Mamba.jl Documentation

Release 0.4.12

Brian J Smith

September 07, 2015

Contents

Version 0.4.12

Requires julia 0.3

Date September 07, 2015

Maintainer Brian J Smith (brian-j-smith@uiowa.edu)

Contributors Benjamin Deonovic (benjamin-deonovic@uiowa.edu), Brian J Smith (brian-j-smith@uiowa.edu), and others

Web site https://github.com/brian-j-smith/Mamba.jl

CHAPTER 1

Overview

1.1 Purpose

Mamba is a julia programming environment and toolset for the implementation and inference of Bayesian models using MCMC sampling. The package provides a framework for (1) specification of hierarchical models through stated relationships between data, parameters, and statistical distributions; (2) block-updating of parameters with samplers provided, defined by the user, or available from other packages; (3) execution of sampling schemes; and (4) posterior inference. It is designed to give users access to all levels of the design and implementation of MCMC simulators to particularly aid in the development of complex models.

Several software options are available for MCMC sampling of Bayesian models. Individuals who are primarily interested in data analysis, unconcerned with the details of MCMC, and have models that can be fit in JAGS, Stan, or OpenBUGS are encouraged to use those programs. *Mamba* is intended for individuals who wish to have access to lower-level MCMC tools, are knowledgeable of MCMC methodologies, and have experience, or wish to gain experience, with their application. The package also provides stand-alone convergence diagnostics and posterior inference *tools*, which are essential for the analysis of MCMC output regardless of the software used to generate it.

1.2 Features

- An interactive and extensible interface.
- Support for a wide range of model and distributional specifications.
- An environment in which all interactions with the software are made through a single, interpreted programming language.
  - Any **julia** operator, function, type, or package can be used for model specification.
  - Custom distributions and samplers can be written in **julia** to extend the package.
- Directed acyclic graph representations of models.
- Arbitrary blocking of model parameters and designation of block-specific samplers.
- Samplers that can be used with the included simulation engine or apart from it, including Slice, adaptive multivariate Metropolis, adaptive Metropolis within Gibbs, and No-U-Turn (Hamiltonian Monte Carlo) samplers.
- Automatic parallel execution of parallel MCMC chains on multi-processor systems.
- Restarting of chains.
- Command-line access to all package functionality, including its simulation API.
- Convergence diagnostics: Gelman, Rubin, and Brooks; Geweke; Heidelberger and Welch; Raftery and Lewis.

- Posterior summaries: moments, quantiles, HPD, cross-covariance, autocorrelation, MCSE, ESS.
- Gadfly plotting: trace, density, running mean, autocorrelation.
- Run-time performance on par with compiled MCMC software.

1.3 Getting Started

The following **julia** command will install the package:

```
julia> Pkg.add("Mamba")
```

CHAPTER 2

Contents

2.1 Introduction

2.1.1 MCMC Software

Markov chain Monte Carlo (MCMC) methods are a class of algorithms for simulating autocorrelated draws from probability distributions *[11][24][35][63]*. They are widely used to obtain empirical estimates for and make inference on multidimensional distributions that often arise in Bayesian statistical modelling, computational physics, and computational biology. Because MCMC provides estimates of *distributions* of interest, and is not limited to *point* estimates and asymptotic standard errors, it facilitates wide ranges of inferences and provides for more realistic prediction errors. An MCMC algorithm can be devised for any probability model. Implementations of algorithms are computational in nature, with the resources needed to execute algorithms directly related to the dimensionality of their associated problems. Rapid increases in computing power and emergence of MCMC software have enabled models of increasing complexity to be fit. For all its advantages, MCMC is regarded as one of the most important developments and powerful tools in modern statistical computing.

Several software programs provide Bayesian modelling via MCMC. Programs range from those designed for general model fitting to those for specific models. *WinBUGS*, its open-source incarnation *OpenBUGS*, and the 'BUGS' clone Just Another Gibbs Sampler (*JAGS*) are among the most widely used programs for general model fitting *[48][57]*. These three provide similar programming syntaxes with which users can specify statistical models by simply stating relationships between data, parameters, and statistical distributions. Once a model is specified, the programs automatically formulate an MCMC sampling scheme to simulate parameter values from their posterior distribution. All aforementioned tasks can be accomplished with minimal programming and without specific knowledge of MCMC methodology. Users who are adept at both and so inclined can write software modules to add new distributions and samplers to *OpenBUGS* and *JAGS [73][75]*. *Stan* is a newer and similar-in-scope program worth noting for its accessible syntax and automatically tuned Hamiltonian Monte Carlo sampling scheme *[78]*. *PyMC* is a Python-based program that allows all modelling tasks to be accomplished in its native language, and gives users more hands-on access to model and sampling scheme specifications *[56]*. Programs like *GRIMS [54]* and *LaplacesDemon [79]* represent another class of programs that fit general models. In their approaches, users work with the functional forms of (unnormalized) probability densities directly, rather a domain specific modelling language (DSL), for model specification. Examples of programs for specific models can be found in the **R** catalogue of packages. For instance, the *arm* package provides Bayesian inference for generalized linear, ordered logistic or probit, and mixed-effects regression models *[30]*, *MCMCpack* fits a wide range of models commonly encountered in the social and behavioral sciences *[49]*, and many others that are more focused on specific classes of models can be found in the "Bayesian Inference" task view on the Comprehensive **R** Archive Network *[55]*.

2.1.2 The Mamba Package

Mamba [68] is a **julia** *[3]* package designed for general Bayesian model fitting via MCMC. Like *OpenBUGS* and *JAGS*, it supports a wide range of model and distributional specifications, and provides a syntax for model specification. Unlike those two, and like *PyMC*, *Mamba* provides a unified environment in which all interactions with the software are made through a single, interpreted language. Any **julia** operator, function, type, or package can be used for model specification; and custom distributions and samplers can be written in **julia** to extend the package. Conversely, interactions with and extensions to *OpenBUGS* and *JAGS* can involve three different programming environments — **R** wrappers used to call the programs, their DSLs, and the underlying implementations in Component Pascal and C++. Advantages of a unified environment include more flexible model specification; tighter integration with supplied functions for convergence diagnostics and posterior inference; and faster development, testing, and debugging of extensions. Advantages of the *BUGS* DSLs include more concise model specification and facilitation of automated sampling scheme formulation. Indeed, sampling schemes must be selected manually in the initial release of *Mamba*. Nevertheless, *Mamba* holds other distinct advantages over existing offerings. In particular, it provides arbitrary blocking of model parameters and designation of block-specific samplers; samplers that can be used with the included simulation engine or apart from it; and command-line access to all package functionality, including its simulation API. Likewise, advantages of the **julia** language include its familiar syntax, focus on technical computing, and benchmarks showing it to be one or more orders of magnitude faster than **R** and **Python** *[2]*. Finally, the intended audience for *Mamba* includes individuals interested in programming in **julia**; who wish to have low-level access to model design and implementation; and, in some cases, are able to derive full conditional distributions of model parameters (up to normalizing constants).

Mamba allows for the implementation of an MCMC sampling scheme to simulate draws for a set of Bayesian model parameters $(\theta_1, \ldots, \theta_p)$ from their joint posterior distribution. The package supports the general Gibbs *[26][31]* scheme outlined in the algorithm below. In its implementation with the package, the user may specify blocks $\{\Theta_j\}_{j=1}^B$ of parameters and corresponding functions $\{f_j\}_{j=1}^B$ to sample each Θ_j from its full conditional distribution $p(\Theta_j|\Theta \setminus \Theta_j)$. Simulation performance (efficiency and runtime) can be affected greatly by the choice of blocking scheme and sampling functions. For some models, an optimal choice may not be obvious, and different choices may need to be tried to find one that gives a desired level of performance. This can be a time-consuming process. The *Mamba* package provides a set of **julia** types and method functions to facilitate the specification of different schemes and functions. Supported sampling functions include those provided by the package, user-defined functions, and functions from other packages; thus providing great flexibility with respect to sampling methods. Furthermore, a sampling engine is provided to save the user from having to implement tasks common to all MCMC simulators. Therefore, time and energy can be focused on implementation aspects that most directly affect performance.

A summary of the steps involved in using the package to perform MCMC simulation for a Bayesian model is given below.

1. Decide on names to use for **julia** objects that will represent the model data structures and parameters $(\theta_1, \ldots, \theta_p)$. For instance, the *Tutorial* section describes a linear regression example in which predictor $\boldsymbol{x}$ and response $\boldsymbol{y}$ are represented by objects `x` and `y`, and regression parameters β_0, β_1, and σ^2 by objects `b0`, `b1`, and `s2`.
2. Create a dictionary to store all structures considered to be fixed in the simulation; e.g., the `line` dictionary in the regression example.
3. Specify the model using the constructors described in the *MCMC Types* section, to create the following:
   (a) A `Stochastic` object for each model term that has a distributional specification. This includes parameters and data, such as the regression parameters `b0`, `b1`, and `s2` that have prior distributions and `y` that has a likelihood specification.
   (b) A vector of `Sampler` objects containing supplied, user-defined, or external functions $\{f_j\}_{j=1}^B$ for sampling each parameter block Θ_j.
   (c) A `Model` object from the resulting stochastic nodes and sampler vector.
4. Simulate parameter values with the *`mcmc()`* function.

Input : Model parameters $\Theta = \{\theta_1, \ldots, \theta_p\}$.
Blocking $\{\Theta_j\}_{j=1}^{B}$ such that $\bigcup_{j=1}^{B} \Theta_j = \Theta$ and $\bigcap_{j=1}^{B} \Theta_j = \varnothing$.
Functions $\{f_j\}_{j=1}^{B}$ such that f_j samples from $p(\Theta_j | \Theta \setminus \Theta_j)$.
Result: Simulated values $\{\Theta^i\}_{i=1}^{N}$ from the joint distribution of Θ.
begin
  $\Theta \leftarrow$ Initialize();
  for $i = 1$ **to** N **do**
    for $j = 1$ **to** B **do**
      $\Theta_j \leftarrow f_j(\Theta)$;
    end
    $\Theta^i \leftarrow \Theta$;
  end
end

Fig. 2.1: *Mamba* Gibbs sampling scheme

5. Use the MCMC output to check convergence and perform posterior inference.

2.2 Tutorial

The complete source code for the examples contained in this tutorial can be obtained `here`.

2.2.1 Bayesian Linear Regression Model

In the sections that follow, the Bayesian simple linear regression example from the *BUGS 0.5* manual *[69]* is used to illustrate features of the package. The example describes a regression relationship between observations $\boldsymbol{x} = (1, 2, 3, 4, 5)^\top$ and $\boldsymbol{y} = (1, 3, 3, 3, 5)^\top$ that can be expressed as

$$\begin{aligned} \boldsymbol{y} &\sim \text{Normal}(\boldsymbol{\mu}, \sigma^2 \boldsymbol{I}) \\ \boldsymbol{\mu} &= \boldsymbol{X}\boldsymbol{\beta} \end{aligned}$$

with prior distribution specifications

$$\begin{aligned} \boldsymbol{\beta} &\sim \text{Normal}\left(\boldsymbol{\mu}_\pi = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \boldsymbol{\Sigma}_\pi = \begin{bmatrix} 1000 & 0 \\ 0 & 1000 \end{bmatrix}\right) \\ \sigma^2 &\sim \text{InverseGamma}(\alpha_\pi = 0.001, \beta_\pi = 0.001), \end{aligned}$$

where $\boldsymbol{\beta} = (\beta_0, \beta_1)^\top$, and $\boldsymbol{X}$ is a design matrix with an intercept vector of ones in the first column and $\boldsymbol{x}$ in the second. Primary interest lies in making inference about the β_0, β_1, and σ^2 parameters, based on their posterior distribution. A computational consideration in this example is that inference cannot be obtain from the joint posterior directly because

of its nonstandard form, derived below up to a constant of proportionality.

$$\begin{aligned} p(\boldsymbol{\beta}, \sigma^2|\boldsymbol{y}) &\propto p(\boldsymbol{y}|\boldsymbol{\beta}, \sigma^2)p(\boldsymbol{\beta})p(\sigma^2) \\ &\propto \left(\sigma^2\right)^{-n/2} \exp\left\{-\frac{1}{2\sigma^2}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})^\top(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})\right\} \\ &\quad \times \exp\left\{-\frac{1}{2}(\boldsymbol{\beta}-\boldsymbol{\mu}_\pi)^\top\boldsymbol{\Sigma}_\pi^{-1}(\boldsymbol{\beta}-\boldsymbol{\mu}_\pi)\right\}\left(\sigma^2\right)^{-\alpha_\pi-1}\exp\left\{-\beta_\pi/\sigma^2\right\} \end{aligned}$$

A common alternative is to make approximate inference based on parameter values simulated from the posterior with MCMC methods.

2.2.2 Model Specification

Nodes

In the *Mamba* package, terms that appear in the Bayesian model specification are referred to as *nodes*. Nodes are classified as one of three types:

- **Stochastic nodes** are any model terms that have likelihood or prior distributional specifications. In the regression example, $\boldsymbol{y}$, $\boldsymbol{\beta}$, and σ^2 are stochastic nodes.
- **Logical nodes** are terms, like $\boldsymbol{\mu}$, that are deterministic functions of other nodes.
- **Input nodes** are any remaining model terms ($\boldsymbol{X}$) and are considered to be fixed quantities in the analysis.

Note that the $\boldsymbol{y}$ node has both a distributional specification and is a fixed quantity. It is designated a stochastic node in *Mamba* because of its distributional specification. This allows for the possibility of model terms with distributional specifications that are a mix of observed and unobserved elements, as in the case of missing values in response vectors.

For model implementation, all nodes are stored in and accessible from a **julia** dictionary structure called `model` with the names (keys) of nodes being symbols. The regression nodes will be named `:y`, `:beta`, `:s2`, `:mu`, and `:xmat` to correspond to the stochastic, logical, and input nodes mentioned above. Implementation begins by instantiating the stochastic and logical nodes using the *Mamba*–supplied constructors `Stochastic` and `Logical`. Stochastic and logical nodes for a model are defined with a call to the `Model` constructor. The model constructor formally defines and assigns names to the nodes. User-specified names are given on the left-hand sides of the arguments to which `Logical` and `Stochastic` nodes are passed.

```
using Mamba

## Model Specification

model = Model(

  y = Stochastic(1,
    quote
      mu = model[:mu]
      s2 = model[:s2]
      MvNormal(mu, sqrt(s2))
    end,
    false
  ),

  mu = Logical(1,
    :(model[:xmat] * model[:beta]),
    false
  ),
```

```
  beta = Stochastic(1,
    :(MvNormal(2, sqrt(1000)))
  ),

  s2 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)
```

A single integer value for the first `Stochastic` constructor argument indicates that the node is an array of the specified dimension. Absence of an integer value implies a scalar node. The next argument is a quoted expression that can contain any valid **julia** code. Expressions for stochastic nodes must return a distribution object from or compatible with the *Distributions* package *[1]*. Such objects represent the nodes' distributional specifications. An optional boolean argument after the expression can be specified to indicate whether values of the node should be monitored (saved) during MCMC simulations (default: `true`).

Stochastic expressions must return a single distribution object that can accommodate the dimensionality of the node, or return an array of (univariate) distribution objects of the same dimension as the node. Examples of alternative, but equivalent, prior distribution specifications for the `beta` node are shown below.

```
# Case 1: Multivariate Normal with independence covariance matrix
beta = Stochastic(1,
  :(MvNormal(2, sqrt(1000)))
)

# Case 2: One common univariate Normal
beta = Stochastic(1,
  :(Normal(0, sqrt(1000)))
)

# Case 3: Array of univariate Normals
beta = Stochastic(1,
  :(Distribution[Normal(0, sqrt(1000)), Normal(0, sqrt(1000))])
)

# Case 4: Array of univariate Normals
beta = Stochastic(1,
  :(Distribution[Normal(0, sqrt(1000)) for i in 1:2])
)
```

Case 1 is one of the multivariate normal distributions available in the *Distributions* package, and the specification used in the example model implementation. In Case 2, a single univariate normal distribution is specified to imply independent priors of the same type for all elements of `beta`. Cases 3 and 4 explicitly specify a univariate prior for each element of `beta` and allow for the possibility of differences among the priors. Both return arrays of Distribution objects, with the last case automating the specification of array elements.

In summary, `y` and `beta` are stochastic vectors, `s2` is a stochastic scalar, and `mu` is a logical vector. We note that the model could have been implemented without `mu`. It is included here primarily to illustrate use of a logical node. Finally, note that nodes `y` and `mu` are not being monitored.

Sampling Schemes

The package provides a flexible system for the specification of schemes to sample stochastic nodes. Arbitrary blocking of nodes and designation of block-specific samplers is supported. Furthermore, block-updating of nodes can be performed with samplers provided, defined by the user, or available from other packages. Schemes are specified as vectors of `Sampler` objects. Constructors are provided for several popular sampling algorithms, including adaptive

Metropolis, No-U-Turn (NUTS), and slice sampling. Example schemes are shown below. In the first one, NUTS is used for the sampling of `beta` and slice for `s2`. The two nodes are block together in the second scheme and sampled jointly with NUTS.

```
## Hybrid No-U-Turn and Slice Sampling Scheme
scheme1 = [NUTS([:beta]),
           Slice([:s2], [3.0])]

## No-U-Turn Sampling Scheme
scheme2 = [NUTS([:beta, :s2])]
```

Additionally, users are free to create their own samplers with the generic `Sampler` constructor. This is particularly useful in settings were full conditional distributions are of standard forms for some nodes and can be sampled from directly. Such is the case for the full conditional of $\boldsymbol{\beta}$ which can be written as

$$\begin{aligned} p(\boldsymbol{\beta}|\sigma^2, \boldsymbol{y}) &\propto p(\boldsymbol{y}|\boldsymbol{\beta}, \sigma^2)p(\boldsymbol{\beta}) \\ &\propto \exp\left\{-\frac{1}{2}(\boldsymbol{\beta}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{\beta}-\boldsymbol{\mu})\right\}, \end{aligned}$$

where $\boldsymbol{\Sigma} = \left(\frac{1}{\sigma^2}\boldsymbol{X}^\top\boldsymbol{X} + \boldsymbol{\Sigma}_\pi^{-1}\right)^{-1}$ and $\boldsymbol{\mu} = \boldsymbol{\Sigma}\left(\frac{1}{\sigma^2}\boldsymbol{X}^\top\boldsymbol{y} + \boldsymbol{\Sigma}_\pi^{-1}\boldsymbol{\mu}_\pi\right)$, and is recognizable as multivariate normal. Likewise,

$$\begin{aligned} p(\sigma^2|\boldsymbol{\beta}, \mathbf{y}) &\propto p(\boldsymbol{y}|\boldsymbol{\beta}, \sigma^2)p(\sigma^2) \\ &\propto \left(\sigma^2\right)^{-(n/2+\alpha_\pi)-1} \exp\left\{-\frac{1}{\sigma^2}\left(\frac{1}{2}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})^\top(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}) + \beta_\pi\right)\right\}, \end{aligned}$$

whose form is inverse gamma with $n/2+\alpha_\pi$ shape and $(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})^\top(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})/2+\beta_\pi$ scale parameters. A user-defined sampling scheme to generate draws from these full conditionals is constructed below.

```
## User-Defined Samplers

Gibbs_beta = Sampler([:beta],
  quote
    beta = model[:beta]
    s2 = model[:s2]
    xmat = model[:xmat]
    y = model[:y]
    beta_mean = mean(beta.distr)
    beta_invcov = invcov(beta.distr)
    Sigma = inv(xmat' * xmat / s2 + beta_invcov)
    mu = Sigma * (xmat' * y / s2 + beta_invcov * beta_mean)
    rand(MvNormal(mu, Sigma))
  end
)

Gibbs_s2 = Sampler([:s2],
  quote
    mu = model[:mu]
    s2 = model[:s2]
    y = model[:y]
    a = length(y) / 2.0 + shape(s2.distr)
    b = sumabs2(y - mu) / 2.0 + scale(s2.distr)
    rand(InverseGamma(a, b))
  end
)

## User-Defined Sampling Scheme
scheme3 = [Gibbs_beta, Gibbs_s2]
```

In these samplers, the respective `MvNormal(2, sqrt(1000))` and `InverseGamma(0.001, 0.001)` priors on stochastic nodes `beta` and `s2` are accessed directly through the `distr` *fields*. Features of the *Distributions* objects returned by `beta.distr` and `s2.distr` can, in turn, be extracted with method functions defined in that package or through their own fields. For instance, `mean(beta.distr)` and `invcov(beta.distr)` apply method functions to extract the mean vector and inverse-covariance matrix of the `beta` prior. Whereas, `shape(s2.distr)` and `scale(s2.distr)` extract the shape and scale parameters from fields of the inverse-gamma prior. *Distributions* method functions can be found in that package's documentation; whereas, fields are found in the source code.

When possible to do so, direct sampling from full conditions is often preferred in practice because it tends to be more efficient than general-purpose algorithms. Schemes that mix the two approaches can be used if full conditionals are available for some model parameters but not for others. Once a sampling scheme is formulated in *Mamba*, it can be assigned to an existing model with a call to the `setsamplers!` function.

```
## Sampling Scheme Assignment
setsamplers!(model, scheme1)
```

Alternatively, a predefined scheme can be passed in to the `Model` constructor at the time of model implementation as the value to its `samplers` argument.

The Model Expression Macro

`@modelexpr`(*args...*)

A macro to automate the declaration of `model` variables in expression supplied to `MCMCStocastic`, `Logical`, and `Sampler` constructors.

Arguments

- `args...` : sequence of one or more arguments, such that the last argument is a single expression or code block, and the previous ones are variable names of model nodes upon which the expression depends.

Value

An expression block of nodal variable declarations followed by the specified expression.

Example

Calls to `@modelexpr` can be used to shorten the expressions specified in the previous `Model` specification and calls to `Sampler`, as shown below. In essence, this macro call automates the tasks of declaring variables `beta`, `s2`, `xmat`, and `y`; and returns the same quoted expressions as before but with less coding required.

```
model = Model(

  y = Stochastic(1,
    @modelexpr(mu, s2,
      MvNormal(mu, sqrt(s2))
    ),
    false
  ),

  mu = Logical(1,
    @modelexpr(xmat, beta,
      xmat * beta
    ),
    false
  ),

  beta = Stochastic(1,
    :(MvNormal(2, sqrt(1000)))
```

```
    ),

    s2 = Stochastic(
      :(InverseGamma(0.001, 0.001))
    )

  )

  Gibbs_beta = Sampler([:beta],
    @modelexpr(beta, s2, xmat, y,
      begin
        beta_mean = mean(beta.distr)
        beta_invcov = invcov(beta.distr)
        Sigma = inv(xmat' * xmat / s2 + beta_invcov)
        mu = Sigma * (xmat' * y / s2 + beta_invcov * beta_mean)
        rand(MvNormal(mu, Sigma))
      end
    )
  )

  Gibbs_s2 = Sampler([:s2],
    @modelexpr(mu, s2, y,
      begin
        a = length(y) / 2.0 + shape(s2.distr)
        b = sumabs2(y - mu) / 2.0 + scale(s2.distr)
        rand(InverseGamma(a, b))
      end
    )
  )
```

2.2.3 Directed Acyclic Graphs

One of the internal structures created by `Model` is a graph representation of the model nodes and their associations. Graphs are managed internally with the *Graphs* package *[77]*. Like *OpenBUGS*, *JAGS*, and other *BUGS* clones, *Mamba* fits models whose nodes form a directed acyclic graph (DAG). A *DAG* is a graph in which nodes are connected by directed edges and no node has a path that loops back to itself. With respect to statistical models, directed edges point from parent nodes to the child nodes that depend on them. Thus, a child node is independent of all others, given its parents.

The DAG representation of a `Model` can be printed out at the command-line or saved to an external file in a format that can be displayed with the Graphviz software.

```
## Graph Representation of Nodes

>>> draw(model)

digraph MambaModel {
  "mu" [shape="diamond", fillcolor="gray85", style="filled"];
    "mu" -> "y";
  "xmat" [shape="box", fillcolor="gray85", style="filled"];
    "xmat" -> "mu";
  "beta" [shape="ellipse"];
    "beta" -> "mu";
  "s2" [shape="ellipse"];
    "s2" -> "y";
  "y" [shape="ellipse", fillcolor="gray85", style="filled"];
}
```

```
>>> draw(model, filename="lineDAG.dot")
```

Either the printed or saved output can be passed manually to the Graphviz software to plot a visual representation of the model. If **julia** is being used with a front-end that supports in-line graphics, like *IJulia [43]*, and the *GraphViz* **julia** package *[22]* is installed, then the following code will plot the graph automatically.

```
using GraphViz

>>> display(Graph(graph2dot(model)))
```

A generated plot of the regression model graph is show in the figure below.

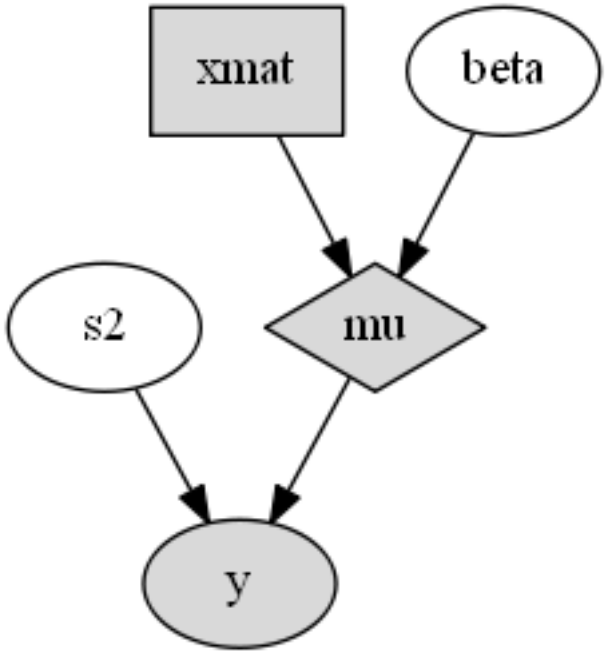

Fig. 2.2: Directed acyclic graph representation of the example regression model nodes.

Stochastic, logical, and input nodes are represented by ellipses, diamonds, and rectangles, respectively. Gray-colored nodes are ones designated as unmonitored in MCMC simulations. The DAG not only allows the user to visually check that the model specification is the intended one, but is also used internally to check that nodal relationships are acyclic.

2.2.4 MCMC Simulation

Data

For the example, observations $(\boldsymbol{x}, \boldsymbol{y})$ are stored in a **julia** dictionary defined in the code block below. Included are predictor and response vectors `:x` and `:y` as well as a design matrix `:xmat` corresponding to the model matrix $\boldsymbol{X}$.

```
## Data
line = (Symbol => Any)[
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
]
line[:xmat] = [ones(5) line[:x]]
```

Initial Values

A **julia** vector of dictionaries containing initial values for all stochastic nodes must be created. The dictionary keys should match the node names, and their values should be vectors whose elements are the same type of structures as the nodes. Three sets of initial values for the regression example are created in with the following code.

```
## Initial Values
inits = [[:y => line[:y],
          :beta => rand(Normal(0, 1), 2),
          :s2 => rand(Gamma(1, 1))]
         for i in 1:3]
```

Initial values for `y` are those in the observed response vector. Likewise, the node is not updated in the sampling schemes defined earlier and thus retains its initial values throughout MCMC simulations. Initial values are generated for `beta` from a normal distribution and for `s2` from a gamma distribution.

MCMC Engine

MCMC simulation of draws from the posterior distribution of a declared set of model nodes and sampling scheme is performed with the *`mcmc()`* function. As shown below, the first three arguments are a `Model` object, a dictionary of values for input nodes, and a dictionary vector of initial values. The number of draws to generate in each simulation run is given as the fourth argument. The remaining arguments are named such that `burnin` is the number of initial values to discard to allow for convergence; `thin` defines the interval between draws to be retained in the output; and `chains` specifies the number of times to run the simulator. The simulation of multiple chains will be executed in parallel automatically if **julia** is running in multiprocessor mode on a multiprocessor system. Multiprocessor mode can be started with the command line argument `julia -p n`, where `n` is the number of available processors. See the **julia** documentation on parallel computing for details.

```
## MCMC Simulations

setsamplers!(model, scheme1)
sim1 = mcmc(model, line, inits, 10000, burnin=250, thin=2, chains=3)

setsamplers!(model, scheme2)
sim2 = mcmc(model, line, inits, 10000, burnin=250, thin=2, chains=3)

setsamplers!(model, scheme3)
sim3 = mcmc(model, line, inits, 10000, burnin=250, thin=2, chains=3)
```

Results are retuned as `Chains` objects on which methods for posterior inference are defined.

2.2.5 Posterior Inference

Convergence Diagnostics

Checks of MCMC output should be performed to assess convergence of simulated draws to the posterior distribution. Checks can be performed with a variety of methods. The diagnostic of Gelman, Rubin, and Brooks *[29][10]* is one method for assessing convergence of posterior mean estimates. Values of the diagnostic's *potential scale reduction factor (PSRF)* that are close to one suggest convergence. As a rule-of-thumb, convergence is rejected if the 97.5 percentile of a PSRF is greater than 1.2.

```
>>> gelmandiag(sim1, mpsrf=true, transform=true) |> showall

Gelman, Rubin, and Brooks Diagnostic:
               PSRF 97.5%
```

```
          s2 1.005 1.010
     beta[1] 1.006 1.006
     beta[2] 1.006 1.006
Multivariate 1.004   NaN
```

The diagnostic of Geweke *[33]* tests for non-convergence of posterior mean estimates. It provides chain-specific test p-values. Convergence is rejected for significant p-values, like those obtained for `s2`.

```
>>> gewekediag(sim1) |> showall

Geweke Diagnostic:
First Window Fraction = 0.1
Second Window Fraction = 0.5

         Z-score p-value
     s2  -2.321  0.0203
beta[1]   0.381  0.7035
beta[2]  -0.273  0.7851

         Z-score p-value
     s2   0.079  0.9370
beta[1]   0.700  0.4839
beta[2]  -0.651  0.5150

         Z-score p-value
     s2  -2.101  0.0356
beta[1]   0.932  0.3515
beta[2]  -0.685  0.4934
```

The diagnostic of Heidelberger and Welch *[41]* tests for non-convergence (non-stationarity) and whether ratios of estimation interval halfwidths to means are within a target ratio. Stationarity is rejected (0) for significant test p-values. Halfwidth tests are rejected (0) if observed ratios are greater than the target, as is the case for `s2` and `beta[1]`.

```
>>> heideldiag(sim1) |> showall

Heidelberger and Welch Diagnostic:
Target Halfwidth Ratio = 0.1
Alpha = 0.05

         Burn-in Stationarity p-value    Mean     Halfwidth  Test
     s2      251            1  0.2407 1.54572606 0.559718246    0
beta[1]      251            1  0.5330 0.53489109 0.065861054    0
beta[2]      251            1  0.5058 0.81767861 0.018822591    1

         Burn-in Stationarity p-value    Mean     Halfwidth  Test
     s2      251            1  0.8672 1.26092566  0.27625544    0
beta[1]      251            1  0.8806 0.55820771  0.07672180    0
beta[2]      251            1  0.9330 0.81398205  0.02254785    1

         Burn-in Stationarity p-value    Mean     Halfwidth  Test
     s2      251            1  0.8145 1.12827403 0.196179015    0
beta[1]      251            1  0.4017 0.55923590 0.056504387    0
beta[2]      251            1  0.4216 0.81202489 0.016274601    1
```

The diagnostic of Raftery and Lewis *[60][61]* is used to determine the number of iterations required to estimate a specified quantile within a desired degree of accuracy. For example, below are required total numbers of iterations, numbers to discard as burn-in sequences, and thinning intervals for estimating 0.025 quantiles so that their estimated cumulative probabilities are within 0.025 ± 0.005 with probability 0.95.

```
>>> rafterydiag(sim1) |> showall

Raftery and Lewis Diagnostic:
Quantile (q) = 0.025
Accuracy (r) = 0.005
Probability (s) = 0.95

        Thinning Burn-in    Total   Nmin Dependence Factor
     s2        2     255 8.1370x103 3746         2.1721837
beta[1]        4     283 3.7515x104 3746        10.0146823
beta[2]        2     267 1.8257x104 3746         4.8737320

        Thinning Burn-in    Total   Nmin Dependence Factor
     s2        2     257 8.2730x103 3746          2.208489
beta[1]        4     279 3.0899x104 3746          8.248532
beta[2]        2     267 1.7209x104 3746          4.593967

        Thinning Burn-in    Total   Nmin Dependence Factor
     s2        2     253 7.7470x103 3746         2.0680726
beta[1]        2     273 2.4635x104 3746         6.5763481
beta[2]        4     271 2.4375x104 3746         6.5069407
```

More information on the diagnostic functions can be found in the *Convergence Diagnostics* section.

