Bayesian Modelling with JAGS and R

Martyn Plummer

International Agency for
Research on Cancer

Rencontres R, 3 July 2012

CRAN Task View "Bayesian Inference"

The CRAN Task View "Bayesian Inference" is maintained by Jong Hee Park

Classification	N Packages
General model fitting	8
Specific models or methods	73
Post-estimation tools	5
Learning Bayesian statistics	5
Linking R to other sampling engines	8
Total	99

Most of these packages use Markov Chain Monte Carlo (MCMC) or other simulation methods (sequential Monte Carlo, approximate Bayesian computing, importance sampling, ...).

BUGS: Bayesian Inference using Gibbs sampling

- The design of BUGS is based on developments in Artificial Intelligence in the 1980s (Lunn et al 2009)
  - Separation of *knowledge base* from *inference engine*
  - Knowledge base represented in a declarative form that "express local relationships between entities"
- Knowledge base represented by directed acyclic graph.
- Uncertainty represented with probability distribution on the graph.

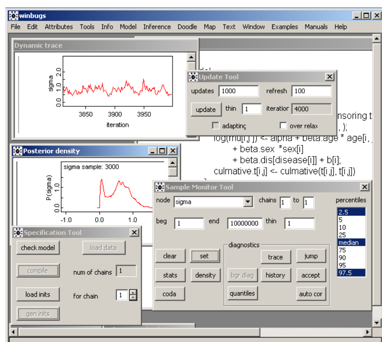
Brief MCMC overview

- MCMC creates a Markov chain with a given target distribution (in this case, the posterior distribution of the parameters) as its equilibrium distribution
- Under regularity conditions, the Markov chain will converge to the equilibrium distribution.
- MCMC then generates dependent samples from the target distribution
- Inference is conducted using the empirical distribution of the samples.

1980s-1990s: The growth of MCMC

- In the 1990s it became feasible to use simulation-based methods to analyse reasonably large statistical problems
- Markov Chain Monte Carlo (MCMC) techniques rediscovered
  - Gibbs sampling (Geman and Geman, 1989)
  - Metropolis-Hastings sampling (Metropolis et al 1950; Hastings 1970)
- BUGS combined Bayesian networks with MCMC.

1995: BUGS becomes WinBUGS



- Rewritten in Component Pascal
- Depends on the Black Box component framework
- Runs on Windows only
- Scripting language replaced by point-and-click interface (scripting re-introduced in later versions)

JAGS is Just Another Gibbs Sampler

Motivations for JAGS:

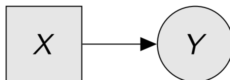
1. To have an alternative BUGS language engine that
   - is extensible
   - is cross-platform
   - can interface to R (rjags, R2jags, runjags)
2. To create a platform for exploring ideas in Bayesian modelling

Other Bayesian software libraries

PyMC	MCMC for Python
	http://code.google.com/p/pymc
HBC	Hierarchical Bayes Compiler
	http://www.cs.utah.edu/~hal/HBC
YADAS	Yet Another Data Analysis System
	http://www.stat.lanl.gov/yadas
HYDRA	MCMC library
	http://sourceforge.net/projects/hydra-mcmc
Scythe	Statistical library (MCMCpack)
	http://scythe.wustl.edu
CppBUGS	C++ version of BUGS
	https://github.com/armstrtw/CppBugs
Stan	A C++ library for probability and sampling
	http://code.google.com/p/stan/

Statistical models as graphs

- In a graphical model, random variables are represented as nodes, and the relations between them by edges.

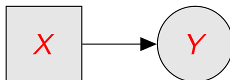


In this simple model, Y is the outcome variable and X is a vector of predictor variables, or covariates.

- Graphical models become more interesting when we have multiple variables, and the relations between them become more complex.
- The BUGS language is an S-like language for describing graphical models.

Statistical models as graphs

- In a graphical model, random variables are represented as nodes, and the relations between them by edges.

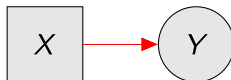


  In this simple model, Y is the outcome variable and X is a vector of predictor variables, or covariates.

- Graphical models become more interesting when we have multiple variables, and the relations between them become more complex.
- The BUGS language is an S-like language for describing graphical models.

