i])
        }
        # prior for hyperparameters
        a[i] ~ dgamma(1, 0.001)
        b[i] ~ dgamma(1, 0.001)
    }
})
```

NIMBLE: Litters Example

- Code on previous slide can also be stored in a text file
- Key with NIMBLE is flexibility — you can rely on defaults or fine-tune modelling (advanced)
- Building the model has two stages: “define” and then “compile”
- Aside: calling `nimbleModel` is not always necessary, but we gain flexibility by creating a model object

NIMBLE: Build a Model

```
## data and constants as R objects
G <- 2
N <- 16
n <- matrix(c(13, 12, 12, ...,10 ,7), nrow = 2)
r <- matrix(c(13, 12, 12, ..., 7, 0), nrow = 2)

littersConsts <- list(G = G, N = N, n = n)
littersData <- list(r = r)
littersInits <- list( a = c(2, 2), b=c(2, 2) )

## create the NIMBLE model object
littersModel <- nimbleModel(littersCode,
   data = littersData, constants = littersConsts,
   inits = littersInits)
```

NIMBLE: Build a Model

- Compile model using
  - `cLittersModel <- compileNimble(littersModel)`
- Note, can inspect objects within the compiled model object, for example
  - `cLittersModel$p`
  - `cLittersModel$r`
  - `cLittersModel$calculate('a')`
- Final line gives the log-prior density (for `a`)

NIMBLE: Run MCMC

The steps for running MCMC in NIMBLE are:

1. Configure the MCMC (via `configureMCMC()`)
2. Build the MCMC (via `buildMCMC()`)
3. Create a compiled version of the MCMC (via `compileNimble()`)
4. Run the MCMC (via `runMCMC()`)
5. Assess and use the MCMC samples (study traces; calculate means, densities etc)

NIMBLE: Run MCMC

- Aside: Steps 1 to 4 on previous slide can be combined, using `nimbleMCMC()` as a short-cut
- However, prefer to retain flexibility (to modify samplers, repeat runs etc)

NIMBLE: Run MCMC

- Firstly, *configure* the MCMC
- This sets up the samplers to be used for each node/parameter or group of nodes/parameters
- NIMBLE provides a default configuration for ease of use

```
littersConf <-
configureMCMC(littersModel, print = TRUE)
```

NIMBLE: Run MCMC

- Can also add to the list of nodes/parameters to be monitored
- This ensures we store the samples we want
- By default, NIMBLE will store only the "top-level" nodes, i.e., hyperparameters with no stochastic parents
- We need to make sure we add any derived quantities to the list

```
littersConf$addMonitors(c('a', 'b', 'p'))
```

NIMBLE: Run MCMC

- Next, the MCMC algorithm must be “built”
- After this, it should be compiled (into C++) — this will enable *much* faster computation
- Illustration below uses the `project` argument; in general, projects can be referenced using the name of the original uncompiled model

```
littersMCMC <- buildMCMC(littersConf)
cLittersMCMC <-
compileNimble(littersMCMC, project =
littersModel)
```

NIMBLE: Run MCMC

- At last, we can run the MCMC to generate samples of nodes/parameters
- Running the R version (prior to compilation) can be very slow, so not recommended
- Next two slides show running the R version and the C++ version for comparison. . .

NIMBLE: Run MCMC

```
niter <- 1000
nburn <- 100

set.seed(1)