Posterior Summaries

Once convergence has been assessed, sample statistics may be computed on the MCMC output to estimate features of the posterior distribution. The *StatsBase* package *[46]* is utilized in the calculation of many posterior estimates. Some of the available posterior summaries are illustrated in the code block below.

```
## Summary Statistics
>>> describe(sim1)

Iterations = 252:10000
Thinning interval = 2
Chains = 1,2,3
Samples per chain = 4875

Empirical Posterior Estimates:
           Mean        SD       Naive SE      MCSE        ESS
     s2 1.31164192 2.11700926 0.017505512 0.084760964  623.81203
beta[1] 0.55077823 1.22684809 0.010144785 0.021035908 3401.41041
beta[2] 0.81456185 0.36999413 0.003059475 0.005983602 3823.52816

Quantiles:
           2.5%       25.0%       50.0%       75.0%     97.5%
     s2  0.1689205  0.37858233 0.642708783 1.29554584 6.971778
beta[1] -2.0786622 -0.00365545 0.598234075 1.15533747 2.927107
beta[2]  0.0958658  0.63363928 0.805067956 0.98038846 1.607176

## Highest Posterior Density Intervals
>>> hpd(sim1)

         95% Lower 95% Upper
     s2  0.0746761 4.6285413
beta[1] -1.9919011 3.0059020
beta[2]  0.1162965 1.6180369
```

```
## Cross-Correlations
>>> cor(sim1)

             s2       beta[1]    beta[2]
     s2  1.0000000 -0.1035043  0.0970532
beta[1] -0.1035043  1.0000000 -0.9120741
beta[2]  0.0970532 -0.9120741  1.0000000

## Lag-Autocorrelations
>>> autocor(sim1)

           Lag 2     Lag 10      Lag 20      Lag 100
     s2  0.9305324  0.7147161  0.50020397 -0.0425768760
beta[1]  0.2841654  0.0144642  0.01890702  0.0169609340
beta[2]  0.2419890  0.0556535  0.03274191  0.0147079793

           Lag 2     Lag 10      Lag 20      Lag 100
     s2  0.8328732 0.46783691  0.20728343  -0.015633535
beta[1]  0.3692985 0.04823331 -0.00047505  -0.027301651
beta[2]  0.3336704 0.01756540  0.02817078  -0.029797132

           Lag 2     Lag 10      Lag 20      Lag 100
     s2 0.79994494  0.3954458  0.17855388    0.03735549
beta[1] 0.29036852  0.0151255  0.01251444   -0.00971026
beta[2] 0.23588485  0.0097962  0.01725959   -0.01162341

## State Space Change Rate (per Iteration)
>>> changerate(sim1)

             Change Rate
          s2       1.000
     beta[1]       0.782
     beta[2]       0.782
Multivariate       1.000

## Deviance Information Criterion
>>> dic(sim1)

      DIC    Effective Parameters
pD 13.811678             1.022158
pV 24.423410             6.328024
```

Output Subsetting

Additionally, sampler output can be subsetted to perform posterior inference on select iterations, parameters, and chains.

```
## Subset Sampler Output
>>> sim = sim1[1000:5000, ["beta[1]", "beta[2]"], :]
>>> describe(sim)

Iterations = 1000:5000
Thinning interval = 2
Chains = 1,2,3
Samples per chain = 2001
```

```
Empirical Posterior Estimates:
           Mean        SD      Naive SE      MCSE       ESS
beta[1] 0.54294803 1.2535281 0.016178934 0.028153921 1982.3958
beta[2] 0.81654896 0.3761126 0.004854379 0.007729269 2367.8756

Quantiles:
           2.5%      25.0%      50.0%      75.0%     97.5%
beta[1] -2.078668 -0.0116601 0.59713701 1.13635020 2.9206261
beta[2]  0.082051  0.6346812 0.80236351 0.98308268 1.6087557
```

Restarting the Sampler

Convergence diagnostics or posterior summaries may indicate that additional draws from the posterior are needed for inference. In such cases, the *mcmc()* function can be used to restart the sampler with previously generated output, as illustrated below.

```
## Restart the Sampler
>>> sim = mcmc(sim1, 5000)
>>> describe(sim)

Iterations = 252:15000
Thinning interval = 2
Chains = 1,2,3
Samples per chain = 7375

Empirical Posterior Estimates:
           Mean        SD      Naive SE      MCSE       ESS
     s2 1.29104854 2.0429169 0.0137343798 0.06552817  971.9531
beta[1] 0.56604188 1.2026624 0.0080854110 0.01584611 5760.2600
beta[2] 0.80989285 0.3628646 0.0024395117 0.00454669 6369.4100

Quantiles:
           2.5%       25.0%      50.0%       75.0%     97.5%
     s2  0.1696926 0.38320727 0.649505944 1.29165161 6.8578108
beta[1] -1.9796920 0.00363272 0.603057680 1.17107396 2.8925465
beta[2]  0.1070020 0.62822379 0.801115311 0.97744272 1.5792629
```

Plotting

Plotting of sampler output in *Mamba* is based on the *Gadfly* package *[44]*. Summary plots can be created and written to files using the `plot` and `draw` functions.

```
## Default summary plot (trace and density plots)
p = plot(sim1)

## Write plot to file
draw(p, filename="summaryplot.svg")
```

The `plot` function can also be used to make autocorrelation and running means plots. Legends can be added with the optional `legend` argument. Arrays of plots can be created and passed to the `draw` function. Use `nrow` and `ncol` to determine how many rows and columns of plots to include in each drawing.

```
## Autocorrelation and running mean plots
p = plot(sim1, [:autocor, :mean], legend=true)
draw(p, nrow=3, ncol=2, filename="autocormeanplot.svg")
```

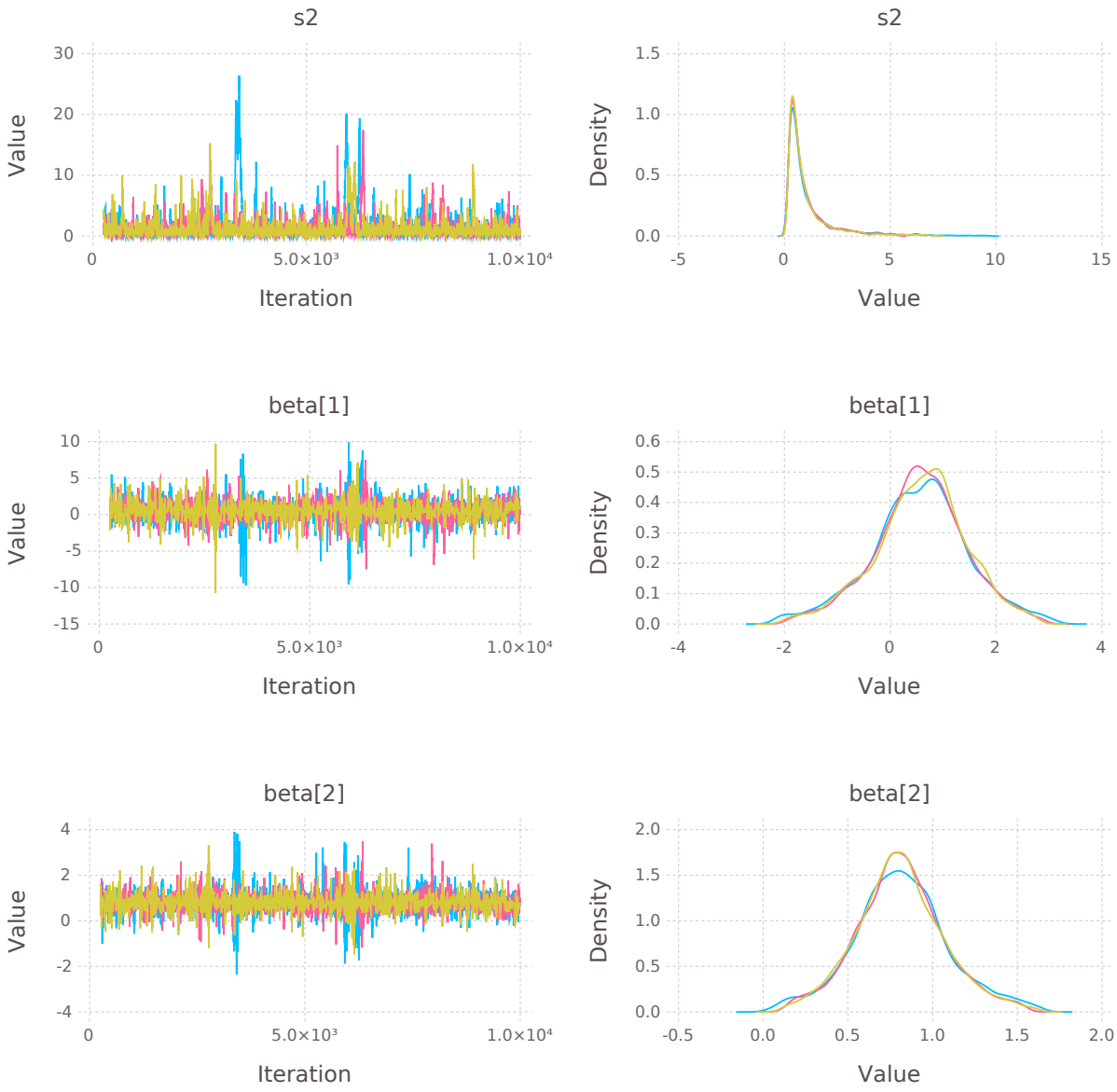

Fig. 2.3: Trace and density plots.

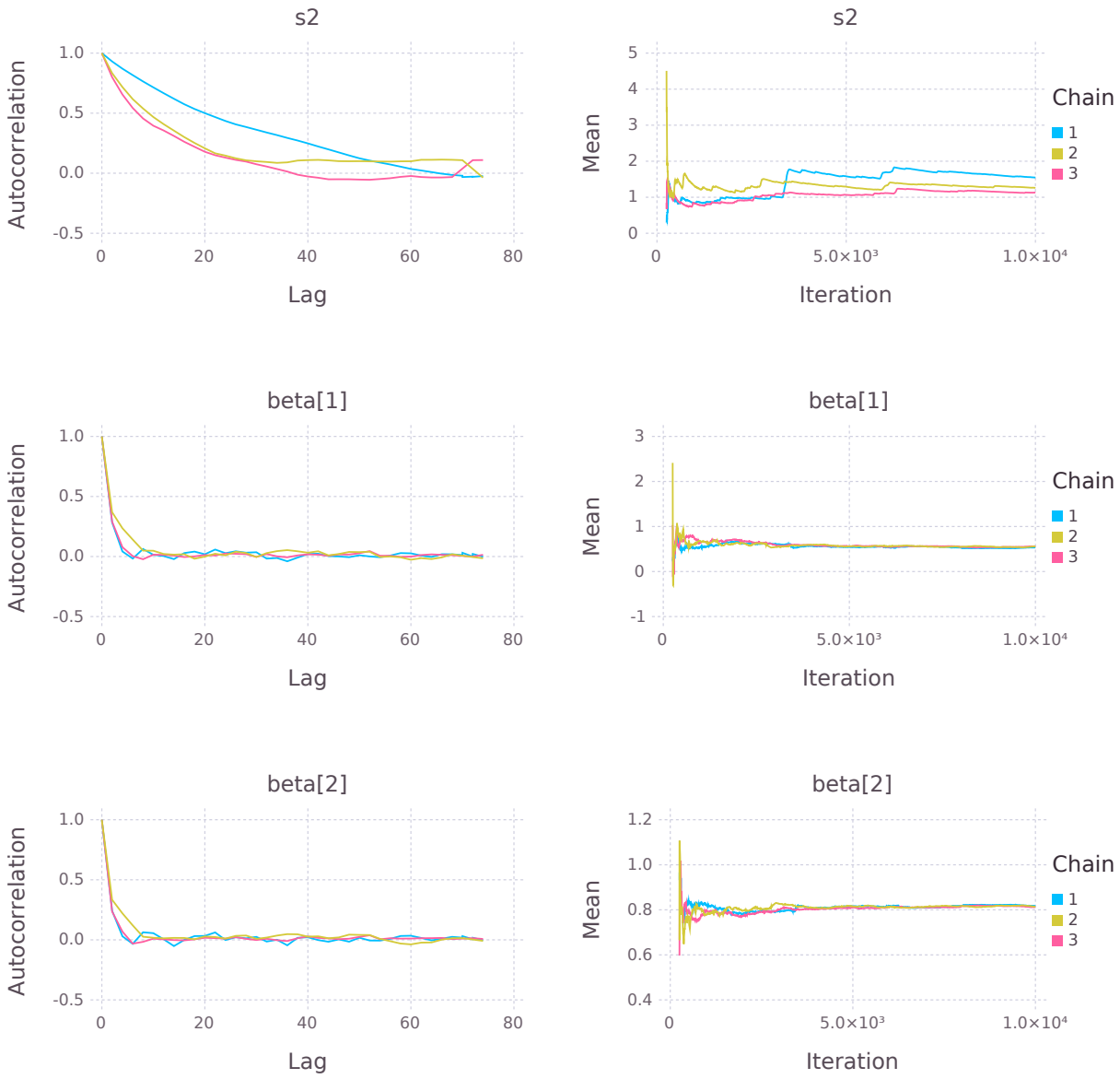

Fig. 2.4: Autocorrelation and running mean plots.

2.2.6 Computational Performance

Computing runtimes were recorded for different sampling algorithms applied to the regression example. Runs wer performed on a desktop computer with an Intel i5-2500 CPU @ 3.30GHz. Results are summarized in the table below. Note that these are only intended to measure the raw computing performance of the package, and do not account for different efficiencies in output generated by the sampling algorithms.

Table 2.1: Number of draws per second for select sampling algorithms in Mamba.

Adaptive Metropolis				Slice	
Within Gibbs	Multivariate	Gibbs	NUTS	Within Gibbs	Multivariate
16,700	11,100	27,300	2,600	13,600	17,600

2.2.7 Development and Testing

Command-line access is provided for all package functionality to aid in the development and testing of models. Examples of available functions are shown in the code block below. Documentation for these and other related functions can be found in the *MCMC Types* section.

```
## Development and Testing

setinputs!(model, line)             # Set input node values
setinits!(model, inits[1])          # Set initial values
setsamplers!(model, scheme1)        # Set sampling scheme

showall(model)                      # Show detailed node information

logpdf(model, 1)                    # Log-density sum for block 1
logpdf(model, 2)                    # Block 2
logpdf(model)                       # All blocks

simulate!(model, 1)                 # Simulate draws for block 1
simulate!(model, 2)                 # Block 2
simulate!(model)                    # All blocks
```

In this example, functions `setinputs!`, `setinits!`, and `setsampler!` allow the user to manually set the input node values, the initial values, and the sampling scheme form the `model` object, and would need to be called prior to `logpdf` and `simulate!`. Updated model objects should be returned when called; otherwise, a problem with the supplied values may exist. Method `showall` prints a detailed summary of all model nodes, their values, and attributes; `logpdf` sums the log-densities over nodes associated with a specified sampling block (second argument); and `simulate!` generates an MCMC draw for the nodes. Non-numeric results may indicate problems with distributional specifications in the second case or with sampling functions in the last case. The block arguments are optional; and, if left unspecified, will cause the corresponding functions to be applied over all sampling blocks. This allows testing of some or all of the samplers.

2.3 Variate Types

2.3.1 Variate

`Variate` is an abstract type that serves as the basis for several concrete types in the *Mamba* package. Conceptually, it represents a data structure that stores numeric values sampled from a target distribution. As an abstract type, `Variate` cannot be instantiated and cannot have fields. It can, however, have method functions, which descendant subtypes will inherit. Such inheritance allows one to endow a core set of functionality to all subtypes by simply

defining the functionality once on the abstract type (see julia Types). Accordingly, a core set of functionality is defined for the `Variate` type through the field and method functions summarized below. Although the (abstract) type does not have fields, its method functions assume that all subtypes will be declared with the `value` field shown.

Declaration

```
abstract Variate{T<:Union(VariateType, Array{VariateType})}
```

Aliases

```
typealias VariateType Float64

typealias UniVariate Variate{VariateType}
typealias MultiVariate{N} Variate{Array{VariateType,N}}
typealias VectorVariate MultiVariate{1}
typealias MatrixVariate MultiVariate{2}
```

Field

- `value::T` : a scalar or array of `VariateType` values that represent samples from a target distribution.

Methods

Method functions supported on all `Variate` types are summarized in the following sections; and, unless otherwise specified, are detailed in The Julia Standard Library documentation.

Array Functions

cummin	cumsum	maximum	prod
cummax	cumsum_kbn	minimum	sum
cumprod	diff	norm	sum_kbn

Collections

endof	size	show
length	getindex	showcompact
ndims	setindex!	

Distributions

The univariate, multivariate, and matrix distributions described in the *Distributions* Section are supported.

Linear Algebra

dot

Mathematical Operators and Elementary Functions

The basic numerical Mathematical Operators and Elementary Functions of the **julia** language are supported, and the ones below added.

Function	Description
`logit(x)`	log-odds
`invlogit(x)`	inverse log-odds

Statistics

```
cor       median      var
cov       std         varm
mean      stdm
```

2.3.2 Subtypes

Subtypes of `Variate` include the *Dependent*, *Logical*, and *Stochastic* types, as well as the those defined for supplied *Sampling Functions*.

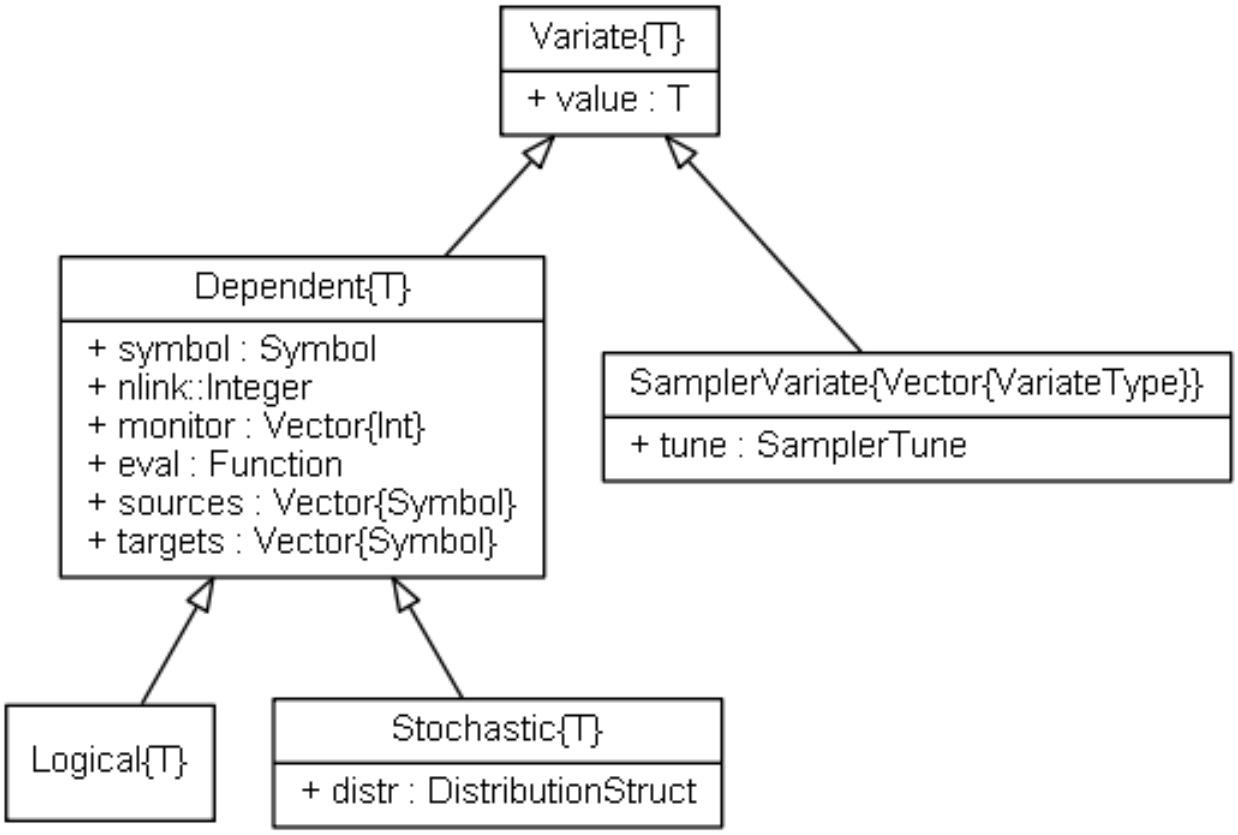

Fig. 2.5: UML relational diagram of `Variate` types and their fields.

2.4 MCMC Types

The *MCMC* types and their relationships are depicted below with a Unified Modelling Language (UML) diagram. In the diagram, types are represented with boxes that display their respective names in the top-most panels, and fields in the second panels. By convention, plus signs denote fields that are publicly accessible, which is always the case for these structures in **julia**. Hollow triangle arrows point to types that the originator extends. Solid diamond arrows indicate that a number of instances of the type being pointed to are contained in the originator. The undirected line between `Sampler` and `Stochastic` represents a bi-directional association. Numbers on the graph indicate that there is one (1), zero or more (0..*), or one or more (1..*) instances of a type at the corresponding end of a relationship.

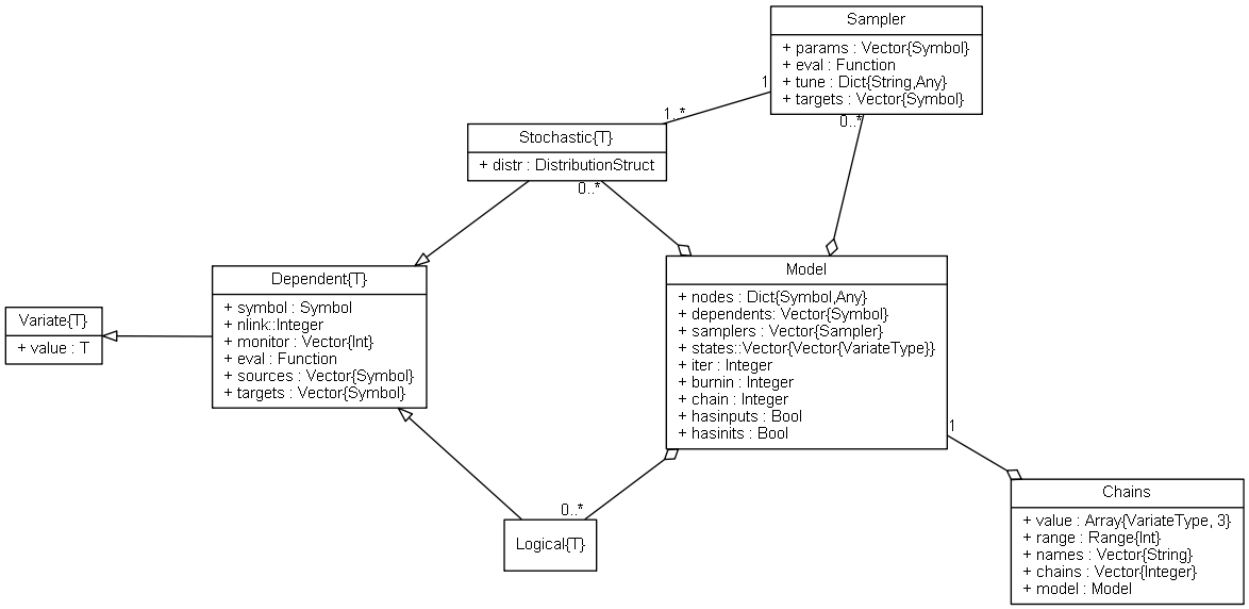

Fig. 2.6: UML relational diagram of *MCMC* types and their fields.

The relationships are as follows. Type `Model` contains a dictionary field (`Dict{Symbol,Any}`) of model nodes and a field (`Vector{Sampler}`) of one or more sampling functions. Nodes can be one of three types:

- **Stochastic nodes** (`Stochastic`) are any model terms that have likelihood or prior distributional specifications.
- **Logical nodes** (`Logical`) are terms that are deterministic functions of other nodes.
- **Input nodes** (not shown) are any other model terms and data types that are considered to be fixed quantities in the analysis.

`Stochastic` and `Logical` are inherited from the base *Variate* type and can be used with operators and in functions defined for that type. The sampling functions in `Model` each correspond to a block of one or more model parameters (stochastic nodes) to be sampled from a target distribution (e.g. full conditional) during the simulation. Finally, `Chains` stores simulation output for a given model. Detailed information about each type is provided in the subsequent sections.

2.4.1 Dependent

`Dependent` is an abstract type designed to store values and attributes of model nodes, including parameters $\theta_1, \ldots, \theta_p$ to be simulated via MCMC, functions of the parameters, and likelihood specifications on observed data.

It extends the base `Variate` type with method functions defined for the fields summarized below. Like the type it extends, values are stored in a `value` field and can be used with method functions that accept `Variate` type objects.

Since parameter values in the `Dependent` structure are stored as a scalar or array, objects of this type can be created for model parameters of corresponding dimensions, with the choice between the two being user and application-specific. At one end of the spectrum, a model might be formulated in terms of parameters that are all scalars, with a separate instances of `Dependent` for each one. At the other end, a formulation might be made in terms of a single parameter array, with one corresponding instance of `Dependent`. Whether to formulate parameters as scalars or arrays will depend on the application at hand. Array formulations should be considered for parameters and data that have multivariate distributions, or are to be used as such in numeric operations and functions. In other cases, scalar parametrizations may be preferable. Situations in which parameter arrays are often used include the specification of regression coefficients and random effects.

Declaration

```
abstract Dependent{T} <:  Variate{T}
```

Fields

- `value::T` : a scalar or array of `VariateType` values that represent samples from a target distribution.
- `symbol::Symbol` : an identifying symbol for the node.
- `nlink::Integer` : number of elements returned by the `link` method defined on the type. Generally, this will be the number of unique elements in the node. In most cases, `nlink` will be equal to `length(value)`. However, for some structures, like stochastic covariance matrices, `nlink` may be smaller.
- `monitor::Vector{Int}` : indices identifying elements of the `value` field to include in monitored MCMC sampler output.
- `eval::Function` : a function for updating the state of the node.
- `sources::Vector{Symbol}` : symbols of other nodes upon whom the values of this one depends.
- `targets::Vector{Symbol}` : symbols of `Dependent` nodes that depend on this one. Elements of `targets` are topologically sorted so that a given node in the vector is conditionally independent of subsequent nodes, given the previous ones.

Methods

`invlink`(*d::Dependent*, *x*, *transform::Bool=true*)

Apply a node-specific inverse-link transformation. In this method, the link transformation is defined to be the identity function. This method may be redefined for subtypes of `Dependent` to implement different link transformations.

Arguments

- `d` : a node on which a `link()` transformation method is defined.
- `x` : an object to which to apply the inverse-link transformation.
- `transform` : whether to transform `x` or assume an identity link.

Value

Returns the inverse-link-transformed version of `x`.

link (*d::Dependent*, *x*, *transform::Bool=true*)
Apply a node-specific link transformation. In this method, the link transformation is defined to be the identity function. This method function may be redefined for subtypes of `Dependent` to implement different link transformations.

Arguments

- `d` : a node on which a `link()` transformation method is defined.
- `x` : an object to which to apply the link transformation.
- `transform` : whether to transform `x` or assume an identity link.

Value

Returns the link-transformed version of `x`.

logpdf (*d::Dependent*, *transform::Bool=false*)
Evaluate the log-density function for a node. In this method, no density function is assumed for the node, and a constant value of 0 is returned. This method function may be redefined for subtypes of `Dependent` that have distributional specifications.

Arguments

- `d` : a node containing values at which to compute the log-density.
- `transform` : whether to evaluate the log-density on the link-transformed scale.

Value

The resulting numeric value of the log-density.

setmonitor! (*d::Dependent*, *monitor::Bool*)
setmonitor! (*d::Dependent*, *monitor::Vector{Int}*)
Specify node elements to be included in monitored MCMC sampler output.

Arguments

- `d` : a node whose elements contain sampled MCMC values.
- `monitor` : a boolean indicating whether all elements are monitored, or a vector of element-wise indices of elements to monitor.

Value

Returns `d` with its `monitor` field updated to reflect the specified monitoring.

show (*d::Dependent*)
Write a text representation of nodal values and attributes to the current output stream.

showall (*d::Dependent*)
Write a verbose text representation of nodal values and attributes to the current output stream.

2.4.2 Logical

Type `Logical` inherits the fields and method functions from the `Dependent` type, and adds the constructors and methods listed below. It is designed for nodes that are deterministic functions of model parameters and data. Stored in the field `eval` is an anonymous function defined as

```
function(model::Mamba.Model)
```

where `model` contains all model nodes. The function can contain any valid **julia** expression or code block written in terms of other nodes and data structures. It should return values with which to update the node in the same type as the `value` field of the node.

Declaration

```
type Logical{T} <:  Dependent{T}
```

Fields

- `value::T` : a scalar or array of `VariateType` values that represent samples from a target distribution.
- `symbol::Symbol` : an identifying symbol for the node.
- `nlink::Integer` : number of elements returned by the `link` method defined on the type.
- `monitor::Vector{Int}` : indices identifying elements of the `value` field to include in monitored MCMC sampler output.
- `eval::Function` : a function for updating values stored in `value`.
- `sources::Vector{Symbol}` : symbols of other nodes upon whom the values of this one depends.
- `targets::Vector{Symbol}` : symbols of `Dependent` nodes that depend on this one. Elements of `targets` are topologically sorted so that a given node in the vector is conditionally independent of subsequent nodes, given the previous ones.

Constructors

`Logical`(*expr::Expr*, *monitor::Union(Bool*, *Vector{Int})=true*)
`Logical`(*d::Integer*, *expr::Expr*, *monitor::Union(Bool*, *Vector{Int})=true*)
Construct a `Logical` object that defines a logical model node.

Arguments

- `d` : number of dimensions for array nodes.
- `expr` : a quoted expression or code-block defining the body of the function stored in the `eval` field.
- `monitor` : a boolean indicating whether all elements are monitored, or a vector of element-wise indices of elements to monitor.

Value

Returns a `Logical{Array{VariateType,d}}` if the dimension argument `d` is specified, and a `Logical{VariateType}` if not.

Example

See the *Model Specification* section of the tutorial.

Methods

`setinits!`(*l::Logical*, *m::Model*, *::Any=nothing*)
Set initial values for a logical node.

Arguments

- `l` : a logical node to which to assign initial values.
- `m` : a model that contains the node.

Value

Returns the result of a call to `update!(l, m)`.

update! (*l::Logical*, *m::Model*)

Update the values of a logical node according to its relationship with others in a model.

Arguments

- `l` : a logical node to update.
- `m` : a model that contains the node.

Value

Returns the node with its values updated.

2.4.3 Stochastic

Type `Stochastic` inherits the fields and method functions from the `Dependent` type, and adds the additional ones listed below. It is designed for model parameters or data that have distributional or likelihood specifications, respectively. Its stochastic relationship to other nodes and data structures is represented by the `Distributions` structure stored in field `distr`. Stored in the field `eval` is an anonymous function defined as

```
function(model::Mamba.Model)
```

where `model` contains all model nodes. The function can contain any valid **julia** expression or code-block. It should return a single *Distributions* object for all node elements or a structure of the same type as the node with element-specific *Distributions* objects.

Declaration

```
type Stochastic{T} <:  Dependent{T}
```

Fields

- `value::T` : a scalar or array of `VariateType` values that represent samples from a target distribution.
- `symbol::Symbol` : an identifying symbol for the node.
- `nlink::Integer` : number of elements returned by the `link` method defined on the type.
- `monitor::Vector{Int}` : indices identifying elements of the `value` field to include in monitored MCMC sampler output.
- `eval::Function` : a function for updating the `distr` field for the node.
- `sources::Vector{Symbol}` : symbols of other nodes upon whom the distributional specification for this one depends.
- `targets::Vector{Symbol}` : symbols of `Dependent` nodes that depend on this one. Elements of `targets` are topologically sorted so that a given node in the vector is conditionally independent of subsequent nodes, given the previous ones.
- `distr::DistributionStruct` : the distributional specification for the node.

Aliases

```
typealias DistributionStruct Union(Distribution, Array{Distribution})
```

Constructors

Stochastic(*expr::Expr*, *monitor::Union(Bool*, *Vector{Int})=true*)
Stochastic(*d::Integer*, *expr::Expr*, *monitor::Union(Bool*, *Vector{Int})=true*)
Construct a `Stochastic` object that defines a stochastic model node.

Arguments

- `d` : number of dimensions for array nodes.
- `expr` : a quoted expression or code-block defining the body of the function stored in the `eval` field.
- `monitor` : a boolean indicating whether all elements are monitored, or a vector of element-wise indices of elements to monitor.

Value

Returns a `Stochastic{Array{VariateType,d}}` if the dimension argument `d` is specified, and a `Stochastic{VariateType}` if not.

Example

See the *Model Specification* section of the tutorial.

Methods

insupport(*s::Stochastic*)
Check whether stochastic node values are within the support of its distribution.

