Mamba.jl Documentation

Release 0.11.2

Brian J Smith

Mar 26, 2018

Contents

1 Overview **3**
- 1.1 Purpose . . . 3
- 1.2 Features . . . 3
- 1.3 Getting Started . . . 4

2 Contents **5**
- 2.1 Introduction . . . 5
- 2.2 Tutorial . . . 7
- 2.3 MCMC Types . . . 21
- 2.4 Sampling Functions . . . 56
- 2.5 Examples . . . 92
- 2.6 Discussion . . . 162
- 2.7 Supplement . . . 163
- 2.8 References . . . 163
- 2.9 Indices . . . 163

Bibliography **165**

Version 0.11.2

Requires julia releases 0.6.0 or later

Date Mar 26, 2018

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

License MIT

CHAPTER 1

Overview

1.1 Purpose

Mamba is an open platform for the implementation and application of MCMC methods to perform Bayesian analysis in julia. 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 intended to give users access to all levels of the design and implementation of MCMC simulators to particularly aid in the development of new methods.

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

  - adaptive Metropolis within Gibbs and multivariate Metropolis,
  - approximate Bayesian computation,
  - binary,
  - Hamiltonian Monte Carlo (simple and No-U-Turn),
  - simplex, and
  - slice 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.
- Importing of sampler output saved in the CODA file format.
- 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 *[13][28][39][78]*. 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 *[57][71]*. 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 [94][96]*. General model fitting is also available with the MCMC procedure found in the SAS/STAT ® software *[49]*. *Stan* is a newer and similar-in-scope program worth noting for its accessible syntax and automatically tuned Hamiltonian Monte Carlo sampling scheme *[98]*. *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 *[70]*. Programs like *GRIMS [66]* and *LaplacesDemon [99]* 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 pro-

vides Bayesian inference for generalized linear, ordered logistic or probit, and mixed-effects regression models *[34]*, *MCMCpack* fits a wide range of models commonly encountered in the social and behavioral sciences *[60]*, 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 *[69]*.

2.1.2 The Mamba Package

Mamba [88] is a **julia** *[4]* 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** *[3]*. 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 *[30][35]* 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.

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

(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.
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 *[89]* 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.

Model implementation begins by instantiating 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,
    (mu, s2) ->  MvNormal(mu, sqrt(s2)),
    false
  ),

  mu = Logical(1,
    (xmat, beta) -> xmat * 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 function that may contain any valid **julia** code. Functions should be defined to take, as their arguments, the inputs upon which their nodes depend and, for stochastic nodes, return distribution objects or arrays of objects compatible with the *Distributions* package *[2]*. Such objects represent the nodes' distributional specifications. An optional boolean argument after the function can be specified to indicate whether values of the node should be monitored (saved) during MCMC simulations (default: `true`).

Stochastic functions must return a single distribution object that can accommodate the dimensionality of the node, or return an array of (univariate or multivariate) distribution objects of the same dimension as the node. Examples of alternative, but equivalent, prior distributional 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,
  () -> UnivariateDistribution[Normal(0, sqrt(1000)), Normal(0, sqrt(1000))]
)

# Case 4: Array of univariate Normals
beta = Stochastic(1,
  () -> UnivariateDistribution[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. Another example is given in the *Pollution* example for the sampling of multiple nodes.

```
## User-Defined Samplers

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

Gibbs_s2 = Sampler([:s2],
  (mu, s2, y) ->
    begin
      a = length(y) / 2.0 + shape(s2.distr)
      b = sum(abs2, 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.

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 *LightGraphs* package *[11]*. 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 [50]*, and the *GraphViz* **julia** package *[26]* 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.

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.

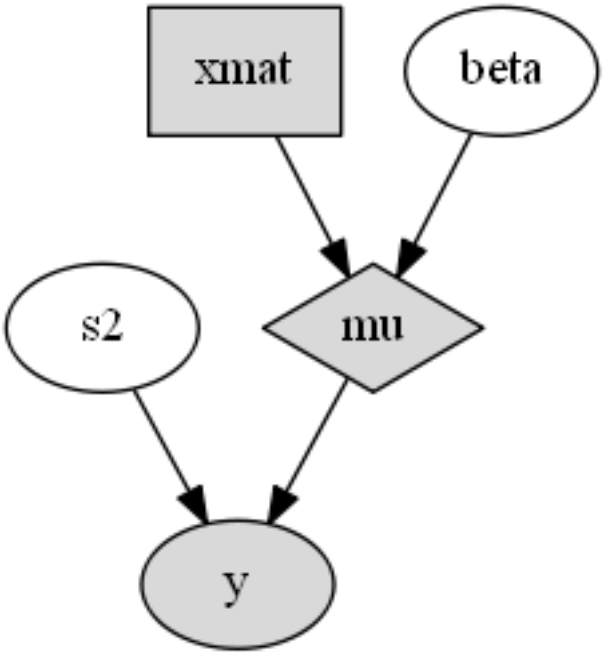

Fig. 2.2: Directed acyclic graph representation of the example regression model nodes.

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 = Dict{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 = [
  Dict{Symbol, Any}(
    :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. Results are returned as `Chains` objects on which methods for posterior inference are defined.

```
## 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)
```

Parallel Computing

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.

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 *[33][12]* 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%
     beta[1] 1.009 1.010
     beta[2] 1.009 1.010
          s2 1.008 1.016
Multivariate 1.006   NaN
```

The diagnostic of Geweke *[37]* 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
beta[1]   1.237  0.2162
beta[2]  -1.568  0.1168
     s2   1.710  0.0872

        Z-score p-value
beta[1]  -1.457  0.1452
beta[2]   1.752  0.0797
     s2  -1.428  0.1534

        Z-score p-value
beta[1]   0.550  0.5824
beta[2]  -0.440  0.6597
     s2   0.583  0.5596
```

The diagnostic of Heidelberger and Welch *[47]* 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
beta[1]     251            1  0.0680 0.57366275 0.053311283    1
beta[2]     738            1  0.0677 0.81285744 0.015404173    1
     s2     738            1  0.0700 1.00825202 0.094300432    1

        Burn-in Stationarity p-value    Mean     Halfwidth  Test
beta[1]     251            1  0.1356  0.6293320 0.065092099    0
beta[2]     251            1  0.0711  0.7934633 0.019215278    1
     s2     251            1  0.4435  1.4635400 0.588158612    0

        Burn-in Stationarity p-value    Mean     Halfwidth  Test
beta[1]     251            1  0.0515  0.5883602 0.058928034    0
beta[2]    1225            1  0.1479  0.8086080 0.018478999    1
     s2     251            1  0.6664  0.9942853 0.127959523    0
```

The diagnostic of Raftery and Lewis *[74][75]* 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
beta[1]        2     267      17897 3746          4.7776295
beta[2]        2     267      17897 3746          4.7776295
     s2        2     257       8689 3746          2.3195408

        Thinning Burn-in    Total   Nmin Dependence Factor
beta[1]        4     271 2.1759x104 3746          5.8085958
beta[2]        4     275 2.8795x104 3746          7.6868660
     s2        2     257 8.3450x103 3746          2.2277096