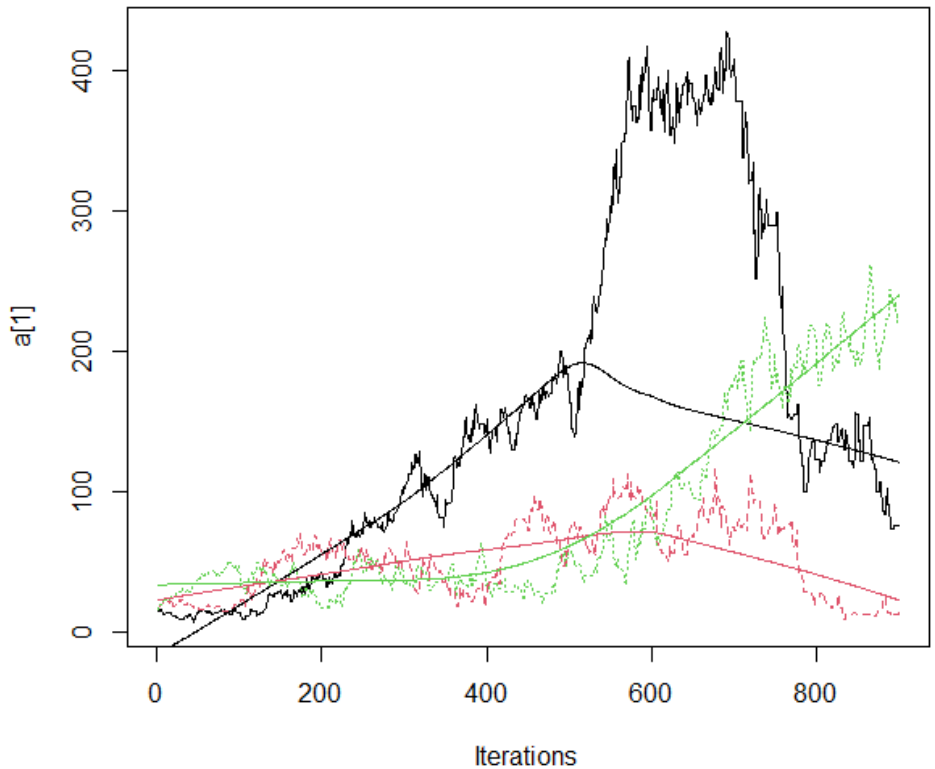
inits <- function() {
   a <- runif(G, 1, 20)
   b <- runif(G, 1, 20)
   p <- rbind(rbeta(N, a[1], b[1]), rbeta(N,
a[2], b[2]))
   return(list(a = a, b = b, p = p))
}
```

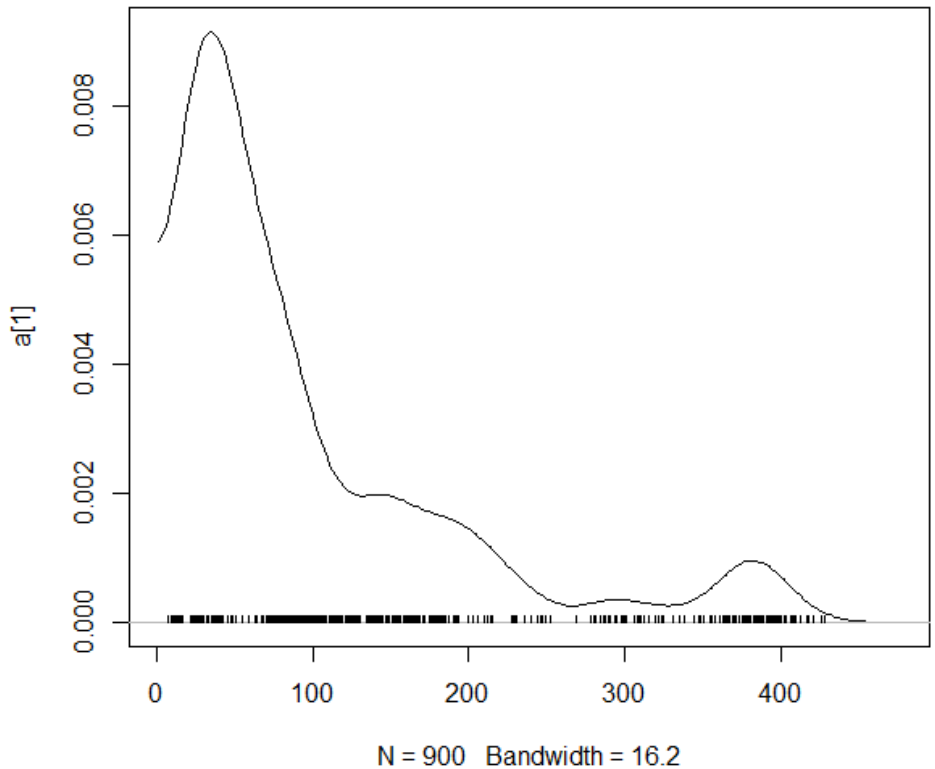
NIMBLE: Run MCMC