Arguments

- `s` : a stochastic node on which to perform the check.

Value

Returns `true` if all values are within the support, and `false` otherwise.

invlink(*s::Stochastic*, *x*, *transform::Bool=true*)
Apply an inverse-link transformation to map transformed values back to the original distributional scale of a stochastic node.

Arguments

- `s` : a stochastic node on which a `link()` transformation method is defined.
- `x` : an object to which to apply the inverse-link transformation.
- `transform` : whether to transform `x` or assume an identity link.

Value

Returns the inverse-link-transformed version of `x`.

link(*s::Stochastic*, *x*, *transform::Bool=true*)
Apply a link transformation to map values in a constrained distributional support to an unconstrained space. Supports for continuous, univariate distributions and positive-definite matrix distributions (Wishart or inverse-Wishart) are transformed as follows:

- Lower and upper bounded: scaled and shifted to the unit interval and logit-transformed.
- Lower bounded: shifted to zero and log-transformed.
- Upper bounded: scaled by -1, shifted to zero, and log-transformed.

- Positive-definite matrix: compute the (upper-triangular) Cholesky decomposition, and return its log-transformed diagonal elements prepended to the remaining upper-triangular part as a vector of length $n(n+1)/2$, where n is the matrix dimension.

Arguments

- `s` : a stochastic node on which a `link()` transformation method is defined.
- `x` : an object to which to apply the link transformation.
- `transform` : whether to transform `x` or assume an identity link.

Value

Returns the link-transformed version of `x`.

`logpdf`(*s::MCMStochastic*, *transform::Bool=false*)
Evaluate the log-density function for a stochastic node.

Arguments

- `s` : a stochastic node containing values at which to compute the log-density.
- `transform` : whether to evaluate the log-density on the link-transformed scale.

Value

The resulting numeric value of the log-density.

`setinits!`(*s::Stochastic*, *m::Model*, *x=nothing*)
Set initial values for a stochastic node.

Arguments

- `s` : a stochastic node to which to assign initial values.
- `m` : a model that contains the node.
- `x` : values to assign to the node.

Value

Returns the node with its assigned initial values.

`update!`(*s::Stochastic*, *m::Model*)
Update the values of a stochastic node according to its relationship with others in a model.

Arguments

- `s` : a stochastic node to update.
- `m` : a model that contains the node.

Value

Returns the node with its values updated.

2.4.4 Distributions

Given in this section are distributions, as provided by the *Distributions [1]* and *Mamba* packages, supported for the specification of *Stochastic* nodes. Truncated versions of continuous univariate distributions are also supported.

Univariate Distributions

Distributions Package Univariate Types

The following univariate types from the *Distributions* package are supported.

Arcsine	Cosine	Hypergeometric	NegativeBinomial	Rayleigh
Bernoulli	DiscreteUniform	InverseGamma	NoncentralBeta	Skellam
Beta	Edgeworth	InverseGaussian	NoncentralChisq	TDist
BetaPrime	Erlang	KSDist	NoncentralF	TriangularDist
Binomial	Exponential	KSOneSided	NoncentralT	Uniform
Categorical	FDist	Laplace	Normal	Weibull
Cauchy	Gamma	Levy	NormalCanon	
Chi	Geometric	Logistic	Pareto	
Chisq	Gumbel	LogNormal	Poisson	

Flat Distribution

A Flat distribution is supplied with the degenerate probability density function:

$$f(x) \propto 1, \quad -\infty < x < \infty.$$

```
Flat()     # Flat distribution
```

User-Defined Univariate Distributions

New known, unknown, or unnormalized univariate distributions can be created and added to *Mamba* as subtypes of the *Distributions* package `ContinuousUnivariateDistribution` or `DiscreteUnivariateDistribution` types. *Mamba* requires only a partial implementation of the method functions described in the full instructions for creating univariate distributions. The specific workflow is given below.

1. Create a `quote` block for the new distribution. Assign the block a variable name, say `extensions`, preceded by the `@everywhere` macro to ensure compatibility when **julia** is run in multi-processor mode.
2. The *Distributions* package contains types and method definitions for new distributions. Load the package and import the package's methods (indicated below) to be extended.
3. Declare a new distribution subtype, say `D`, within the block. Create a constructor for the subtype that accepts un-typed arguments and explicitly converts them in the constructor body to the proper types for the fields of `D`. Implementing the constructor in this way ensures that it will be callable with the *Mamba* `Stochastic` and `Logical` types.
4. Extend/define the following *Distributions* package methods for the new distribution `D`.

   minimum(*d::D*)
   Return the lower bound of the support of `d`.

   maximum(*d::D*)
   Return the upper bound of the support of `d`.

   logpdf(*d::D*, *x::Real*)
   Return the normalized or unnomalized log-density evaluated at `x`.

5. Test the subtype.
6. Add the `quote` block (new distribution) to *Mamba* with the following calls.

```
        using Mamba
        @everywhere eval(Mamba, extensions)
```

Below is a univariate example based on the linear regression model in the *Tutorial*.

```
## Define a new univariate Distribution type for Mamba.
## The definition must be placed within an unevaluated quote block.
@everywhere extensions = quote

  ## Load needed packages and import methods to be extended
  using Distributions
  import Distributions: minimum, maximum, logpdf

  ## Type declaration
  type NewUnivarDist <: ContinuousUnivariateDistribution
    mu::Float64
    sigma::Float64

    ## Constructor
    function NewUnivarDist(mu, sigma)
      new(convert(Float64, mu), convert(Float64, sigma))
    end
  end

  ## The following method functions must be implemented

  ## Minimum and maximum support values
  minimum(d::NewUnivarDist) = -Inf
  maximum(d::NewUnivarDist) = Inf

  ## Normalized or unnormalized log-density value
  function logpdf(d::NewUnivarDist, x::Real)
    -log(d.sigma) - 0.5 * ((x - d.mu) / d.sigma)^2
  end

end

## Test the extensions in a temporary module (optional)
module Testing end
eval(Testing, extensions)
d = Testing.NewUnivarDist(0.0, 1.0)
Testing.minimum(d)
Testing.maximum(d)
Testing.insupport(d, 2.0)
Testing.logpdf(d, 2.0)

## Add the extensions to Mamba
using Mamba
@everywhere eval(Mamba, extensions)

## Implement a Mamba model using the new distribution
model = Model(

  y = Stochastic(1,
    @modelexpr(mu, s2,
      begin
        sigma = sqrt(s2)
        Distribution[NewUnivarDist(mu[i], sigma) for i in 1:length(mu)]
```

```
      end
    ),
    false
  ),

  mu = Logical(1,
    @modelexpr(xmat, beta,
      xmat * beta
    ),
    false
  ),

  beta = Stochastic(1,
    :(MvNormal(2, sqrt(1000)))
  ),

  s2 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)

## Sampling Scheme
scheme = [NUTS([:beta]),
          Slice([:s2], [3.0])]

## Sampling Scheme Assignment
setsamplers!(model, scheme)

## Data
line = (Symbol => Any)[
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
]
line[:xmat] = [ones(5) line[:x]]

## Initial Values
inits = Dict{Symbol,Any}[
  [:y => line[:y],
   :beta => rand(Normal(0, 1), 2),
   :s2 => rand(Gamma(1, 1))]
  for i in 1:3]

## MCMC Simulation
sim = mcmc(model, line, inits, 10000, burnin=250, thin=2, chains=3)
describe(sim)
```

Multivariate Distributions

Distributions Package Multivariate Types

The following multivariate types from the *Distributions* package are supported.

```
Dirichlet     Multinomial     MvNormal          MvNormalCannon
MvTDist       VonMisesFisher
```

Block-Diagonal Multivariate Normal Distribution

A Block-Diagonal Multivariate Normal distribution is supplied with the probability density function:

$$f(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right), \quad -\infty < \boldsymbol{x} < \infty,$$

where

$$\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma_1} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Sigma_2} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \boldsymbol{\Sigma}_m \end{bmatrix}.$$

```
BDiagNormal(mu, C)      # multivariate normal with mean vector mu and block-
                        # diagonal covariance matrix Sigma such that
                        # length(mu) = dim(Sigma), and Sigma_1 = ... = Sigma_m = C
                        # for a matrix C or Sigma_1 = C[1], ..., Sigma_m = C[m]
                        # for a vector of matrices C.
```

User-Defined Multivariate Distributions

New known, unknown, or unnormalized multivariate distributions can be created and added to *Mamba* as subtypes of the *Distributions* package `ContinuousMultivariateDistribution` or `DiscreteMultivariateDistribution` types. *Mamba* requires only a partial implementation of the method functions described in the full instructions for creating multivariate distributions. The specific workflow is given below.

1. Create a `quote` block for the new distribution. Assign the block a variable name, say `extensions`, preceded by the `@everywhere` macro to ensure compatibility when **julia** is run in multi-processor mode.
2. The *Distributions* package contains types and method definitions for new distributions. Load the package and import the package's methods (indicated below) to be extended.
3. Declare a new distribution subtype, say `D`, within the block. Create a constructor for the subtype that accepts un-typed arguments and explicitly converts them in the constructor body to the proper types for the fields of `D`. Implementing the constructor in this way ensures that it will be callable with the *Mamba* `Stochastic` and `Logical` types.
4. Extend/define the following *Distributions* package methods for the new distribution `D`.

   `length`(*d::D*)
   Return the sample space size (dimension) of `d`.

   `insupport{T<:Real}`(*d::D*, *x::Vector{T}*)
   Return a logical indicating whether `x` is in the support of `d`.

   `_logpdf{T<:Real}`(*d::D*, *x::Vector{T}*)
   Return the normalized or unnomalized log-density evaluated at `x`.

5. Test the subtype.
6. Add the `quote` block (new distribution) to *Mamba* with the following calls.

```
using Mamba
@everywhere eval(Mamba, extensions)
```

Below is a multivariate example based on the linear regression model in the *Tutorial*.

```
## Define a new multivariate Distribution type for Mamba.
## The definition must be placed within an unevaluated quote block.
@everywhere extensions = quote

  ## Load needed packages and import methods to be extended
  using Distributions
  import Distributions: length, insupport, _logpdf

  ## Type declaration
  type NewMultivarDist <: ContinuousMultivariateDistribution
    mu::Vector{Float64}
    sigma::Float64

    ## Constructor
    function NewMultivarDist(mu, sigma)
      new(convert(Vector{Float64}, mu), convert(Float64, sigma))
    end
  end

  ## The following method functions must be implemented

  ## Dimension of the distribution
  length(d::NewMultivarDist) = length(d.mu)

  ## Logicals indicating whether elements of x are in the support
  function insupport{T<:Real}(d::NewMultivarDist, x::Vector{T})
    length(d) == length(x) && all(isfinite(x))
  end

  ## Normalized or unnormalized log-density value
  function _logpdf{T<:Real}(d::NewMultivarDist, x::Vector{T})
    -length(x) * log(d.sigma) - 0.5 * sumabs2(x - d.mu) / d.sigma^2
  end

end

## Test the extensions in a temporary module (optional)
module Testing end
eval(Testing, extensions)
d = Testing.NewMultivarDist([0.0, 0.0], 1.0)
Testing.insupport(d, [2.0, 3.0])
Testing.logpdf(d, [2.0, 3.0])

## Add the extensions to Mamba
using Mamba
@everywhere eval(Mamba, extensions)

## Implement a Mamba model using the new distribution
model = Model(

  y = Stochastic(1,
    @modelexpr(mu, s2,
      NewMultivarDist(mu, sqrt(s2))
    ),
    false
  ),

  mu = Logical(1,
```

```
    @modelexpr(xmat, beta,
      xmat * beta
    ),
    false
  ),

  beta = Stochastic(1,
    :(MvNormal(2, sqrt(1000)))
  ),

  s2 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)

## Sampling Scheme
scheme = [NUTS([:beta]),
          Slice([:s2], [3.0])]

## Sampling Scheme Assignment
setsamplers!(model, scheme)

## Data
line = (Symbol => Any)[
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
]
line[:xmat] = [ones(5) line[:x]]

## Initial Values
inits = Dict{Symbol,Any}[
  [:y => line[:y],
   :beta => rand(Normal(0, 1), 2),
   :s2 => rand(Gamma(1, 1))]
  for i in 1:3]

## MCMC Simulation
sim = mcmc(model, line, inits, 10000, burnin=250, thin=2, chains=3)
describe(sim)
```

Matrix-Variate Distributions

Distributions Package Matrix-Variate Types

The following matrix-variate types from the *Distributions* package are supported.

```
InverseWishart   Wishart
```

2.4.5 Sampler

Each of the $\{f_j\}_{j=1}^{B}$ sampling functions of the *Mamba Gibbs sampling scheme* is implemented as a `Sampler` type object, whose fields are summarized herein. The `eval` field is an anonymous function defined as

```
function(model::Mamba.Model, block::Integer)
```

where `model` contains all model nodes, and `block` is an index identifying the corresponding sampling function in a vector of all samplers for the associated model. Through the arguments, all model nodes and fields can be accessed in the body of the function. The function may return an updated sample for the nodes identified in its `params` field. Such a return value can be a structure of the same type as the node if the block consists of only one node, or a dictionary of node structures with keys equal to the block node symbols if one or more. Alternatively, a value of `nothing` may be returned. Return values that are not `nothing` will be used to automatically update the node values and propagate them to dependent nodes. No automatic updating will be done if `nothing` is returned.

Declaration

```
type Sampler
```

Fields

- `params::Vector{Symbol}` : symbols of stochastic nodes in the block being updated by the sampler.
- `eval::Function` : a sampling function that updates values of the `params` nodes.
- `tune::Dict{String,Any}` : any tuning parameters needed by the sampling function.
- `targets::Vector{Symbol}` : symbols of `Dependent` nodes that depend on and whose states must be updated after `params`. Elements of `targets` are topologically sorted so that a given node in the vector is conditionally independent of subsequent nodes, given the previous ones.

Constructor

`Sampler`(*params::Vector{Symbol}*, *expr::Expr*, *tune::Dict=Dict()*)

Construct a `Sampler` object that defines a sampling function for a block of stochastic nodes.

Arguments

- `params` : symbols of nodes that are being block-updated by the sampler.
- `expr` : a quoted expression that makes up the body of the sampling function whose definition is described above.
- `tune` : tuning parameters needed by the sampling function.

Value

Returns a `Sampler` type object.

Example

See the *Model Specification* section of the tutorial.

Methods

`show`(*s::Sampler*)

Write a text representation of the defined sampling function to the current output stream.

`showall`(*s::Sampler*)

Write a verbose text representation of the defined sampling function to the current output stream.

2.4.6 Model

The `Model` type is designed to store the set of all model nodes, including parameter set Θ as denoted in the *Mamba Gibbs sampling scheme*. In particular, it stores `Dependent` type objects in its `nodes` dictionary field. Valid models are ones whose nodes form directed acyclic graphs (DAGs). Sampling functions $\{f_j\}_{j=1}^{B}$ are saved as `Sampler` objects in the vector of field `samplers`. Vector elements $j = 1, \ldots, B$ correspond to sampling blocks $\{\Theta_j\}_{j=1}^{B}$.

Declaration

```
type Model
```

Fields

- `nodes::Dict{Symbol,Any}` : a dictionary containing all input, logical, and stochastic model nodes.
- `dependents::Vector{Symbol}` : symbols of all `Dependent` nodes in topologically sorted order so that a given node in the vector is conditionally independent of subsequent nodes, given the previous ones.
- `samplers::Vector{Sampler}` : sampling functions for updating blocks of stochastic nodes.
- `states::Vector{Vector{VariateType}}` : states of chains at the end of a possible series of MCMC runs.
- `iter::Integer` : current MCMC draw from the target distribution.
- `burnin::Integer` : number of initial draws to discard as a burn-in sequence to allow for convergence.
- `chain::Integer` : current run of an MCMC simulator in a possible series of runs.
- `hasinputs::Bool` : whether values have been assigned to the input nodes.
- `hasinits::Bool` : whether initial values have been assigned to stochastic nodes.

Constructor

`Model`(*; iter::Integer=0, burnin::Integer=0, chain::Integer=1, samplers::Vector{Sampler}=Array(Sampler, 0), nodes...*)

Construct a `Model` object that defines a model for MCMC simulation.

Arguments

- `iter` : current iteration of the MCMC simulation.
- `burnin` : number of initial draws to be discarded as a burn-in sequence to allow for convergence.
- `chain` : current run of the MCMC simulator in a possible sequence of runs.
- `samplers` : a vector of block-specific sampling functions.
- `nodes...` : an arbitrary number of user-specified arguments defining logical and stochastic nodes in the model. Argument values must be `Logical` or `Stochastic` type objects. Their names in the model will be taken from the argument names.

Value

Returns a `Model` type object.

Example

See the *Model Specification* section of the tutorial.

Methods

draw(*m::Model; filename::String=""*)
Draw a GraphViz DOT-formatted graph representation of model nodes and their relationships.

Arguments

- `m` : a model for which to construct a graph.
- `filename` : an external file to which to save the resulting graph, or an empty string to draw to standard output (default). If a supplied external file name does not include a dot (`.`), the file extension `.dot` will be appended automatically.

Value

The model drawn to an external file or standard output. Stochastic, logical, and input nodes will be represented by ellipses, diamonds, and rectangles, respectively. Nodes that are unmonitored in MCMC simulations will be gray-colored.

Example

See the *Directed Acyclic Graphs* section of the tutorial.

getindex(*m::Model*, *key::Symbol*)
Returns a model node identified by its symbol. The syntax `m[key]` is converted to `getindex(m, key)`.

Arguments

- `m` : a model contining the node to get.
- `key` : symbol of the node to get.

Value

The specified node.

gradlogpdf(*m::Model*, *block::Integer=0*, *transform::Bool=false; dtype::Symbol=:forward*)
gradlogpdf(*m::Model*, *x::Vector{T<:Real}*, *block::Integer=0*, *transform::Bool=false; dtype::Symbol=:forward*)
gradlogpdf!(*m::Model*, *x::Vector{T<:Real}*, *block::Integer=0*, *transform::Bool=false; dtype::Symbol=:forward*)
Compute the gradient of log-densities for stochastic nodes.

Arguments

- `m` : a model containing the stochastic nodes for which to compute the gradient.
- `x` : a value (possibly different than the current one) at which to compute the gradient.
- `block` : the sampling block of stochastic nodes for which to compute the gradient, if specified; otherwise, all sampling blocks are included.
- `transform` : whether to compute the gradient of block parameters on the link–transformed scale.
- **`dtype`** [type of differentiation for gradient calculations. Options are]
  - `:central` : central differencing.
  - `:forward` : forward differencing.

Value

The resulting gradient vector. Method `gradlogpdf!()` additionally updates model `m` with supplied values `x`.

Note

Numerical approximation of derivatives by central and forward differencing is performed with the *Calculus* package *[76]*.

graph (*m::Model*)
Construct a graph representation of model nodes and their relationships.

Arguments

- `m` : a model for which to construct a graph.

Value

Returns a `GenericGraph` type object as defined in the Graphs package.

graph2dot (*m::Model*)
Draw a GraphViz DOT-formatted graph representation of model nodes and their relationships.

Arguments

- `m` : a model for which to construct a graph.

Value

A character string representation of the graph suitable for in-line processing. Stochastic, logical, and input nodes will be represented by ellipses, diamonds, and rectangles, respectively. Nodes that are unmonitored in MCMC simulations will be gray-colored.

Example

See the *Directed Acyclic Graphs* section of the tutorial.

keys (*m::Model*, *ntype::Symbol=:assigned*, *block::Integer=0*)
Return the symbols of nodes of a specified type.

Arguments

- `m` : a model containing the nodes of interest.
- **`ntype`** [the type of nodes to return. Options are]
  - `:all` : all input, logical, and stochastic model nodes.
  - `:assigned` : nodes that have been assigned values.
  - `:block` : stochastic nodes being block-sampled.
  - `:dependent` : logical or stochastic (dependent) nodes.
  - `:independent` or `:input` : input (independent) nodes.
  - `:logical` : logical nodes.
  - `:monitor` : stochastic nodes being monitored in MCMC sampler output.
  - `:output` : stochastic nodes upon which no other stochastic nodes depend.
  - `:stochastic` : stochastic nodes.
- `block` : the block for which to return nodes if `ntype = :block`, or all blocks if `block = 0` (default).

Value

A vector of node symbols.

logpdf (*m::Model*, *block::Integer=0*, *transform::Bool=false*)
logpdf (*m::Model*, *x::Vector{T<:Real}*, *block::Integer=0*, *transform::Bool=false*)

logpdf!(*m::Model*, *x::Vector{T<:Real}*, *block::Integer=0*, *transform::Bool=false*)
Compute the sum of log-densities for stochastic nodes.

Arguments

- `m` : a model containing the stochastic nodes for which to evaluate log-densities.
- `x` : a value (possibly different than the current one) at which to evaluate densities.
- `block` : the sampling block of stochastic nodes over which to sum densities, if specified; otherwise, all stochastic nodes are included.
- `transform` : whether to evaluate evaluate log-densities of block parameters on the link–transformed scale.

Value

The resulting numeric value of summed log-densities. Method `logpdf!()` additionally updates model `m` with supplied values `x`.

mcmc(*m::Model*, *inputs::Dict{Symbol}*, *inits::Vector{Dict{Symbol*, *Any}}*, *iters::Integer; burnin::Integer=0*, *thin::Integer=1*, *chains::Integer=1*, *verbose::Bool=true*)
mcmc(*c::Chains*, *iters::Integer; verbose::Bool=true*)
Simulate MCMC draws for a specified model.

Arguments

- `m` : a specified mode.
- `c` : chains from a previous call to `mcmc` for which to simulate additional draws.
- `inputs` : a dictionary of values for input model nodes. Dictionary keys and values should be given for each input node.
- `inits` : a vector of dictionaries that contain initial values for stochastic model nodes. Dictionary keys and values should be given for each stochastic node. Consecutive runs of the simulator will iterate through the vector's dictionary elements.
- `iters` : number of draws to generate for each simulation run.
- `burnin` : numer of initial draws to discard as a burn-in sequence to allow for convergence.
- `thin` : step-size between draws to output.
- `chains` : number of simulation runs to perform.
- `verbose` : whether to print sampler progress at the console.

Value

A `Chains` type object of simulated draws.

Example

See the *MCMC Simulation* section of the tutorial.

relist(*m::Model*, *values::Vector{T<:Real}*, *block::Integer=0*, *transform::Bool=false*)
relist(*m::Model*, *values::Vector{T<:Real}*, *nkeys::Vector{Symbol}*, *transform::Bool=false*)
Convert a vector of values to a set of logical and/or stochastic node values.

Arguments

- `m` : a model with nodes to serve as the template for conversion.
- `values` : values to convert.
- `block` : the sampling block of nodes to which to convert `values`. Defaults to all blocks.

- `nkeys` : a vector of symbols identifying the nodes to which to convert `values`.
- `transform` : whether to apply an inverse-link transformation in the conversion.

Value

A dictionary of node symbols and converted values.

relist! (*m::Model*, *values::Vector{T<:Real}*, *block::Integer=0*, *transform::Bool=false*)
relist! (*m::Model*, *values::Vector{T<:Real}*, *nkeys::Vector{Symbol}*, *transform::Bool=false*)
Copy a vector of values to a set of logical and/or stochastic nodes.

Arguments

- `m` : a model with nodes to which values will be copied.
- `values` : values to copy.
- `block` : the sampling block of nodes to which to copy `values`. Defaults to all blocks.
- `nkeys` : a vector of symbols identifying the nodes to which to copy `values`.
- `transform` : whether to apply an inverse-link transformation in the copy.

Value

Returns the model with copied node values.

setinits! (*m::Model*, *inits::Dict{Symbol, Any}*)
Set the initial values of stochastic model nodes.

Arguments

- `m` : a model with nodes to be initialized.
- `inits` : a dictionary of initial values for stochastic model nodes. Dictionary keys and values should be given for each stochastic node.

Value

Returns the model with stochastic nodes initialized and the `iter` field set equal to 0.

Example

See the *Development and Testing* section of the tutorial.

setinputs! (*m::Model*, *inputs::Dict{Symbol, Any}*)
Set the values of input model nodes.

Arguments

- `m` : a model with input nodes to be assigned.
- `inputs` : a dictionary of values for input model nodes. Dictionary keys and values should be given for each input node.

Value

Returns the model with values assigned to input nodes.

Example

See the *Development and Testing* section of the tutorial.

setsamplers! (*m::Model*, *samplers::Vector{Sampler}*)
Set the block-samplers for stochastic model nodes.

Arguments

- `m` : a model with stochastic nodes to be sampled.
- `samplers` : block-specific samplers.

Values:

Returns the model updated with the block-samplers.

Example

See the *Model Specification* and *MCMC Simulation* sections of the tutorial.

`show`(*m::Model*)
Write a text representation of the model, nodes, and attributes to the current output stream.

`showall`(*m::Model*)
Write a verbose text representation of the model, nodes, and attributes to the current output stream.

`simulate!`(*m::Model*, *block::Integer=0*)
Simulate one MCMC draw from a specified model.

Argument:

- `m` : a model specification.
- `block` : the block for which to simulate an MCMC draw, if specified; otherwise, simulate draws for all blocks (default).

Value

Returns the model updated with the MCMC draw and, in the case of `block=0`, the `iter` field incremented by 1.

Example

See the *Development and Testing* section of the tutorial.

`tune`(*m::Model*, *block::Integer=0*)
Get block-sampler tuning parameters.

Arguments

- `m` : a model with block-samplers.
- `block` : the block for which to return the tuning parameters, if specified; otherwise, the tuning parameters for all blocks.

Value

If `block = 0`, a vector of dictionaries containing block-specific tuning parameters; otherwise, one block-specific dictionary.

`unlist`(*m::Model*, *block::Integer=0*, *transform::Bool=false*)
`unlist`(*m::Model*, *nkeys::Vector{Symbol}*, *transform::Bool=false*)
Convert a set of logical and/or stochastic node values to a vector.

Arguments

- `m` : a model with nodes to be converted.
- `block` : the sampling block of nodes to be converted. Defaults to all blocks.
- `nkeys` : a vector of symbols identifying the nodes to be converted.
- `transform` : whether to apply a link transformation in the conversion.

Value

A vector of concatenated node values.

update! (*m::Model*, *block::Integer=0*)

Update values of logical and stochastic model node according to their relationship with others in a model.

Arguments

- `m` : a mode with nodes to be updated.
- `block` : the sampling block of nodes to be updated. Defaults to all blocks.

Value

Returns the model with updated nodes.

2.4.7 Chains

The `Chains` type stores output from one or more runs (chains) of an MCMC sampler. It serves as the container for output generated by the *`mcmc()`* function, and supplies methods for convergence diagnostics and posterior inference. Moreover, it can be used as a stand-alone container for any user-generated MCMC output, and is thus a **julia** analogue to the *boa [66][67]* and *coda [58][59]* **R** packages.

Declaration

```
immutable Chains
```

Fields

- `value::Array{VariateType,3}` : a 3-dimensional array of sampled values whose first, second, and third dimensions index the iterations, parameter elements, and runs of an MCMC sampler, respectively.
- `range::Range{Int}` : range of iterations stored in the rows of the `value` array.
- `names::Vector{String}` : names assigned to the parameter elements.
- `chains::Vector{Integer}` : indices to the MCMC runs.
- `model::Model` : the model from which the sampled values were generated.

Constructors

Chains (*iters::Integer*, *params::Integer;* *start::Integer=1*, *thin::Integer=1*, *chains::Integer=1*, *names::Vector{T<:String}=Array(String, 0)*, *model::Model=Model()*)
Chains (*value::Array{T<:Real, 3}; start::Integer=1*, *thin::Integer=1*, *names::Vector{U<:String}=Array(String, 0)*, *chains::Vector{V<:Integer}=Array(Integer, 0)*, *model::Model=Model()*)
Chains (*value::Matrix{T<:Real}; start::Integer=1*, *thin::Integer=1*, *names::Vector{U<:String}=Array(String, 0)*, *chains::Integer=1*, *model::Model=Model()*)
Chains (*value::Vector{T<:Real};* *start::Integer=1*, *thin::Integer=1*, *names::String="Param1"*, *chains::Integer=1*, *model::Model=Model()*)

Construct a `Chains` object that stores MCMC sampler output.

Arguments

- `iters` : total number of iterations in each sampler run, of which `length(start:thin:iters)` outputted iterations will be stored in the object.
- `params` : number of parameters to store.

- `value` : an array whose first, second (optional), and third (optional) dimensions index outputted iterations, parameter elements, and runs of an MCMC sampler, respectively.
- `start` : number of the first iteration to be stored.
- `thin` : number of steps between consecutive iterations to be stored.
- `chains` : number of simulation runs for which to store output, or indices to the runs (default: 1, 2, ...).
- `names` : names to assign to the parameter elements (default: `"Param1"`, `"Param2"`, ...).
- `model` : the model for which the simulation was run.

Value

Returns a `Chains` type object.

Example

See the *AMM*, *AMWG*, *NUTS*, and *Slice* examples.

Indexing

`getindex`(*c::Chains*, *inds...*)

Subset MCMC sampler output. The syntax `c[i, j, k]` is converted to `getindex(c, i, j, k)`.

Arguments

- `c` : sampler output to subset.
- `inds...` : a tuple of `i, j, k` indices to the iterations, parameters, and chains to be subsetted. Indices of the form `start:stop` or `start:thin:stop` can be used to subset iterations, where `start` and `stop` define a range for the subset and `thin` will apply additional thinning to existing sampler output. Indices for subsetting of parameters can be specified as strings, integers, or booleans identifying parameters to be kept. Indices for chains can be integers or booleans. A value of `:` can be specified for any of the dimensions to indicate no subsetting.

Value

Returns a `Chains` object with the subsetted sampler output.

Example

See the *Output Subsetting* section of the tutorial.

`setindex!`(*c::Chains*, *value*, *inds...*)

Store MCMC sampler output at a given index. The syntax `c[i, j, k] = value` is converted to `setindex!(c, value, i, j, k)`.

Arguments

- `c` : object within which to store sampler output.
- `value` : sampler output.
- `inds...` : a tuple of `i, j, k` indices to iterations, parameters, and chains within the object. Iterations can be indexed as a `start:stop` or `start:thin:stop` range, a single numeric index, or a vector of indices; and are taken to be relative to the index range store in the `c.range` field. Indices for subsetting of parameters can be specified as strings, integers, or booleans. Indices for chains can be integers or booleans. A value of `:` can be specified for the parameters or chains to index all corresponding elements.

Value

Returns a `Chains` object with the sampler output stored in the specified indices.

Example

See the *AMM*, *AMWG*, *NUTS*, and *Slice* examples.

Convergence Diagnostics

MCMC simulation provides autocorrelated samples from a target distribution. Because of computational complexities in implementing MCMC algorithms, the autocorrelated nature of samples, and the need to choose initial sampling values at different points in target distributions; it is important to evaluate the quality of resulting output. Specifically, one should check that MCMC samples have converged to the target (or, more commonly, are stationary) and that the number of convergent samples provides sufficiently accurate and precise estimates of posterior statistics.

Several established convergence diagnostics are supplied by *Mamba*. The diagnostics and their features are summarized in the table below and described in detail in the subsequent function descriptions. They differ with respect to the posterior statistic being assessed (mean vs. quantile), whether the application is to parameters univariately or multivariately, and the number of chains required for calculations. Diagnostics may assess stationarity, estimation accuracy and precision, or both. A more comprehensive comparative review can be found in *[16]*. Since diagnostics differ in their focus and design, it is often good practice to employ more than one to assess convergence. Note too that diagnostics generally test for non-convergence and that non-significant test results do not prove convergence. Thus, non-significant results should be interpreted with care.

Table 2.2: Comparative summary of features for the supplied MCMC convergence diagnostics.

				Convergence Assessments	
Diagnostic	Statistic	Parameters	Chains	Stationarity	Estimation
Gelman, Rubin, and Brooks	Mean	Univariate	2+	Yes	No
		Multivariate	2+	Yes	No
Geweke	Mean	Univariate	1	Yes	No
Heidelberger and Welch	Mean	Univariate	1	Yes	Yes
Raftery and Lewis	Quantile	Univariate	1	Yes	Yes

Gelman, Rubin, and Brooks Diagnostics

gelmandiag (*c::Chains; alpha::Real=0.05*, *mpsrf::Bool=false*, *transform::Bool=false*)

Compute the convergence diagnostics of Gelman, Rubin, and Brooks *[29][10]* for MCMC sampler output. The diagnostics are designed to asses convergence of posterior means estimated with multiple autocorrelated samples (chains). They does so by comparing the between and within-chain variances with metrics called *potential scale reduction factors (PSRF)*. Both univariate and multivariate factors are available to assess the convergence of parameters individually and jointly. Scale factors close to one are indicative of convergence. As a rule of thumb, convergence is concluded if the 0.975 quantile of an estimated factor is less than 1.2. Multiple chains are required for calculations. It is recommended that at least three chains be generated, each with different starting values chosen to be diffuse with respect to the anticipated posterior distribution. Use of multiple chains in the diagnostic provides for more robust assessment of convergence than is possible with single chain diagnostics.