        Thinning Burn-in    Total   Nmin Dependence Factor
beta[1]        2     269 2.0647x104 3746          5.5117459
beta[2]        2     263 1.4523x104 3746          3.8769354
     s2        2     255 7.8770x103 3746          2.1027763
```

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 *[53]* 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
beta[1] 0.5971183 1.14894446 0.0095006014 0.016925598 4607.9743
beta[2] 0.8017036 0.34632566 0.0028637608 0.004793345 4875.0000
     s2 1.2203777 2.00876760 0.0166104638 0.101798287  389.3843

Quantiles:
            2.5%       25.0%       50.0%     75.0%     97.5%
beta[1] -1.74343373 0.026573102 0.59122696 1.1878720 2.8308472
beta[2]  0.12168742 0.628297573 0.80357822 0.9719441 1.5051573
     s2  0.17091385 0.383671702 0.65371989 1.2206381 6.0313970

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

         95% Lower  95% Upper
beta[1] -1.75436235 2.8109571
beta[2]  0.09721501 1.4733163
     s2  0.08338409 3.8706865

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

           beta[1]      beta[2]        s2
beta[1]  1.000000000 -0.905245029  0.027467317
```

```
beta[2] -0.905245029  1.000000000 -0.024489462
     s2  0.027467317 -0.024489462  1.000000000

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

           Lag 2       Lag 10        Lag 20       Lag 100
beta[1] 0.24521566  -0.021411797 -0.0077424153  -0.044989417
beta[2] 0.20402485  -0.019107846  0.0033980453  -0.053869216
     s2 0.85931351   0.568056917  0.3248136852   0.024157524

           Lag 2       Lag 10        Lag 20       Lag 100
beta[1] 0.28180489  -0.031007672    0.03930888  0.0394895028
beta[2] 0.25905976  -0.017946010    0.03613043  0.0227758214
     s2 0.92905843   0.761339226    0.58455868  0.0050215824

           Lag 2       Lag 10        Lag 20       Lag 100
beta[1] 0.38634357 -0.0029361782  -0.032310111  0.0028806786
beta[2] 0.32822879 -0.0056670786  -0.020100663 -0.0062622517
     s2 0.68812720  0.2420402859   0.080495078 -0.0290205896

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

             Change Rate
     beta[1]       0.844
     beta[2]       0.844
          s2       1.000
Multivariate       1.000

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

      DIC    Effective Parameters
pD 13.828540            1.1661193
pV 22.624104            5.5639015
```

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] 0.62445845 1.0285709 0.013275474 0.023818436 1864.8416
beta[2] 0.79392648 0.3096614 0.003996712 0.006516677 2001.0000
```

```
Quantiles:
            2.5%       25.0%       50.0%     75.0%     97.5%
beta[1] -1.53050898 0.076745702 0.61120944 1.2174641 2.6906753
beta[2]  0.18846617 0.618849048 0.79323126 0.9619767 1.4502109
```

File I/O

For cases in which it is desirable to store sampler output in external files for processing in future **julia** sessions, read and write methods are provided.

```
## Write to and Read from an External File
write("sim1.jls", sim1)
sim1 = read("sim1.jls", ModelChains)
```

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
beta[1] 0.59655228 1.1225920 0.0075471034 0.014053505 6380.79199
beta[2] 0.80144540 0.3395731 0.0022829250 0.003954871 7372.28048
     s2 1.18366563 1.8163096 0.0122109158 0.070481708  664.08995

Quantiles:
            2.5%       25.0%       50.0%     75.0%     97.5%
beta[1] -1.70512374 0.031582137 0.58989089 1.1783924 2.8253668
beta[2]  0.12399073 0.630638800 0.80358526 0.9703569 1.4939817
     s2  0.17075261 0.382963160 0.65372440 1.2210168 5.7449800
```

Plotting

Plotting of sampler output in *Mamba* is based on the *Gadfly* package *[51]*. 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")
```

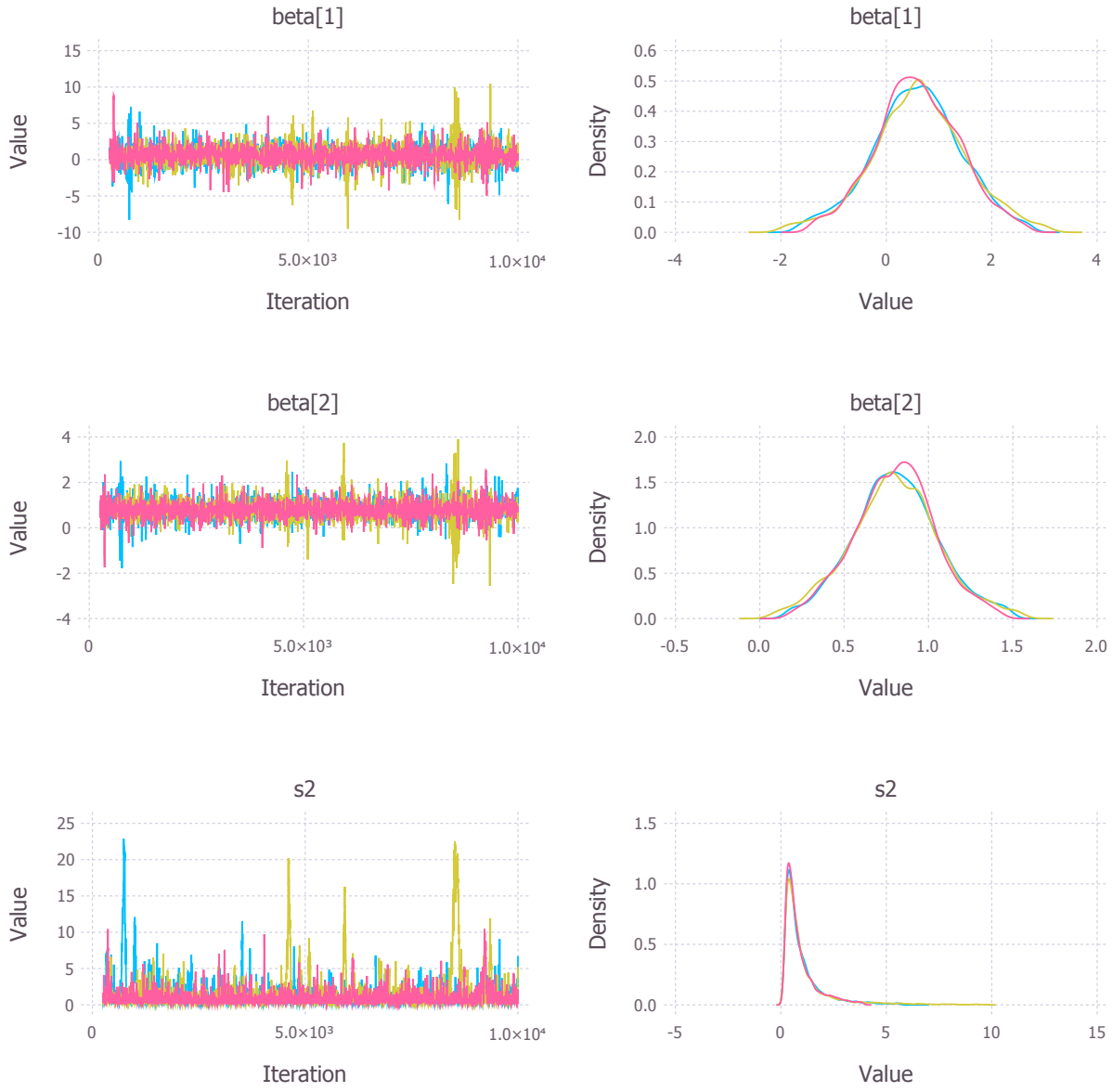

Fig. 2.3: Trace and density plots.

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")
```