Statistical models as graphs

- In a graphical model, random variables are represented as nodes, and the relations between them by edges.

  In this simple model, Y is the outcome variable and X is a vector of predictor variables, or covariates.
- Graphical models become more interesting when we have multiple variables, and the relations between them become more complex.
- The BUGS language is an S-like language for describing graphical models.

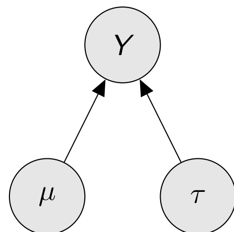
Stochastic Relations

The relation

$$Y \sim N(\mu, \tau^{-1})$$

is written as

```
Y ~ dnorm(mu, tau)
```



This relation can be represented by a graph in which Y, μ, τ are nodes. The dependency of Y on parameters μ, τ is represented by directed edges.

Stochastic relations

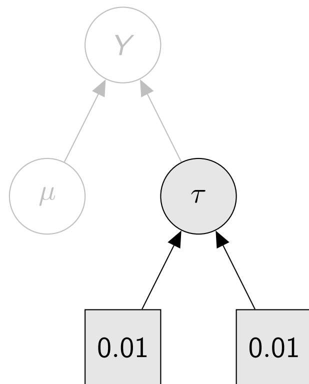
A parameter can itself have a distribution with its own hyper-parameters

$$\tau \sim \Gamma(0.01, 0.01)$$

In the BUGS language

```
tau ~ dgamma(0.01, 0.01)
```

This fits with the Bayesian approach to statistical inference, in which the parameters of a model are also random variables.



Deterministic relations

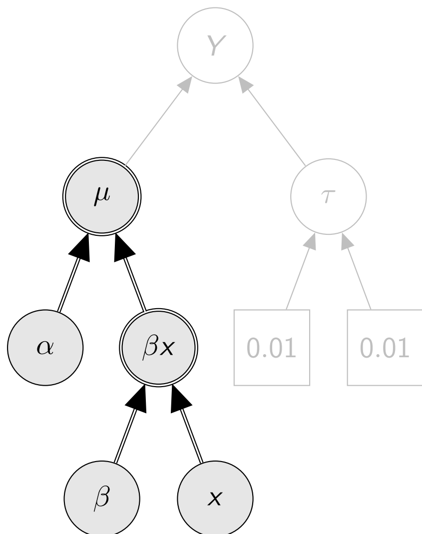
We can also describe deterministic relationships between variables

$$\mu = \alpha + \beta x$$

In BUGS:

```
mu <- alpha + beta * x
```

They are represented by double arrows.



Arrays and for loops

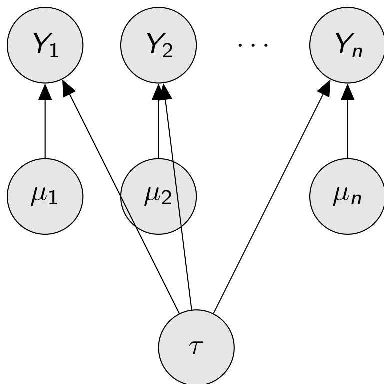
Repeated structures in the graph can be simplified using arrays and for loops.

$$Y_i \sim N(\mu_i, \tau^{-1}) \;\; i = 1 \ldots n$$

In BUGS:

```
for (i in 1:n) {
   Y[i] ~ dnorm(mu[i], \tau)
}
```

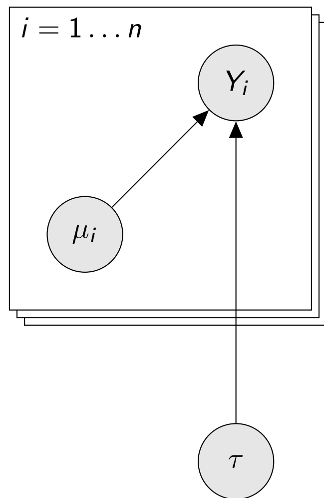
Here the nodes Y[1] to Y[n] are embedded in the array Y. Matrices and higher-dimensional arrays can also be used.