Arguments

- `c` : sampler output on which to perform calculations.
- `alpha` : quantile (`1 - alpha / 2`) at which to estimate the upper limits of scale reduction factors.
- `mpsrf` : whether to compute the multivariate potential scale reduction factor. This factor will not be calculable if any one of the parameters in the output is a linear combination of others.
- `transform` : whether to apply log or logit transformations, as appropriate, to parameters in the chain to potentially produce output that is more normally distributed, an assumption of the PSRF formulations.

Value

A `ChainSummary` type object of the form:

```
immutable ChainSummary
  value::Array{Float64,3}
  rownames::Vector{String}
  colnames::Vector{String}
  header::String
end
```

with parameters contained in the rows of the `value` field, and scale reduction factors and upper-limit quantiles in the first and second columns.

Example

See the *Convergence Diagnostics* section of the tutorial.

Geweke Diagnostic

gewekediag(*c::Chains; first::Real=0.1*, *last::Real=0.5*, *etype=:imse*, *args...*)

Compute the convergence diagnostic of Geweke *[33]* for MCMC sampler output. The diagnostic is designed to asses convergence of posterior means estimated with autocorrelated samples. It computes a normal-based test statistic comparing the sample means in two windows containing proportions of the first and last iterations. Users should ensure that there is sufficient separation between the two windows to assume that their samples are independent. A non-significant test p-value indicates convergence. Significant p-values indicate non-convergence and the possible need to discard initial samples as a burn-in sequence or to simulate additional samples.

Arguments

- `c` : sampler output on which to perform calculations.
- `first` : proportion of iterations to include in the first window.
- `last` : proportion of iterations to include in the last window.
- `etype` : method for computing Monte Carlo standard errors. See *`mcse()`* for options.
- `args...` : additional arguments to be passed to the `etype` method.

Value

A `ChainSummary` type object with parameters contained in the rows of the `value` field, and test Z-scores and p-values in the first and second columns. Results are chain-specific.

Example

See the *Convergence Diagnostics* section of the tutorial.

Heidelberger and Welch Diagnostic

heideldiag(*c::Chains; alpha::Real=0.05*, *eps::Real=0.1*, *etype=:imse*, *args...*)

Compute the convergence diagnostic of Heidelberger and Welch *[41]* for MCMC sampler output. The diagnostic is designed to assess convergence of posterior means estimated with autocorrelated samples and to determine whether a target degree of accuracy is achieved. A stationarity test is performed for convergence assessment by iteratively discarding 10% of the initial samples until the test p-value is non-significant and stationarity is concluded or until 50% have been discarded and stationarity is rejected, whichever occurs first. Then, a halfwidth test is performed by calculating the relative halfwidth of a posterior mean estimation interval as $z_{1-\alpha/2}\hat{s}/|\bar{\theta}|$; where z is a standard normal quantile, $\hat{s}$ is the Monte Carlo standard error, and $\bar{\theta}$ is the estimated

posterior mean. If the relative halfwidth is greater than a target ratio, the test is rejected. Rejection of the stationarity or halfwidth test suggests that additional samples are needed.

Arguments

- `c` : sampler output on which to perform calculations.
- `alpha` : significance level for evaluations of stationarity tests and calculations of relative estimation interval halfwidths.
- `eps` : target ratio for the relative halfwidths.
- `etype` : method for computing Monte Carlo standard errors. See *`mcse()`* for options.
- `args...` : additional arguments to be passed to the `etype` method.

Value

A `ChainSummary` type object with parameters contained in the rows of the `value` field, and numbers of burn-in sequences to discard, whether the stationarity tests are passed (1 = yes, 0 = no), their p-values ($p > \alpha$ implies stationarity), posterior means, halfwidths of their $(1 - \alpha)100\%$ estimation intervals, and whether the halfwidth tests are passed (1 = yes, 0 = no) in the columns. Results are chain-specific.

Example

See the *Convergence Diagnostics* section of the tutorial.

Raftery and Lewis Diagnostic

`rafterydiag`(*c::Chains; q::Real=0.025*, *r::Real=0.005*, *s::Real=0.95*, *eps::Real=0.001*)

Compute the convergence diagnostic of Raftery and Lewis *[60][61]* for MCMC sampler output. The diagnostic is designed to determine the number of autocorrelated samples required to estimate a specified quantile θ_q, such that $\Pr(\theta \leq \theta_q) = q$, within a desired degree of accuracy. In particular, if $\hat{\theta}_q$ is the estimand and $\Pr(\theta \leq \hat{\theta}_q) = \hat{P}_q$ the estimated cumulative probability, then accuracy is specified in terms of r and s, where $\Pr(q - r < \hat{P}_q < q + r) = s$. Thinning may be employed in the calculation of the diagnostic to satisfy its underlying assumptions. However, users may not want to apply the same (or any) thinning when estimating posterior summary statistics because doing so results in a loss of information. Accordingly, sample sizes estimated by the diagnostic tend to be conservative (too large).

Arguments

- `c` : sampler output on which to perform calculations.
- `q` : posterior quantile of interest.
- `r` : margin of error for estimated cumulative probabilities.
- `s` : probability for the margin of error.
- `eps` : tolerance within which the probabilities of transitioning from initial to retained iterations are within the equilibrium probabilities for the chain. This argument determines the number of samples to discard as a burn-in sequence and is typically left at its default value.

Value

A `ChainSummary` type object with parameters contained in the rows of the `value` field, and thinning intervals employed, numbers of samples to discard as burn-in sequences, total numbers (N) to burn-in and retain, numbers of independent samples that would be needed ($Nmin$), and dependence factors ($N/Nmin$) in the columns. Results are chain-specific.

Example

See the *Convergence Diagnostics* section of the tutorial.

Posterior Summary Statistics

autocor(*c::Chains; lags::Vector=[1,5,10,50], relative::Bool=true*)
Compute lag-k autocorrelations for MCMC sampler output.

Arguments

- `c` : sampler output on which to perform calculations.
- `lags` : lags at which to compute autocorrelations.
- `relative` : whether the lags are relative to the thinning interval of the output (`true`) or relative to the absolute iteration numbers (`false`).

Value

A `ChainSummary` type object with model parameters indexed by the first dimension of `value`, lag-autocorrelations by the second, and chains by the third.

Example

See the *Posterior Summaries* section of the tutorial.

changerate(*c::Chains*)
Estimate the probability, or rate per iteration, $\Pr(\theta^i \neq \theta^{i-1})$ of a state space change for iterations $i = 2, \ldots, N$ in MCMC sampler output. Estimation is performed for each parameter univariately as well as for the full parameter vector multivariately. For continuous output generated from samplers, like Metropolis-Hastings, whose algorithms conditionally accept candidate draws, the probability can be viewed as the acceptance rate.

Arguments

- `c` : sampler output on which to perform calculations.

Value

A `ChainSummary` type object with parameters in the rows of the `value` field, and the estimated rates in the column. Results are for all chains combined.

Example

See the *Posterior Summaries* section of the tutorial.

cor(*c::Chains*)
Compute cross-correlations for MCMC sampler output.

Arguments

- `c` : sampler output on which to perform calculations.

Value

A `ChainSummary` type object with the first and second dimensions of the `value` field indexing the model parameters between which correlations. Results are for all chains combined.

Example

See the *Posterior Summaries* section of the tutorial.

describe(*c::Chains; q::Vector=[0.025, 0.25, 0.5, 0.75, 0.975], etype=:bm, args...*)
Compute summary statistics for MCMC sampler output.

Arguments

- `c` : sampler output on which to perform calculations.

- `q` : probabilities at which to calculate quantiles.
- `etype` : method for computing Monte Carlo standard errors. See *`mcse()`* for options.
- `args...` : additional arguments to be passed to the `etype` method.

Value

Results from calls to `summarystats(c, etype, args...)` and `quantile(c, q)` are printed for all chains combined, and a value of `nothing` is returned.

Example

See the *Posterior Summaries* section of the tutorial.

`hpd`(*c::Chains; alpha::Real=0.05*)

Compute highest posterior density (HPD) intervals of Chen and Shao *[14]* for MCMC sampler output. HPD intervals have the desirable property of being the smallest intervals that contain a given probability. However, their calculation assumes unimodal marginal posterior distributions, and they are not invariant to transformations of parameters like central (quantile-based) posterior intervals.

Arguments

- `c` : sampler output on which to perform calculations.
- `alpha` : the `100 * (1 - alpha)` % interval to compute.

Value

A `ChainSummary` type object with parameters contained in the rows of the `value` field, and lower and upper intervals in the first and second columns. Results are for all chains combined.

Example

See the *Posterior Summaries* section of the tutorial.

`mcse`(*x::Vector{T<:Real}*, *method::Symbol=:imse; args...*)

Compute Monte Carlo standard errors.

Arguments

- `x` : a time series of values on which to perform calculations.
- **`method`** [method used for the calculations. Options are]
  - `:bm` : batch means *[36]*, with optional argument `size::Integer=100` determining the number of sequential values to include in each batch. This method requires that the number of values in `x` is at least 2 times the batch size.
  - `:imse` : initial monotone sequence estimator *[34]*.
  - `:ipse` : initial positive sequence estimator *[34]*.
- `args...` : additional arguments for the calculation method.

Value

The numeric standard error value.

`quantile`(*c::Chains; q::Vector=[0.025, 0.25, 0.5, 0.75, 0.975]*)

Compute posterior quantiles for MCMC sampler output.

Arguments

- `c` : sampler output on which to perform calculations.
- `q` : probabilities at which to compute quantiles.

Value

A `ChainSummary` type object with parameters contained in the rows of the `value` field, and quantiles in the columns. Results are for all chains combined.

summarystats(*c::Chains; etype=:bm, args...*)

Compute posterior summary statistics for MCMC sampler output.

Arguments

- `c` : sampler output on which to perform calculations.
- `etype` : method for computing Monte Carlo standard errors. See *mcse()* for options.
- `args...` : additional arguments to be passed to the `etype` method.

Value

A `ChainSummary` type object with parameters in the rows of the `value` field; and the sample mean, standard deviation, standard error, Monte Carlo standard error, and effective sample size in the columns. Results are for all chains combined.

Model-Based Inference

dic(*c::Chains*)

Compute the Deviance Information Criterion (DIC) of Spiegelhalter et al. *[70]* and Gelman et al. *[27]* from MCMC sampler output.

Arguments

- `c` : sampler output from a model fit with the *mcmc()* function and for which all sampled nodes are monitored.

Value

A `ChainSummary` type object with DIC results from the methods of Spiegelhalter and Gelman in the first and second rows of the `value` field, and the DIC value and effective numbers of parameters in the first and second columns; where

$$\text{DIC} = -2\mathcal{L}(\bar{\Theta}) + 2p,$$

such that $\mathcal{L}(\bar{\Theta})$ is the log-likelihood of model outputs given the expected values of model parameters Θ, and p is the effective number of parameters. The latter is defined as $p_D = -2\bar{\mathcal{L}}(\Theta) + 2\mathcal{L}(\bar{\Theta})$ for the method of Spiegelhalter and as $p_V = \frac{1}{2}\,\text{var}(-2\mathcal{L}(\Theta))$ for the method of Gelman. Results are for all chains combined.

Example

See the *Posterior Summaries* section of the tutorial.

predict(*c::Chains, key::Symbol*)

Generate MCMC draws from a posterior predictive distribution.

Arguments

- `c`: sampler output from a model fit with the *mcmc()* function.
- `key`: name of an observed Stochastic model node for which to generate draws from its predictive distribution.

Value

A `Chain` object of simulated draws. For observed data node y, simulation is from the posterior predictive distribution

$$p(\tilde{y}|y) = \int p(\tilde{y}|\Theta)p(\Theta|y)d\Theta,$$

where $\tilde{y}$ is an unknown observation on the node, $p(\tilde{y}|\Theta)$ is the data likelihood, and $p(\Theta|y)$ is the posterior distribution of unobserved parameters Θ.

Example

See the *Pumps* example.

Plotting

plot (*c::Chains*, *ptype::Vector{Symbol}=[:trace*, *:density]; legend::Bool=false*, *args...*)
plot (*c::Chains*, *ptype::Symbol; legend::Bool=false*, *args...*)

Various plots to summarize a `Chains` object. Separate plots are produced for each parameter.

Arguments

- `c` : sampler output to plot.
- **`ptype`** [plot type(s). Options are]
  - `:autocor` : autocorrelation plots, with optional argument `maxlag::Integer=int(10*log10(length(c.range)))` determining the maximum autocorrelation lag to plot. Lags are plotted relative to the thinning interval of the output.
  - `:density` : density plots. Optional argument `trim::(Real,Real)=(.025,.975)` trims off lower and upper quantiles of density.
  - `:mean` : running mean plots.
  - `:trace` : trace plots.
- `legend` : whether to include legends in the plots to identify chain-specific results.
- `args...` : additional arguments to be passed to the `ptype` method, as described above.

Value

Returns a `Vector{Plot}` whose elements are individual parameter plots of the specified type if `ptype` is a symbol, and a `Matrix{Plot}` with plot types in the rows and parameters in the columns if `ptype` is a vector. The result can be displayed or saved to a file with `draw()`.

Note

Plots are created using the *Gadfly* package *[44]*.

Example

See the *Plotting* section of the tutorial.

draw (*p::Array{Plot}; fmt::Symbol=:svg*, *filename::String=""*, *width::MeasureOrNumber=8inch*, *height::MeasureOrNumber=8inch*, *nrow::Integer=3*, *ncol::Integer=2*, *byrow::Bool=true*, *ask::Bool=true*)

Draw plots produced by *`plot()`* into display grids containing a default of 3 rows and 2 columns of plots.

Arguments

- `p` : array of plots to be drawn. Elements of `p` are read in the order stored by **julia** (e.g. column-major order for matrices) and written to the display grid according to the `byrow` argument. Grids will be filled sequentially until all plots have been drawn.

- **fmt** [output format. Options are]
  - :pdf : Portable Document Format (.pdf).
  - :png : Portable Network Graphics (.png).
  - :ps : Postscript (.ps).
  - :svg : Scalable Vector Graphics (.svg).
- filename : an external file to which to save the display grids as they are drawn, or an empty string to draw to the display device (default). If a supplied external file name does not include a dot (.), then a hyphen followed by the grid sequence number and then the format extension will be appended automatically. In the case of multiple grids, the former file name behavior will write all grids to the single named file, but prompt users before advancing to the next grid and overwriting the file; the latter behavior will write each grid to a different file.
- width/height : grid widths/heights in cm, mm, inch, pt, or px units.
- nrow/ncol : number of rows/columns in the display grids.
- byrow : whether the display grids should be filled by row.
- ask : whether to prompt users before displaying subsequent grids to a single named file or the display device.

Value

Grids drawn to an external file or the display device.

Example

See the *Plotting* section of the tutorial.

2.5 Sampling Functions

A collection of functions are provided for simulating draws from from distributions that can be specified up to constants of proportionalities. Included are stand-alone functions as well as *Sampler* constructors for use with the *mcmc()* engine.

2.5.1 Adaptive Mixture Metropolis (AMM)

Implementation of the Roberts and Rosenthal *[64]* adaptive (multivariate) mixture Metropolis *[39][40][51]* sampler for simulating autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Stand-Alone Function

amm! (*v::AMMVariate*, *SigmaF::Cholesky{Float64}*, *logf::Function; adapt::Bool=true*)

Simulate one draw from a target distribution using an adaptive mixture Metropolis sampler. Parameters are assumed to be continuous and unconstrained.

Arguments

- v : current state of parameters to be simulated. When running the sampler in adaptive mode, the v argument in a successive call to the function should contain the tune field returned by the previous call.
- SigmaF : Cholesky factorization of the covariance matrix for the non-adaptive multivariate normal proposal distribution.

- `logf` : function to compute the log-transformed density (up to a normalizing constant) at `v.value`.
- `adapt` : whether to adaptively update the proposal distribution.

Value

Returns `v` updated with simulated values and associated tuning parameters.

Example

The following example samples parameters in a simple linear regression model. Details of the model specification and posterior distribution can be found in the *Supplement*.

```
################################################################################
## Linear Regression
##   y ~ N(b0 + b1 * x, s2)
##   b0, b1 ~ N(0, 1000)
##   s2 ~ invgamma(0.001, 0.001)
################################################################################

using Mamba

## Data
data = [
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
]

## Log-transformed Posterior(b0, b1, log(s2)) + Constant
logf = function(x)
   b0 = x[1]
   b1 = x[2]
   logs2 = x[3]
   r = data[:y] - b0 - b1 * data[:x]
   (-0.5 * length(data[:y]) - 0.001) * logs2 -
     (0.5 * dot(r, r) + 0.001) / exp(logs2) -
     0.5 * b0^2 / 1000 - 0.5 * b1^2 / 1000
end

## MCMC Simulation with Adaptive Multivariate Metopolis Sampling
n = 5000
burnin = 1000
sim = Chains(n, 3, names = ["b0", "b1", "s2"])
theta = AMMVariate([0.0, 0.0, 0.0])
SigmaF = cholfact(eye(3))
for i in 1:n
  amm!(theta, SigmaF, logf, adapt = (i <= burnin))
  sim[i,:,1] = [theta[1:2], exp(theta[3])]
end
describe(sim)
```

AMMVariate Type

Declaration

```
AMMVariate <:  VectorVariate
```

Fields

- `value::Vector{VariateType}` : vector of sampled values.
- `tune::AMMTune` : tuning parameters for the sampling algorithm.

Constructors

AMMVariate(*x::Vector{VariateType}*, *tune::AMMTune*)
AMMVariate(*x::Vector{VariateType}*, *tune=nothing*)

Construct a `AMMVariate` object that stores sampled values and tuning parameters for adaptive mixture Metropolis sampling.

Arguments

- `x` : vector of sampled values.
- `tune` : tuning parameters for the sampling algorithm. If `nothing` is supplied, parameters are set to their defaults.

Value

Returns a `AMMVariate` type object with fields pointing to the values supplied to arguments `x` and `tune`.

AMMTune Type

Declaration

```
type AMMTune
```

Fields

- `adapt::Bool` : whether the proposal distribution has been adaptively tuned.
- `beta::Real` : proportion of weight given to draws from the non-adaptive proposal with covariance factorization `SigmaF`, relative to draws from the adaptively tuned proposal with covariance factorization `SigmaLm`, during adaptive updating. Fixed at `beta = 0.05`.
- `m::Integer` : number of adaptive update iterations that have been performed.
- `Mv::Vector{Float64}` : running mean of draws `v` during adaptive updating. Used in the calculation of `SigmaLm`.
- `Mvv::Vector{Float64}` : running mean of `v * v'` during adaptive updating. Used in the calculation of `SigmaLm`.
- `scale::Real` : fixed value `2.38^2` in the factor (`scale / length(v)`) by which the adaptively updated covariance matrix is scaled—adopted from Gelman, Roberts, and Gilks *[28]*.
- `SigmaF::Cholesky{Float64}` : factorization of the non-adaptive covariance matrix.
- `SigmaLm::Matrix{Float64}` : lower-triangular factorization of the adaptively tuned covariance matrix.

Sampler Constructor

AMM(*params::Vector{Symbol}*, *Sigma::Matrix{T<:Real}; adapt::Symbol=:all*)

Construct a `Sampler` object for adaptive mixture Metropolis sampling. Parameters are assumed to be continuous, but may be constrained or unconstrained.

Arguments

- `params` : stochastic nodes to be updated with the sampler. Constrained parameters are mapped to unconstrained space according to transformations defined by the *Stochastic* `link()` function.
- `Sigma` : covariance matrix for the non-adaptive multivariate normal proposal distribution. The covariance matrix is relative to the unconstrained parameter space, where candidate draws are generated.
- **`adapt`** [type of adaptation phase. Options are]
  - `:all` : adapt proposal during all iterations.
  - `:burnin` : adapt proposal during burn-in iterations.
  - `:none` : no adaptation (multivariate Metropolis sampling with fixed proposal).

Value

Returns a `Sampler` type object.

Example

See the *Examples* section.

2.5.2 Adaptive Metropolis within Gibbs (AMWG)

Implementation of a Metropolis-within-Gibbs sampler *[51][64][74]* for iteratively simulating autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Stand-Alone Function

amwg!(*v::AMWGVariate*, *sigma::Vector{Float64}*, *logf::Function; adapt::Bool=true*, *batchsize::Integer=50*, *target::Real=0.44*)

Simulate one draw from a target distribution using an adaptive Metropolis-within-Gibbs sampler. Parameters are assumed to be continuous and unconstrained.

Arguments

- `v` : current state of parameters to be simulated. When running the sampler in adaptive mode, the `v` argument in a successive call to the function should contain the `tune` field returned by the previous call.
- `sigma` : initial standard deviations for the univariate normal proposal distributions.
- `logf` : function to compute the log-transformed density (up to a normalizing constant) at `v.value`.
- `adapt` : whether to adaptively update the proposal distribution.
- `batchsize` : number of samples that must be newly accumulated before applying an adaptive update to the proposal distributions.
- `target` : a target acceptance rate for the adaptive algorithm.

Value

Returns `v` updated with simulated values and associated tuning parameters.

Example

The following example samples parameters in a simple linear regression model. Details of the model specification and posterior distribution can be found in the *Supplement*.

```
################################################################################
## Linear Regression
##   y ~ N(b0 + b1 * x, s2)
##   b0, b1 ~ N(0, 1000)
##   s2 ~ invgamma(0.001, 0.001)
################################################################################

using Mamba

## Data
data = [
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
]

## Log-transformed Posterior(b0, b1, log(s2)) + Constant
logf = function(x)
   b0 = x[1]
   b1 = x[2]
   logs2 = x[3]
   r = data[:y] - b0 - b1 * data[:x]
   (-0.5 * length(data[:y]) - 0.001) * logs2 -
     (0.5 * dot(r, r) + 0.001) / exp(logs2) -
     0.5 * b0^2 / 1000 - 0.5 * b1^2 / 1000
end

## MCMC Simulation with Adaptive Metopolis-within-Gibbs Sampling
n = 5000
burnin = 1000
sim = Chains(n, 3, names = ["b0", "b1", "s2"])
theta = AMWGVariate([0.0, 0.0, 0.0])
sigma = ones(3)
for i in 1:n
  amwg!(theta, sigma, logf, adapt = (i <= burnin))
  sim[i,:,1] = [theta[1:2], exp(theta[3])]
end
describe(sim)
```

AMWGVariate Type

Declaration

```
AMWGVariate <:  VectorVariate
```

Fields

- `value::Vector{VariateType}` : vector of sampled values.
- `tune::AMWGTune` : tuning parameters for the sampling algorithm.

Constructors

AMWGVariate(*x::Vector{VariateType}*, *tune::AMWGTune*)
AMWGVariate(*x::Vector{VariateType}*, *tune=nothing*)

Construct a `AMWGVariate` object that stores sampled values and tuning parameters for adaptive Metropolis-within-Gibbs sampling.

Arguments

- `x` : vector of sampled values.
- `tune` : tuning parameters for the sampling algorithm. If `nothing` is supplied, parameters are set to their defaults.

Value

Returns a `AMWGVariate` type object with fields pointing to the values supplied to arguments `x` and `tune`.

AMWGTune Type

Declaration

```
type AMWGTune
```

Fields

- `adapt::Bool` : whether the proposal distribution has been adaptively tuned.
- `accept::Vector{Integer}` : number of accepted candidate draws generated for each element of the parameter vector during adaptive updating.
- `batchsize::Integer` : number of samples that must be accumulated before applying an adaptive update to the proposal distributions.
- `m::Integer` : number of adaptive update iterations that have been performed.
- `sigma::Vector{Float64}` : updated values of the proposal standard deviations if `adapt = true`, and the user-defined values otherwise.
- `target::Real` : target acceptance rate for the adaptive algorithm.

Sampler Constructor

AMWG(*params::Vector{Symbol}*, *sigma::Vector{T<:Real}*; *adapt::Symbol=:all*, *batchsize::Integer=50*, *target::Real=0.44*)

Construct a `Sampler` object for adaptive Metropolis-within-Gibbs sampling. Parameters are assumed to be continuous, but may be constrained or unconstrained.

Arguments

- `params` : stochastic nodes to be updated with the sampler. Constrained parameters are mapped to unconstrained space according to transformations defined by the *Stochastic* `link()` function.
- `sigma` : initial standard deviations for the univariate normal proposal distributions. The standard deviations are relative to the unconstrained parameter space, where candidate draws are generated.
- **`adapt`** [type of adaptation phase. Options are]

- `:all` : adapt proposals during all iterations.
- `:burnin` : adapt proposals during burn-in iterations.
- `:none` : no adaptation (Metropolis-within-Gibbs sampling with fixed proposals).

•`batchsize` : number of samples that must be accumulated before applying an adaptive update to the proposal distributions.

•`target` : a target acceptance rate for the algorithm.

Value

Returns a `Sampler` type object.

Example

See the *Examples* section.

2.5.3 Direct Grid Sampler (DGS)

Implementation of a sampler for the simulation of draws from discrete univariate distributions with finite supports. Draws are simulated directly from the individual full conditional probability mass functions for each of the specified model parameters.

Sampler Constructor

DGS (*params::Vector{Symbol}*)

Construct a `Sampler` object for direct grid sampling. Parameters are assumed to have discrete uniform distributions with finite supports.

Arguments

•`params` : stochastic nodes to be updated with the sampler.

Value

Returns a `Sampler` type object.

Example

See the *Eyes* example.

2.5.4 Missing Values Sampler (MISS)

A sampler to simulate missing output values from their likelihood distributions.

Sampler Constructor

MISS (*params::Vector{Symbol}*)

Construct a `Sampler` object to sampling missing output values. The constructor should only be used to sample stochastic nodes upon which no other stochastic node depends. So-called 'output nodes' can be identified with the *`keys()`* function. Moreover, when the `MISS` constructor is included in a vector of `Sampler` objects to define a sampling scheme, it should be positioned at the beginning of the vector. This ensures that missing output values are updated before any other samplers are executed.

Arguments

•`params` : stochastic nodes that contain missing values (`NaN`) to be updated with the sampler.

Value

Returns a `Sampler` type object.

Example

See the *Bones* and *Mice* examples.

2.5.5 No-U-Turn Sampler (NUTS)

Implementation of the NUTS extension (algorithm 6) *[42]* to Hamiltonian Monte Carlo *[53]* for simulating autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Stand-Alone Functions

`nutsepsilon`(*v::NUTSVariate*, *fx::Function*)

Generate an initial value for the step size parameter of the No-U-Turn sampler. Parameters are assumed to be continuous and unconstrained.

Arguments

- `v` : the current state of parameters to be simulated.
- `fx` : function to compute the log-transformed density (up to a normalizing constant) and gradient vector at `v.value`, and to return the respective results as a tuple.

Value

A numeric step size value.

`nuts!`(*v::NUTSVariate*, *epsilon::Real*, *fx::Function; adapt::Bool=false*, *target::Real=0.6*)

Simulate one draw from a target distribution using the No-U-Turn sampler. Parameters are assumed to be continuous and unconstrained.

Arguments

- `v` : current state of parameters to be simulated. When running the sampler in adaptive mode, the `v` argument in a successive call to the function should contain the `tune` field returned by the previous call.
- `epsilon` : the NUTS algorithm step size parameter.
- `fx` : function to compute the log-transformed density (up to a normalizing constant) and gradient vector at `v.value`, and to return the respective results as a tuple.
- `adapt` : whether to adaptively update the `epsilon` step size parameter.
- `target` : a target acceptance rate for the algorithm.

Value

Returns `v` updated with simulated values and associated tuning parameters.

Example

The following example samples parameters in a simple linear regression model. Details of the model specification and posterior distribution can be found in the *Supplement*.

```
################################################################################
## Linear Regression
##   y ~ N(b0 + b1 * x, s2)
##   b0, b1 ~ N(0, 1000)
##   s2 ~ invgamma(0.001, 0.001)
```

```
################################################################################

using Mamba

## Data
data = [
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
]

## Log-transformed Posterior(b0, b1, log(s2)) + Constant and Gradient Vector
fx = function(x)
  b0 = x[1]
  b1 = x[2]
  logs2 = x[3]
  r = data[:y] - b0 - b1 * data[:x]
  logf = (-0.5 * length(data[:y]) - 0.001) * logs2 -
           (0.5 * dot(r, r) + 0.001) / exp(logs2) -
           0.5 * b0^2 / 1000 - 0.5 * b1^2 / 1000
  grad = [
    sum(r) / exp(logs2) - b0 / 1000,
    sum(data[:x] .* r) / exp(logs2) - b1 / 1000,
    -0.5 * length(data[:y]) - 0.001 + (0.5 * dot(r, r) + 0.001) / exp(logs2)
  ]
  logf, grad
end

## MCMC Simulation with No-U-Turn Sampling
n = 5000
burnin = 1000
sim = Chains(n, 3, start = (burnin + 1), names = ["b0", "b1", "s2"])
theta = NUTSVariate([0.0, 0.0, 0.0])
epsilon = nutsepsilon(theta, fx)
for i in 1:n
  nuts!(theta, epsilon, fx, adapt = (i <= burnin))
  if i > burnin
    sim[i,:,1] = [theta[1:2], exp(theta[3])]
  end
end
describe(sim)
```

NUTSVariate Type

Declaration

```
NUTSVariate <:  VectorVariate
```

Fields

- `value::Vector{VariateType}` : vector of sampled values.
- `tune::NUTSTune` : tuning parameters for the sampling algorithm.

Constructors

NUTSVariate (*x::Vector{VariateType}*, *tune::NUTSTune*)
NUTSVariate (*x::Vector{VariateType}*, *tune=nothing*)

Construct a `NUTSVariate` object that stores sampled values and tuning parameters for No-U-Turn sampling.

Arguments

- `x` : vector of sampled values.
- `tune` : tuning parameters for the sampling algorithm. If `nothing` is supplied, parameters are set to their defaults.

Value

Returns a `NUTSVariate` type object with fields pointing to the values supplied to arguments `x` and `tune`.

NUTSTune Type

Declaration

```
type NUTSTune
```

Fields

- `adapt::Bool` : whether the proposal distribution has been adaptively tuned.
- `alpha::Float64` : cumulative acceptance probabilities α from leapfrog steps.
- `epsilon::Float64` : updated value of the step size parameter $\epsilon_m = \exp\left(\mu - \sqrt{m}\bar{H}_m/\gamma\right)$ if `adapt = true`, and the user-defined value otherwise.
- `epsbar::Float64` : dual averaging parameter, defined as $\bar{\epsilon}_m = \exp\left(m^{-\kappa}\log(\epsilon_m) + (1 - m^{-\kappa})\log(\bar{\epsilon}_{m-1})\right)$.
- `gamma::Float64` : dual averaging parameter, fixed at $\gamma = 0.05$.
- `Hbar::Float64` : dual averaging parameter, defied as $\bar{H}_m = \left(1 - \frac{1}{m+t_0}\right)\bar{H}_{m-1} + \frac{1}{m+t_0}\left(\text{target} - \frac{\alpha}{n_\alpha}\right)$.
- `kappa::Float64` : dual averaging parameter, fixed at $\kappa = 0.05$.
- `m::Integer` : number of adaptive update iterations m that have been performed.
- `mu::Float64` : dual averaging parameter, defined as $\mu = \log(10\epsilon_0)$.
- `nalpha::Integer` : the total number n_α of leapfrog steps performed.
- `t0::Float64` : dual averaging parameter, fixed at $t_0 = 10$.
- `target::Float64` : target acceptance rate for the adaptive algorithm.

Sampler Constructor

NUTS (*params::Vector{Symbol}; dtype::Symbol=:forward*, *target::Real=0.6*)

Construct a `Sampler` object for No-U-Turn sampling, with the algorithm's step size parameter adaptively tuned during burn-in iterations. Parameters are assumed to be continuous, but may be constrained or unconstrained.

Arguments

- `params` : stochastic nodes to be updated with the sampler. Constrained parameters are mapped to unconstrained space according to transformations defined by the *Stochastic* `link()` function.
- **`dtype`** [type of differentiation for gradient calculations. Options are]
  - `:central` : central differencing.
  - `:forward` : forward differencing.
- `target` : a target acceptance rate for the algorithm.

Value

Returns a `Sampler` type object.

Example

See the *Examples* section.

2.5.6 Shrinkage Slice (Slice)

Implementation of the shrinkage slice sampler of Neal *[52]* for simulating autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Stand-Alone Function

`slice!`(*v::SliceVariate*, *width::Vector{Float64}*, *logf::Function*, *stype::Symbol=:multivar*)

Simulate one draw from a target distribution using a shrinkage slice sampler. Parameters are assumed to be continuous, but may be constrained or unconstrained.

Arguments

- `v` : current state of parameters to be simulated.
- `width` : vector of the same length as `v`, defining initial widths of a hyperrectangle from which to simulate values.
- `logf` : function to compute the log-transformed density (up to a normalizing constant) at `v.value`.
- **`stype`** [sampler type. Options are]
  - `:multivar` : Joint multivariate sampling of parameters.
  - `:univar` : Sequential univariate sampling.