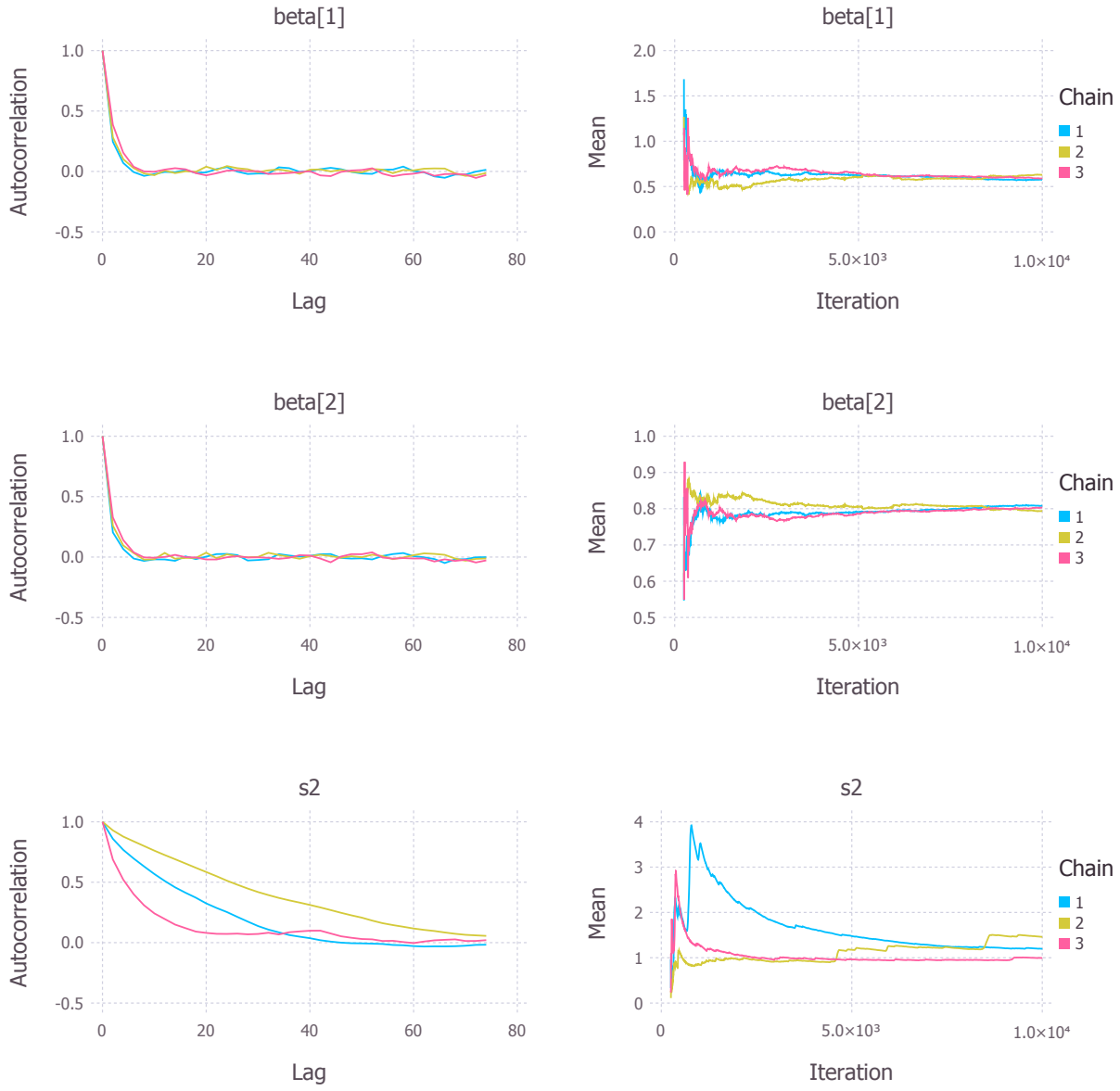

Fig. 2.4: Autocorrelation and running mean plots.

```
## Pairwise contour plots
p = plot(sim1, :contour)
draw(p, nrow=2, ncol=2, filename="contourplot.svg")
```

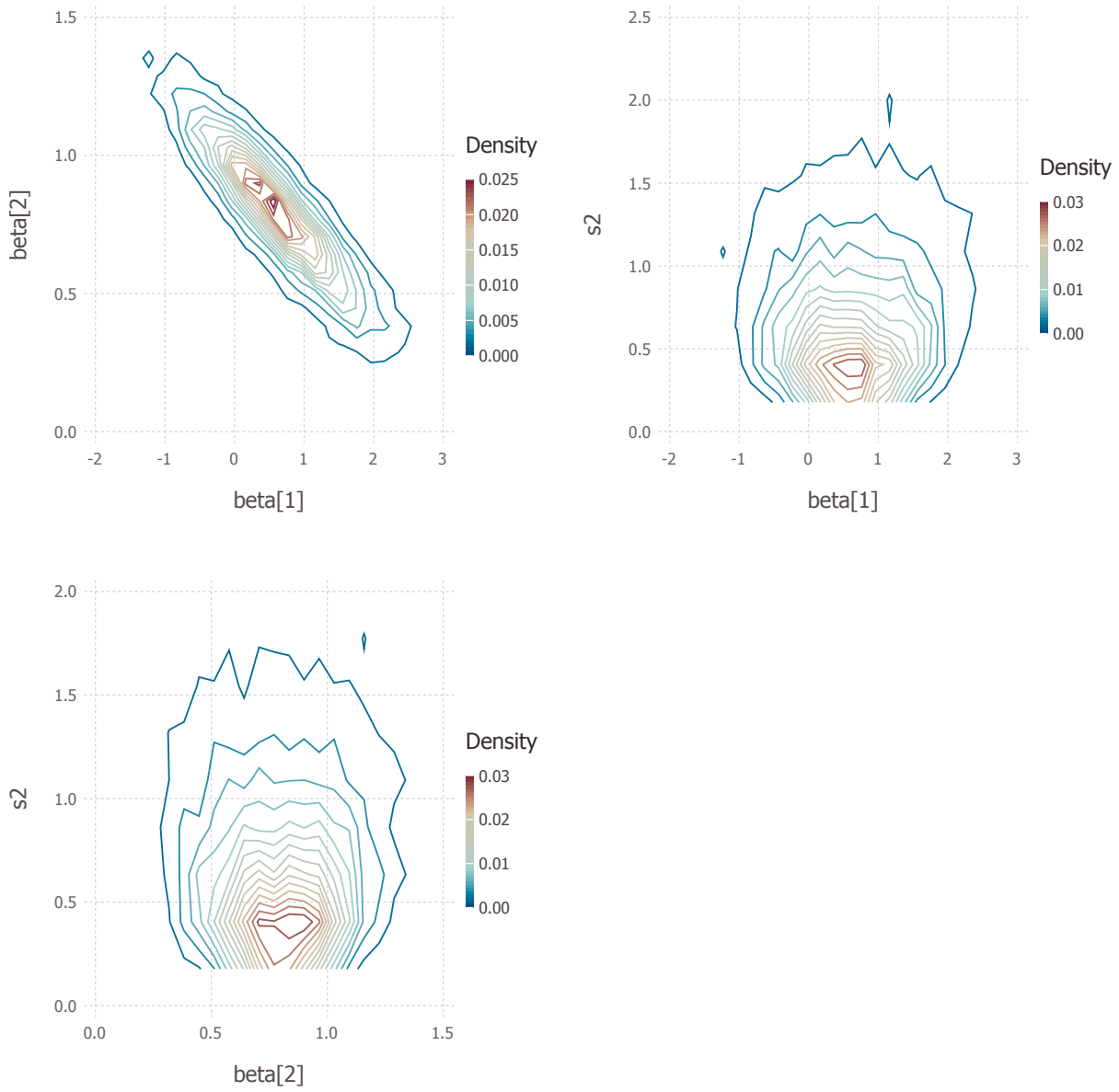

Fig. 2.5: Pairwise posterior density contour plots.

2.2.6 Computational Performance

Computing runtimes were recorded for different sampling algorithms applied to the regression example. Runs were 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

sample!(model, 1)                   # Sample values for block 1
sample!(model, 2)                   # Block 2
sample!(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 `sample!`. 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 `sample!` generates an MCMC sample of values 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 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.