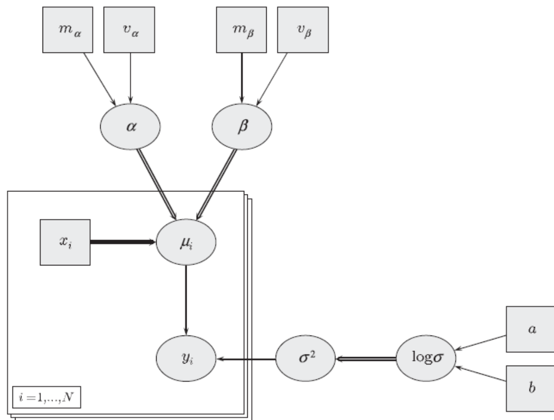
Plates

Repeated structures can make the graph hard to read.
To simplify drawing of the graph, we use a "plate" notation.

- Only one entry in the for loop is shown.
- The rest are implied by the stack of plates



A linear regression example



Code for the linear regression example

In BUGS

```
for (i in 1:N) {
   y[i] ~ dnorm(mu[i], tau)
   mu[i] <- alpha + beta * x[i]
}
alpha ~ dnorm(m.alpha, p.alpha)
beta ~ dnorm(m.beta, p.beta)
log.sigma ~ dunif(a, b)
sigma <- exp(log.sigma)
sigma.sq <- pow(sigma, 2)
tau <- 1 / sigma.sq
```

Code for the linear regression example

In BUGS

```
for (i in 1:N) {
   y[i] ~ dnorm(mu[i], tau)
   mu[i] <- alpha + beta * x[i]
}
alpha ~ dnorm(m.alpha, p.alpha)
beta ~ dnorm(m.beta, p.beta)
log.sigma ~ dunif(a, b)
sigma <- exp(log.sigma)
sigma.sq <- pow(sigma, 2)
tau <- 1 / sigma.sq
```

In R

```
lm(y ~ x)
```

BUGS code can be verbose

All parts of the model must be explicitly defined.

```
for (i in 1:N) {
   y[i] ~ dnorm(mu[i], tau)
   mu[i] <- alpha + beta * x[i]
}
alpha ~ dnorm(m.alpha, p.alpha)
beta ~ dnorm(m.beta, p.beta)
log.sigma ~ dunif(a, b)
sigma <- exp(log.sigma)
sigma.sq <- pow(sigma, 2)
tau <- 1 / sigma.sq
```

- You need to specify the *parameters* as well as the data.
- Parameters need to have explicit *prior distributions*
- Unlike R, the language is not vectorized, so you need `for` loops for repeated calculations.
- The model may include parameter transformations.

BUGS code can be verbose

All parts of the model must be explicitly defined.

```
for (i in 1:N) {
   y[i] ~ dnorm(mu[i], tau)
   mu[i] <- alpha + beta * x[i]
}
alpha ~ dnorm(m.alpha, p.alpha)
beta ~ dnorm(m.beta, p.beta)
log.sigma ~ dunif(a, b)
sigma <- exp(log.sigma)
sigma.sq <- pow(sigma, 2)
tau <- 1 / sigma.sq
```

- You need to specify the *parameters* as well as the data.
- Parameters need to have explicit *prior distributions*
- Unlike R, the language is not vectorized, so you need `for` loops for repeated calculations.
- The model may include parameter transformations.

BUGS code can be verbose

All parts of the model must be explicitly defined.

```
for (i in 1:N) {
   y[i] ~ dnorm(mu[i], tau)
   mu[i] <- alpha + beta * x[i]
}
alpha ~ dnorm(m.alpha, p.alpha)
beta ~ dnorm(m.beta, p.beta)
log.sigma ~ dunif(a, b)
sigma <- exp(log.sigma)
sigma.sq <- pow(sigma, 2)
tau <- 1 / sigma.sq
```

- You need to specify the *parameters* as well as the data.
- Parameters need to have explicit *prior distributions*
- Unlike R, the language is not vectorized, so you need `for` loops for repeated calculations.
- The model may include parameter transformations.

BUGS code can be verbose

All parts of the model must be explicitly defined.

```
for (i in 1:N) {
   y[i] ~ dnorm(mu[i], tau)
   mu[i] <- alpha + beta * x[i]
}
alpha ~ dnorm(m.alpha, p.alpha)
beta ~ dnorm(m.beta, p.beta)
log.sigma ~ dunif(a, b)
sigma <- exp(log.sigma)
sigma.sq <- pow(sigma, 2)
tau <- 1 / sigma.sq
```

- You need to specify the *parameters* as well as the data.
- Parameters need to have explicit *prior distributions*
- Unlike R, the language is not vectorized, so you need `for` loops for repeated calculations.
- The model may include parameter transformations.