Value

Returns `v` updated with simulated values and associated tuning parameters.

Example

The following example samples parameters in a simple linear regression model. Details of the model specification and posterior distribution can be found in the *Supplement*.

```
#############################################################################
## Linear Regression
##   y ~ N(b0 + b1 * x, s2)
##   b0, b1 ~ N(0, 1000)
##   s2 ~ invgamma(0.001, 0.001)
#############################################################################

using Mamba
```

```
## Data
data = [
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
]

## Log-transformed Posterior(b0, b1, log(s2)) + Constant
logf = function(x)
   b0 = x[1]
   b1 = x[2]
   logs2 = x[3]
   r = data[:y] - b0 - b1 * data[:x]
   (-0.5 * length(data[:y]) - 0.001) * logs2 -
     (0.5 * dot(r, r) + 0.001) / exp(logs2) -
     0.5 * b0^2 / 1000 - 0.5 * b1^2 / 1000
end

## MCMC Simulation with Multivariate Slice Sampling
n = 5000
sim1 = Chains(n, 3, names = ["b0", "b1", "s2"])
theta = SliceVariate([0.0, 0.0, 0.0])
width = [1.0, 1.0, 2.0]
for i in 1:n
  slice!(theta, width, logf, :multivar)
  sim1[i,:,1] = [theta[1:2], exp(theta[3])]
end
describe(sim1)

## MCMC Simulation with Univariate Slice Sampling
n = 5000
sim2 = Chains(n, 3, names = ["b0", "b1", "s2"])
theta = SliceVariate([0.0, 0.0, 0.0])
width = [1.0, 1.0, 2.0]
for i in 1:n
  slice!(theta, width, logf, :univar)
  sim2[i,:,1] = [theta[1:2], exp(theta[3])]
end
describe(sim2)
```

SliceVariate Type

Declaration

`SliceVariate <: VectorVariate`

Fields

- `value::Vector{VariateType}` : vector of sampled values.
- `tune::SliceTune` : tuning parameters for the sampling algorithm.

Constructors

`SliceVariate`(*x::Vector{VariateType}*, *tune::SliceTune*)

SliceVariate(*x::Vector{VariateType}*, *tune=nothing*)
Construct a `SliceVariate` object that stores sampled values and tuning parameters for slice sampling.

Arguments

- `x` : vector of sampled values.
- `tune` : tuning parameters for the sampling algorithm. If `nothing` is supplied, parameters are set to their defaults.

Value

Returns a `SliceVariate` type object with fields pointing to the values supplied to arguments `x` and `tune`.

SliceTune Type

Declaration

```
type SliceTune
```

Fields

- `width::Vector{Float64}` : vector of initial widths defining hyperrectangles from which to simulate values.

Sampler Constructor

Slice(*params::Vector{Symbol}*, *width::Vector{T<:Real}*, *stype::Symbol=:multivar; transform::Bool=false*)
Construct a `Sampler` object for shrinkage slice sampling. Parameters are assumed to be continuous, but may be constrained or unconstrained.

Arguments

- `params` : stochastic nodes to be updated with the sampler.
- `width` : vector of the same length as the combined elements of nodes `params`, defining initial widths of a hyperrectangle from which to simulate values.
- **`stype`** [sampler type. Options are]
  - `:multivar` : Joint multivariate sampling of parameters.
  - `:univar` : Sequential univariate sampling.
- `transform` : whether to sample parameters on the link-transformed scale (unconstrained parameter space). If `true`, then constrained parameters are mapped to unconstrained space according to transformations defined by the *Stochastic* `link()` function, and `width` is interpreted as being relative to the unconstrained parameter space. Otherwise, sampling is relative to the untransformed space.

Value

Returns a `Sampler` type object.

Example

See the *Examples* section.

2.6 Examples

The following examples are taken from OpenBUGS *[38]*, and were used in the development and testing of *Mamba*. They are provide to illustrate model specification and fitting with the package, and how its syntax compares to other Bayesian modelling software.

2.6.1 Rats: A Normal Hierarchical Model

An example from OpenBUGS *[38]* and section 6 of Gelfand *et al.* *[25]* concerning 30 rats whose weights were measured at each of five consecutive weeks.

Model

Weights are modeled as

$$\begin{aligned} y_{i,j} &\sim \text{Normal}\left(\alpha_i + \beta_i(x_j - \bar{x}), \sigma_c\right) \qquad i = 1, \ldots, 30; j = 1, \ldots, 5 \\ \alpha_i &\sim \text{Normal}(\mu_\alpha, \sigma_\alpha) \\ \beta_i &\sim \text{Normal}(\mu_\beta, \sigma_\beta) \\ \mu_\alpha, \mu_\beta &\sim \text{Normal}(0, 1000) \\ \sigma_\alpha^2, \sigma_\beta^2, \sigma_c^2 &\sim \text{InverseGamma}(0.001, 0.001), \end{aligned}$$

where $y_{i,j}$ is repeated weight measurement j on rat i, and x_j is the day on which the measurement was taken.

Analysis Program

```
using Mamba

## Data
rats = (Symbol => Any)[
  :y =>
    [151, 199, 246, 283, 320,
     145, 199, 249, 293, 354,
     147, 214, 263, 312, 328,
     155, 200, 237, 272, 297,
     135, 188, 230, 280, 323,
     159, 210, 252, 298, 331,
     141, 189, 231, 275, 305,
     159, 201, 248, 297, 338,
     177, 236, 285, 350, 376,
     134, 182, 220, 260, 296,
     160, 208, 261, 313, 352,
     143, 188, 220, 273, 314,
     154, 200, 244, 289, 325,
     171, 221, 270, 326, 358,
     163, 216, 242, 281, 312,
     160, 207, 248, 288, 324,
     142, 187, 234, 280, 316,
     156, 203, 243, 283, 317,
     157, 212, 259, 307, 336,
     152, 203, 246, 286, 321,
     154, 205, 253, 298, 334,
     139, 190, 225, 267, 302,
```

```
    146, 191, 229, 272, 302,
    157, 211, 250, 285, 323,
    132, 185, 237, 286, 331,
    160, 207, 257, 303, 345,
    169, 216, 261, 295, 333,
    157, 205, 248, 289, 316,
    137, 180, 219, 258, 291,
    153, 200, 244, 286, 324],
  :x => [8.0, 15.0, 22.0, 29.0, 36.0]
]
rats[:xbar] = mean(rats[:x])
rats[:N] = size(rats[:y], 1)
rats[:T] = size(rats[:y], 2)

rats[:rat] = Integer[div(i - 1, 5) + 1 for i in 1:150]
rats[:week] = Integer[(i - 1) % 5 + 1 for i in 1:150]
rats[:X] = rats[:x][rats[:week]]
rats[:Xm] = rats[:X] - rats[:xbar]


## Model Specification

model = Model(

  y = Stochastic(1,
    @modelexpr(alpha, beta, rat, Xm, s2_c,
      begin
        mu = alpha[rat] + beta[rat] .* Xm
        MvNormal(mu, sqrt(s2_c))
      end
    ),
    false
  ),

  alpha = Stochastic(1,
    @modelexpr(mu_alpha, s2_alpha,
      Normal(mu_alpha, sqrt(s2_alpha))
    ),
    false
  ),

  alpha0 = Logical(
    @modelexpr(mu_alpha, xbar, mu_beta,
      mu_alpha - xbar * mu_beta
    )
  ),

  mu_alpha = Stochastic(
    :(Normal(0.0, 1000)),
    false
  ),

  s2_alpha = Stochastic(
    :(InverseGamma(0.001, 0.001)),
    false
  ),

  beta = Stochastic(1,
```

```
    @modelexpr(mu_beta, s2_beta,
      Normal(mu_beta, sqrt(s2_beta))
    ),
    false
  ),

  mu_beta = Stochastic(
    :(Normal(0.0, 1000))
  ),

  s2_beta = Stochastic(
    :(InverseGamma(0.001, 0.001)),
    false
  ),

  s2_c = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)


## Initial Values
inits = [
  [:y => rats[:y], :alpha => fill(250, 30), :beta => fill(6, 30),
   :mu_alpha => 150, :mu_beta => 10, :s2_c => 1, :s2_alpha => 1,
   :s2_beta => 1],
  [:y => rats[:y], :alpha => fill(20, 30), :beta => fill(0.6, 30),
   :mu_alpha => 15, :mu_beta => 1, :s2_c => 10, :s2_alpha => 10,
   :s2_beta => 10]
]


## Sampling Scheme
scheme = [Slice([:s2_c], [10.0]),
          AMWG([:alpha], fill(100.0, 30)),
          Slice([:mu_alpha, :s2_alpha], [100.0, 10.0], :univar),
          AMWG([:beta], ones(30)),
          Slice([:mu_beta, :s2_beta], [1.0, 1.0], :univar)]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, rats, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
           Mean       SD      Naive SE     MCSE       ESS
mu_beta   6.18623 0.1072236 0.001238112 0.00215008 2486.9858
 alpha0 106.57458 3.6532621 0.042184237 0.05738100 4053.4550
```

```
   s2_c  37.01996 5.5173157 0.063708474 0.18973116  845.6260

Quantiles:
          2.5%     25.0%     50.0%     75.0%     97.5%
mu_beta  5.97845   6.11535   6.18663   6.25654   6.40124
 alpha0 99.36560 104.13484 106.54939 109.01257 113.78295
   s2_c 27.69329  33.09522  36.58565  40.29530  49.37235
```

2.6.2 Pumps: Gamma-Poisson Hierarchical Model

An example from OpenBUGS *[38]* and George *et al.* *[32]* concerning the number of failures of 10 power plant pumps.

Model

Pump failure are modelled as

$$
\begin{aligned}
y_i &\sim \text{Poisson}(\theta_i t_i) \qquad i = 1, \ldots, 10 \\
\theta_i &\sim \text{Gamma}(\alpha, 1/\beta) \\
\alpha &\sim \text{Gamma}(1, 1) \\
\beta &\sim \text{Gamma}(0.1, 1),
\end{aligned}
$$

where y_i is the number of times that pump i failed, and t_i is the operation time of the pump (in 1000s of hours).

Analysis Program

```
using Mamba

## Data
pumps = (Symbol => Any)[
  :y => [5, 1, 5, 14, 3, 19, 1, 1, 4, 22],
  :t => [94.3, 15.7, 62.9, 126, 5.24, 31.4, 1.05, 1.05, 2.1, 10.5]
]
pumps[:N] = length(pumps[:y])


## Model Specification

model = Model(

  y = Stochastic(1,
    @modelexpr(theta, t, N,
      Distribution[
        begin
          lambda = theta[i] * t[i]
          Poisson(lambda)
        end
        for i in 1:N
      ]
    ),
    false
  ),

  theta = Stochastic(1,
```

```
    @modelexpr(alpha, beta,
      Gamma(alpha, 1 / beta)
    ),
    true
  ),

  alpha = Stochastic(
    :(Exponential(1.0))
  ),

  beta = Stochastic(
    :(Gamma(0.1, 1.0))
  )

)


## Initial Values
inits = [
  [:y => pumps[:y], :alpha => 1.0, :beta => 1.0,
   :theta => rand(Gamma(1.0, 1.0), pumps[:N])],
  [:y => pumps[:y], :alpha => 10.0, :beta => 10.0,
   :theta => rand(Gamma(10.0, 10.0), pumps[:N])]
]


## Sampling Scheme
scheme = [Slice([:alpha, :beta], [1.0, 1.0], :univar),
          Slice([:theta], ones(pumps[:N]), :univar)]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, pumps, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)


## Posterior Predictive Distribution
ppd = predict(sim, :y)
describe(ppd)
```

Results

```
## MCMC Simulations

Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
             Mean        SD       Naive SE      MCSE       ESS
 theta[1] 0.059728 0.02511183 0.000289966 0.000357021 4947.3086
 theta[2] 0.099763 0.07733395 0.000892976 0.001131733 4669.3099
 theta[3] 0.088984 0.03795487 0.000438265 0.000510879 5519.5005
 theta[4] 0.116179 0.03056397 0.000352922 0.000382914 6371.1555
 theta[5] 0.603654 0.32235984 0.003722291 0.006123252 2771.5172
```

```
 theta[6] 0.608916 0.13940546 0.001609716 0.001556592 7500.0000
 theta[7] 0.894307 0.69157282 0.007985595 0.030109699  527.5492
 theta[8] 0.884304 0.72603888 0.008383575 0.025229264  828.1530
 theta[9] 1.562347 0.75587826 0.008728130 0.023807874 1008.0045
theta[10] 1.987852 0.42786838 0.004940598 0.010220427 1752.5979
    alpha 0.679247 0.26700149 0.003083068 0.007082723 1421.1072
     beta 0.893892 0.52823112 0.006099488 0.018254904  837.3150

Quantiles:
             2.5%     25.0%     50.0%     75.0%     97.5%
 theta[1] 0.0210364 0.0413043 0.0561389 0.0742908 0.117973
 theta[2] 0.0077032 0.0433944 0.0812108 0.1359765 0.296096
 theta[3] 0.0309151 0.0616544 0.0841478 0.1106976 0.178913
 theta[4] 0.0635652 0.0944482 0.1137455 0.1351764 0.182949
 theta[5] 0.1481702 0.3696211 0.5496033 0.7757793 1.380921
 theta[6] 0.3666778 0.5100374 0.5997593 0.6959183 0.915158
 theta[7] 0.0781523 0.3743788 0.7229133 1.2273922 2.689196
 theta[8] 0.0734788 0.3736729 0.6900152 1.1995264 2.729814
 theta[9] 0.4625337 1.0045124 1.4399533 1.9909942 3.359561
theta[10] 1.2345761 1.6807668 1.9568796 2.2550151 2.907759
    alpha 0.2785980 0.4856731 0.6407352 0.8256277 1.324442
     beta 0.1770170 0.5026369 0.7839917 1.1795453 2.199956

## Posterior Predictive Distribution

Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
         Mean      SD    Naive SE     MCSE       ESS
 y[1]  5.668000 3.41520 0.03943534 0.04402538 6017.6410
 y[2]  1.533467 1.70394 0.01967537 0.02404013 5023.8127
 y[3]  5.568533 3.38075 0.03903758 0.04069074 6902.9658
 y[4] 14.647333 5.45170 0.06295083 0.06597961 6827.2327
 y[5]  3.176400 2.46365 0.02844775 0.03740773 4337.4491
 y[6] 19.165467 6.20492 0.07164830 0.06647720 7500.0000
 y[7]  0.938400 1.20349 0.01389676 0.03474854 1199.5408
 y[8]  0.937200 1.22294 0.01412133 0.02790754 1920.2997
 y[9]  3.289333 2.39193 0.02761960 0.05164792 2144.8171
y[10] 20.863467 6.39303 0.07382033 0.11124836 3302.3719

Quantiles:
      2.5% 25.0% 50.0% 75.0% 97.5%
 y[1]    1     3     5     8    14
 y[2]    0     0     1     2     6
 y[3]    1     3     5     7    14
 y[4]    5    11    14    18    26
 y[5]    0     1     3     4     9
 y[6]    9    15    19    23    33
 y[7]    0     0     1     1     4
 y[8]    0     0     1     1     4
 y[9]    0     2     3     5     9
y[10]   10    16    20    25    35
```

2.6.3 Dogs: Loglinear Model for Binary Data

An example from OpenBUGS *[38]*, Lindley and Smith *[47]*, and Kalbfleisch *[45]* concerning the Solomon-Wynne experiment on dogs. In the experiment, 30 dogs were subjected to 25 trials. On each trial, a barrier was raised, and an electric shock was administered 10 seconds later if the dog did not jump the barrier.

Model

Failures to jump the barriers in time are modelled as

$$\begin{aligned} y_{i,j} &= \text{Bernoulli}(\pi_{i,j}) \qquad i = 1, \ldots, 30; j = 2, \ldots, 25 \\ \log(\pi_{i,j}) &= \alpha x_{i,j-1} + \beta(j - 1 - x_{i,j-1}) \\ \alpha, \beta &\sim \text{Flat}(-\infty, -0.00001) \end{aligned}$$

where $y_{i,j} = 1$ if dog i fails to jump the barrier before the shock on trial j, and 0 otherwise; $x_{i,j-1}$ is the number of successful jumps prior to trial j; and $\pi_{i,j}$ is the probability of failure.

Analysis Program

```
using Mamba

## Data
dogs = (Symbol => Any)[
  :Y =>
    [0 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 0 1 1 0 1 1 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1
     0 1 1 0 0 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 0 0 1 1 1 1 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 0 0 0 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 0 1 0 1 0 1 1 0 1 0 0 0 1 1 1 1 1 0 1 1 0
     0 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 0 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 1 1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 1 0 1 0 0 0 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1
     0 1 0 0 0 0 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 1 1 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1
     0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 1 1 0 1 1 1 1 1
     0 0 0 0 0 0 0 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1
     0 0 1 0 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     0 0 0 0 1 1 0 0 1 1 1 0 1 0 1 0 1 0 1 1 1 1 1 1 1
     0 0 0 0 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1]
]
```

```
dogs[:Dogs] = size(dogs[:Y], 1)
dogs[:Trials] = size(dogs[:Y], 2)

dogs[:xa] = mapslices(cumsum, dogs[:Y], 2)
dogs[:xs] = mapslices(x -> [1:25] - x, dogs[:xa], 2)
dogs[:y] = 1 - dogs[:Y][:, 2:25]


## Model Specification

model = Model(

  y = Stochastic(2,
    @modelexpr(Dogs, Trials, alpha, xa, beta, xs,
      Distribution[
        begin
          p = exp(alpha * xa[i,j] + beta * xs[i,j])
          Bernoulli(p)
        end
        for i in 1:Dogs, j in 1:Trials-1
      ]
    ),
    false
  ),

  alpha = Stochastic(
    :(Truncated(Flat(), -Inf, -1e-5))
  ),

  A = Logical(
    @modelexpr(alpha,
      exp(alpha)
    )
  ),

  beta = Stochastic(
    :(Truncated(Flat(), -Inf, -1e-5))
  ),

  B = Logical(
    @modelexpr(beta,
      exp(beta)
    )
  )

)


## Initial Values
inits = [
  [:y => dogs[:y], :alpha => -1, :beta => -1],
  [:y => dogs[:y], :alpha => -2, :beta => -2]
]


## Sampling Scheme
scheme = [Slice([:alpha, :beta], [1.0, 1.0])]
setsamplers!(model, scheme)
```

```
## MCMC Simulations
sim = mcmc(model, dogs, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
         Mean         SD         Naive SE        MCSE         ESS
    A  0.783585 0.018821132 0.00021732772 0.00034707278 2940.6978
    B  0.924234 0.010903201 0.00012589932 0.00016800825 4211.5969
alpha -0.244165 0.024064384 0.00027787158 0.00044330447 2946.7635
 beta -0.078860 0.011812880 0.00013640339 0.00018220980 4203.0847

Quantiles:
         2.5%       25.0%      50.0%      75.0%      97.5%
    A  0.7462511  0.7709223  0.7836328  0.7962491  0.8197714
    B  0.9018854  0.9170128  0.9245159  0.9319588  0.9445650
alpha -0.2926932 -0.2601676 -0.2438148 -0.2278432 -0.1987298
 beta -0.1032678 -0.0866338 -0.0784850 -0.0704666 -0.0570308
```

2.6.4 Seeds: Random Effect Logistic Regression

An example from OpenBUGS *[38]*, Crowder *[17]*, and Breslow and Clayton *[9]* concerning the proportion of seeds that germinated on each of 21 plates arranged according to a 2 by 2 factorial layout by seed and type of root extract.

Model

Germinations are modelled as

$$\begin{aligned} r_i &\sim \text{Binomial}(n_i, p_i) \qquad i = 1, \ldots, 21 \\ \text{logit}(p_i) &= \alpha_0 + \alpha_1 x_{1i} + \alpha_2 x_{2i} + \alpha_{12} x_{1i} x_{2i} + b_i \\ b_i &\sim \text{Normal}(0, \sigma) \\ \alpha_0, \alpha_1, \alpha_2, \alpha_{12} &\sim \text{Normal}(0, 1000) \\ \sigma^2 &\sim \text{InverseGamma}(0.001, 0.001), \end{aligned}$$

where r_i are the number of seeds, out of n_i, that germinate on plate i; and x_{1i} and x_{2i} are the seed type and root extract.

Analysis Program

```
using Mamba

## Data
seeds = (Symbol => Any)[
  :r => [10, 23, 23, 26, 17, 5, 53, 55, 32, 46, 10, 8, 10, 8, 23, 0, 3, 22, 15,
```

```
          32, 3],
  :n => [39, 62, 81, 51, 39, 6, 74, 72, 51, 79, 13, 16, 30, 28, 45, 4, 12, 41,
         30, 51, 7],
  :x1 => [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
  :x2 => [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
]
seeds[:N] = length(seeds[:r])


## Model Specification

model = Model(

  r = Stochastic(1,
    @modelexpr(alpha0, alpha1, x1, alpha2, x2, alpha12, b, n, N,
      Distribution[
        begin
          p = invlogit(alpha0 + alpha1 * x1[i] + alpha2 * x2[i] +
                       alpha12 * x1[i] * x2[i] + b[i])
          Binomial(n[i], p)
        end
        for i in 1:N
      ]
    ),
    false
  ),

  b = Stochastic(1,
    @modelexpr(s2,
      Normal(0, sqrt(s2))
    ),
    false
  ),

  alpha0 = Stochastic(
    :(Normal(0, 1000))
  ),

  alpha1 = Stochastic(
    :(Normal(0, 1000))
  ),

  alpha2 = Stochastic(
    :(Normal(0, 1000))
  ),

  alpha12 = Stochastic(
    :(Normal(0, 1000))
  ),

  s2 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)


## Initial Values
```

```
inits = [
  [:r => seeds[:r], :alpha0 => 0, :alpha1 => 0, :alpha2 => 0,
   :alpha12 => 0, :s2 => 0.01, :b => zeros(seeds[:N])],
  [:r => seeds[:r], :alpha0 => 0, :alpha1 => 0, :alpha2 => 0,
   :alpha12 => 0, :s2 => 1, :b => zeros(seeds[:N])]
]


## Sampling Scheme
scheme = [AMM([:alpha0, :alpha1, :alpha2, :alpha12], 0.01 * eye(4)),
          AMWG([:b], fill(0.01, seeds[:N])),
          AMWG([:s2], [0.1])]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, seeds, inits, 12500, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:12500
Thinning interval = 2
Chains = 1,2
Samples per chain = 5000

Empirical Posterior Estimates:
           Mean         SD       Naive SE       MCSE       ESS
 alpha2  1.3107281 0.26053104 0.0026053104 0.016391301 252.63418
     s2  0.0857053 0.09738014 0.0009738014 0.008167153 142.16720
 alpha0 -0.5561543 0.17595432 0.0017595432 0.011366011 239.65346
alpha12 -0.7464409 0.43006756 0.0043006756 0.026796428 257.58440
 alpha1  0.0887002 0.26872879 0.0026872879 0.013771345 380.78132

Quantiles:
            2.5%       25.0%      50.0%      75.0%      97.5%
 alpha2  0.80405938  1.1488818  1.3099477  1.4807632  1.8281561
     s2  0.00117398  0.0211176  0.0593763  0.1114008  0.3523464
 alpha0 -0.91491978 -0.6666323 -0.5512929 -0.4426242 -0.2224477
alpha12 -1.54570414 -1.0275765 -0.7572503 -0.4914919  0.1702970
 alpha1 -0.42501648 -0.0936379  0.0943906  0.2600758  0.6235393
```

2.6.5 Surgical: Institutional Ranking

An example from OpenBUGS *[38]* concerning mortality rates in 12 hospitals performing cardiac surgery in infants.

Model

Number of deaths are modelled as

$$\begin{aligned} r_i &\sim \text{Binomial}(n_i, p_i) \qquad i = 1, \ldots, 12 \\ \text{logit}(p_i) &= b_i \\ b_i &\sim \text{Normal}(\mu, \sigma) \\ \mu &\sim \text{Normal}(0, 1000) \\ \sigma^2 &\sim \text{InverseGamma}(0.001, 0001), \end{aligned}$$

where r_i is the number of deaths, out of n_i operations, at hospital i.

Analysis Program

```
using Mamba

## Data
surgical = (Symbol => Any)[
  :r => [0, 18, 8, 46, 8, 13, 9, 31, 14, 8, 29, 24],
  :n => [47, 148, 119, 810, 211, 196, 148, 215, 207, 97, 256, 360]
]
surgical[:N] = length(surgical[:r])


## Model Specification

model = Model(

  r = Stochastic(1,
    @modelexpr(n, p, N,
      Distribution[Binomial(n[i], p[i]) for i in 1:N]
    ),
    false
  ),

  p = Logical(1,
    @modelexpr(b,
      invlogit(b)
    )
  ),

  b = Stochastic(1,
    @modelexpr(mu, s2,
      Normal(mu, sqrt(s2))
    ),
    false
  ),

  mu = Stochastic(
    :(Normal(0, 1000))
  ),

  pop_mean = Logical(
    @modelexpr(mu,
      invlogit(mu)
    )
```

```
  ),

  s2 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)

## Initial Values
inits = [
  [:r => surgical[:r], :b => fill(0.1, surgical[:N]), :s2 => 1, :mu => 0],
  [:r => surgical[:r], :b => fill(0.5, surgical[:N]), :s2 => 10, :mu => 1]
]


## Sampling Scheme
scheme = [NUTS([:b]),
          Slice([:mu, :s2], [1.0, 1.0])]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, surgical, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
             Mean         SD      Naive SE       MCSE        ESS
      mu -2.550741500 0.1567451 0.0018099366 0.0032183095 2372.0965
pop_mean  0.073072811 0.0104407 0.0001205589 0.0002104121 2462.1738
    p[1]  0.052805725 0.0198027 0.0002286619 0.0004412988 2013.6467
    p[2]  0.104001890 0.0219549 0.0002535130 0.0003555323 3813.3232
    p[3]  0.070669708 0.0176251 0.0002035168 0.0002098008 7057.4433
    p[4]  0.059118030 0.0079059 0.0000912898 0.0001290882 3750.8750
    p[5]  0.051504980 0.0132372 0.0001528496 0.0002512701 2775.2895
    p[6]  0.069199820 0.0147703 0.0001705526 0.0002018471 5354.6699
    p[7]  0.067117588 0.0160309 0.0001851085 0.0001921212 6962.4719
    p[8]  0.123097365 0.0219326 0.0002532561 0.0003878670 3197.5350
    p[9]  0.070410452 0.0148534 0.0001715122 0.0001833296 6564.2635
   p[10]  0.078694851 0.0198460 0.0002291623 0.0002492771 6338.4458
   p[11]  0.102801934 0.0176951 0.0002043250 0.0002605003 4614.1109
   p[12]  0.068715136 0.0119246 0.0001376936 0.0001584527 5663.5606
      s2  0.194478287 0.1732356 0.0020003522 0.0056335715  945.5980

Quantiles:
             2.5%        25.0%         50.0%        75.0%       97.5%
      mu -2.886808 -2.641133981 -2.542990063 -2.4516929 -2.26518316
pop_mean  0.052810  0.066537569  0.072898833  0.0793148  0.09404783
    p[1]  0.016802  0.038901688  0.052307730  0.0654430  0.09367208
    p[2]  0.067526  0.087996977  0.101695407  0.1173628  0.15350268
    p[3]  0.039639  0.058578620  0.069377099  0.0811090  0.10938201
```

```
    p[4]  0.044485  0.053534322  0.058953960  0.0642819  0.07502250
    p[5]  0.027655  0.042219334  0.050835609  0.0601217  0.07939497
    p[6]  0.043011  0.058938078  0.068255749  0.0784358  0.10189272
    p[7]  0.038490  0.056074218  0.066107926  0.0769301  0.10139561
    p[8]  0.084454  0.107222812  0.121811458  0.1372187  0.16871609
    p[9]  0.044307  0.059573839  0.069798505  0.0799364  0.10182827
   p[10]  0.044685  0.064689236  0.076790598  0.0905553  0.12280446
   p[11]  0.072230  0.090317164  0.101265471  0.1140400  0.14092283
   p[12]  0.047187  0.060534280  0.068033520  0.0762521  0.09388019
      s2  0.030929  0.090703538  0.147636980  0.2405679  0.62218529
```

2.6.6 Magnesium: Meta-Analysis Prior Sensitivity

An example from OpenBUGS *[38]*.

Model

Number of events reported for treatment and control subjects in 8 studies is modelled as

$$
\begin{aligned}
r_j^c &\sim \text{Binomial}(n_j^c, p_{i,j}^c) \qquad i = 1, \ldots, 6; j = 1, \ldots, 8 \\
p_{i,j}^c &\sim \text{Uniform}(0, 1) \\
r_j^t &\sim \text{Binomial}(n_j^t, p_{i,j}^t) \\
\text{logit}(p_{i,j}^t) &= \theta_{i,j} + \text{logit}(p_{i,j}^c) \\
\theta_{i,j} &\sim \text{Normal}(\mu_i, \tau_i) \\
\mu_i &\sim \text{Uniform}(-10, 10) \\
\tau_i &\sim \text{Different Priors},
\end{aligned}
$$

where r_j^c is the number of control group events, out of n_j^c, in study j; r_j^t is the number of treatment group events; and i indexes differ prior specifications.

Analysis Program

```
using Mamba

## Data
magnesium = (Symbol => Any)[
  :rt => [1, 9, 2, 1, 10, 1, 1, 90],
  :nt => [40, 135, 200, 48, 150, 59, 25, 1159],
  :rc => [2, 23, 7, 1, 8, 9, 3, 118],
  :nc => [36, 135, 200, 46, 148, 56, 23, 1157]
]

magnesium[:rtx] = hcat([magnesium[:rt] for i in 1:6]...)'
magnesium[:rcx] = hcat([magnesium[:rc] for i in 1:6]...)'
magnesium[:s2] = 1 ./ (magnesium[:rt] + 0.5) +
                 1 ./ (magnesium[:nt] - magnesium[:rt] + 0.5) +
                 1 ./ (magnesium[:rc] + 0.5) +
                 1 ./ (magnesium[:nc] - magnesium[:rc] + 0.5)
magnesium[:s2_0] = 1 / mean(1 ./ magnesium[:s2])


## Model Specification
```

```
model = Model(

  rcx = Stochastic(2,
    @modelexpr(nc, pc,
      Distribution[Binomial(nc[j], pc[i,j]) for i in 1:6, j in 1:8]
    ),
    false
  ),

  pc = Stochastic(2,
    :(Uniform(0, 1)),
    false
  ),

  rtx = Stochastic(2,
    @modelexpr(nt, pc, theta,
      Distribution[
        begin
          phi = logit(pc[i,j])
          pt = invlogit(theta[i,j] + phi)
          Binomial(nt[j], pt)
        end
        for i in 1:6, j in 1:8
      ]
    ),
    false
  ),

  theta = Stochastic(2,
    @modelexpr(mu, tau,
      Distribution[Normal(mu[i], tau[i]) for i in 1:6, j in 1:8]
    ),
    false
  ),

  mu = Stochastic(1,
    :(Uniform(-10, 10)),
    false
  ),

  OR = Logical(1,
    @modelexpr(mu,
      exp(mu)
    )
  ),

  tau = Logical(1,
    @modelexpr(priors, s2_0,
      [ sqrt(priors[1]),
        sqrt(priors[2]),
        priors[3],
        sqrt(s2_0 * (1 / priors[4] - 1)),
        sqrt(s2_0) * (1 / priors[5] - 1),
        sqrt(priors[6]) ]
    )
  ),

  priors = Stochastic(1,
```

```
    @modelexpr(s2_0,
      Distribution[
        InverseGamma(0.001, 0.001),
        Uniform(0, 50),
        Uniform(0, 50),
        Uniform(0, 1),
        Uniform(0, 1),
        Truncated(Normal(0, sqrt(s2_0 / erf(0.75))), 0, Inf)
      ]
    ),
    false
  )

)


## Initial Values
inits = [
  [:rcx => magnesium[:rcx], :rtx => magnesium[:rtx],
   :theta => zeros(6, 8), :mu => fill(-0.5, 6),
   :pc => fill(0.5, 6, 8), :priors => [1, 1, 1, 0.5, 0.5, 1]],
  [:rcx => magnesium[:rcx], :rtx => magnesium[:rtx],
   :theta => zeros(6, 8), :mu => fill(0.5, 6),
   :pc => fill(0.5, 6, 8), :priors => [1, 1, 1, 0.5, 0.5, 1]]
]


## Sampling Scheme
scheme = [AMWG([:theta], fill(0.1, 48)),
          AMWG([:mu], fill(0.1, 6)),
          Slice([:pc], fill(0.25, 48), :univar),
          Slice([:priors], [1.0, 5.0, 5.0, 0.25, 0.25, 5.0], :univar)]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, magnesium, inits, 12500, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:12500
Thinning interval = 2
Chains = 1,2
Samples per chain = 5000

Empirical Posterior Estimates:
          Mean       SD       Naive SE      MCSE       ESS
 OR[1] 0.4938813 0.16724218 0.001672422 0.006780965  608.2872
 OR[2] 0.4693250 1.10491202 0.011049120 0.030148804 1343.1213
 OR[3] 0.4449415 0.21477655 0.002147765 0.005421447 1569.4347
 OR[4] 0.4749598 0.13873544 0.001387354 0.005962572  541.3866
 OR[5] 0.4936191 0.15045871 0.001504587 0.006343458  562.5778
 OR[6] 0.4452843 0.14497511 0.001449751 0.005554683  681.1902
tau[1] 0.5005081 0.38002286 0.003800229 0.019431081  382.4948
tau[2] 1.1646043 0.75346006 0.007534601 0.037904158  395.1362
tau[3] 0.8082610 0.49226340 0.004922634 0.018663493  695.6796
```

```
tau[4] 0.4676924 0.27044423 0.002704442 0.011446738  558.2027
tau[5] 0.4523501 0.34711372 0.003471137 0.017258471  404.5190
tau[6] 0.5734504 0.19406724 0.001940672 0.006299041  949.1953

Quantiles:
          2.5%       25.0%      50.0%      75.0%     97.5%
 OR[1] 0.20905560 0.38255153 0.48785122 0.59332617 0.8138231
 OR[2] 0.11223358 0.27027567 0.38254203 0.51449190 1.0372736
 OR[3] 0.15915375 0.31983172 0.42081185 0.53465139 0.8806431
 OR[4] 0.21563818 0.38285241 0.47223709 0.56044575 0.7647249
 OR[5] 0.20841340 0.38723776 0.49357081 0.60852910 0.7656372
 OR[6] 0.21100558 0.34464323 0.42943747 0.52740053 0.7759471
tau[1] 0.03672526 0.20687073 0.43248769 0.69829673 1.4601677
tau[2] 0.29554286 0.68477332 0.98359802 1.40115062 3.3037132
tau[3] 0.13105672 0.48296429 0.71422501 1.02092799 2.0156086
tau[4] 0.07837793 0.28460737 0.41764937 0.59299427 1.1350358
tau[5] 0.02041302 0.18367182 0.39716777 0.63507897 1.2857608
tau[6] 0.20363568 0.43899364 0.57124610 0.70377109 0.9609324
```

2.6.7 Salm: Extra-Poisson Variation in a Dose-Response Study

An example from OpenBUGS *[38]* and Breslow *[8]* concerning mutagenicity assay data on salmonella in three plates exposed to six doses of quinoline.