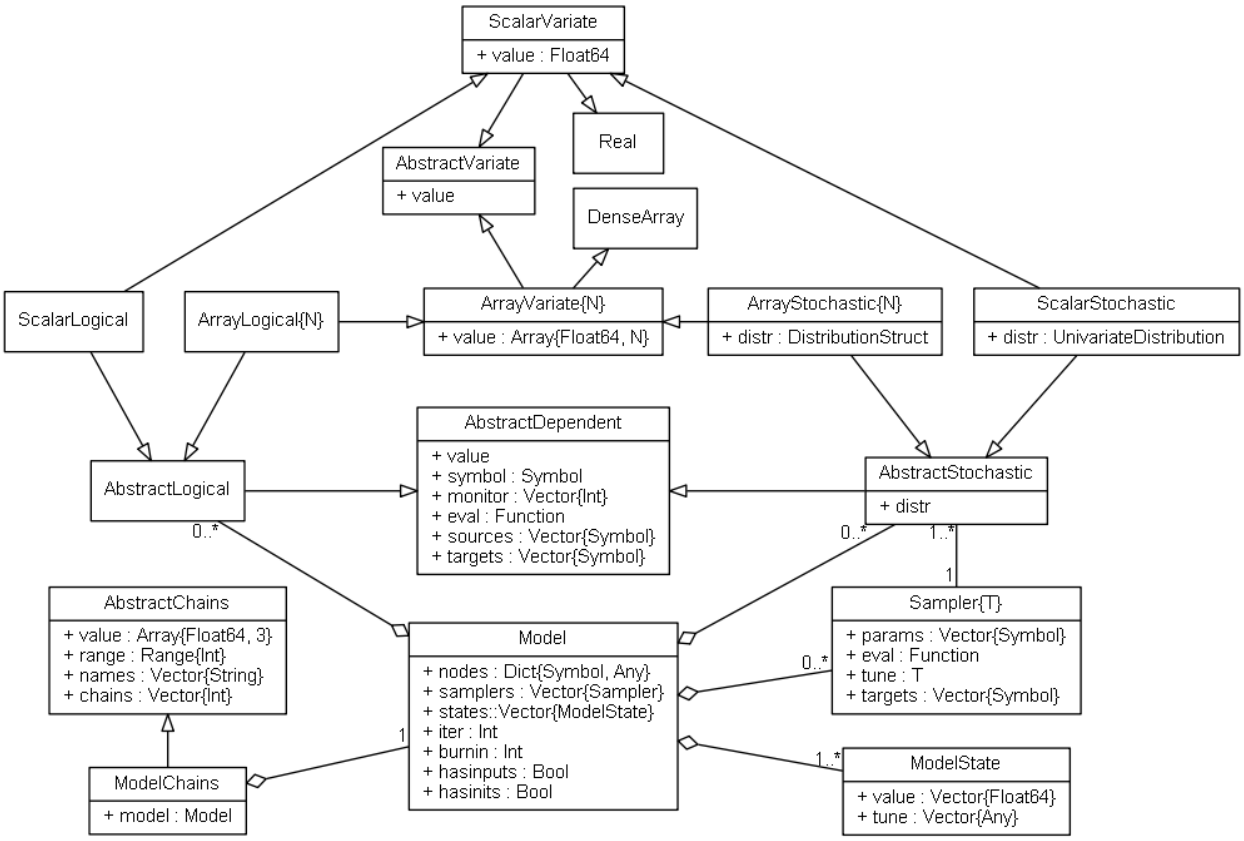

Fig. 2.6: UML relational diagram of *MCMC* types and their fields.

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** (`ScalarStochastic` or `ArrayStochastic`) are any model terms that have likelihood or prior distributional specifications.
- **Logical nodes** (`ScalarLogical` or `ArrayLogical`) 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 *Variate* types 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, `ModelChains` stores simulation output for a given model. Detailed information about each type is provided in the subsequent sections.

2.3.1 Variate

`ScalarVariate` and `ArrayVariate{N}` are abstract types that serve as the basis for several concrete types in the *Mamba* package. Conceptually, they represent data structures that store numeric values simulated from target distributions. Being abstract, these variate types cannot be instantiated and cannot have fields. They 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 method functions once on the abstract types (see julia Types). Accordingly, a core set of functionality is defined for the variate types through the field and method functions discussed below. Although the (abstract) types do not have fields, their method functions assume that all subtypes will be declared with a `value` field.

Declarations

```
abstract type ScalarVariate <: Real
abstract type ArrayVariate{N} <: DenseArray{Float64, N}

const AbstractVariate = Union{ScalarVariate, ArrayVariate}
const VectorVariate = ArrayVariate{1}
const MatrixVariate = ArrayVariate{2}
```

Type Hierarchy

Subtypes of the variate types include the *Dependent*, *Logical*, *Stochastic*, and *SamplerVariate* types.

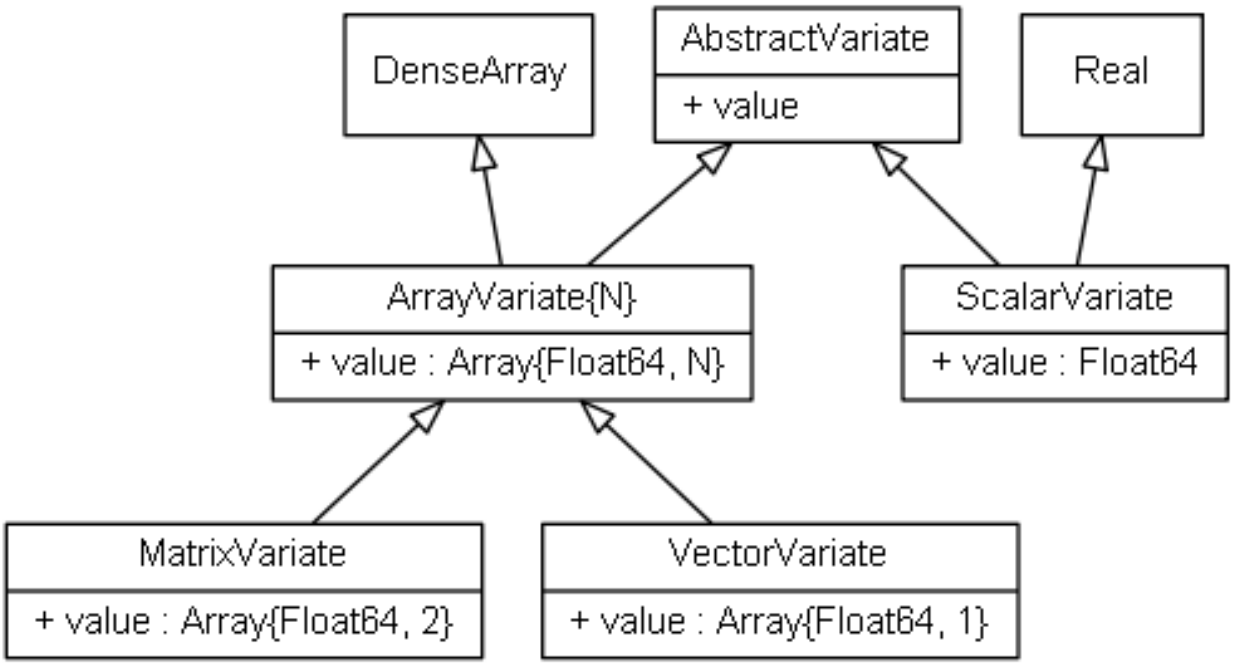

Fig. 2.7: UML relational diagram of `Variate` types and their fields.

Field

- `value::T` : scalar or array of `Float64` values that represent simulated values from a target distribution.

Methods

Methods for `ScalarVariate` and `ArrayVariate` include mathematical operators, mathematical functions, and statistics defined in the base **julia** language for parent types `Real` and `DenseArray`. In addition, the following

functions are provided.

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

2.3.2 Dependent

`AbstractDependent` 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` types 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 `Float64` or `Array{Float64, N}` type objects.

Since parameter values in the `AbstractDependent` 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 `AbstractDependent` for each one. At the other end, a formulation might be made in terms of a single parameter array, with one corresponding instance of `AbstractDependent`. 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

```
const AbstractDependent = Union{AbstractLogical, AbstractStochastic}
```

Fields

- `value::T` : scalar or array of `Float64` values that represent samples from a target distribution.
- `symbol::Symbol` : identifying symbol for the node.
- `monitor::Vector{Int}` : indices identifying elements of the `value` field to include in monitored MCMC sampler output.
- `eval::Function` : function for updating the state of the node.
- `sources::Vector{Symbol}` : other nodes upon whom the values of this one depends.
- `targets::Vector{Symbol}` : `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.