BUGS code can be verbose

All parts of the model must be explicitly defined.

```
for (i in 1:N) {
   y[i] ~ dnorm(mu[i], tau)
   mu[i] <- alpha + beta * x[i]
}
alpha ~ dnorm(m.alpha, p.alpha)
beta ~ dnorm(m.beta, p.beta)
log.sigma ~ dunif(a, b)
sigma <- exp(log.sigma)
sigma.sq <- pow(sigma, 2)
tau <- 1 / sigma.sq
```

- You need to specify the *parameters* as well as the data.
- Parameters need to have explicit *prior distributions*
- Unlike R, the language is not vectorized, so you need `for` loops for repeated calculations.
- The model may include parameter transformations.

BUGS code can be verbose

All parts of the model must be explicitly defined.

```
for (i in 1:N) {
   y[i] ~ dnorm(mu[i], tau)
   mu[i] <- alpha + beta * x[i]
}
alpha ~ dnorm(m.alpha, p.alpha)
beta ~ dnorm(m.beta, p.beta)
log.sigma ~ dunif(a, b)
sigma <- exp(log.sigma)
sigma.sq <- pow(sigma, 2)
tau <- 1 / sigma.sq
```

- You need to specify the *parameters* as well as the data.
- Parameters need to have explicit *prior distributions*
- Unlike R, the language is not vectorized, so you need `for` loops for repeated calculations.
- The model may include parameter transformations.

MCMC on a graphical model

- A BUGS model defines a distribution on a set of nodes $\{v_1, \dots v_n\}$ on a graph.
- The distribution factorizes as

$$p(\mathbf{v}) = \prod_{i=1}^{n} p(v_i \mid \text{Parents}(v_i))$$

- **Gibbs sampling** is an MCMC algorithm that consists of visiting each node in turn, sampling it from its full conditional distribution $p(v_i \mid \mathbf{v}_{-i})$ where $\mathbf{v}_{-i} = \{v_1 \dots v_{i-1}, v_{i+1} \dots v_n\}$
- $p(v_i \mid \mathbf{v}_{-i})$ depends only on local properties of the graph (children, parents, co-parents of v_i).

MCMC on a graphical model

- A BUGS model defines a distribution on a set of nodes $\{v_1, \dots v_n\}$ on a graph.
- The distribution factorizes as

$$p(\mathbf{v}) = \prod_{i=1}^{n} p(v_i \mid \text{Parents}(v_i))$$

- **Gibbs sampling** is an MCMC algorithm that consists of visiting each node in turn, sampling it from its full conditional distribution $p(v_i \mid \mathbf{v}_{-i})$ where $\mathbf{v}_{-i} = \{v_1 \dots v_{i-1}, v_{i+1} \dots v_n\}$
- $p(v_i \mid \mathbf{v}_{-i})$ depends only on local properties of the graph (children, parents, co-parents of v_i).

Metropolis-Hastings within Gibbs sampling

- We do not need to sample from $p(v_i \mid \mathbf{v}_{-i})$ directly.
- It is sufficient to generate a reversible transition $v_i \rightarrow v_i'$ that satisfies the detailed balance relation.

$$p(v_i \mid \mathbf{v}'_{-i}) p_t(v_i \rightarrow v_i') = p(v_i' \mid \mathbf{v}'_{-i}) p_t(v_i' \rightarrow v_i)$$

- **Slice sampling** is the default method in JAGS for generating reversible transitions
- **Metropolis-Hastings** is a general algorithm turning a proposed transition kernel $p_t(v_i \rightarrow v_i')$ into a reversible transition, at the cost of rejecting some proposed moves.