```
print(system.time(samples.slow <-
runMCMC(littersMCMC, niter = niter,
nburnin = nburn, inits = inits,
nchains = 3, samplesAsCodaMCMC = TRUE)))

print(system.time(samples <-
runMCMC(cLittersMCMC, niter = niter,
nburnin = nburn, inits = inits,
nchains = 3, samplesAsCodaMCMC = TRUE)))
```

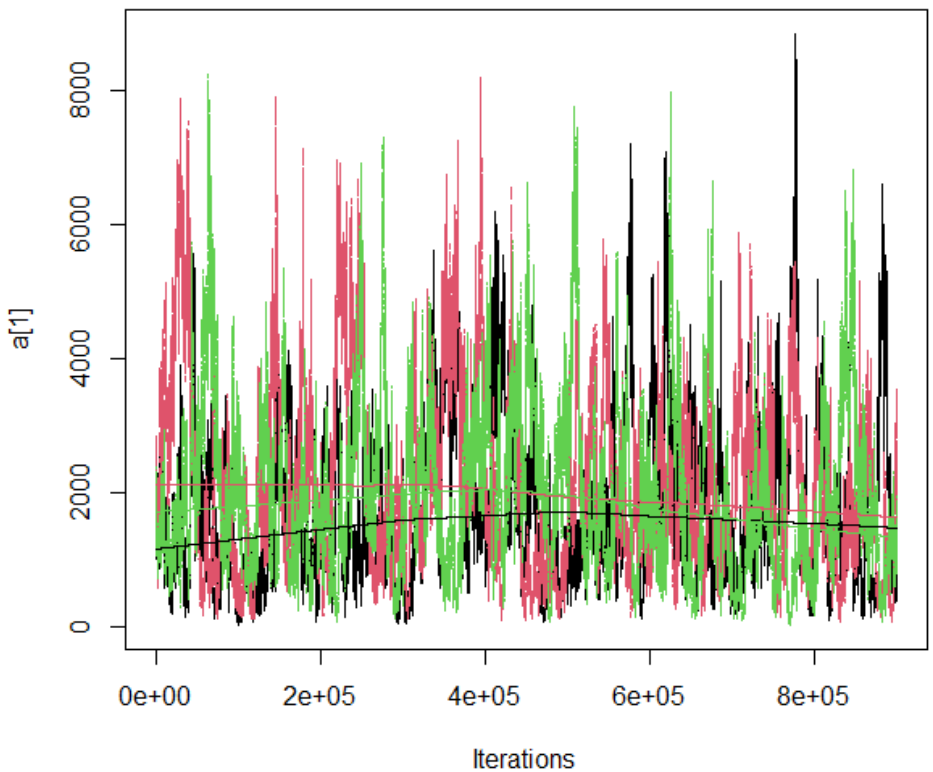
NIMBLE: MCMC Traces for a_1

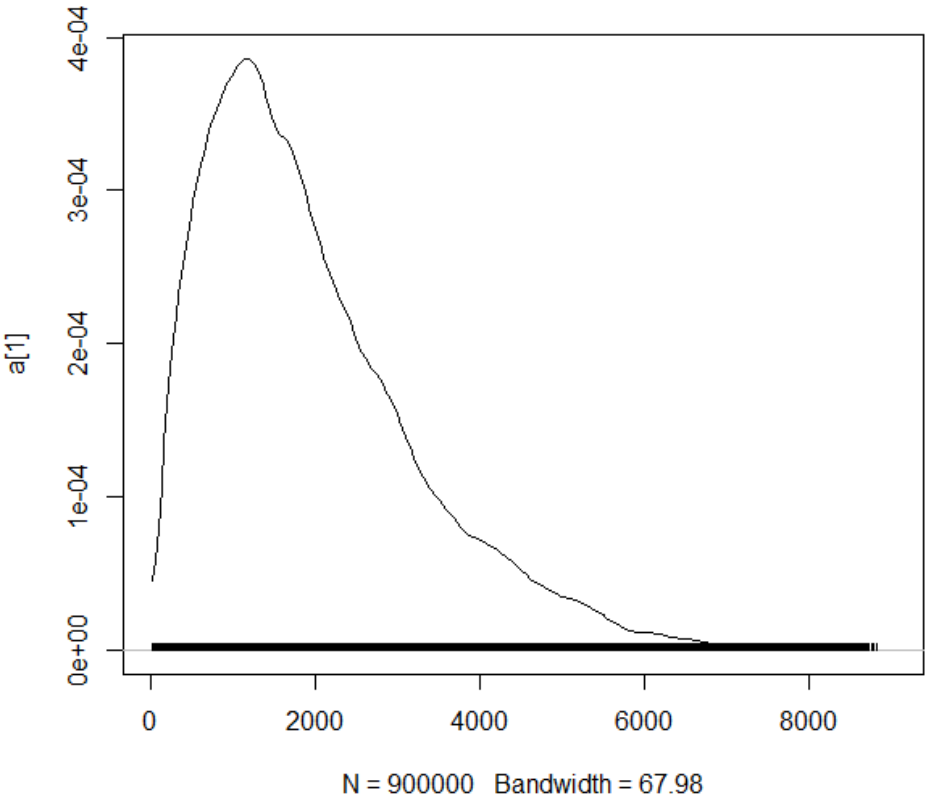




Evidence of non-convergence; may be resolved running longer chains or by blocking (simultaneous updating of parameters)

NIMBLE: MCMC Traces for a_1





Many more iterations! Not perfect, but better…

NIMBLE: Data vs Constants

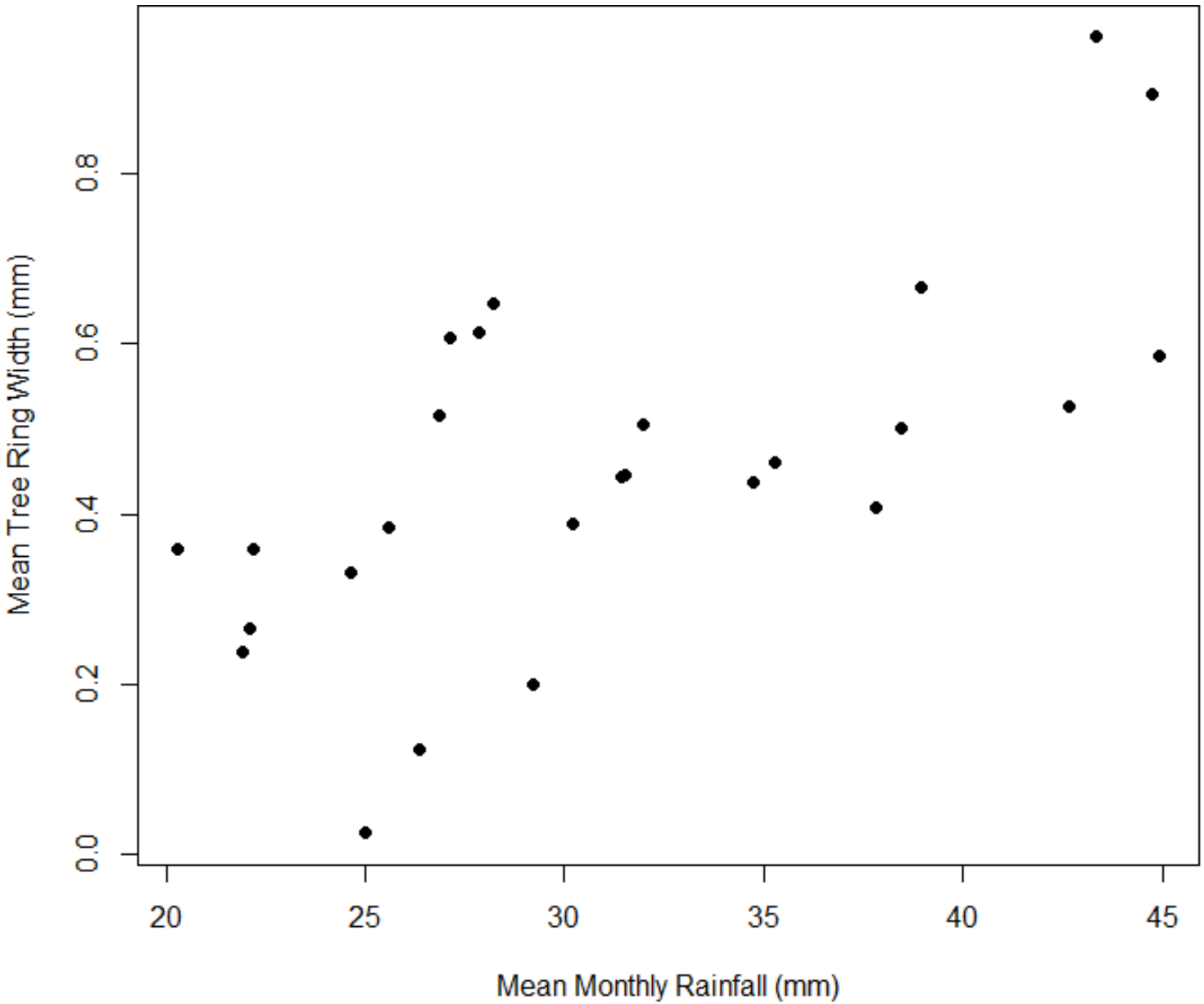
- Constants are values needed to define model relationships
  - Index ranges, `N` in litters example
  - Vectors for indexing, e.g. `mu[block[i]]`
  - Given to `nimbleModel`
- Data is a more general concept: observed values of variables
  - Can be sampled, but not during MCMC
  - Given to `nimbleModel` *or* supplied later

NIMBLE will try to work out what should be which. . .

NIMBLE: Data vs Constants

- `littersModel$isData('r')`
- `littersModel$isData('p')`
- `littersModel$r`
- `littersModel$p`
- `littersModel$simulate('r')`
- `littersModel$simulate('p')`
- `littersModel$simulate('r', includeData = TRUE)`

NIMBLE: Tree Rings PRA



NIMBLE: Tree Rings PRA

- Model: linear regression
- Will supply data length and covariate as constants
- `Model1.Constants <- list( ndata=n, x=Rainfall )`
- Data will be the response variable
- `Model1.Data <- list( z=Ring_Width )`

NIMBLE: Tree Rings PRA