Display

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

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

Initialization

setmonitor! (*d::AbstractDependent*, *monitor::Bool*)
setmonitor! (*d::AbstractDependent*, *monitor::Vector{Int}*)

Specify node elements to be included in monitored MCMC sampler output.

Arguments

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

Value

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

Node Operations

logpdf (*d::AbstractDependent*, *transform::Bool=false*)
logpdf (*d::AbstractDependent*, *x*, *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 `AbstractDependent` that have distributional specifications.

Arguments

- `d` : node for which to evaluate the log-density.
- `x` : value, of the same type and shape as the node value, at which to perform the evaluation. If not specified, the node value is used.
- `transform` : whether the evaluation is on the link-transformed scale.

Value

The resulting numeric value of the log-density.

unlist (*d::AbstractDependent*, *transform::Bool=false*)
unlist (*d::AbstractDependent*, *x::Real*, *transform::Bool=false*)
unlist (*d::AbstractDependent*, *x::AbstractArray*, *transform::Bool=false*)
relist (*d::AbstractDependent*, *x::AbstractArray*, *transform::Bool=false*)

Extract (unlist) node values to a vector, or re-assemble (relist) values to be put into a node. In this generic method, all values are listed. The methods are used internally for the extraction of unique stochastic node values to sample, and can be redefined to implement different behaviors for `AbstractDependent` subtypes.

Arguments

- `d` : node for which to unlist or relist values.
- `x` : values to be listed. If not specified, the node values are used.
- `transform` : whether to apply a link or inverse-link transformation to the values. In this generic method, transformations are defined to be the identity function.

Value

Returns unmodified `x` values as a vector (unlist) or in the same shape as the specified node (relist).

2.3.3 Logical

The `Logical` types inherit fields and method functions from the `AbstractDependent` type, and adds the constructors and methods listed below. It is designed for nodes that are deterministic functions of model parameters and data.

Declarations

```
type ScalarLogical <: ScalarVariate
type ArrayLogical{N} <: ArrayVariate{N}
const AbstractLogical = Union{ScalarLogical, ArrayLogical}
```

Fields

- `value` : values of type `Float64` for `ScalarLogical` nodes and `Array{Float64}` for `ArrayLogical` nodes that represent samples from a target distribution.
- `symbol::Symbol` : identifying symbol for the node.
- `monitor::Vector{Int}` : indices identifying elements of the `value` field to include in monitored MCMC sampler output.
- `eval::Function` : function for updating values stored in `value`.
- `sources::Vector{Symbol}` : other nodes upon whom the values of this one depends.
- `targets::Vector{Symbol}` : `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 (*f::Function*, *monitor::Union{Bool, Vector{Int}}=true*)
Logical (*f::Function*, *d::Integer*, *monitor::Union{Bool, Vector{Int}}=true*)
Logical (*d::Integer*, *f::Function*, *monitor::Union{Bool, Vector{Int}}=true*)

Construct a `Logical` object that defines a logical model node.

Arguments

- `d` : number of dimensions for array nodes.
- `f` : function whose untyped arguments are the other model nodes upon which this one depends. The function may contain any valid **julia** expression or code block. It will be saved in the `eval` field of the constructed logical node and should return a value in the same type as and with which to update the node's `value` field.
- `monitor` : boolean indicating whether all elements are monitored, or vector of element-wise indices of elements to monitor.

Value

Returns an `ArrayLogical` if the dimension argument `d` is specified, and a `ScalarLogical` if not.

Example

See the *Model Specification* section of the tutorial.

Initialization

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

Arguments

- `l` : logical node to which to assign initial values.
- `m` : model containing the node.

Value

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

Node Operations

update!(*l::AbstractLogical*, *m::Model*)
Update the values of a logical node according to its relationship with others in a model.

Arguments

- `l` : logical node to update.
- `m` : model containing the node.

Value

Returns the node with its values updated.

2.3.4 Stochastic

The `Stochastic` types inherit fields and method functions from the `AbstractDependent` 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 structure stored in `distr` field.

Declarations

```
type ScalarStochastic <: ScalarVariate
type ArrayStochastic{N} <: ArrayVariate{N}
const AbstractStochastic = Union{ScalarStochastic, ArrayStochastic}
```

Fields

- `value` : values of type `Float64` for `ScalarStochastic` nodes and `Array{Float64}` for `ArrayStochastic` nodes that represent samples from a target distribution.
- `symbol::Symbol` : identifying symbol for the node.
- `monitor::Vector{Int}` : indices identifying elements of the `value` field to include in monitored MCMC sampler output.
- `eval::Function` : function for updating the `distr` field for the node.
- `sources::Vector{Symbol}` : other nodes upon whom the distributional specification for this one depends.

- `targets::Vector{Symbol}` : `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` : distributional specification of type `UnivariateDistribution` for `ScalarStochastic` nodes and `DistributionStruct` for `ArrayStochastic` nodes.

Distribution Structures

The `DistributionStruct` alias defines the types of distribution structures supported for `AbstractStochastic` nodes. Single `Distribution` types from the *Distributions* section, arrays of `UnivariateDistribution`, and arrays of `MultivariateDistribution` objects are supported. When a `MultivariateDistribution` array is specified for a stochastic node, the node is assumed to be one dimension bigger than the array, with the last dimension containing values from the distributions stored in the previous dimensions. Such arrays may contain distributions of different lengths. Model specification syntax for all three types of distribution structures can be seen in the *Birats Example*.

```
const DistributionStruct = Union{Distribution,
                                 Array{UnivariateDistribution},
                                 Array{MultivariateDistribution}}
```

Constructors

Stochastic(*f::Function*, *monitor::Union{Bool, Vector{Int}}=true*)
Stochastic(*f::Function*, *d::Integer*, *monitor::Union{Bool, Vector{Int}}=true*)
Stochastic(*d::Integer*, *f::Function*, *monitor::Union{Bool, Vector{Int}}=true*)

Construct a `Stochastic` object that defines a stochastic model node.

Arguments

- `d` : number of dimensions for array nodes.
- `f` : function whose untyped arguments are the other model nodes upon which this one depends. The function may contain any valid **julia** expression or code block. It will be saved in the `eval` field of the constructed stochastic node and should return a `DistributionStruct` object to be stored in the node's `distr` field.
- `monitor` : boolean indicating whether all elements are monitored, or vector of element-wise indices of elements to monitor.

Value

Returns an `ArrayStochastic` if the dimension argument `d` is specified, and a `ScalarStochastic` if not.

Example

See the *Model Specification* section of the tutorial.

Initialization

setinits!(*s::Stochastic*, *m::Model*, *x=nothing*)

Set initial values for a stochastic node.

Arguments

- `s` : stochastic node to which to assign initial values.

- `m` : model containing the node.
- `x` : values to assign to the node.

Value

Returns the node with its assigned initial values.

Node Operations

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

Arguments

- `s` : stochastic node for which to evaluate the log-density.
- `x` : value, of the same type and shape as the node value, at which to perform the evaluation. If not specified, the node value is used.
- `transform` : whether the evaluation is on the link-transformed scale.

Value

The resulting numeric value of the log-density.

rand(*s::AbstractStochastic*)
Draw a sample from the distributional specification on a stochastic node.

Arguments

- `s` : stochastic node from which to generate a random sample.

Value

Returns the sampled value(s).

unlist(*s::AbstractStochastic*, *transform::Bool=false*)
unlist(*s::AbstractStochastic*, *x::Real*, *transform::Bool=false*)
unlist(*s::AbstractStochastic*, *x::AbstractArray*, *transform::Bool=false*)
relist(*s::AbstractStochastic*, *x::AbstractArray*, *transform::Bool=false*)
Extract (unlist) stochastic node values to a vector, or re-assemble (relist) values into a format that can be put into a node. These methods are used internally to extract the unique and sampled values of stochastic nodes. They are used, for instance, to extract only the unique, upper-triangular portions of (symmetric) covariance matrices and only the sampled values of `Array{MultivariateDistribution}` specifications whose distributions may be of different lengths.