Model objects in R

In general, a model object in R is created from

- A description of the model
- A data set (or variables from the calling environment)
- Initial values (optional)

```
glm.out <- glm(lot1 ~ log(u), family=Gamma, data=clotting)
```

The model object can be queried via extractor functions to produce:

- Parameter estimates (`summary`, `coef`, `vcov`, `confint`)
- Predictions (predict)

These functions are generic. New methods can be developed for new model objects.

jags.model objects in the rjags package

A JAGS model object is created in the standard way

```
library(rjags)
m <- jags.model("blocker.bug", data, inits, n.chains = 2)
```

- The first argument is the name of a file containing the BUGS code for the model.
- `data` is a named list of data for observed nodes
- `inits` is a list of lists of initial values (one for each chain)
- `n.chains` is the number of parallel chains

But `m` does not represent a “fitted model”. It is a dynamic object that can be queried to generate samples from the posterior.

Burn-in

- Unlike most algorithms used in statistics, the MCMC algorithm does not tell you when it has converged.
- Convergence must be determined empirically by running parallel chains from very different starting values.
  - This is formalized by the Gelman-Rubin convergence diagnostic.
- In any case, we need to discard the initial output from the Markov chain (typically 1000–10000 samples)
- The `update` method runs the Markov chain but discards the sampling history

```
update(m, 3000)
```

Drawing samples from a jags.model object

The coda.samples function updates the model and stores the sampled values of monitored nodes

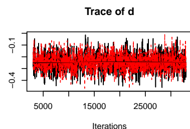
```
x <- coda.samples(m, c("d","delta.new","sigma"),
                  n.iter=30000, thin=10)
```

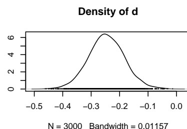
It returns an object of class "mcmc.list".

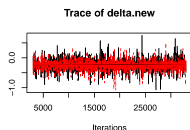
CRAN package coda

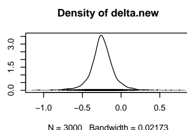
- CODA is (almost) an abbreviation of "Convergence Diagnosis and Output Analysis"
- Provides an object-based infrastructure for representing MCMC output
- Convergence diagnostics are empirical tests of convergence or run length control.
- 93 packages on CRAN depend on, import, or suggest coda
    - Not necessarily in the "Bayesian Inference" task view

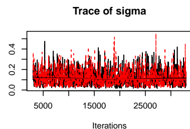
Plotting `mcmc.list` objects

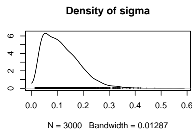








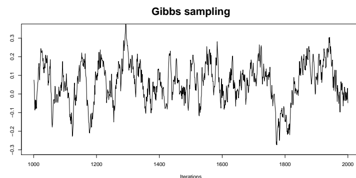


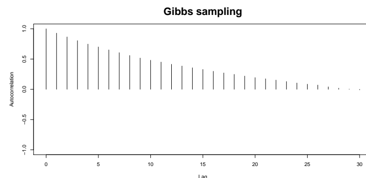


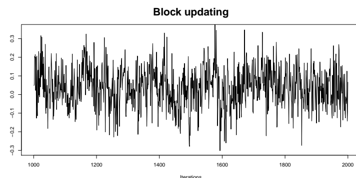
The plot method for mcmc.list objects produces a trace plot and a density plot.
This reflects the dual nature of an `mcmc` object as both time series and empirical estimate of the posterior.

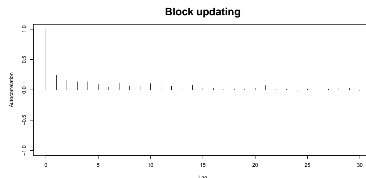
Good and bad "mixing"