```
Model1.Code <- nimbleCode({
    lm.alpha ~ dnorm ( 0, sd=100 )
    lm.beta ~ dnorm ( 0, sd=100 )
    lm.tau ~ dgamma( 0.01, 0.01 )
    lm.sigma <- 1 / sqrt(lm.tau)
    for(i in 1:ndata){
        lm.mu[i] <- lm.alpha + lm.beta*x[i]
        z[i] ~ dnorm( lm.mu[i], sd=lm.sigma )
    }
} )
```

NIMBLE: Tree Rings PRA

```
Model1.Nimble <- nimbleModel( Model1.Code,
constants=Model1.Constants, data=Model1.Data )
Model1.Comp <- compileNimble( Model1.Nimble )

Model1.Conf <- configureMCMC( Model1.Comp )
Model1.Conf$addMonitors( c("lm.sigma") )
Model1.MCMC <- buildMCMC( Model1.Conf )
Model1.MCMC.Comp <- compileNimble( Model1.MCMC )

nsims <- 2000 ; nburnin <- 500
niter <- nsims+nburnin
set.seed(1)
Model1.samples <- runMCMC( Model1.MCMC.Comp,
nburnin=nburnin, niter=niter )
```

NIMBLE: Tree Rings PRA

For a single run you can take a shortcut and compile/execute in one call:

```
Model1.samples.shortcut <- nimbleMCMC(
Model1.Comp, nburnin=nburnin, niter=niter,
monitors=c("lm.alpha.centred", "lm.beta",
"lm.tau", "lm.alpha", "lm.sigma") ))
```

If you want to do repeat runs this will be inefficient, as you will be compiling the C++ every time.

NIMBLE: Tree Rings PRA

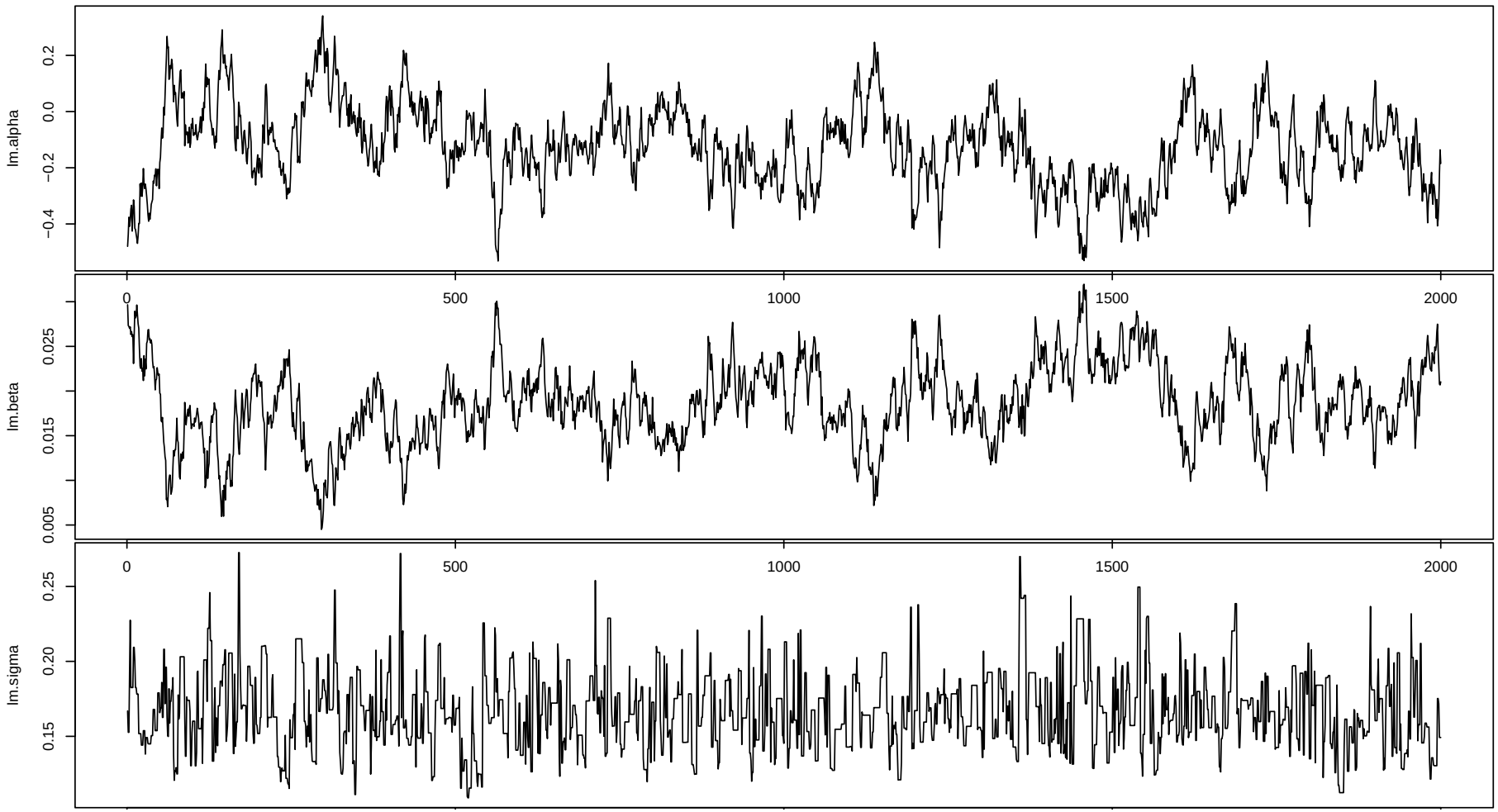
BioSS

You can also run the MCMC in R rather than in C++ — but note it will be a *lot* slower!