Arguments

- `s` : stochastic node for which to unlist or relist values.
- `x` : values to be listed. If not specified, the node values are used.
- `transform` : whether to apply a link transformation, or its inverse, 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 it with the diagonal elements log-transformed.

Value

Returns the extracted `x` values as a vector or the re-assembled values in the same shape as the specified node.

`update!`(*s::AbstractStochastic*, *m::Model*)

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

Arguments

- `s` : stochastic node to update.
- `m` : model containing the node.

Value

Returns the node with its values updated.

2.3.5 Distributions

Given in this section are distributions, as provided by the *Distributions [2]* 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          DiscreteUniform   InverseGamma        NoncentralChisq          ␣
↪SymTriangularDist
Bernoulli        Edgeworth         InverseGaussian     NoncentralF               TDist
Beta             Epanechnikov      Kolmogorov          NoncentralHypergeometric ␣
↪TriangularDist
BetaPrime        Erlang            KSDist              NoncentralT              ␣
↪Triweight
Binomial         Exponential       KSOneSided          Normal                    Uniform
Biweight         FDist             Laplace             NormalCanon               VonMises
Categorical      Frechet           Levy                Pareto                    Weibull
Cauchy           Gamma             Logistic            PoissonBinomial
Chi              Geometric         LogNormal           Poisson
Chisq            Gumbel            NegativeBinomial    Rayleigh
Cosine           Hypergeometric    NoncentralBeta      Skellam
```

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 any of its methods (indicated below) that are extended.
3. Declare the new distribution subtype, say `D`, within the block. Any constructors explicitly defined for the subtype should accept un-typed or abstract-type (`Real`, `AbstractArray`, or `DenseArray`) arguments. Implementing constructors in this way ensures that they 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(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
  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
using Mamba
@everywhere eval(extensions)

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

  y = Stochastic(1,
    (mu, s2) ->
      begin
        sigma = sqrt(s2)
        UnivariateDistribution[
          NewUnivarDist(mu[i], sigma) for i in 1:length(mu)
        ]
      end,
    false
  ),

  mu = Logical(1,
    (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 = Dict{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   MvNormal   MvTDist   Multinomial   MvNormalCanon   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 any of its methods (indicated below) that are extended.
3. Declare the new distribution subtype, say `D`, within the block. Any constructors explicitly defined for the subtype should accept un-typed or abstract-type (`Real`, `AbstractArray`, or `DenseArray`) arguments. Implementing constructors in this way ensures that they 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::AbstractVector{T}*)
   Return a logical indicating whether `x` is in the support of `d`.

   `_logpdf{T<:Real}` (*d::D*, *x::AbstractVector{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(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
  end

  ## The following method functions must be implemented

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

  ## Logical indicating whether x is in the support
  function insupport{T<:Real}(d::NewMultivarDist, x::AbstractVector{T})
    length(d) == length(x) && all(isfinite.(x))
  end

  ## Normalized or unnormalized log-density value
  function _logpdf{T<:Real}(d::NewMultivarDist, x::AbstractVector{T})
    -length(x) * log(d.sigma) - 0.5 * sum(abs2, 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
using Mamba
@everywhere eval(extensions)

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

  y = Stochastic(1,
    (mu, s2) -> NewMultivarDist(mu, sqrt(s2)),
    false
  ),

  mu = Logical(1,
    (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 = Dict{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
```

2.3.6 Sampler

The `Sampler` type stores model-based *Sampling Functions* for use in the *Mamba Gibbs sampling scheme*. Developers can use it as a wrapper for calling stand-alone samplers or as a structure for implementing self-contained samplers.

Declaration

```
type Sampler{T}
```

Fields

- `params::Vector{Symbol}` : stochastic nodes in the block being updated by the sampler.
- `eval::Function` : sampling function that updates values of the `params` nodes.
- `tune::T` : tuning parameters needed by the sampling function.
- `targets::Vector{Symbol}` : `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.

Constructors

`Sampler` (*param::Symbol*, *f::Function*, *tune::Any=Dict()*)
`Sampler` (*params::Vector{Symbol}*, *f::Function*, *tune::Any=Dict()*)

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

Arguments

- `param/params` : node(s) being block-updated by the sampler.
- `f` : function for the `eval` field of the constructed sampler and whose arguments are the other model nodes upon which the sampler depends, typed argument `model::Model` that contains all model nodes, and/or typed argument `block::Integer` that 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.

- `tune` : tuning parameters needed by the sampling function.

Value

Returns a `Sampler{typeof(tune)}` type object.

Example

See the *Sampling Schemes* section of the tutorial.

Display

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.

2.3.7 SamplerVariate

The `SamplerVariate` type is designed to store simulated values from and tuning parameters for stand-alone *Sampling Functions*. It is a parametric type whose parameter can be any subtype of the abstract `SamplerTune` type and serves to identify the family of sampling functions to which the variate belongs.