Model

Number of revertant colonies of salmonella are modelled as

$$\begin{aligned} y_{i,j} &\sim \text{Poisson}(\mu_{i,j}) \qquad i = 1, \ldots, 3; j = 1, \ldots, 6 \\ \log(\mu_{i,j}) &= \alpha + \beta \log(x_j + 10) + \gamma x_j + \lambda_{i,j} \\ \alpha, \beta, \gamma &\sim \text{Normal}(0, 1000) \\ \lambda_{i,j} &\sim \text{Normal}(0, \sigma) \\ \sigma^2 &\sim \text{InverseGamma}(0.001, 0.001), \end{aligned}$$

where y_i is the number of colonies in plate i and dose j.

Analysis Program

```
using Mamba

## Data
salm = (Symbol => Any)[
  :y => reshape(
    [15, 21, 29, 16, 18, 21, 16, 26, 33, 27, 41, 60, 33, 38, 41, 20, 27, 42],
    3, 6),
  :x => [0, 10, 33, 100, 333, 1000],
  :plate => 3,
  :dose => 6
]


## Model Specification
```

```
model = Model(

  y = Stochastic(2,
    @modelexpr(alpha, beta, gamma, x, lambda,
      Distribution[
        begin
          mu = exp(alpha + beta * log(x[j] + 10) + gamma * x[j] + lambda[i,j])
          Poisson(mu)
        end
        for i in 1:3, j in 1:6
      ]
    ),
    false
  ),

  alpha = Stochastic(
    :(Normal(0, 1000))
  ),

  beta = Stochastic(
    :(Normal(0, 1000))
  ),

  gamma = Stochastic(
    :(Normal(0, 1000))
  ),

  lambda = Stochastic(2,
    @modelexpr(s2,
      Normal(0, sqrt(s2))
    ),
    false
  ),

  s2 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)


## Initial Values
inits = [
  [:y => salm[:y], :alpha => 0, :beta => 0, :gamma => 0, :s2 => 10,
   :lambda => zeros(3, 6)],
  [:y => salm[:y], :alpha => 1, :beta => 1, :gamma => 0.01, :s2 => 1,
   :lambda => zeros(3, 6)]
]

## Sampling Scheme
scheme = [Slice([:alpha, :beta, :gamma], [1.0, 1.0, 0.1]),
          AMWG([:lambda, :s2], fill(0.1, 19))]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, salm, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
         Mean         SD       Naive SE      MCSE        ESS
gamma -0.0011251 0.00034537 0.000003988 0.00002158 256.1352
alpha  2.0100584 0.26156943 0.003020344 0.02106994 154.1158
   s2  0.0690770 0.04304237 0.000497010 0.00192980 497.4697
 beta  0.3543443 0.07160779 0.000826856 0.00564423 160.9577

Quantiles:
         2.5%       25.0%      50.0%       75.0%       97.5%
gamma -0.0017930 -0.0013474 -0.001133 -0.0009096 -0.0003927
alpha  1.5060295  1.8471115  2.006273  2.1655604  2.5054785
   s2  0.0135820  0.0397881  0.059831  0.0874088  0.1774730
 beta  0.2073238  0.3127477  0.358298  0.4000692  0.4878904
```

2.6.8 Equiv: Bioequivalence in a Cross-Over Trial

An example from OpenBUGS *[38]* and Gelfand *et al.* *[25]* concerning a two-treatment, cross-over trial with 10 subjects.

Model

Treatment responses are modelled as

$$
\begin{aligned}
y_{i,j} &\sim \text{Normal}(m_{i,j}, \sigma_1) \qquad i = 1, \ldots, 10; j = 1, 2 \\
m_{i,j} &= \mu + (-1)^{T_{i,j}-1}\phi/2 + (-1)^{j-1}\pi/2 + \delta_i \\
\delta_i &\sim \text{Normal}(0, \sigma_2) \\
\mu, \phi, \pi &\sim \text{Normal}(0, 1000) \\
\sigma_1^2, \sigma_2^2 &\sim \text{InverseGamma}(0.001, 0.001)
\end{aligned}
$$

where $y_{i,j}$ is the response for patient i in period j; and $T_{i,j} = 1, 2$ is the treatment received.

Analysis Program

```
using Mamba

## Data
equiv = (Symbol => Any)[
  :group => [1, 1, 2, 2, 2, 1, 1, 1, 2, 2],
  :y =>
    [1.40 1.65
     1.64 1.57
     1.44 1.58
     1.36 1.68
     1.65 1.69
     1.08 1.31
     1.09 1.43
```

```
     1.25 1.44
     1.25 1.39
     1.30 1.52]
]
equiv[:N] = size(equiv[:y], 1)
equiv[:P] = size(equiv[:y], 2)

equiv[:T] = [equiv[:group] 3 - equiv[:group]]


## Model Specification

model = Model(

  y = Stochastic(2,
    @modelexpr(delta, mu, phi, pi, s2_1, T,
      begin
        sigma = sqrt(s2_1)
        Distribution[
          begin
            m = mu + (-1)^(T[i,j]-1) * phi / 2 + (-1)^(j-1) * pi / 2 +
                delta[i,j]
            Normal(m, sigma)
          end
          for i in 1:10, j in 1:2
        ]
      end
    ),
    false
  ),

  delta = Stochastic(2,
    @modelexpr(s2_2,
      Normal(0, sqrt(s2_2))
    ),
    false
  ),

  mu = Stochastic(
    :(Normal(0, 1000))
  ),

  phi = Stochastic(
    :(Normal(0, 1000))
  ),

  theta = Logical(
    @modelexpr(phi,
      exp(phi)
    )
  ),

  pi = Stochastic(
    :(Normal(0, 1000))
  ),

  s2_1 = Stochastic(
    :(InverseGamma(0.001, 0.001))
```

```
  ),

  s2_2 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  ),

  equiv = Logical(
    @modelexpr(theta,
      int(0.8 < theta < 1.2)
    )
  )

)


## Initial Values
inits = [
  [:y => equiv[:y], :delta => zeros(10, 2), :mu => 0, :phi => 0,
   :pi => 0, :s2_1 => 1, :s2_2 => 1],
  [:y => equiv[:y], :delta => zeros(10, 2), :mu => 10, :phi => 10,
   :pi => 10, :s2_1 => 10, :s2_2 => 10]
]


## Sampling Scheme
scheme = [NUTS([:delta]),
          Slice([:mu, :phi, :pi], fill(1.0, 3)),
          Slice([:s2_1, :s2_2], ones(2), :univar)]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, equiv, inits, 12500, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:12500
Thinning interval = 2
Chains = 1,2
Samples per chain = 5000

Empirical Posterior Estimates:
         Mean        SD       Naive SE       MCSE        ESS
   pi -0.1788624 0.07963600 0.0007963600 0.0022537740 1248.5276
   mu  1.4432301 0.04735444 0.0004735444 0.0020359933  540.9645
equiv  0.9835000 0.12739456 0.0012739456 0.0022978910 3073.5685
 s2_2  0.0187006 0.01466228 0.0001466228 0.0004799037  933.4580
 s2_1  0.0164033 0.01412396 0.0001412396 0.0005079176  773.2610
theta  0.9837569 0.07936595 0.0007936595 0.0025041274 1004.5132
  phi -0.0195929 0.08005300 0.0008005300 0.0025432838  990.7533

Quantiles:
         2.5%      25.0%     50.0%      75.0%      97.5%
   pi -0.332750 -0.2265245 -0.184088 -0.1285565 -0.0142326
   mu  1.353882  1.4111562  1.441424  1.4740774  1.5345210
equiv  1.000000  1.0000000  1.000000  1.0000000  1.0000000
```

```
 s2_2  0.001568  0.0061461  0.016302  0.0273322  0.0530448
 s2_1  0.001035  0.0043785  0.013677  0.0242892  0.0507633
theta  0.838904  0.9336518  0.974953  1.0320046  1.1567524
  phi -0.175659 -0.0686517 -0.025366  0.0315031  0.1456164
```

2.6.9 Dyes: Variance Components Model

An example from OpenBUGS *[38]*, Davies *[18]*, and Box and Tiao *[7]* concerning batch-to-batch variation in yields from six batches and five samples of dyestuff.

Model

Yields are modelled as

$$\begin{aligned} y_{i,j} &\sim \text{Normal}(\mu_i, \sigma_{\text{within}}) \qquad i = 1, \ldots, 6; j = 1, \ldots, 5 \\ \mu_i &\sim \text{Normal}(\theta, \sigma_{\text{between}}) \\ \theta &\sim \text{Normal}(0, 1000) \\ \sigma^2_{\text{within}}, \sigma^2_{\text{between}} &\sim \text{InverseGamma}(0.001, 0.001), \end{aligned}$$

where $y_{i,j}$ is the response for batch i and sample j.

Analysis Program

```
using Mamba

## Data
dyes = (Symbol => Any)[
  :y =>
    [1545, 1440, 1440, 1520, 1580,
     1540, 1555, 1490, 1560, 1495,
     1595, 1550, 1605, 1510, 1560,
     1445, 1440, 1595, 1465, 1545,
     1595, 1630, 1515, 1635, 1625,
     1520, 1455, 1450, 1480, 1445],
  :batches => 6,
  :samples => 5
]

dyes[:batch] = vcat([fill(i, dyes[:samples]) for i in 1:dyes[:batches]]...)
dyes[:sample] = vcat(fill([1:dyes[:samples]], dyes[:batches])...)


## Model Specification

model = Model(

  y = Stochastic(1,
    @modelexpr(mu, batch, s2_within,
      MvNormal(mu[batch], sqrt(s2_within))
    ),
    false
  ),
```

```
  mu = Stochastic(1,
    @modelexpr(theta, batches, s2_between,
      Normal(theta, sqrt(s2_between))
    ),
    false
  ),

  theta = Stochastic(
    :(Normal(0, 1000))
  ),

  s2_within = Stochastic(
    :(InverseGamma(0.001, 0.001))
  ),

  s2_between = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)


## Initial Values
inits = [
  [:y => dyes[:y], :theta => 1500, :s2_within => 1, :s2_between => 1,
   :mu => fill(1500, dyes[:batches])],
  [:y => dyes[:y], :theta => 3000, :s2_within => 10, :s2_between => 10,
   :mu => fill(3000, dyes[:batches])]
]


## Sampling Scheme
scheme = [NUTS([:mu, :theta]),
          Slice([:s2_within, :s2_between], [1000.0, 1000.0])]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, dyes, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
              Mean        SD     Naive SE    MCSE       ESS
s2_between 2504.9567 2821.1909 32.576307 201.76845  195.50525
 s2_within 2870.5356  997.8584 11.522276  50.36398  392.55247
     theta 1527.0634   22.9340  0.264819   0.39541 3363.98979

Quantiles:
               2.5%      25.0%     50.0%    75.0%       97.5%
s2_between  152.111234  850.5886 1676.491 3000.7898 1.287454x104
```

```
 s2_within 1532.847859 2175.9060 2665.165 3328.3947  5.54381x103
     theta 1481.227885 1512.8876 1527.653 1540.3759  1.57204x103
```

2.6.10 Stacks: Robust Regression

An example from OpenBUGS *[38]*, Brownlee *[12]*, and Birkes and Dodge *[4]* concerning 21 daily responses of stack loss, the amount of ammonia escaping, as a function of air flow, temperature, and acid concentration.

Model

Losses are modelled as

$$\begin{aligned} y_i &\sim \text{Laplace}(\mu_i, \sigma^2) \qquad i = 1, \ldots, 21 \\ \mu_i &= \beta_0 + \beta_1 z_{1i} + \beta_2 z_{2i} + \beta_3 z_{3i} \\ \beta_0, \beta_1, \beta_2, \beta_3 &\sim \text{Normal}(0, 1000) \\ \sigma^2 &\sim \text{InverseGamma}(0.001, 0.001), \end{aligned}$$

where y_i is the stack loss on day i; and z_{1i}, z_{2i}, z_{3i} are standardized predictors.

Analysis Program

```
using Mamba

## Data
stacks = (Symbol => Any)[
  :y => [42, 37, 37, 28, 18, 18, 19, 20, 15, 14, 14, 13, 11, 12, 8, 7, 8, 8, 9,
         15, 15],
  :x =>
    [80 27 89
     80 27 88
     75 25 90
     62 24 87
     62 22 87
     62 23 87
     62 24 93
     62 24 93
     58 23 87
     58 18 80
     58 18 89
     58 17 88
     58 18 82
     58 19 93
     50 18 89
     50 18 86
     50 19 72
     50 19 79
     50 20 80
     56 20 82
     70 20 91]
]
stacks[:N] = size(stacks[:x], 1)
stacks[:p] = size(stacks[:x], 2)
```

```
stacks[:meanx] = map(j -> mean(stacks[:x][:,j]), 1:stacks[:p])
stacks[:sdx] = map(j -> std(stacks[:x][:,j]), 1:stacks[:p])
stacks[:z] = Float64[
  (stacks[:x][i,j] - stacks[:meanx][j]) / stacks[:sdx][j]
  for i in 1:stacks[:N], j in 1:stacks[:p]
]


## Model Specification

model = Model(

  y = Stochastic(1,
    @modelexpr(mu, s2, N,
      begin
        Distribution[Laplace(mu[i], s2) for i in 1:N]
      end
    ),
    false
  ),

  beta0 = Stochastic(
    :(Normal(0, 1000)),
    false
  ),

  beta = Stochastic(1,
    :(Normal(0, 1000)),
    false
  ),

  mu = Logical(1,
    @modelexpr(beta0, z, beta,
      beta0 + z * beta
    ),
    false
  ),

  s2 = Stochastic(
    :(InverseGamma(0.001, 0.001)),
    false
  ),

  sigma = Logical(
    @modelexpr(s2,
      sqrt(2.0) * s2
    )
  ),

  b0 = Logical(
    @modelexpr(beta0, b, meanx,
      beta0 - dot(b, meanx)
    )
  ),

  b = Logical(1,
    @modelexpr(beta, sdx,
      beta ./ sdx
```

```
    )
  ),

  outlier = Logical(1,
    @modelexpr(y, mu, sigma, N,
      Float64[abs((y[i] - mu[i]) / sigma) > 2.5 for i in 1:N]
    ),
    [1,3,4,21]
  )

)


## Initial Values
inits = [
  [:y => stacks[:y], :beta0 => 10, :beta => [0, 0, 0], :s2 => 10],
  [:y => stacks[:y], :beta0 => 1, :beta => [1, 1, 1], :s2 => 1]
]


## Sampling Scheme
scheme = [NUTS([:beta0, :beta]),
          Slice([:s2], [1.0])]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, stacks, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
                Mean        SD       Naive SE      MCSE        ESS
       b[1]   0.83445 0.1297656 0.001498404 0.00227082 3265.534
       b[2]   0.75151 0.3329099 0.003844112 0.00562761 3499.489
       b[3]  -0.11714 0.1196021 0.001381046 0.00160727 5537.350
 outlier[1]   0.03893 0.1934490 0.002233757 0.00285110 4603.713
 outlier[3]   0.05587 0.2296794 0.002652109 0.00357717 4122.536
 outlier[4]   0.30000 0.4582881 0.005291855 0.00861175 2832.010
outlier[21]   0.60987 0.4878125 0.005632774 0.01106491 1943.615
         b0 -38.73747 8.6797300 0.100224890 0.10329322 7061.042
      sigma   3.46651 0.8548797 0.009871301 0.02758864  960.173

Quantiles:
                2.5%       25.0%       50.0%       75.0%       97.5%
       b[1]   0.568641   0.754152   0.835908   0.916990   1.089509
       b[2]   0.178021   0.525812   0.723047   0.947780   1.480699
       b[3]  -0.366040  -0.189466  -0.112007  -0.041001   0.111335
 outlier[1]   0.000000   0.000000   0.000000   0.000000   1.000000
 outlier[3]   0.000000   0.000000   0.000000   0.000000   1.000000
 outlier[4]   0.000000   0.000000   0.000000   1.000000   1.000000
```

```
outlier[21]   0.000000   0.000000   1.000000   1.000000   1.000000
         b0 -56.503611 -44.046775 -38.745722 -33.417515 -21.737427
      sigma   2.177428   2.860547   3.337315   3.941882   5.509634
```

2.6.11 Epilepsy: Repeated Measures on Poisson Counts

An example from OpenBUGS *[38]*, Thall and Vail *[71]* Breslow and Clayton *[9]* concerning the effects of treatment, baseline seizure counts, and age on follow-up seizure counts at four visits in 59 patients.

Model

Counts are modelled as

$$
\begin{aligned}
y_{i,j} &\sim \text{Poisson}(\mu_{i,j}) \qquad i = 1, \ldots, 59; j = 1, \ldots, 4 \\
\log(\mu_{i,j}) &= \alpha_0 + \alpha_{\text{Base}} \log(\text{Base}_i/4) + \alpha_{\text{Trt}}\text{Trt}_i + \alpha_{\text{BT}}\text{Trt}_i \log(\text{Base}_i/4) + \\
&\quad \alpha_{\text{Age}} \log(\text{Age}_i) + \alpha_{\text{V4}}\text{V}_4 + \text{b1}_i + \text{b}_{i,j} \\
\text{b1}_i &\sim \text{Normal}(0, \sigma_{\text{b1}}) \\
\text{b}_{i,j} &\sim \text{Normal}(0, \sigma_{\text{b}}) \\
\alpha_* &\sim \text{Normal}(0, 100) \\
\sigma^2_{\text{b1}}, \sigma^2_{\text{b}} &\sim \text{InverseGamma}(0.001, 0.001),
\end{aligned}
$$

where y_{ij} are the counts on patient i at visit j, Trt is a treatment indicator, Base is baseline seizure counts, Age is age in years, and V_4 is an indicator for the fourth visit.

Analysis Program

```
using Mamba

## Data
epil = (Symbol => Any)[
  :y =>
    [ 5  3  3  3
      3  5  3  3
      2  4  0  5
      4  4  1  4
      7 18  9 21
      5  2  8  7
      6  4  0  2
     40 20 21 12
      5  6  6  5
     14 13  6  0
     26 12  6 22
     12  6  8  4
      4  4  6  2
      7  9 12 14
     16 24 10  9
     11  0  0  5
      0  0  3  3
     37 29 28 29
      3  5  2  5
      3  0  6  7
      3  4  3  4
```

```
      3  4  3  4
      2  3  3  5
      8 12  2  8
     18 24 76 25
      2  1  2  1
      3  1  4  2
     13 15 13 12
     11 14  9  8
      8  7  9  4
      0  4  3  0
      3  6  1  3
      2  6  7  4
      4  3  1  3
     22 17 19 16
      5  4  7  4
      2  4  0  4
      3  7  7  7
      4 18  2  5
      2  1  1  0
      0  2  4  0
      5  4  0  3
     11 14 25 15
     10  5  3  8
     19  7  6  7
      1  1  2  3
      6 10  8  8
      2  1  0  0
    102 65 72 63
      4  3  2  4
      8  6  5  7
      1  3  1  5
     18 11 28 13
      6  3  4  0
      3  5  4  3
      1 23 19  8
      2  3  0  1
      0  0  0  0
      1  4  3  2],
  :Trt =>
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
     1, 1, 1, 1, 1, 1, 1, 1, 1],
  :Base =>
    [11, 11, 6, 8, 66, 27, 12, 52, 23, 10, 52, 33, 18, 42, 87, 50, 18, 111, 18,
     20, 12, 9, 17, 28, 55, 9, 10, 47, 76, 38, 19, 10, 19, 24, 31, 14, 11, 67,
     41, 7, 22, 13, 46, 36, 38, 7, 36, 11, 151, 22, 41, 32, 56, 24, 16, 22, 25,
     13, 12],
  :Age =>
    [31, 30, 25, 36, 22, 29, 31, 42, 37, 28, 36, 24, 23, 36, 26, 26, 28, 31, 32,
     21, 29, 21, 32, 25, 30, 40, 19, 22, 18, 32, 20, 30, 18, 24, 30, 35, 27, 20,
     22, 28, 23, 40, 33, 21, 35, 25, 26, 25, 22, 32, 25, 35, 21, 41, 32, 26, 21,
     36, 37],
  :V4 => [0, 0, 0, 1]
]
epil[:N] = size(epil[:y], 1)
epil[:T] = size(epil[:y], 2)

epil[:logBase4] = log(epil[:Base] / 4)
```

```
epil[:BT] = epil[:logBase4] .* epil[:Trt]
epil[:logAge] = log(epil[:Age])
map(key -> epil[symbol(string(key, "bar"))] = mean(epil[key]),
    [:logBase4, :Trt, :BT, :logAge, :V4])


## Model Specification

model = Model(

  y = Stochastic(2,
    @modelexpr(a0, alpha_Base, logBase4, logBase4bar, alpha_Trt, Trt, Trtbar,
               alpha_BT, BT, BTbar, alpha_Age, logAge, logAgebar, alpha_V4, V4,
               V4bar, b1, b, N, T,
      Distribution[
        begin
          mu = exp(a0 + alpha_Base * (logBase4[i] - logBase4bar) +
                   alpha_Trt * (Trt[i] - Trtbar) + alpha_BT * (BT[i] - BTbar) +
                   alpha_Age * (logAge[i] - logAgebar) +
                   alpha_V4 * (V4[j] - V4bar) + b1[i] +
                   b[i,j])
          Poisson(mu)
        end
        for i in 1:N, j in 1:T
      ]
    ),
    false
  ),

  b1 = Stochastic(1,
    @modelexpr(s2_b1,
      Normal(0, sqrt(s2_b1))
    ),
    false
  ),

  b = Stochastic(2,
    @modelexpr(s2_b,
      Normal(0, sqrt(s2_b))
    ),
    false
  ),

  a0 = Stochastic(
    :(Normal(0, 100)),
    false
  ),

  alpha_Base = Stochastic(
    :(Normal(0, 100))
  ),

  alpha_Trt = Stochastic(
    :(Normal(0, 100))
  ),

  alpha_BT = Stochastic(
    :(Normal(0, 100))
```

```
  ),

  alpha_Age = Stochastic(
    :(Normal(0, 100))
  ),

  alpha_V4 = Stochastic(
    :(Normal(0, 100))
  ),

  alpha0 = Logical(
    @modelexpr(a0, alpha_Base, logBase4bar, alpha_Trt, Trtbar, alpha_BT, BTbar,
               alpha_Age, logAgebar, alpha_V4, V4bar,
      a0 - alpha_Base * logBase4bar - alpha_Trt * Trtbar - alpha_BT * BTbar -
        alpha_Age * logAgebar - alpha_V4 * V4bar
    )
  ),

  s2_b1 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  ),

  s2_b = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)


## Initial Values
inits = [
  [:y => epil[:y], :a0 => 0, :alpha_Base => 0, :alpha_Trt => 0,
   :alpha_BT => 0, :alpha_Age => 0, :alpha_V4 => 0, :s2_b1 => 1,
   :s2_b => 1, :b1 => zeros(epil[:N]), :b => zeros(epil[:N], epil[:T])],
  [:y => epil[:y], :a0 => 1, :alpha_Base => 1, :alpha_Trt => 1,
   :alpha_BT => 1, :alpha_Age => 1, :alpha_V4 => 1, :s2_b1 => 10,
   :s2_b => 10, :b1 => zeros(epil[:N]), :b => zeros(epil[:N], epil[:T])]
]


## Sampling Scheme
scheme = [AMWG([:a0, :alpha_Base, :alpha_Trt, :alpha_BT, :alpha_Age,
                :alpha_V4], fill(0.1, 6)),
          Slice([:b1], fill(0.5, epil[:N])),
          Slice([:b], fill(0.5, epil[:N] * epil[:T])),
          Slice([:s2_b1, :s2_b], ones(2))]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, epil, inits, 15000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:15000
Thinning interval = 2
Chains = 1,2
Samples per chain = 6250

Empirical Posterior Estimates:
              Mean         SD       Naive SE        MCSE       ESS
 alpha_Age  0.4583090 0.3945362 0.0035288392 0.020336364 376.38053
    alpha0 -1.3561708 1.3132402 0.0117459774 0.072100024 331.75503
  alpha_BT  0.2421700 0.1905664 0.0017044781 0.010759318 313.70641
alpha_Base  0.9110497 0.1353545 0.0012106472 0.007208435 352.58447
      s2_b  0.1352375 0.0318193 0.0002846002 0.001551352 420.68781
 alpha_Trt -0.7593139 0.3977342 0.0035574432 0.023478808 286.96826
     s2_b1  0.2491188 0.0731667 0.0006544231 0.002900632 636.27088
  alpha_V4 -0.0928793 0.0836669 0.0007483393 0.003604194 538.87837

Quantiles:
              2.5%      25.0%      50.0%      75.0%      97.5%
 alpha_Age -0.1966699  0.176356  0.4160869  0.6966479  1.3050754
    alpha0 -4.1688878 -2.157933 -1.2634314 -0.4362265  0.8661958
  alpha_BT -0.0902501  0.108103  0.2265617  0.3583543  0.6578050
alpha_Base  0.6631882  0.817701  0.9026821  0.9974174  1.2006197
      s2_b  0.0715581  0.112591  0.1362650  0.1580326  0.1937159
 alpha_Trt -1.6368221 -1.011391 -0.7565400 -0.4808709 -0.0161134
     s2_b1  0.1381748  0.197135  0.2376114  0.2896550  0.4228051
  alpha_V4 -0.2550453 -0.148157 -0.0931360 -0.0366814  0.0720990
```

2.6.12 Blocker: Random Effects Meta-Analysis of Clinical Trials

An example from OpenBUGS *[38]* and Carlin *[13]* concerning a meta-analysis of 22 clinical trials to prevent mortality after myocardial infarction.

Model

Events are modelled as

$$
\begin{aligned}
r_i^c &\sim \text{Binomial}(n_i^c, p_i^c) \qquad i = 1, \ldots, 22 \\
r_i^t &\sim \text{Binomial}(n_i^t, p_i^t) \\
\text{logit}(p_i^c) &= \mu_i \\
\text{logit}(p_i^t) &= \mu_i + \delta_i \\
\mu_i &\sim \text{Normal}(0, 1000) \\
\delta_i &\sim \text{Normal}(d, \sigma) \\
d &\sim \text{Normal}(0, 1000) \\
\sigma^2 &\sim \text{InverseGamma}(0.001, 0.001),
\end{aligned}
$$

where r_i^c is the number of control group events, out of n_i^c, in study i; and r_i^t is the number of treatment group events.

Analysis Program

```
using Mamba

## Data
blocker = (Symbol => Any)[
  :rt =>
    [3, 7, 5, 102, 28, 4, 98, 60, 25, 138, 64, 45, 9, 57, 25, 33, 28, 8, 6, 32,
     27, 22],
  :nt =>
    [38, 114, 69, 1533, 355, 59, 945, 632, 278, 1916, 873, 263, 291, 858, 154,
     207, 251, 151, 174, 209, 391, 680],
  :rc =>
    [3, 14, 11, 127, 27, 6, 152, 48, 37, 188, 52, 47, 16, 45, 31, 38, 12, 6, 3,
     40, 43, 39],
  :nc =>
    [39, 116, 93, 1520, 365, 52, 939, 471, 282, 1921, 583, 266, 293, 883, 147,
     213, 122, 154, 134, 218, 364, 674]
]
blocker[:N] = length(blocker[:rt])


## Model Specification

model = Model(

  rc = Stochastic(1,
    @modelexpr(mu, nc, N,
      begin
        pc = invlogit(mu)
        Distribution[Binomial(nc[i], pc[i]) for i in 1:N]
      end
    ),
    false
  ),

  rt = Stochastic(1,
    @modelexpr(mu, delta, nt, N,
      begin
        pt = invlogit(mu + delta)
        Distribution[Binomial(nt[i], pt[i]) for i in 1:N]
      end
    ),
    false
  ),

  mu = Stochastic(1,
    :(Normal(0, 1000)),
    false
  ),

  delta = Stochastic(1,
    @modelexpr(d, s2,
      Normal(d, sqrt(s2))
    ),
    false
  ),
```

```
  delta_new = Stochastic(
    @modelexpr(d, s2,
      Normal(d, sqrt(s2))
    )
  ),

  d = Stochastic(
    :(Normal(0, 1000))
  ),

  s2 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)


## Initial Values
inits = [
  [:rc => blocker[:rc], :rt => blocker[:rt], :d => 0, :delta_new => 0,
   :s2 => 1, :mu => zeros(blocker[:N]), :delta => zeros(blocker[:N])],
  [:rc => blocker[:rc], :rt => blocker[:rt], :d => 2, :delta_new => 2,
   :s2 => 10, :mu => fill(2, blocker[:N]), :delta => fill(2, blocker[:N])]
]


## Sampling Scheme
scheme = [AMWG([:mu], fill(0.1, blocker[:N])),
          AMWG([:delta, :delta_new], fill(0.1, blocker[:N] + 1)),
          Slice([:d, :s2], [1.0, 1.0])]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, blocker, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
              Mean        SD       Naive SE       MCSE         ESS
delta_new -0.25005767 0.15032528 0.0017358068 0.0042970294 1223.84778
       s2  0.01822186 0.02112127 0.0002438874 0.0012497271  285.63372
        d -0.25563567 0.06184194 0.0007140893 0.0030457284  412.27207

Quantiles:
              2.5%       25.0%       50.0%       75.0%       97.5%
delta_new -0.53854055 -0.32799584 -0.25578493 -0.17758841  0.07986060
       s2  0.00068555  0.00416488  0.01076157  0.02444208  0.07735715
        d -0.37341230 -0.29591698 -0.25818488 -0.21834138 -0.12842580
```

2.6.13 Oxford: Smooth Fit to Log-Odds Ratios

An example from OpenBUGS *[38]* and Breslow and Clayton *[9]* concerning the association between death from childhood cancer and maternal exposure to X-rays, for subjects partitioned into 120 age and birth-year strata.

Model

Deaths are modelled as

$$
\begin{aligned}
r_i^0 &\sim \text{Binomial}(n_i^0, p_i^0) \qquad i = 1, \ldots, 120 \\
r_i^1 &\sim \text{Binomial}(n_i^1, p_i^1) \\
\text{logit}(p_i^0) &= \mu_i \\
\text{logit}(p_i^1) &= \mu_i + \log(\psi_i) \\
\log(\psi) &= \alpha + \beta_1 \text{year}_i + \beta_2(\text{year}_i^2 - 22) + b_i \\
\mu_i &\sim \text{Normal}(0, 1000) \\
b_i &\sim \text{Normal}(0, \sigma) \\
\alpha, \beta_1, \beta_2 &\sim \text{Normal}(0, 1000) \\
\sigma^2 &\sim \text{InverseGamma}(0.001, 0.001),
\end{aligned}
$$

where r_i^0 is the number of deaths among unexposed subjects in stratum i, r_i^1 is the number among exposed subjects, and year_i is the stratum-specific birth year (relative to 1954).