An inefficient MCMC sampler exhibits random walk behaviour



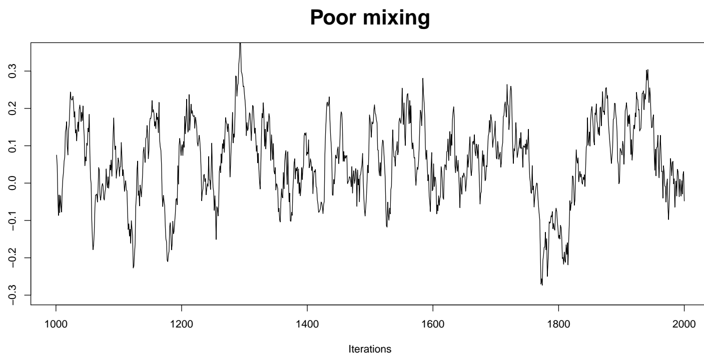




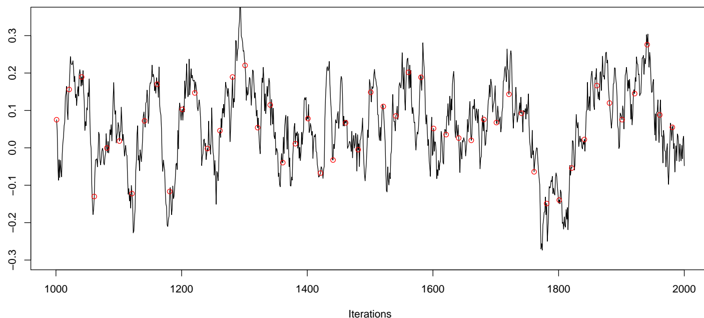


This can also be diagnosed by an autocorrelation plot.

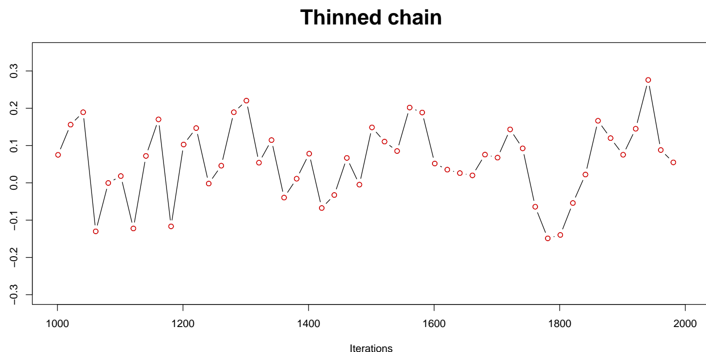
"Solving" autocorrelation by thinning



"Solving" autocorrelation by thinning

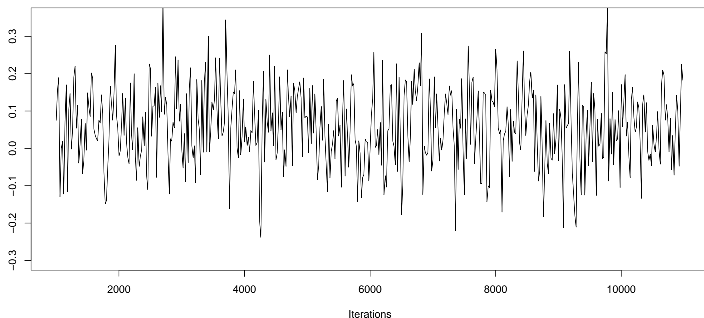


"Solving" autocorrelation by thinning



"Solving" autocorrelation by thinning

"Solving" autocorrelation by thinning



Effective number of samples

The `effectiveSize` function shows the equivalent number of independent samples from the posterior that contain the same information.

```
> niter(x)*nchain(x) #Number of actual samples
[1] 6000
> effectiveSize(x)
        d delta.new     sigma
1603.1158 4162.9912  956.4735
```

JAGS library

The JAGS library consists of

- A compiler that creates a virtual graphical model from a BUGS language description
- C++ classes for representing all the objects used by a virtual graphical model (Nodes, Graphs, Samplers, Monitors,...)
- A "Console" class provides a clean, safe interface to the JAGS library (e.g. it catches all exceptions)

JAGS modules

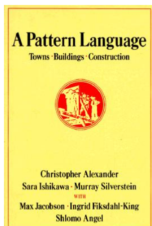
Modules are dynamically loadable extensions to the JAGS library. They can provide new

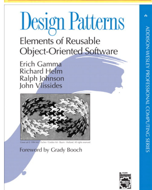
- Functions
- Distributions
- Samplers (More efficient ways of sampling)
- Monitors (Sequentially updated summary statistics)
- Random Number Generators

Standard jags modules

basemod	Functions and distributions built into the compiler. "Universal" samplers.
bugs	Functions and distributions from OpenBUGS. Conjugate samplers.
dic	Deviance statistics
mix	Distributions for finite mixture models. Simulated tempering sampler.
glm	Block samplers for generalized linear (mixed) models.
lecuyer	Pierre L'Ecuyer's rngstreams.