Declaration

```
abstract type SamplerTune
type SamplerVariate{T<:SamplerTune} <: VectorVariate
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::T` : tuning parameters.

Constructors

SamplerVariate(*x::AbstractVector{U<:Real}*, *tune::SamplerTune*)
SamplerVariate{T<:SamplerTune}(*x::AbstractVector{U<:Real}*, *tune::T*)
SamplerVariate{T<:SamplerTune}(*x::AbstractVector{U<:Real}*, *pargs...; kargs...*)
Construct a `SamplerVariate` object for storing simulated values and tuning parameters.

Arguments

- `x` : simulated values.
- `tune` : tuning parameters. If not specified, the tuning parameter constructor is called with the `value` field of the variate to instantiate the parameters.

- `T` : explicit tuning parameter type for the variate. If not specified, the type is inferred from the `tune` argument.
- `pargs...`, `kargs...` : variable positional and keyword arguments that are passed, along with the `value` field of the variate, to the tuning parameter constructor as `T(value, pargs...; kargs...)`. Accordingly, the arguments that this version of the `SamplerVariate{T}` constructor accepts are defined by the `T` constructors implemented for it.

Value

Returns a `SamplerVariate{T}` type object with fields set to the supplied `x` and tuning parameter values.

2.3.8 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}` : all input, logical, and stochastic model nodes.
- `samplers::Vector{Sampler}` : sampling functions for updating blocks of stochastic nodes.
- `states::Vector{ModelState}` : states of chains at the end of an MCMC run in a possible series of runs, where `ModelState` has fields `value::Vector{Float64}` and `tune::Vector{Any}` to store the last values of sampled nodes and block-sampler tuning parameters, respectively.
- `iter::Int` : current MCMC draw from the target distribution.
- `burnin::Int` : number of initial draws to discard as a burn-in sequence to allow for convergence.
- `hasinputs::Bool` : whether values have been assigned to input nodes.
- `hasinits::Bool` : whether initial values have been assigned to stochastic nodes.

Constructor

`Model`(*; iter::Integer=0*, *burnin::Integer=0*, *samplers::Vector{Sampler}=Sampler[]*, *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.
- `samplers` : block-specific sampling functions.
- `nodes...` : 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.

MCMC Engine

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(*mc::ModelChains*, *iters::Integer; verbose::Bool=true*)
Simulate MCMC draws for a specified model.

Arguments

- `m` : specified model.
- `mc` : chains from a previous call to `mcmc` for which to simulate additional draws.
- `inputs` : values for input model nodes. Dictionary keys and values should be given for each input node.
- `inits` : 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 `ModelChains` type object of simulated draws.

Example

See the *MCMC Simulation* section of the tutorial.

Indexing

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

Arguments

- `m` : model containing the node to get.
- `nodekey` : node to get.

Value

The specified node.

keys(*m::Model*)

keys (*m::Model*, *ntype::Symbol*, *at...*)
Extract the symbols (keys) for all existing nodes or for nodes of a specified type.

Arguments

- `m` : model containing the nodes of interest.
- **`ntype`** [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 updated by the sampling block(s) `at::Integer=0` (default: all blocks).
  - `:dependent` : logical and stochastic (dependent) nodes in topologically sorted order.
  - `: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.
  - `:source` : nodes upon which the node `at::Symbol` or vector of nodes `at::Vector{Symbol}` depends.
  - `:stochastic` : stochastic nodes.
  - `:target` : topologically sorted nodes that depend on the sampling block(s) `at::Integer=0` (default: all blocks), node `at::Symbol`, or vector of nodes `at::Vector{Symbol}`.
- `at...` : additional positional arguments to be passed to the `ntype` options, as described above.

Value

A vector of node symbols.

Display

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

Arguments

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

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

Arguments

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

Value

Returns a `ModelGraph` type object with field `graph` containing a `DiGraph` representation of indices, as defined in the LightGraphs package, to a vector of node symbols in field `keys`.

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

Arguments

- `m` : 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.

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.

Initialization

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

Arguments

- `m` : model with nodes to be initialized.
- `inits` : 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` : model with input nodes to be assigned.
- `inputs` : 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{T<:Sampler}*)
Set the block-samplers for stochastic model nodes.

Arguments

- `m` : 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.

Parameter Block Operations

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

Arguments

- `m` : model with block-samplers.
- `block` : block for which to get the tuning parameters (default: all blocks).

Value

A `Vector{Any}` of all block-specific tuning parameters if `block=0`, and turning parameters for the specified block otherwise.

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

Arguments

- `m` : model containing the stochastic nodes for which to compute the gradient.
- `block` : sampling block of stochastic nodes for which to compute the gradient (default: all stochastic nodes).
- `x` : value (possibly different than the current one) at which to compute the gradient.
- `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 *[97]*.

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

Arguments

- `m` : model containing the stochastic nodes for which to evaluate log-densities.
- `block` : sampling block of stochastic nodes over which to sum densities (default: all stochastic nodes).
- `nodekeys` : nodes over which to sum densities.
- `x` : value (possibly different than the current one) at which to evaluate densities.
- `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`.

sample!(*m::Model*, *block::Integer=0*)
Generate one MCMC sample of values for a specified model.

Argument:

- `m` : model specification.
- `block` : block for which to sample values (default: all blocks).

Value

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

Example

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

unlist(*m::Model*, *block::Integer=0*, *transform::Bool=false*)
unlist(*m::Model*, *nodekeys::Vector{Symbol}*, *transform::Bool=false*)
relist(*m::Model*, *x::AbstractArray{T<:Real}*, *block::Integer=0*, *transform::Bool=false*)
relist(*m::Model*, *x::AbstractArray{T<:Real}*, *nodekeys::Vector{Symbol}*, *transform::Bool=false*)
relist!(*m::Model*, *x::AbstractArray{T<:Real}*, *block::Integer=0*, *transform::Bool=false*)
relist!(*m::Model*, *x::AbstractArray{T<:Real}*, *nodekey::Symbol*, *transform::Bool=false*)
Convert (unlist) sets of logical and/or stochastic node values to vectors, or reverse (relist) the process.

Arguments

- `m` : model containing nodes to be unlisted or relisted.
- `block` : sampling block of nodes to be listed (default: all blocks).
- `nodekey/nodekeys` : node(s) to be listed.
- `x` : values to re-list.
- `transform` : whether to apply a link transformation in the conversion.

Value

The `unlist` methods return vectors of concatenated node values, `relist` return dictionaries of symbol keys and values for the specified nodes, and `relist!` return their model argument with values copied to the nodes.

update! (*m::Model*, *block::Integer=0*)
update! (*m::Model*, *nodekeys::Vector{Symbol}*)

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

Arguments

- `m` : mode with nodes to be updated.
- `block` : sampling block of nodes to be updated (default: all blocks).
- `nodekeys` : nodes to be updated in the given order.

Value

Returns the model with updated nodes.

2.3.9 Chains

`AbstractChains` subtypes store output from one or more runs (chains) of an MCMC sampler. They serve as containers for output generated by the `mcmc()` function, and supply methods for convergence diagnostics and posterior inference. Moreover, they can be used as stand-alone containers for any user-generated MCMC output, and are thus a **julia** analogue to the *boa [86][87]* and *coda [72][73]* **R** packages.

Declarations

```
abstract type AbstractChains
immutable Chains <: AbstractChains
immutable ModelChains <: AbstractChains
```

Fields

- `value::Array{Float64, 3}` : 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{AbstractString}` : names assigned to the parameter elements.
- `chains::Vector{Int}` : indices to the MCMC runs.
- `model::Model` : model from which the sampled values were generated (`ModelChains` only).

Constructors

Chains (*iters::Integer*, *params::Integer;* *start::Integer=1*, *thin::Integer=1*, *chains::Integer=1*, *names::Vector{T<:AbstractString}=AbstractString[]*)
Chains (*value::Array{T<:Real, 3}; start::Integer=1*, *thin::Integer=1*, *names::Vector{U<:AbstractString}=AbstractString[]*, *chains::Vector{V<:Integer}=Int[]*)
Chains (*value::Matrix{T<:Real}; start::Integer=1*, *thin::Integer=1*, *names::Vector{U<:AbstractString}=AbstractString[]*, *chains::Integer=1*)

Chains(*value::Vector{T<:Real}; start::Integer=1, thin::Integer=1, names::AbstractString="Param1", chains::Integer=1*)
ModelChains(*c::Chains*, *m::Model*)

Construct a `Chains` or `ModelChains` 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` : 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"`, ...).
- `m` : model for which simulated values were generated.

Value

Returns an object of type `Chains` or `ModelChains` according to the name of the constructor called.

Example

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

Indexing and Concatenation

cat(*dim::Integer, chains::AbstractChains...*)
vcat(*chains::AbstractChains...*)
hcat(*chains::AbstractChains...*)

Concatenate input MCMC chains along a specified dimension. For dimensions other than the specified one, all input chains must have the same sizes, which will also be the sizes of the output chain. The size of the output chain along the specified dimension will be the sum of the sizes of the input chains in that dimension. `vcat` concatenates vertically along dimension 1, and has the alternative syntax `[chain1; chain2; ...]`. `hcat` concatenates horizontally along dimension 2, and has the alternative syntax `[chain1 chain2 ...]`.

Arguments

- `dim` : dimension (1, 2, or 3) along which to concatenate the input chains.
- `chains` : chains to concatenate.

Value

A `Chains` object containing the concatenated input.

Example

See the `readcoda()` example.

first(*c::AbstractChains*)
step(*c::AbstractChains*)

last (*c::AbstractChains*)
Get the first iteration, step-size (thinning), or last iteration of MCMC sampler output.

Arguments

- `c` : sampler output for which to return results.

Value

Integer value of the requested iteration type.

getindex (*c::Chains*, *window*, *names*, *chains*)
getindex (*mc::ModelChains*, *window*, *names*, *chains*)
Subset MCMC sampler output. The syntax `c[i, j, k]` is converted to `getindex(c, i, j, k)`.

Arguments

- `c` : sampler output to subset.
- `window` : 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.
- `names` : indices for subsetting of parameters that can be specified as strings, integers, or booleans identifying parameters to be kept. `ModelChains` may additionally be indexed by model node symbols.
- `chains` : indices for chains can be integers or booleans.

A value of `:` can be specified for any of the dimensions to indicate no subsetting.

Value

Subsetted sampler output stored in the same type of object as that supplied in the call.

Example

See the *Output Subsetting* section of the tutorial.

setindex! (*c::AbstractChains*, *value*, *iters*, *names*, *chains*)
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.
- `iters` : 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.
- `names` : indices for subsetting of parameters can be specified as strings, integers, or booleans.
- `chains` : indices for chains can be integers or booleans.

A value of `:` can be specified for any of the dimensions to index all corresponding elements.

Value

An object of the same type as `c` with the sampler output stored in the specified indices.

Example

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

File I/O

read(*name::AbstractString*, *::Type{T<:AbstractChains}*)
write(*name::AbstractString*, *c::AbstractChains*)

Read a chain from or write one to an external file.

Arguments

- `name` : file to read or write. Recommended convention is for the file name to be specified with a `.jls` extension.
- `T` : chain type to read.
- `c` : chain to write.

Value

An `AbstractChains` subtype read from an external file, or a written external file containing a subtype.

Example

See the *File I/O* section of the tutorial.

readcoda(*output::AbstractString*, *index::AbstractString*)

Read MCMC sampler output generated in the CODA format by OpenBUGS *[90]*. The function only retains those sampler iterations at which all model parameters were monitored.

Arguments

- `output` : text file containing the iteration numbers and sampled values for the model parameters.
- `index` : text file containing the names of the parameters, followed by the first and last rows in which their output can be found in the `output` file.

Value

A `Chains` object containing the read sampler output.

Example

The following example reads sampler output contained in the CODA files `line1.out`, `line1.ind`, `line2.out`, and `line2.ind`.