Analysis Program

```
using Mamba

## Data
oxford = (Symbol=> Any)[
  :r1 =>
    [3, 5, 2, 7, 7, 2, 5, 3, 5, 11, 6, 6, 11, 4, 4, 2, 8, 8, 6, 5, 15, 4, 9, 9,
     4, 12, 8, 8, 6, 8, 12, 4, 7, 16, 12, 9, 4, 7, 8, 11, 5, 12, 8, 17, 9, 3, 2,
     7, 6, 5, 11, 14, 13, 8, 6, 4, 8, 4, 8, 7, 15, 15, 9, 9, 5, 6, 3, 9, 12, 14,
     16, 17, 8, 8, 9, 5, 9, 11, 6, 14, 21, 16, 6, 9, 8, 9, 8, 4, 11, 11, 6, 9,
     4, 4, 9, 9, 10, 14, 6, 3, 4, 6, 10, 4, 3, 3, 10, 4, 10, 5, 4, 3, 13, 1, 7,
     5, 7, 6, 3, 7],
  :n1 =>
    [28, 21, 32, 35, 35, 38, 30, 43, 49, 53, 31, 35, 46, 53, 61, 40, 29, 44, 52,
     55, 61, 31, 48, 44, 42, 53, 56, 71, 43, 43, 43, 40, 44, 70, 75, 71, 37, 31,
     42, 46, 47, 55, 63, 91, 43, 39, 35, 32, 53, 49, 75, 64, 69, 64, 49, 29, 40,
     27, 48, 43, 61, 77, 55, 60, 46, 28, 33, 32, 46, 57, 56, 78, 58, 52, 31, 28,
     46, 42, 45, 63, 71, 69, 43, 50, 31, 34, 54, 46, 58, 62, 52, 41, 34, 52, 63,
     59, 88, 62, 47, 53, 57, 74, 68, 61, 45, 45, 62, 73, 53, 39, 45, 51, 55, 41,
     53, 51, 42, 46, 54, 32],
  :r0 =>
    [0, 2, 2, 1, 2, 0, 1, 1, 1, 2, 4, 4, 2, 1, 7, 4, 3, 5, 3, 2, 4, 1, 4, 5, 2,
     7, 5, 8, 2, 3, 5, 4, 1, 6, 5, 11, 5, 2, 5, 8, 5, 6, 6, 10, 7, 5, 5, 2, 8,
     1, 13, 9, 11, 9, 4, 4, 8, 6, 8, 6, 8, 14, 6, 5, 5, 2, 4, 2, 9, 5, 6, 7, 5,
     10, 3, 2, 1, 7, 9, 13, 9, 11, 4, 8, 2, 3, 7, 4, 7, 5, 6, 6, 5, 6, 9, 7, 7,
     7, 4, 2, 3, 4, 10, 3, 4, 2, 10, 5, 4, 5, 4, 6, 5, 3, 2, 2, 4, 6, 4, 1],
  :n0 =>
    [28, 21, 32, 35, 35, 38, 30, 43, 49, 53, 31, 35, 46, 53, 61, 40, 29, 44, 52,
     55, 61, 31, 48, 44, 42, 53, 56, 71, 43, 43, 43, 40, 44, 70, 75, 71, 37, 31,
     42, 46, 47, 55, 63, 91, 43, 39, 35, 32, 53, 49, 75, 64, 69, 64, 49, 29, 40,
```

```
    27, 48, 43, 61, 77, 55, 60, 46, 28, 33, 32, 46, 57, 56, 78, 58, 52, 31, 28,
    46, 42, 45, 63, 71, 69, 43, 50, 31, 34, 54, 46, 58, 62, 52, 41, 34, 52, 63,
    59, 88, 62, 47, 53, 57, 74, 68, 61, 45, 45, 62, 73, 53, 39, 45, 51, 55, 41,
    53, 51, 42, 46, 54, 32],
  :year =>
    [-10, -9, -9, -8, -8, -8, -7, -7, -7, -7, -6, -6, -6, -6, -6, -5, -5, -5,
    -5, -5, -5, -4, -4, -4, -4, -4, -4, -4, -3, -3, -3, -3, -3, -3, -3, -3, -2,
    -2, -2, -2, -2, -2, -2, -2, -2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 6,
    6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 9, 9, 10],
  :K => 120
]
oxford[:K] = length(oxford[:r1])


## Model Specification

model = Model(

  r0 = Stochastic(1,
    @modelexpr(mu, n0, K,
      begin
        p = invlogit(mu)
        Distribution[Binomial(n0[i], p[i]) for i in 1:K]
      end
    ),
    false
  ),

  r1 = Stochastic(1,
    @modelexpr(mu, alpha, beta1, beta2, year, b, n1, K,
      Distribution[
        begin
          p = invlogit(mu[i] + alpha + beta1 * year[i] +
                       beta2 * (year[i]^2 - 22.0) + b[i])
          Binomial(n1[i], p)
        end
        for i in 1:K
      ]
    ),
    false
  ),

  b = Stochastic(1,
    @modelexpr(s2,
      Normal(0, sqrt(s2))
    ),
    false
  ),

  mu = Stochastic(1,
    :(Normal(0, 1000)),
    false
  ),

  alpha = Stochastic(
    :(Normal(0, 1000))
```

```
  ),

  beta1 = Stochastic(
    :(Normal(0, 1000))
  ),

  beta2 = Stochastic(
    :(Normal(0, 1000))
  ),

  s2 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)


## Initial Values
inits = [
  [:r0 => oxford[:r0], :r1 => oxford[:r1], :alpha => 0, :beta1 => 0,
   :beta2 => 0, :s2 => 1, :b => zeros(oxford[:K]), :mu => zeros(oxford[:K])],
  [:r0 => oxford[:r0], :r1 => oxford[:r1], :alpha => 1, :beta1 => 1,
   :beta2 => 1, :s2 => 10, :b => zeros(oxford[:K]), :mu => zeros(oxford[:K])]
]


## Sampling Scheme
scheme = [AMWG([:alpha, :beta1, :beta2], fill(1.0, 3)),
          Slice([:s2], [1.0]),
          Slice([:mu], ones(oxford[:K])),
          Slice([:b], ones(oxford[:K]))]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, oxford, inits, 12500, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:12500
Thinning interval = 2
Chains = 1,2
Samples per chain = 5000

Empirical Posterior Estimates:
         Mean          SD        Naive SE       MCSE        ESS
alpha  0.5657848 0.063005090 0.00063005090 0.0034007318 343.24680
   s2  0.0262390 0.030798915 0.00030798915 0.0026076857 139.49555
beta1 -0.0433363 0.016175426 0.00016175426 0.0011077969 213.20196
beta2  0.0054771 0.003567575 0.00003567575 0.0002358424 228.82453

Quantiles:
          2.5%       25.0%      50.0%      75.0%       97.5%
alpha  0.44382579  0.5238801  0.5675039  0.60514271  0.6959681
   s2  0.00071344  0.0033353  0.0146737  0.03971325  0.1182023
beta1 -0.07451524 -0.0543180 -0.0434426 -0.03212161 -0.0099208
```

```
beta2 -0.00104990  0.0028489  0.0056500  0.00774736  0.0136309
```

2.6.14 LSAT: Item Response

An example from OpenBUGS *[38]* and Boch and Lieberman *[5]* concerning a 5-item multiple choice test (32 possible response patters) given to 1000 students.

Model

Item responses are modelled as

$$\begin{aligned} r_{i,j} &\sim \text{Bernoulli}(p_{i,j}) \qquad i = 1, \ldots, 1000; j = 1, \ldots, 5 \\ \text{logit}(p_{i,j}) &= \beta\theta_i - \alpha_j \\ \theta_i &\sim \text{Normal}(0, 1) \\ \alpha_j &\sim \text{Normal}(0, 100) \\ \beta &\sim \text{Flat}(0, \infty), \end{aligned}$$

where $r_{i,j}$ is an indicator for correct response by student i to questions j.

Analysis Program

```
using Mamba

## Data
lsat = (Symbol => Any)[
  :culm =>
    [3, 9, 11, 22, 23, 24, 27, 31, 32, 40, 40, 56, 56, 59, 61, 76, 86, 115, 129,
     210, 213, 241, 256, 336, 352, 408, 429, 602, 613, 674, 702, 1000],
  :response =>
    [0 0 0 0 0
     0 0 0 0 1
     0 0 0 1 0
     0 0 0 1 1
     0 0 1 0 0
     0 0 1 0 1
     0 0 1 1 0
     0 0 1 1 1
     0 1 0 0 0
     0 1 0 0 1
     0 1 0 1 0
     0 1 0 1 1
     0 1 1 0 0
     0 1 1 0 1
     0 1 1 1 0
     0 1 1 1 1
     1 0 0 0 0
     1 0 0 0 1
     1 0 0 1 0
     1 0 0 1 1
     1 0 1 0 0
     1 0 1 0 1
     1 0 1 1 0
     1 0 1 1 1
```

```
     1 1 0 0 0
     1 1 0 0 1
     1 1 0 1 0
     1 1 0 1 1
     1 1 1 0 0
     1 1 1 0 1
     1 1 1 1 0
     1 1 1 1 1],
  :N => 1000
]
lsat[:R] = size(lsat[:response], 1)
lsat[:T] = size(lsat[:response], 2)

n = [lsat[:culm][1], diff(lsat[:culm])]
idx = mapreduce(i -> fill(i, n[i]), vcat, 1:length(n))
lsat[:r] = lsat[:response][idx,:]


## Model Specification

model = Model(

  r = Stochastic(2,
    @modelexpr(beta, theta, alpha, N, T,
      Distribution[
        begin
          p = invlogit(beta * theta[i] - alpha[j])
          Bernoulli(p)
        end
        for i in 1:N, j in 1:T
      ]
    ),
    false
  ),

  theta = Stochastic(1,
    :(Normal(0, 1)),
    false
  ),

  alpha = Stochastic(1,
    :(Normal(0, 100)),
    false
  ),

  a = Logical(1,
    @modelexpr(alpha,
      alpha - mean(alpha)
    )
  ),

  beta = Stochastic(
    :(Truncated(Flat(), 0, Inf))
  )

)
```

```
## Initial Values
inits = [
  [:r => lsat[:r], :alpha => zeros(lsat[:T]), :beta => 1,
   :theta => zeros(lsat[:N])],
  [:r => lsat[:r], :alpha => ones(lsat[:T]), :beta => 2,
   :theta => zeros(lsat[:N])]
]


## Sampling Scheme
scheme = [AMWG([:alpha], fill(0.1, lsat[:T])),
          Slice([:beta], [1.0]),
          Slice([:theta], fill(0.5, lsat[:N]))]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, lsat, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
        Mean         SD       Naive SE        MCSE         ESS
a[1] -1.2631137 0.1050811 0.0012133719 0.00222282916 2234.791
a[2]  0.4781765 0.0690242 0.0007970227 0.00126502181 2977.191
a[3]  1.2401762 0.0679403 0.0007845071 0.00155219697 1915.850
a[4]  0.1705201 0.0716647 0.0008275129 0.00146377676 2396.962
a[5] -0.6257592 0.0858278 0.0009910540 0.00156546941 3005.846
beta  0.7648299 0.0567109 0.0006548411 0.00277090485  418.880

Quantiles:
        2.5%       25.0%     50.0%      75.0%      97.5%
a[1] -1.4797642 -1.3325895 -1.262183 -1.1898184 -1.0647215
a[2]  0.3493393  0.4300223  0.477481  0.5253468  0.6143681
a[3]  1.1090783  1.1936967  1.240509  1.2865947  1.3699112
a[4]  0.0285454  0.1221579  0.170171  0.2191623  0.3100302
a[5] -0.7970834 -0.6828463 -0.625013 -0.5685627 -0.4590098
beta  0.6618355  0.7249738  0.761457  0.8013506  0.8839082
```

2.6.15 Bones: Latent Trait Model for Multiple Ordered Categorical Responses

An example from OpenBUGS *[38]*, Roche *et al.* *[65]*, and Thissen *[72]* concerning skeletal age in 13 boys predicted from 34 radiograph indicators of skeletal maturity.

Model

Skeletal ages are modelled as

$$\begin{aligned} \text{logit}(Q_{i,j,k}) &= \delta_j(\theta_i - \gamma_{j,k}) \qquad i = 1,\ldots,13; j = 1,\ldots,34; k = 1,\ldots,4 \\ \theta_i &\sim \text{Normal}(0, 100), \end{aligned}$$

where δ_j is a discriminability parameter for indicator j, $\gamma_{j,k}$ is a threshold parameter, and $Q_{i,j,k}$ is the cumulative probability that boy i with skeletal age θ_i is assigned a more mature grade than k.

Analysis Program

```
using Mamba

## Data
bones = (Symbol => Any)[
  :gamma => reshape(
    [ 0.7425,     NaN,     NaN,    NaN,  10.2670,     NaN,     NaN,     NaN,
     10.5215,     NaN,     NaN,    NaN,   9.3877,     NaN,     NaN,     NaN,
      0.2593,     NaN,     NaN,    NaN,  -0.5998,     NaN,     NaN,     NaN,
     10.5891,     NaN,     NaN,    NaN,   6.6701,     NaN,     NaN,     NaN,
      8.8921,     NaN,     NaN,    NaN,  12.4275,     NaN,     NaN,     NaN,
     12.4788,     NaN,     NaN,    NaN,  13.7778,     NaN,     NaN,     NaN,
      5.8374,     NaN,     NaN,    NaN,   6.9485,     NaN,     NaN,     NaN,
     13.7184,     NaN,     NaN,    NaN,  14.3476,     NaN,     NaN,     NaN,
      4.8066,     NaN,     NaN,    NaN,   9.1037,     NaN,     NaN,     NaN,
     10.7483,     NaN,     NaN,    NaN,   0.3887,  1.0153,     NaN,     NaN,
      3.2573,  7.0421,     NaN,    NaN,  11.6273, 14.4242,     NaN,     NaN,
     15.8842, 17.4685,     NaN,    NaN,  14.8926, 16.7409,     NaN,     NaN,
     15.5487, 16.8720,     NaN,    NaN,  15.4091, 17.0061,     NaN,     NaN,
      3.9216,  5.2099,     NaN,    NaN,  15.4750, 16.9406, 17.4944,     NaN,
      0.4927,  1.3556,  2.3016,  3.2535,  1.3059,  1.8793,  2.4970,  3.2306,
      1.5012,  1.8902,  2.3689,  2.9495,  0.8021,  2.3873,  3.9525,  5.3198,
      5.0022,  6.3704,  8.2832, 10.4988,  4.0168,  5.1537,  7.1053, 10.3038],
    4, 34)',
  :delta =>
    [2.9541, 0.6603, 0.7965, 1.0495, 5.7874, 3.8376, 0.6324,
     0.8272, 0.6968, 0.8747, 0.8136, 0.8246, 0.6711, 0.978,
     1.1528, 1.6923, 1.0331, 0.5381, 1.0688, 8.1123, 0.9974,
     1.2656, 1.1802, 1.368,  1.5435, 1.5006, 1.6766, 1.4297,
     3.385,  3.3085, 3.4007, 2.0906, 1.0954, 1.5329],
  :ncat =>
    [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
     3, 3, 3, 3, 3, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5],
  :grade => reshape(
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,2,1,1,2,1,1,
     2,1,1,1,2,2,1,1,1,1,1,1,1,1,1,1,1,1,1,3,1,1,1,1,1,1,1,1,3,1,1,2,1,1,
     2,1,1,1,2,2,1,1,1,1,1,1,1,1,1,1,1,1,1,3,1,1,1,1,1,1,1,1,4,3,3,3,1,1,
     2,1,1,1,2,2,1,1,1,1,1,1,NaN,1,1,1,1,1,1,3,1,1,1,1,1,1,1,1,4,5,4,3,1,1,
     2,1,1,1,2,2,1,1,2,1,1,1,1,1,1,1,2,1,1,3,2,1,1,1,1,1,3,1,5,5,5,4,2,3,
     2,1,1,1,2,2,1,2,1,1,1,1,1,2,1,1,2,NaN,1,3,2,1,1,1,1,1,3,1,5,5,5,5,3,3,
     2,1,1,1,2,2,1,1,1,NaN,NaN,1,1,1,1,1,2,NaN,1,3,3,1,1,1,1,1,3,1,5,5,5,5,3,3,
     2,1,2,2,2,2,2,2,1,NaN,NaN,1,2,2,1,1,2,2,1,3,2,1,1,1,1,1,3,1,5,5,5,5,3,4,
     2,1,1,2,2,2,2,2,2,1,1,1,2,1,1,1,2,1,1,3,3,1,1,1,1,1,3,1,5,5,5,5,4,4,
     2,1,2,2,2,2,2,2,2,1,1,1,2,2,2,1,2,NaN,2,3,3,1,1,1,1,1,3,1,5,5,5,5,5,5,
     2,1,NaN,2,2,2,NaN,2,2,1,NaN,NaN,2,2,NaN,NaN,2,1,2,3,3,NaN,1,NaN,1,1,3,1,5,5,5,5,5,5,
     2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,3,3,1,NaN,2,1,3,2,5,5,5,5,5,5,
```

```
      2,2,2,2,2,2,2,2,2,2,NaN,2,2,2,2,2,2,2,2,3,3,3,NaN,2,NaN,2,3,4,5,5,5,5,5,5],
     34, 13)',
  :nChild => 13,
  :nInd => 34
]


## Model Specification

model = Model(

  grade = Stochastic(2,
    @modelexpr(ncat, delta, theta, gamma, nChild, nInd,
      begin
        p = Array(Float64, 5)
        Distribution[
          begin
            n = ncat[j]
            p[1] = 1.0
            for k in 1:n-1
              Q = invlogit(delta[j] * (theta[i] - gamma[j,k]))
              p[k] -= Q
              p[k+1] = Q
            end
            Categorical(p[1:n])
          end
          for i in 1:nChild, j in 1:nInd
        ]
      end
    ),
    false
  ),

  theta = Stochastic(1,
    :(Normal(0, 100))
  )

)


## Initial Values
inits = [
  [:grade => bones[:grade], :theta => [0.5,1,2,3,5,6,7,8,9,12,13,16,18]],
  [:grade => bones[:grade], :theta => [1,2,3,4,5,6,7,8,9,10,11,12,13]]
]


## Sampling Scheme
scheme = [MISS([:grade]),
          AMWG([:theta], fill(0.1, bones[:nChild]))]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, bones, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
              Mean        SD      Naive SE       MCSE        ESS
 theta[1]  0.32603385 0.2064087 0.0023834028 0.005562488 1376.94938
 theta[2]  1.37861692 0.2582431 0.0029819342 0.007697049 1125.66422
 theta[3]  2.35227822 0.2799853 0.0032329913 0.008869417  996.50667
 theta[4]  2.90165730 0.2971332 0.0034309987 0.009730398  932.48357
 theta[5]  5.54427283 0.5024232 0.0058014839 0.022915189  480.72039
 theta[6]  6.70804782 0.5720689 0.0066056827 0.029251931  382.46138
 theta[7]  6.49138381 0.6015462 0.0069460578 0.030356237  392.68306
 theta[8]  8.93701249 0.7363614 0.0085027686 0.042741328  296.81508
 theta[9]  9.03585289 0.6517250 0.0075254717 0.031204552  436.20719
theta[10] 11.93125529 0.6936092 0.0080091090 0.039048344  315.51820
theta[11] 11.53686992 0.9227166 0.0106546132 0.065584299  197.94151
theta[12] 15.81482824 0.5426174 0.0062656056 0.028535483  361.59041
theta[13] 16.93028146 0.7245874 0.0083668145 0.043348628  279.40285

Quantiles:
             2.5%       25.0%      50.0%       75.0%        97.5%
 theta[1] -0.1121555  0.1955782  0.33881555  0.45840506  0.717456283
 theta[2]  0.9170535  1.1996943  1.36116575  1.53751273  1.946611947
 theta[3]  1.7828759  2.1713678  2.35623189  2.53035766  2.921158047
 theta[4]  2.3294082  2.6962175  2.89121336  3.10758151  3.494534287
 theta[5]  4.5914295  5.2054331  5.53392246  5.86525435  6.586724493
 theta[6]  5.5664907  6.3098380  6.70666338  7.09569168  7.822987248
 theta[7]  5.3866373  6.0762806  6.46533033  6.88636840  7.705137414
 theta[8]  7.4730453  8.4312561  8.96072241  9.45344704 10.285673300
 theta[9]  7.8047792  8.6055914  9.01498109  9.46962522 10.302472161
theta[10] 10.6412916 11.4837953 11.89611699 12.37737647 13.387304288
theta[11]  9.8355861 10.8871750 11.49029895 12.15757004 13.426345115
theta[12] 14.7925044 15.4588947 15.79840132 16.15824313 16.959330990
theta[13] 15.6184307 16.4228972 16.90719268 17.41900248 18.389576101
```

2.6.16 Inhalers: Ordered Categorical Data

An example from OpenBUGS *[38]* and Ezzet and Whitehead *[21]* concerning a two-treatment, two-period crossover trial comparing salbutamol inhalation devices in 286 asthma patients.

Model

Treatment responses are modelled as

$$
\begin{aligned}
R_{i,t} &= j \quad \text{if } Y_{i,t} \in [a_{j-1}, a_j) \qquad i = 1, \ldots, 4; t = 1, 2; j = 1, \ldots, 3 \\
\text{logit}(Q_{i,t,j}) &= -(a_j + \mu_{s_i,t} + b_i) \\
\mu_{1,1} &= \beta/2 + \pi/2 \\
\mu_{1,2} &= -\beta/2 - \pi/2 - \kappa \\
\mu_{2,1} &= -\beta/2 + \pi/2 \\
\mu_{2,2} &= \beta/2 - \pi/2 + \kappa \\
b_i &\sim \text{Normal}(0, \sigma) \\
a[1] &\sim \text{Flat}(-1000, a[2]) \\
a[2] &\sim \text{Flat}(-1000, a[3]) \\
a[3] &\sim \text{Flat}(-1000, 1000) \\
\beta &\sim \text{Normal}(0, 1000) \\
\pi &\sim \text{Normal}(0, 1000) \\
\kappa &\sim \text{Normal}(0, 1000) \\
\sigma^2 &\sim \text{InverseGamma}(0.001, 0.001),
\end{aligned}
$$

where $R_{i,t}$ is a 4-point ordinal rating of the device used by patient i, and $Q_{i,t,j}$ is the cumulative probability of the rating in treatment period t being worse than category j.

Analysis Program

```
using Mamba

## Data
inhalers = (Symbol => Any)[
  :pattern =>
    [1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4
     1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4]',
  :Ncum =>
    [ 59 157 173 175 186 253 270 271 271 278 280 281 282 285 285 286
     122 170 173 175 226 268 270 271 278 280 281 281 284 285 286 286]',
  :treat =>
    [ 1 -1
     -1  1],
  :period =>
    [1 -1
     1 -1],
  :carry =>
    [0 -1
     0  1],
  :N => 286,
  :T => 2,
  :G => 2,
  :Npattern => 16,
  :Ncut => 3
]

inhalers[:group] = Array(Int, inhalers[:N])
inhalers[:response] = Array(Int, inhalers[:N], inhalers[:T])
```

```
i = 1
for k in 1:inhalers[:Npattern], g in 1:inhalers[:G]
  while i <= inhalers[:Ncum][k,g]
    inhalers[:group][i] = g
    for t in 1:inhalers[:T]
      inhalers[:response][i,t] = inhalers[:pattern][k,t]
    end
    i += 1
  end
end


## Model Specification

model = Model(

  response = Stochastic(2,
    @modelexpr(a1, a2, a3, mu, group, b, N, T,
      begin
        a = Float64[a1, a2, a3]
        Distribution[
          begin
            eta = mu[group[i],t] + b[i]
            p = ones(4)
            for j in 1:3
              Q = invlogit(-(a[j] + eta))
              p[j] -= Q
              p[j+1] = Q
            end
            Categorical(p)
          end
          for i in 1:N, t in 1:T
        ]
      end
    ),
    false
  ),

  mu = Logical(2,
    @modelexpr(beta, treat, pi, period, kappa, carry, G, T,
      [ beta * treat[g,t] / 2 + pi * period[g,t] / 2 + kappa * carry[g,t]
        for g in 1:G, t in 1:T ]
    ),
    false
  ),

  b = Stochastic(1,
    @modelexpr(s2,
      Normal(0, sqrt(s2))
    ),
    false
  ),

  a1 = Stochastic(
    @modelexpr(a2,
      Truncated(Flat(), -1000, a2)
    )
  ),
```

```
  a2 = Stochastic(
    @modelexpr(a3,
      Truncated(Flat(), -1000, a3)
    )
  ),

  a3 = Stochastic(
    :(Truncated(Flat(), -1000, 1000))
  ),

  beta = Stochastic(
    :(Normal(0, 1000))
  ),

  pi = Stochastic(
    :(Normal(0, 1000))
  ),

  kappa = Stochastic(
    :(Normal(0, 1000))
  ),

  s2 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)


## Initial Values
inits = [
  [:response => inhalers[:response], :beta => 0, :pi => 0, :kappa => 0,
   :a1 => 2, :a2 => 3, :a3 => 4, :s2 => 1, :b => zeros(inhalers[:N])],
  [:response => inhalers[:response], :beta => 1, :pi => 1, :kappa => 0,
   :a1 => 3, :a2 => 4, :a3 => 5, :s2 => 10, :b => zeros(inhalers[:N])]
]


## Sampling Scheme
scheme = [AMWG([:b], fill(0.1, inhalers[:N])),
          Slice([:a1, :a2, :a3], fill(2.0, 3)),
          Slice([:beta, :pi, :kappa, :s2], fill(1.0, 4), :univar)]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, inhalers, inits, 5000, burnin=1000, thin=2, chains=2)
describe(sim)
```

Results

2.6.17 Mice: Weibull Regression

An example from OpenBUGS *[38]*, Grieve *[37]*, and Dellaportas and Smith *[19]* concerning time to death or censoring among four groups of 20 mice each.

Model

Time to events are modelled as

$$\begin{aligned} t_i &\sim \text{Weibull}(r, 1/\mu_i^r) \qquad i = 1, \ldots, 20 \\ \log(\mu_i) &= \boldsymbol{z}_i^\top \boldsymbol{\beta} \\ \beta_k &\sim \text{Normal}(0, 10) \\ r &\sim \text{Exponential}(1000), \end{aligned}$$

where t_i is the time of death for mouse i, and $\boldsymbol{z}_i$ is a vector of covariates.

Analysis Program

```
using Mamba

## Data
mice = (Symbol => Any)[
  :t =>
    [12  1  21 25 11  26  27  30 13 12  21 20  23  25  23  29 35 NaN 31 36
     32 27  23 12 18 NaN NaN  38 29 30 NaN 32 NaN NaN NaN NaN 25  30 37 27
     22 26 NaN 28 19  15  12  35 35 10  22 18 NaN  12 NaN NaN 31  24 37 29
     27 18  22 13 18  29  28 NaN 16 22  26 19 NaN NaN  17  28 26  12 17 26],
  :tcensor =>
    [0 0  0 0 0  0  0  0 0 0  0 0  0  0  0  0 0 40 0 0
     0 0  0 0 0 40 40  0 0 0 40 0 40 40 40 40 0  0 0 0
     0 0 10 0 0  0  0  0 0 0  0 0 24  0 40 40 0  0 0 0
     0 0  0 0 0  0  0 20 0 0  0 0 29 10  0  0 0  0 0 0]
]
mice[:M] = size(mice[:t], 1)
mice[:N] = size(mice[:t], 2)


## Model Specification

model = Model(

  t = Stochastic(2,
    @modelexpr(r, beta, tcensor, M, N,
      Distribution[
        begin
          lambda = exp(-beta[i] / r)
          0 < lambda < Inf ?
            Truncated(Weibull(r, lambda), tcensor[i,j], Inf) :
            Uniform(0, Inf)
        end
        for i in 1:M, j in 1:N
      ]
    ),
    false
  ),

  r = Stochastic(
    :(Exponential(1000))
  ),

  beta = Stochastic(1,
```

```
    :(Normal(0, 10)),
    false
  ),

  median = Logical(1,
    @modelexpr(beta, r,
      exp(-beta / r) * log(2)^(1/r)
    )
  ),

  veh_control = Logical(
    @modelexpr(beta,
      beta[2] - beta[1]
    )
  ),

  test_sub = Logical(
    @modelexpr(beta,
      beta[3] - beta[1]
    )
  ),

  pos_control = Logical(
    @modelexpr(beta,
      beta[4] - beta[1]
    )
  )

)


## Initial Values
inits = [
  [:t => mice[:t], :beta => fill(-1, mice[:M]), :r => 1.0],
  [:t => mice[:t], :beta => fill(-2, mice[:M]), :r => 1.0]
]


## Sampling Scheme
scheme = [MISS([:t]),
          Slice([:beta], fill(1.0, mice[:M]), :univar),
          Slice([:r], [0.25])]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, mice, inits, 20000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

2.6.18 Leuk: Cox Regression

An example from OpenBUGS *[38]* and Ezzet and Whitehead *[23]* concerning survival in 42 leukemia patients treated with 6-mercaptopurine or placebo.

Model

Times to death are modelled using the Bayesian Cox proportional hazards model, formulated by Clayton *[15]* as

$$
\begin{aligned}
dN_i(t) &\sim \text{Poisson}(I_i(t)dt) \qquad i = 1, \ldots, 42 \\
I_i(t)dt &= Y_i(t)\exp(\beta Z_i)d\Lambda_0(t) \\
\beta &\sim \text{Normal}(0, 1000) \\
d\Lambda_0(t) &\sim \text{Gamma}(cd\Lambda_0^*(t), 1/c) \\
d\Lambda_0^*(t) &= rdt \\
c &= 0.001 \\
r &= 0.1,
\end{aligned}
$$

where $dN_i(t)$ is a counting process increment in time interval $[t, t + dt)$ for patient i; $Y_i(t)$ is an indicator of whether the patient is observed at time t; $\boldsymbol{z}_i$ is a vector of covariates; and $d\Lambda_0(t)$ is the increment in the integrated baseline hazard function during $[t, t + dt)$.

Analysis Program

```
using Mamba

## Data
leuk = (Symbol => Any)[
  :t_obs =>
    [1, 1, 2, 2, 3, 4, 4, 5, 5, 8, 8, 8, 8, 11, 11, 12, 12, 15, 17, 22, 23, 6,
     6, 6, 6, 7, 9, 10, 10, 11, 13, 16, 17, 19, 20, 22, 23, 25, 32, 32, 34, 35],
  :fail =>
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
     1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0],
  :Z =>
    [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
     0.5, 0.5, 0.5, 0.5, 0.5, 0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5,
     -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5,
     -0.5, -0.5],
  :t => [1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 15, 16, 17, 22, 23, 35]
]
leuk[:N] = N = length(leuk[:t_obs])
leuk[:T] = T = length(leuk[:t]) - 1

leuk[:Y] = Array(Integer, N, T)
leuk[:dN] = Array(Integer, N, T)
for i in 1:N
  for j in 1:T
    leuk[:dN][i,j] = leuk[:fail][i] * (leuk[:t_obs][i] == leuk[:t][j])
    leuk[:Y][i,j] = int(leuk[:t_obs][i] >= leuk[:t][j])
  end
end

leuk[:c] = 0.001
leuk[:r] = 0.1


## Model Specification

model = Model(
```

```
  dN = Stochastic(2,
    @modelexpr(Y, beta, Z, dL0, N, T,
      Distribution[
        Y[i,j] > 0 ? Poisson(exp(beta * Z[i]) * dL0[j]) : Flat()
        for i in 1:N, j in 1:T
      ]
    ),
    false
  ),

  mu = Logical(1,
    @modelexpr(c, r, t,
      c * r * (t[2:end] - t[1:end-1])
    ),
    false
  ),

  dL0 = Stochastic(1,
    @modelexpr(mu, c, T,
      Distribution[Gamma(mu[j], 1 / c) for j in 1:T]
    ),
    false
  ),

  beta = Stochastic(
    :(Normal(0, 1000))
  ),

  S0 = Logical(1,
    @modelexpr(dL0,
      exp(-cumsum(dL0))
    ),
    false
  ),

  S_treat = Logical(1,
    @modelexpr(S0, beta,
      S0.^exp(-0.5 * beta)
    )
  ),

  S_placebo = Logical(1,
    @modelexpr(S0, beta,
      S0.^exp(0.5 * beta)
    )
  )

)


## Initial Values
inits = [
  [:dN => leuk[:dN], :beta => 0, :dL0 => fill(1, leuk[:T])],
  [:dN => leuk[:dN], :beta => 1, :dL0 => fill(2, leuk[:T])]
]