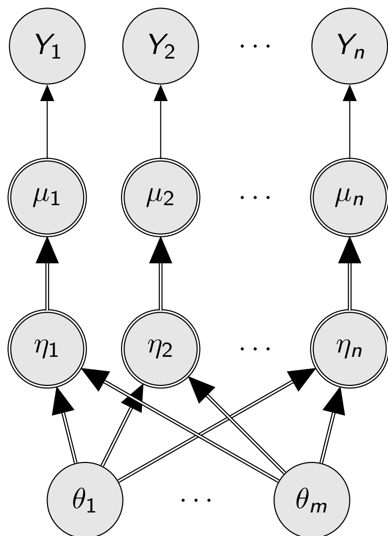
Design Patterns





- Design patterns are reusable solutions to commonly recurring design problems
- Originally developed in architecture, the patterns concept has been translated to software development.
- This may be a useful way of thinking about efficient sampling of graphical models, which often have a rich structure.
- First we need to look for recurring design motifs

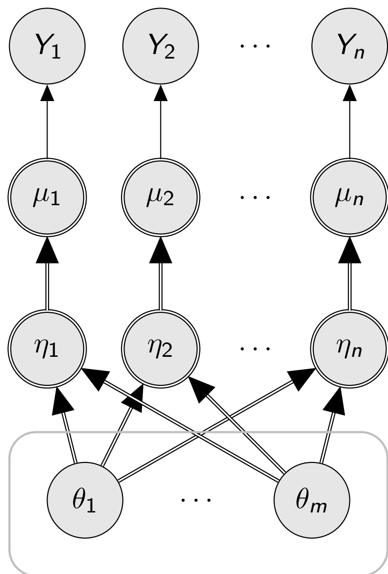
GLM as a design motif



A GLM is a sub-graph with the following elements

- **parameters** θ with prior normal distribution
- **linear predictors** η are linear functions of the parameters (intermediate nodes omitted).
- **link functions** transform linear predictor η to mean value μ
- **Outcome variables Y** depend on parameters θ via the mean μ

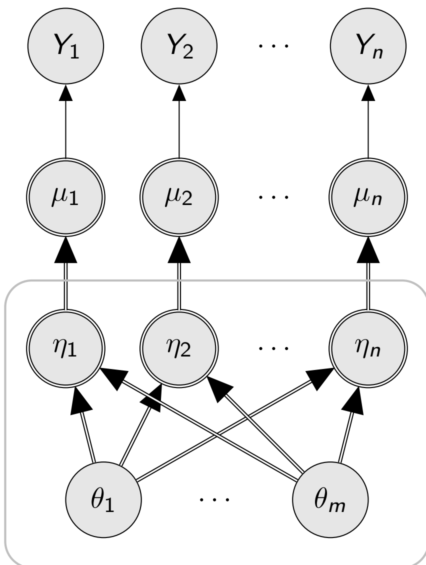
GLM as a design motif



A GLM is a sub-graph with the following elements

- **parameters θ** with prior normal distribution
- **linear predictors** η are linear functions of the parameters (intermediate nodes omitted).
- **link functions** transform linear predictor η to mean value μ
- **Outcome variables Y** depend on parameters θ via the mean μ

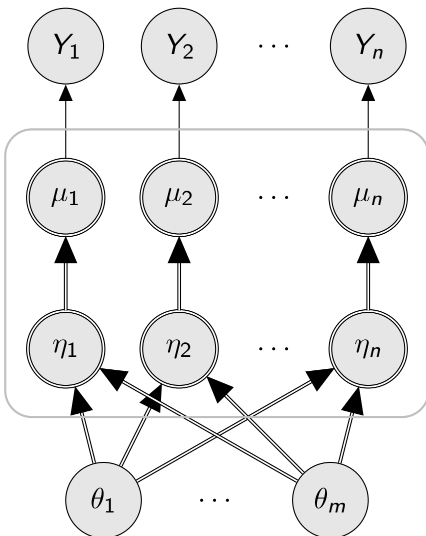
GLM as a design motif



A GLM is a sub-graph with the following elements

- **parameters** θ with prior normal distribution
- **linear predictors** η are linear functions of the parameters (intermediate nodes omitted).
- **link functions** transform linear predictor η to mean value μ
- **Outcome variables Y** depend on parameters θ via the mean μ