```
Model1.samples.shortcut <- nimbleMCMC(
Model1.Comp, nburnin=nburnin, niter=niter,
monitors=c("lm.alpha.centred", "lm.beta",
"lm.tau", "lm.alpha", "lm.sigma") ))
```

It can however be useful for diagnosing problems in models, especially complex models with user-defined distributions etc.

NIMBLE: Tree Rings PRA



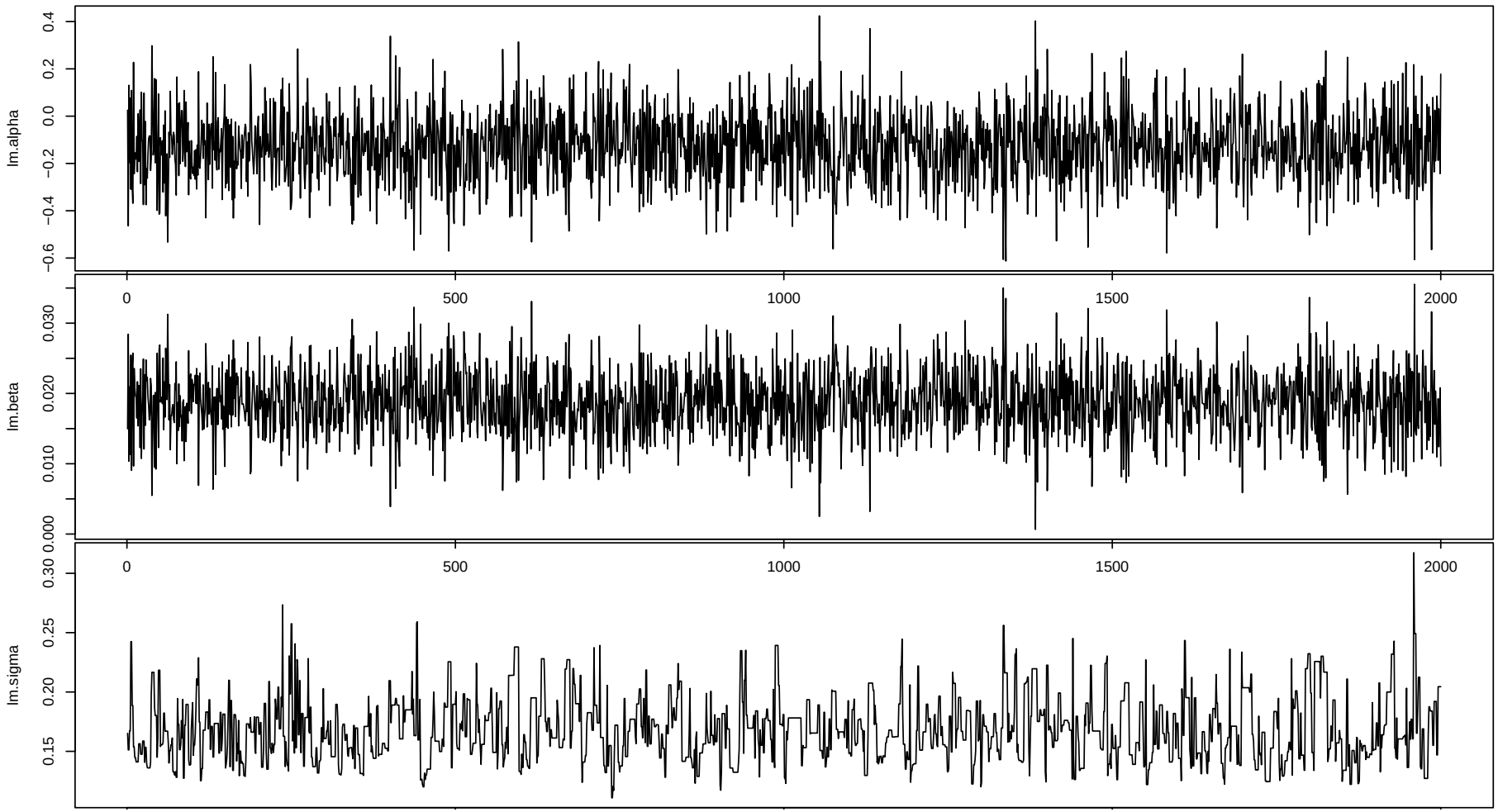
NIMBLE: Tree Rings PRA

- Convergence here looks slow
- Caused by high correlation between regression intercept and slope
- `cor(Model1.samples[ , "lm.alpha"],Model1.samples[ , "lm.beta"])`
- A simple fix is to mean-centre the covariate

NIMBLE: Tree Rings PRA

```
Model1.Code <- nimbleCode({
    lm.alpha.centred ~ dnorm ( 0, sd=100 )
    lm.alpha <- lm.alpha.centred - lm.beta*xmean
    lm.beta ~ dnorm ( 0, sd=100 )
    lm.tau ~ dgamma( 0.01, 0.01 )
    lm.sigma <- 1 / sqrt(lm.tau)
    xmean <- mean( x[1:ndata] )
    for(i in 1:ndata){
        lm.mu[i] <- lm.alpha.centred +
lm.beta*(x[i]-xmean)
        z[i] ~ dnorm( lm.mu[i], sd=lm.sigma )
    }
} )
```

NIMBLE: Tree Rings PRA



NIMBLE: Tree Rings PRA

- Convergence much better, reduced correlation in sampling (faster moving)
- `cor(Model1.samples[ , "lm.alpha.centred"],Model1.samples[ , "lm.beta"])`
- Can also investigate samples — they are samples from the marginal posterior distributions
- Calculate summary statistics, plot densities etc

NIMBLE: Tree Rings PRA

- End goal here is to plot the values of vulnerability and risk by threshold
- It is possible to include this in the MCMC, but for simplicity we illustrate via a further sampling stage
- We use the full set of MCMC samples in order to retain the full uncertainty

NIMBLE: Tree Rings PRA