## Sampling Scheme
```

```
scheme = [AMWG([:dL0], fill(0.1, leuk[:T])),
          Slice([:beta], [3.0])]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, leuk, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
                 Mean          SD        Naive SE       MCSE        ESS
   S_treat[1] 0.98302184 0.014032858 0.00016203749 0.0006383128  483.3092
   S_treat[2] 0.96624643 0.020819923 0.00024040776 0.0007446993  781.6213
   S_treat[3] 0.95620713 0.024534430 0.00028329919 0.0009465788  671.7975
   S_treat[4] 0.93643295 0.031636973 0.00036531230 0.0013470515  551.5965
   S_treat[5] 0.91398224 0.037982886 0.00043858859 0.0015468336  602.9603
   S_treat[6] 0.87937232 0.047653044 0.00055024996 0.0019138759  619.9458
   S_treat[7] 0.86752882 0.051602796 0.00059585777 0.0021775423  561.5821
   S_treat[8] 0.82175096 0.064689997 0.00074697574 0.0026883215  579.0444
   S_treat[9] 0.80555769 0.068612901 0.00079227353 0.0028610945  575.1050
  S_treat[10] 0.77183154 0.076942474 0.00088845517 0.0032446983  562.3202
  S_treat[11] 0.73565702 0.085534730 0.00098766999 0.0037453183  521.5639
  S_treat[12] 0.71260021 0.089598298 0.00103459203 0.0035843383  624.8583
  S_treat[13] 0.69113536 0.094685927 0.00109333891 0.0039840537  564.8336
  S_treat[14] 0.66442349 0.098857036 0.00114150273 0.0043081307  526.5474
  S_treat[15] 0.63642300 0.102857125 0.00118769178 0.0044398707  536.6957
  S_treat[16] 0.56561600 0.112893079 0.00130357699 0.0049775555  514.4017
  S_treat[17] 0.47103433 0.120102602 0.00138682539 0.0051081652  552.8088
 S_placebo[1] 0.92778986 0.050170096 0.00057931437 0.0023131639  470.4106
 S_placebo[2] 0.85943183 0.067291385 0.00077701399 0.0022336707  907.5710
 S_placebo[3] 0.82080457 0.074342778 0.00085843646 0.0025543630  847.0564
 S_placebo[4] 0.74834420 0.085488930 0.00098714114 0.0032803421  679.1747
 S_placebo[5] 0.67108872 0.090818803 0.00104868521 0.0031480313  832.2877
 S_placebo[6] 0.56465561 0.097783399 0.00112910544 0.0033397837  857.2225
 S_placebo[7] 0.53217331 0.099628193 0.00115040728 0.0036366205  750.5308
 S_placebo[8] 0.41887442 0.097451300 0.00112527068 0.0033070732  868.3357
 S_placebo[9] 0.38321055 0.095687796 0.00110490749 0.0031667575  913.0267
S_placebo[10] 0.31712209 0.091201496 0.00105310416 0.0029020050  987.6603
S_placebo[11] 0.25673919 0.086400289 0.00099766461 0.0026927518 1029.5270
S_placebo[12] 0.22301415 0.082260904 0.00094986710 0.0021934098 1406.5249
S_placebo[13] 0.19554723 0.079430544 0.00091718492 0.0024666426 1036.9615
S_placebo[14] 0.16485544 0.074276591 0.00085767220 0.0025275203  863.6040
S_placebo[15] 0.13703275 0.068763778 0.00079401571 0.0024652242  778.0484
S_placebo[16] 0.08379600 0.054748991 0.00063218689 0.0020924163  684.6302
S_placebo[17] 0.04092034 0.037737842 0.00043575906 0.0013746671  753.6315
         beta 1.55206443 0.424977799 0.00490722093 0.0111217466 1460.1131

Quantiles:
                2.5%       25.0%      50.0%       75.0%       97.5%
   S_treat[1] 0.94621416 0.97721126 0.986955387 0.992864719 0.998368771
```

```
   S_treat[2] 0.91509516 0.95573219 0.970764331 0.981528195 0.993124279
   S_treat[3] 0.89749932 0.94319490 0.960671242 0.974341293 0.989582052
   S_treat[4] 0.85990686 0.91902353 0.941699242 0.959578177 0.981962464
   S_treat[5] 0.82397551 0.89249914 0.919841738 0.941506275 0.971669638
   S_treat[6] 0.77387551 0.85077283 0.885177533 0.913944652 0.955211053
   S_treat[7] 0.75318224 0.83615928 0.873267366 0.904856224 0.950378212
   S_treat[8] 0.68017136 0.78096730 0.828861810 0.868789540 0.928833884
   S_treat[9] 0.65553285 0.76283692 0.813277192 0.854664506 0.919302649
  S_treat[10] 0.60268487 0.72360801 0.779448002 0.826762737 0.901699132
  S_treat[11] 0.55030830 0.68188164 0.743447395 0.797449392 0.879907216
  S_treat[12] 0.52335655 0.65497052 0.720927823 0.777608202 0.864661623
  S_treat[13] 0.49228689 0.63108504 0.699537585 0.758920006 0.852628141
  S_treat[14] 0.45972979 0.59839293 0.670665550 0.735415731 0.838250322
  S_treat[15] 0.42697507 0.56653713 0.641228870 0.709931594 0.822082952
  S_treat[16] 0.34387359 0.48788044 0.566936069 0.644918756 0.777491390
  S_treat[17] 0.24667350 0.38339560 0.469857377 0.555064846 0.702808815
 S_placebo[1] 0.79979042 0.90318503 0.939595572 0.964924422 0.990890790
 S_placebo[2] 0.70559587 0.81931331 0.869352795 0.908385797 0.963186970
 S_placebo[3] 0.65639888 0.77312423 0.828159645 0.876023887 0.943371309
 S_placebo[4] 0.56607932 0.69244455 0.754818343 0.810412223 0.894624799
 S_placebo[5] 0.48516052 0.60948198 0.675431460 0.736195314 0.836155269
 S_placebo[6] 0.36968729 0.49714945 0.566089714 0.632288659 0.749328787
 S_placebo[7] 0.34028564 0.46293915 0.532290556 0.601897323 0.724841278
 S_placebo[8] 0.23796143 0.34940586 0.415075601 0.483568964 0.620503275
 S_placebo[9] 0.20912939 0.31494786 0.378301304 0.447909630 0.582109553
S_placebo[10] 0.15624653 0.25026510 0.311673305 0.378896633 0.510893913
S_placebo[11] 0.10926572 0.19326201 0.249637651 0.312589668 0.443600588
S_placebo[12] 0.08456597 0.16203246 0.216266931 0.276356661 0.404484250
S_placebo[13] 0.06502276 0.13721863 0.187654643 0.246139543 0.373111561
S_placebo[14] 0.04794312 0.11042966 0.156032281 0.209723502 0.331684320
S_placebo[15] 0.03427845 0.08617175 0.127242826 0.176712122 0.295007629
S_placebo[16] 0.01255699 0.04295264 0.072131767 0.112032446 0.220466119
S_placebo[17] 0.00216784 0.01395971 0.029822826 0.055822282 0.140786390
         beta 0.75242446 1.25909288 1.540715015 1.829373993 2.418625723
```

2.6.19 Jaws: Repeated Measures Analysis of Variance

An example from OpenBUGS *[38]* and Elston and Grizzle *[20]* concerning jaw bone heights measured repeatedly in a cohort of 20 boys at ages 8, 8.5, 9, and 9.5 years.

Model

Bone heights are modelled as

$$
\begin{aligned}
\boldsymbol{y}_i &\sim \text{Normal}(\boldsymbol{X}\boldsymbol{\beta}, \boldsymbol{\Sigma}) \qquad i = 1, \ldots, 20 \\
\boldsymbol{X} &= \begin{bmatrix} 1 & 8 \\ 1 & 8.5 \\ 1 & 9 \\ 1 & 9.5 \end{bmatrix} \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} \quad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_{1,1} & \sigma_{1,2} & \sigma_{1,3} & \sigma_{1,4} \\ \sigma_{2,1} & \sigma_{2,2} & \sigma_{2,3} & \sigma_{2,4} \\ \sigma_{3,1} & \sigma_{3,2} & \sigma_{3,3} & \sigma_{3,4} \\ \sigma_{4,1} & \sigma_{4,2} & \sigma_{4,3} & \sigma_{4,4} \end{bmatrix} \\
\beta_0, \beta_1 &\sim \text{Normal}(0, \sqrt{1000}) \\
\boldsymbol{\Sigma} &\sim \text{InverseWishart}(4, \boldsymbol{I})
\end{aligned}
$$

where $\boldsymbol{y}_i$ is a vector of the four repeated measurements for boy i. In the model specification below, the bone heights are arranged into a 1-dimensional vector on which a *Block-Diagonal Multivariate Normal Distribution* is specified.

Furthermore, since $\mathbf{\Sigma}$ is a covariance matrix, it is symmetric with `M * (M + 1) / 2` unique (upper or lower triangular) parameters, where `M` is the matrix dimension. Consequently, that is the number of parameters to account for when defining samplers for $\mathbf{\Sigma}$; e.g., `AMWG([:Sigma], fill(0.1, int(M * (M + 1) / 2)))`.

Analysis Program

```
using Mamba

## Data
jaws = (Symbol => Any)[
  :Y =>
   [47.8 48.8 49.0 49.7
    46.4 47.3 47.7 48.4
    46.3 46.8 47.8 48.5
    45.1 45.3 46.1 47.2
    47.6 48.5 48.9 49.3
    52.5 53.2 53.3 53.7
    51.2 53.0 54.3 54.5
    49.8 50.0 50.3 52.7
    48.1 50.8 52.3 54.4
    45.0 47.0 47.3 48.3
    51.2 51.4 51.6 51.9
    48.5 49.2 53.0 55.5
    52.1 52.8 53.7 55.0
    48.2 48.9 49.3 49.8
    49.6 50.4 51.2 51.8
    50.7 51.7 52.7 53.3
    47.2 47.7 48.4 49.5
    53.3 54.6 55.1 55.3
    46.2 47.5 48.1 48.4
    46.3 47.6 51.3 51.8],
  :age => [8.0, 8.5, 9.0, 9.5]
]
M = jaws[:M] = size(jaws[:Y], 2)
N = jaws[:N] = size(jaws[:Y], 1)
jaws[:y] = vec(jaws[:Y]')
jaws[:x] = kron(ones(jaws[:N]), jaws[:age])


## Model Specification
model = Model(

  y = Stochastic(1,
    @modelexpr(beta0, beta1, x, Sigma,
      BDiagNormal(beta0 + beta1 * x, Sigma)
    ),
    false
  ),

  beta0 = Stochastic(
    :(Normal(0, sqrt(1000)))
  ),

  beta1 = Stochastic(
    :(Normal(0, sqrt(1000)))
  ),
```

```
  Sigma = Stochastic(2,
    @modelexpr(M,
      InverseWishart(4.0, eye(M))
    )
  )

)


## Initial Values
inits = [
  [:y => jaws[:y], :beta0 => 40, :beta1 => 1, :Sigma => eye(M)],
  [:y => jaws[:y], :beta0 => 10, :beta1 => 10, :Sigma => eye(M)]
]


## Sampling Scheme
scheme = [Slice([:beta0, :beta1], [10, 1]),
          AMWG([:Sigma], fill(0.1, int(M * (M + 1) / 2)))]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, jaws, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
             Mean       SD      Naive SE     MCSE       ESS
     beta0 33.64206 1.9166361 0.022131407 0.06737008 809.36637
     beta1  1.87509 0.2164271 0.002499085 0.00776451 776.95308
Sigma[1,1]  7.38108 2.7056209 0.031241819 0.18590159 211.82035
Sigma[1,2]  7.19741 2.6922115 0.031086981 0.19045182 199.82422
Sigma[1,3]  6.78381 2.6747486 0.030885337 0.19225335 193.56111
Sigma[1,4]  6.53015 2.6851416 0.031005345 0.19272356 194.11754
Sigma[2,2]  7.53852 2.7705502 0.031991558 0.19576958 200.28188
Sigma[2,3]  7.20718 2.7647599 0.031924698 0.19814088 194.70032
Sigma[2,4]  6.96772 2.7775563 0.032072457 0.19836708 196.05889
Sigma[3,3]  8.03805 2.9649386 0.034236162 0.20539431 208.37930
Sigma[3,4]  8.00528 3.0132157 0.034793618 0.20660383 212.70794
Sigma[4,4]  8.58703 3.1870478 0.036800859 0.21045921 229.31967

Quantiles:
             2.5%     25.0%     50.0%     75.0%     97.5%
     beta0 29.82471 32.408405 33.654913 34.908169 37.422755
     beta1  1.45350  1.726719  1.873572  2.015534  2.307439
Sigma[1,1]  3.87079  5.403112  6.721379  8.779604 14.320056
Sigma[1,2]  3.69391  5.274856  6.515303  8.567864 14.053785
Sigma[1,3]  3.31306  4.888463  6.137073  8.156816 13.266370
Sigma[1,4]  3.02151  4.636266  5.913888  7.860503 13.159120
Sigma[2,2]  3.92078  5.576799  6.857077  8.950953 14.535064
```

```
Sigma[2,3]  3.61053  5.233882  6.521158  8.641108 14.040245
Sigma[2,4]  3.29096  4.997585  6.297002  8.382338 13.679823
Sigma[3,3]  4.16453  5.904915  7.321269  9.615860 15.037631
Sigma[3,4]  4.04242  5.850981  7.295268  9.625204 15.275975
Sigma[4,4]  4.37143  6.340437  7.843085 10.166456 16.465667
```

2.6.20 Eyes: Normal Mixture Model

An example from OpenBUGS *[38]*, Bowmaker *[6]*, and Robert *[62]* concerning 48 peak sensitivity wavelength measurements taken on a set of monkey's eyes.

Model

Measurements are modelled as the mixture distribution

$$
\begin{aligned}
y_i &\sim \text{Normal}(\lambda_{T_i}, \sigma) \qquad i = 1, \ldots, 48 \\
T_i &\sim \text{Categorical}(p, 1-p) \\
\lambda_1 &\sim \text{Normal}(0, 1000) \\
\lambda_2 &= \lambda_1 + \theta \\
\theta &\sim \text{Uniform}(0, 1000) \\
\sigma^2 &\sim \text{InverseGamma}(0.001, 0.001) \\
p &= \text{Uniform}(0, 1)
\end{aligned}
$$

where y_i is the measurement on monkey i.

Analysis Program

```
using Mamba

## Data
eyes = [
  :y =>
    [529.0, 530.0, 532.0, 533.1, 533.4, 533.6, 533.7, 534.1, 534.8, 535.3,
     535.4, 535.9, 536.1, 536.3, 536.4, 536.6, 537.0, 537.4, 537.5, 538.3,
     538.5, 538.6, 539.4, 539.6, 540.4, 540.8, 542.0, 542.8, 543.0, 543.5,
     543.8, 543.9, 545.3, 546.2, 548.8, 548.7, 548.9, 549.0, 549.4, 549.9,
     550.6, 551.2, 551.4, 551.5, 551.6, 552.8, 552.9, 553.2],
  :N => 48,
  :alpha => [1, 1]
]


## Model Specification

model = Model(

  y = Stochastic(1,
    @modelexpr(lambda, T, s2, N,
      begin
        sigma = sqrt(s2)
        Distribution[
          begin
```

```
            mu = lambda[T[i]]
            Normal(mu, sigma)
          end
          for i in 1:N
        ]
      end
    ),
    false
  ),

  T = Stochastic(1,
    @modelexpr(p, N,
      begin
        P = Float64[p, 1 - p]
        Distribution[Categorical(P) for i in 1:N]
      end
    ),
    false
  ),

  p = Stochastic(
    :(Uniform(0, 1))
  ),

  lambda = Logical(1,
    @modelexpr(lambda0, theta,
      Float64[lambda0, lambda0 + theta]
    )
  ),

  lambda0 = Stochastic(
    :(Normal(0.0, 1000.0)),
    false
  ),

  theta = Stochastic(
    :(Uniform(0.0, 1000.0)),
    false
  ),

  s2 = Stochastic(
    :(InverseGamma(0.001, 0.001))
  )

)


## Initial Values
inits = [
  [:y => eyes[:y], :T => fill(1, eyes[:N]), :p => 0.5, :lambda0 => 535,
   :theta => 5, :s2 => 10],
  [:y => eyes[:y], :T => fill(1, eyes[:N]), :p => 0.5, :lambda0 => 550,
   :theta => 1, :s2 => 1]
]


## Sampling Scheme
scheme = [DGS([:T]),
```

```
          Slice([:p, :lambda0, :theta, :s2], fill(1.0, 4), :univar, transform=true)]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, eyes, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

Empirical Posterior Estimates:
             Mean       SD      Naive SE      MCSE       ESS
lambda[1] 536.81290 0.9883483 0.011412463 0.052323509 356.8012
lambda[2] 548.75963 1.4660892 0.016928940 0.061653235 565.4693
        p   0.59646 0.0909798 0.001050544 0.003005307 916.4579
       s2  15.37891 7.1915331 0.083040671 0.406764080 312.5775

Quantiles:
             2.5%    25.0%     50.0%     75.0%     97.5%
lambda[1] 535.086 536.17470 536.75185 537.36318 538.9397
lambda[2] 545.199 548.07943 548.86953 549.63026 551.1739
        p   0.413   0.54206   0.60080   0.65619   0.7602
       s2   8.637  11.40687  13.68101  16.78394  39.0676
```

2.7 Conclusion

Mamba is a platform for the development and application of MCMC simulators for Bayesian modelling. Such simulators can be difficult to implement in practice. *Mamba* eases that task by standardizing and automating the generation of initial values, specification of distributions, iteration of Gibbs steps, updating of parameters, and running of MCMC chains. It automatically evaluates (unnormalized) full conditionals and allows MCMC simulators to be implemented by simply stating relationships between data, parameters, and statistical distributions, similar to the 'BUGS' clones and Stan program. In general, the package is designed to give users access to all levels of MCMC design and implementation. To that end, its toolset includes: 1) a model specification syntax, 2) stand-alone and integrated sampling functions, 3) a simulation engine and API, and 4) functions for convergence diagnostics and posterior inference. Moreover, its tools are designed to be modular so that they can be combined, extended, and used in ways that best meet users' needs.

Mamba can accommodate a wide range of model formulations as well as combinations of user-defined, supplied, and external samplers and distributions. It handles routine implementation tasks, thus allowing users to focus on design issues and making the package well-suited for

- developing new Bayesian models,
- implementing simulators for classes of models,
- testing new sampling algorithms,
- prototyping MCMC schemes for software development, and
- teaching MCMC methods.

Furthermore, output is easily generated with the simulation engine in a standardized format that can be analyzed directly with supplied or user-defined tools for convergence diagnostics and inference. Use of the package is illustrated with several examples. Future plans include additional sampler implementations and optimizations, development of alternative model-specification interfaces, and automatic specification of sampling schemes. The software is freely available and licensed under the open-source MIT license. It is hoped that the package will foster MCMC methods developed by researchers in the field and make their methods more accessible to a broader scientific community.

2.8 Supplement

2.8.1 Bayesian Linear Regression Model

The unnormalized posterior distribution was given for a *Bayesian Linear Regression Model* in the tutorial. Additional forms of that posterior are given in the following section.

Log-Transformed Distribution and Gradient

Let $\mathcal{L}$ denote the logarithm of a density of interest up to a normalizing constant, and $\nabla\mathcal{L}$ its gradient. Then, the following are obtained for the regression example parameters $\boldsymbol{\beta}$ and $\theta = \log(\sigma^2)$, for samplers, like NUTS, that can utilize both.

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\beta}, \theta|\boldsymbol{y}) &= \log(p(\boldsymbol{y}|\boldsymbol{\beta}, \theta)p(\boldsymbol{\beta})p(\theta)) \\
&= (-n/2 - \alpha_\pi)\theta - \frac{1}{\exp\{\theta\}}\left(\frac{1}{2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \beta_\pi\right) \\
&\quad - \frac{1}{2}(\boldsymbol{\beta} - \boldsymbol{\mu}_\pi)^\top\boldsymbol{\Sigma}_\pi^{-1}(\boldsymbol{\beta} - \boldsymbol{\mu}_\pi) \\
\nabla\mathcal{L}(\boldsymbol{\beta}, \theta|\boldsymbol{y}) &= \begin{bmatrix} \frac{1}{\exp\{\theta\}}\boldsymbol{X}^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) - \boldsymbol{\Sigma}_\pi^{-1}(\boldsymbol{\beta} - \boldsymbol{\mu}_\pi) \\ -n/2 - \alpha_\pi + \frac{1}{\exp\{\theta\}}\left(\frac{1}{2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \beta_\pi\right) \end{bmatrix}
\end{aligned}$$

2.9 References

2.10 Indices

- genindex

Bibliography

[1] D Bates, J M White, J Bezanson, S Karpinski, V B Shah, and other contributors. *Distributions*. 2014. julia software package. URL: https://github.com/JuliaStats/Distributions.jl.

[2] J Bezanson, S Karpinski, V B Shah, and A Edelman. Julia: a fast dynamic language for technical computing. *arXiv:1209.5145 [cs.PL]*, 2012. URL: http://arxiv.org/abs/1209.5145.

[3] J Bezanson, S Karpinski, V B Shah, and other contributors. The Julia Language. 2014. URL: http://julialang.org/.

[4] D Birkes and Y Dodge, editors. *Alternative Methods of Regression*. Wiley, New York, 1993.

[5] R D Boch and M Lieberman. Fitting a response model for n dichotomously scored items. *Psychometrika*, 35:179–197, 1970.

[6] J K Bowmaker, G H Jacobs, D J Spiegelhalter, and J D Mollon. Two types of trichromatic squirrel monkey share a pigment in the red-green region. *Vision Research*, 25:1937–1946, 1985.

[7] G E Box and G C Tiao, editors. *Bayesian Inference in Statistical Analysis*. Addison Wesley, Reading, MA, 1973.

[8] N E Breslow. Extra-Poisson variation in log-linear models. *Applied Statistics*, 33:38–44, 1984.

[9] N E Breslow and D G Clayton. Approximate inference in generalized linear mixed models. *Journal of the American Statistical Association*, 88:9–25, 1993.

[10] S Brooks and A Gelman. General methods for monitoring convergence of iterative simulations. *Journal of Computational and Graphical Statistics*, 7(4):434–455, 1998.

[11] S Brooks, A Gelman, G L Jones, and X-L Meng, editors. *Handbook of Markov Chain Monte Carlo*. Chapman & Hall/CRC, Boca Raton, FL, 2011.

[12] K A Brownlee, editor. *Statistical Theory and Methodology in Science and Engineering*. Wiley, New York, 1965.

[13] J B Carlin. Meta-analysis for 2 x 2 tables: a Bayesian approach. *Statistics in Medicine*, 11:141–159, 1992.

[14] M-H Chen and Q-M Shao. Monte Carlo estimation of Bayesian credible and HPD intervals. *Journal of Computational and Graphical Statistics*, 8(1):69–92, 1999.

[15] D Clayton. Bayesian analysis of frailty models. Technical Report, Medical Research Council Biostatistics Unit, Cambridge, 1994.

[16] M K Cowles and B P Carlin. Markov chain Monte Carlo convergence diagnostics: a comparative review. *Journal of the American Statistical Association*, 91:883–904, 1996.

[17] M Crowder. Beta-Binomial ANOVA for proportions. *Applied Statistics*, 27:34–37, 1978.

[18] O L Davies. *Statistical Methods in Research and Production*. Olver & Boyd, Edinburgh and London, 1967.

[19] P Dellaportas and A F M Smith. Bayesian inference for generalized linear and proportional hazards model via Gibbs sampling. *Applied Statistics*, 42:443–460, 1993.

[20] R C Elston and J E Grizzle. Estimation of time-response curves and their confidence bounds. *Biometrics*, 18:148–159, 1962.

[21] F Ezzet and J Whitehead. A random effects model for ordinal responses from a crossover trial. *Statistics in Medicine*, 10:901–907, 1993.

[22] Keno Fischer and other contributors. *GraphViz*. 2014. julia software package. URL: https://github.com/Keno/GraphViz.jl.

[23] E Frierich and E Gehan. The effect of 6-mercaptopurine on the duration of steroid-induced remissions in acute leukaemia: a model for evaluation of other potentially useful therapy. *Blood*, 21:699–716, 1963.

[24] D Gamerman. *Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference*. Chapman & Hall/CRC, Boca Raton, FL, 1997.

[25] A E Gelfand, S Hills, A Racine-Poon, and A F M Smith. Illustration of Bayesian inference in normal data models using Gibbs sampling. *Journal of the American Statistical Association*, 85:972–985, 1990.

[26] A E Gelfand and A F M Smith. Sampling based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 85:398–409, 1990.

[27] A Gelman, J B Carlin, H S Stern, D B Dunson, A V Vehtari, and Rubin D B. *Bayesian Data Analysis: Third Edition*. CRC Press, 2013.

[28] A Gelman, G O Roberts, and W R Gilks. Efficient Metropolis jumping rules. *Bayesian Statistics*, 5:599–607, 1996.

[29] A Gelman and D B Rubin. Inference from iterative simulation using multiple sequences. *Statistical Science*, 7:457–511, 1992.

[30] A Gelman, Y-S Su, M Yajima, J Hill, M G Pittau, J Kerman, T Zheng, and V Dorie. *arm: Data Analysis Using Regression and Multilevel/Hierarchical Models*. 2014. R software package. URL: http://CRAN.R-project.org/package=arm.

[31] S Geman and D Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.

[32] E I George, U E Makov, and A F M Smith. Conjugate likelihood distributions. *Scandinavian Journal of Statistics*, 20:147–156, 1993.

[33] J Geweke. *Bayesian Statistics*., chapter Evaluating the Accuracy of Sampling-Based Approaches to Calculating Posterior Moments. volume 4. Oxford University Press, New York, 1992.

[34] C J Geyer. Practical Markov chain Monte Carlo. *Statistical Science*, 7:473–511, 1992.

[35] W R Gilks, S Richardson, and D J Spiegelhalter, editors. *Monte Carlo in Practice*. Chapman & Hall/CRC, Boca Raton, FL, 1996.

[36] P W Glynn and W Whitt. Estimating the asymptotic variance with batch means. *Operations Research Letters*, 10:431–435, 1991.

[37] A P Grieve. Applications of Bayesian software: two examples. *Statistician*, 36:283–288, 1987.

[38] OpenBUGS Project Management Group. *OpenBUGS Examples Volume I*. 2014. version 3.2.3. URL: http://www.openbugs.net/Examples/Volumei.html.

[39] H Haario, E Saksman, and J Tamminen. An adaptive Metropolis algorithm. *Bernoulli*, 7:223–242, 2001.

[40] W K Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[41] P Heidelberger and P Welch. Simulation run length control in the presence of an initial transient. *Operations Research*, 31:1109–1144, 1983.

[42] M D Hoffman and A Gelman. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15:1593–1623, 2014. URL: http://jmlr.org/papers/v15/hoffman14a.html.

[43] Steven G Johnson, Fernando Perez, Jeff Bezanson, Stefan Karpinski, Keno Fischer, and other contributors. *IJulia*. 2015. julia software package. URL: https://github.com/JuliaLang/IJulia.jl.

[44] D C Jones. *Gadfly*. 2014. julia software package. URL: https://github.com/dcjones/Gadfly.jl.

[45] J G Kalbfleisch. *Probability and Statistical Inference: Volume 2*. Springer-Verlag, New York, 1985.

[46] D Lin, S Byrne, A N Jensen, D Bates, J M White, S Kornblith, and other contributors. *StatsBase*. 2014. julia software package. URL: https://github.com/JuliaStats/StatsBase.jl.

[47] D V Lindley and A F M Smith. Bayes estimates for the linear model (with discussion). *Journal of the Royal Statistical Society: Series B*, 34:1–44, 1972.

[48] D Lunn, D Spiegelhalter, A Thomas, and N Best. The BUGS project: evolution, critique and future directions. *Statistics in Medicine*, 28(25):3049–3067, 2009.

[49] A D Martin, K M Quinn, and J H Park. *MCMCpack: Markov Chain Monte Carlo (MCMC) Package*. 2013. R software package. URL: http://CRAN.R-project.org/package=MCMCpack.

[50] C McGilchrist and C Aisbett. Regression with frailty in survival analysis. *Biometrics*, 47:461–466, 1991.

[51] N Metropolis, A W Rosenbluth, M N Rosenbluth, A H Teller, and E Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[52] R M Neal. Slice sampling (with discussion). *Annals of Statistics*, 31:705–767, 2003.

[53] R M Neal. *Handbook of Markov Chain Monte Carlo*., chapter MCMC Using Hamiltonian Dynamics, pages 113–162. CRC Press, 2011.

[54] R M Neal. GRIMS – general R interface for Markov sampling. 2012. [Online; accessed 5-March-2014]. URL: http://www.cs.toronto.edu/~radford/GRIMS.html.

[55] J H Park. *CRAN Task View: Bayesian Inference*. 2014. version 2014-05-16. URL: http://cran.r-project.org/web/views/Bayesian.html.

[56] A Patil, D Huard, and C J Fonnesbeck. PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software*, 35(4):1–81, 2010.

[57] M Plummer. JAGS: a program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*. Vienna, Austria, March 20–22 2003. ISSN 1609-395X.

[58] M Plummer, N Best, K Cowles, and K Vines. CODA: convergence diagnosis and output analysis for MCMC. *R News*, 6(1):7–11, 2006.

[59] M Plummer, N Best, K Cowles, K Vines, D Sarkar, and R Almond. *coda: Output Analysis and Diagnostics for MCMC*. 2012. R software package. URL: http://CRAN.R-project.org/package=coda.

[60] A L Raftery and S Lewis. Comment: One long run with diagnostics: implementation strategies for Markov chain Monte Carlo. *Statistical Science*, 7(4):493–497, 1992.

[61] A L Raftery and S Lewis. *Bayesian Statistics*., chapter How Many Iterations in the Gibbs Sampler? volume 4. Oxford University Press, New York, 1992.

[62] C Robert. *Markov chain Monte Carlo in practice*., chapter Mixtures of distributions: inference and estimation. Chapman & Hall, 1994.

[63] C Robert and G Casella. *Monte Carlo Statistical Methods*. Springer, New York, 2nd edition, 2004.

[64] G O Roberts and J S Rosenthal. Examples of adaptive MCMC. *Journal of Computational and Graphical Statistics*, 18(2):349–367, 2009.

[65] A F Roche, H Wainer, and D Thissen. *Skeletal maturity: The knee joint as a biological indicator*. Plenum, New York, 1975.

[66] B J Smith. boa: an R package for MCMC output convergence assessment and posterior inference. *Journal of Statistical Computing*, 21(11):1–37, 2007.

[67] B J Smith. *boa: Bayesian Output Analysis Program for MCMC*. 2008. R software package. URL: http://CRAN.R-project.org/package=boa.

[68] B J Smith and other contributors. *Mamba: Markov Chain Monte Carlo for Bayesian Analysis in julia*. 2014. julia software package. URL: https://github.com/brian-j-smith/Mamba.jl.

[69] D Spiegelhalter, A Thomas, N Best, and W Gilks. *BUGS 0.5 Bayesian Inference Using Gibbs Sampling Manual (version ii)*. MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK, August 1996.

[70] D J Spiegelhalter, N G Best, B P Carlin, and A van der Linde. Bayesian measures of model complexity and fit (with discussion). *Journal of the Royal Statistical Society, Series B*, 64(4):583–639, 2002.

[71] P F Thall and S C Vail. Some covariance models for longitudinal count data with overdispersion. *Biometrics*, 46:657–671, 1990.

[72] D Thissen. *MULITLOG Version 5: User's Guide*. Scientific Software, Mooresville, IN, 5th edition, 1986.

[73] A Thomas. *OpenBUGS Developer Manual*. March 2014. version 3.2.3. URL: http://www.openbugs.net/Manuals/Developer/Manual.html.

[74] L Tierney. Markov chains for exploring posterior distributions (with discussion). *Annals of Statistics*, 22:1701–1762, 1994.

[75] D Wabersich and J Vandekerckhove. Extending JAGS: a tutorial on adding custom distributions to JAGS (with a diffusion model example). *Behavior Research Methods*, 2013. DOI 10.3758/s13428-013-0369-3.

[76] J M White and other contributors. *Calculus*. 2014. julia software package. URL: https://github.com/johnmyleswhite/Calculus.jl.

[77] J M White and other contributors. *Graphs*. 2014. julia software package. URL: https://github.com/JuliaLang/Graphs.jl.

[78] Stan Development Team. Stan: a C++ library for probability and sampling. 2014. URL: http://mc-stan.org/.

[79] Statisticat, LLC. *LaplacesDemon: Complete Environment for Bayesian Inference*. 2014. R software package. URL: http://www.bayesian-inference.com/software.

Symbols

A

C

D

E

G

V