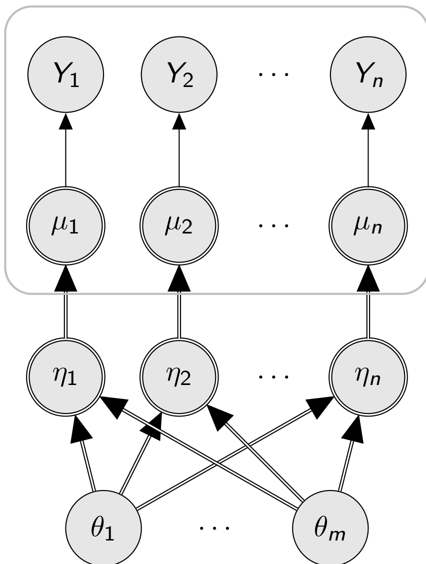
GLM as a design motif



A GLM is a sub-graph with the following elements

- **parameters** $\boldsymbol{\theta}$ with prior normal distribution
- **linear predictors** $\boldsymbol{\eta}$ are linear functions of the parameters (intermediate nodes omitted).
- **link functions** transform linear predictor $\boldsymbol{\eta}$ to mean value $\boldsymbol{\mu}$
- **Outcome variables Y** depend on parameters θ via the mean μ

GLM as a design motif

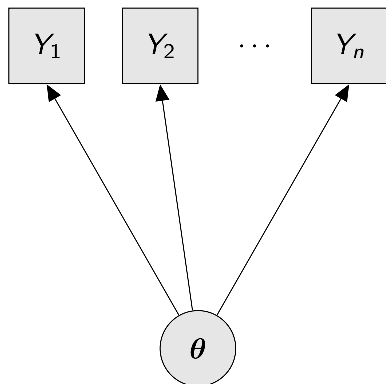


A GLM is a sub-graph with the following elements

- **parameters** $\boldsymbol{\theta}$ with prior normal distribution
- **linear predictors** $\boldsymbol{\eta}$ are linear functions of the parameters (intermediate nodes omitted).
- **link functions** transform linear predictor $\boldsymbol{\eta}$ to mean value $\boldsymbol{\mu}$
- **Outcome variables Y** depend on parameters $\boldsymbol{\theta}$ via the mean $\boldsymbol{\mu}$

Module "glm": Linearisation by data augmentation

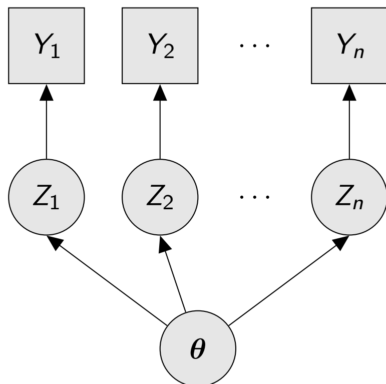
Some generalized linear models can be turned into linear models by data augmentation.



- Original idea by Albert and Chib (1993) for binary regression with a probit link.
- Refined by Holmes and Held (2006) and extended to logistic regression.
- Poisson and Binomial regression handled by Fruhwirth-Schnatter et al (2009).

Module "glm": Linearisation by data augmentation

Some generalized linear models can be turned into linear models by data augmentation.



- Original idea by Albert and Chib (1993) for binary regression with a probit link.
- Refined by Holmes and Held (2006) and extended to logistic regression.
- Poisson and Binomial regression handled by Fruhwirth-Schnatter et al (2009).

Strengths

- Universal language (BUGS) for describing Bayesian models
- Universal method (MCMC) for analyzing them
- Perfect concordance between graphical models and MCMC.
  - Density calculations remain feasible even in large graphical models.
  - The density depends only on local features of the graph.

Weaknesses

- MCMC may not converge.
- Long runs may be required to get target effective sample size.
- Large memory overhead associated with virtual graphical model.
- Restricted to fully parametric models.
- Restricted to models of fixed dimension.

Future plans

- Rewriting coda
- Improving sampling methods in modules.
- Parallelization (See the `dclone` package)
- Documentation of API to encourage third-party modules

Future perspectives

Computing is evolving towards parallel architecture. Can MCMC – an inherently sequential algorithm – survive?