```
using Mamba

## Get the directory in which the example CODA files are saved
dir = dirname(@__FILE__)

## Read the MCMC sampler output from the files
c1 = readcoda(joinpath(dir, "line1.out"), joinpath(dir, "line1.ind"))
c2 = readcoda(joinpath(dir, "line2.out"), joinpath(dir, "line2.ind"))

## Concatenate the resulting chains
c = cat(3, c1, c2)

## Compute summary statistics
describe(c)
```

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 *[18]*. 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
Discrete	Mean	Univariate	2+	Yes	No
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

Discrete Diagnostic

discretediag(*c::AbstractChains; frac::Real=0.3*, *method::Symbol=:weiss*, *nsim::Int=1000*)

Compute the convergence diagnostic for a discrete variable. Several options are available by choosing *method* to be one of *:hangartner, :weiss, :DARBOOT, MCBOOT, :billinsgley, :billingsleyBOOT*. The first four are based off of Pearson's chi-square test of homogeneity. The diagnostic tests whether the proportion of the categories of the discrete variable are similar in each chain. The last two methods test whether the transition probabilities between each category are similar between each chain. Along with a between chain assessment of convergence, a within-chain assessment is carried out by comparing a specified fraction (*frac*), or window, of the beginning of a chain to the specified fraction of the end of the chain. For within-chain assessment, users should ensure that there is sufficient separation between the 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.
- `frac` : proportion of iterations to include in the first window.
- `method` : Specify which method to use. One of :hangartner, :weiss, :DARBOOT, MCBOOT, :billinsgley, :billingsleyBOOT‘.
- `nsim` : For the bootstrap methods (*:DARBOOT*, *:MCBOOT*, and *:billingsleyBOOT*) the number of bootstrap simulations.

Value

A `ChainSummary` type object with parameters contained in the rows of the `value` field. The first three columns correspond to the test statistic, degrees of freedom, and p-value of the between-chain assessment. The next columns are the test statistic, degrees of freedom, and p-value for each chain of the within-chain assessment.

Example

See the *Pollution* example.

Gelman, Rubin, and Brooks Diagnostics

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

Compute the convergence diagnostics of Gelman, Rubin, and Brooks *[33][12]* 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{AbstractString}
  colnames::Vector{AbstractString}
  header::AbstractString
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::AbstractChains; first::Real=0.1*, *last::Real=0.5*, *etype=:imse*, *args...*)

Compute the convergence diagnostic of Geweke *[37]* 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::AbstractChains; alpha::Real=0.05*, *eps::Real=0.1*, *etype=:imse*, *args...*)

Compute the convergence diagnostic of Heidelberger and Welch *[47]* 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::AbstractChains; q::Real=0.025, r::Real=0.005, s::Real=0.95, eps::Real=0.001*)

Compute the convergence diagnostic of Raftery and Lewis *[74][75]* 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::AbstractChains; 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::AbstractChains*)

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::AbstractChains*)

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::AbstractChains; 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::AbstractChains; alpha::Real=0.05*)

Compute highest posterior density (HPD) intervals of Chen and Shao *[16]* 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` : time series of values on which to perform calculations.
- **`method`** [method used for the calculations. Options are]
  - `:bm` : batch means *[41]*, 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 *[38]*.
  - `:ipse` : initial positive sequence estimator *[38]*.
- `args...` : additional arguments for the calculation method.

Value

The numeric standard error value.

`quantile` (*c::AbstractChains; 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::AbstractChains; 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 (*mc::ModelChains*)

Compute the Deviance Information Criterion (DIC) of Spiegelhalter et al. *[91]* and Gelman et al. *[31]* from MCMC sampler output.

Arguments

- `mc` : sampler output from a model fit with the `mcmc()` function.

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.

logpdf (*mc::ModelChains*, *nodekey::Symbol*)
logpdf (*mc::ModelChains*, *nodekeys::Vector{Symbol}=keys(mc.model, :stochastic)*)

Compute the sum of log-densities at each iteration of MCMC output for stochastic nodes.

Arguments

- `mc`: sampler output from a model fit with the `mcmc()` function.
- `nodekey/nodekeys`: stochastic model node(s) over which to sum densities (default: all).

Value

A `ModelChains` object of resulting summed log-densities at each MCMC iteration of the supplied chain.

predict (*mc::ModelChains*, *nodekey::Symbol*)
predict (*mc::ModelChains*, *nodekeys::Vector{Symbol}=keys(mc.model, :output)*)

Generate MCMC draws from a posterior predictive distribution.

Arguments

- `mc`: sampler output from a model fit with the `mcmc()` function.
- `nodekey/nodekeys`: observed Stochastic model node(s) for which to generate draws from the predictive distribution (default: all observed data nodes).

Value

A `ModelChains` object of draws simulated at each MCMC iteration of the supplied chain. 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::AbstractChains*, *ptype::Vector{Symbol}=[:trace, :density]; legend::Bool=false*, *args...*)
plot(*c::AbstractChains*, *ptype::Symbol; legend::Bool=false*, *args...*)
Various plots to summarize sampler output stored in `AbstractChains` subtypes. Separate plots are produced for each sampled parameter.

Arguments

- `c` : sampler output to plot.
- **`ptype`** [plot type(s). Options are]
  - `:autocor` : autocorrelation plots, with optional argument `maxlag::Integer=round(Int, 10 * log10(length(c.range)))` determining the maximum autocorrelation lag to plot. Lags are plotted relative to the thinning interval of the output.
  - `:bar` : bar plots. Optional argument `position::Symbol=:stack` controls whether bars should be stacked on top of each other (default) or placed side by side (`:dodge`).
  - `:contour` : pairwise posterior density contour plots. Optional argument `bins::Integer=100` controls the plot resolution.
  - `:density` : density plots. Optional argument `trim::Tuple{Real, Real}=(0.025, 0.975)` trims off lower and upper quantiles of density.
  - `:mean` : running mean plots.
  - `:mixeddensity` : bar plots (`:bar`) for parameters with integer values within bounds defined by optional argument `barbounds::Tuple{Real, Real}=(0, Inf)`, and density plots (`:density`) otherwise.
  - `:trace` : trace plots.
- `legend` : whether to include legends in the plots to identify chain-specific results.
- `args...` : additional keyword arguments to be passed to the `ptype` options, 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 *[51]*.

Example

See the *Plotting* section of the tutorial.

draw(*p::Array{Plot}; fmt::Symbol=:svg*, *filename::AbstractString=""*, *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` : 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).
  - `:pgf` : Portable Graphics Format (.pgf).
  - `:png` : Portable Network Graphics (.png).
  - `:ps` : Postscript (.ps).
  - `:svg` : Scalable Vector Graphics (.svg).
- `filename` : 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.

2.4 Sampling Functions

Listed below are the sampling methods for which functions are provided to simulating draws from distributions that can be specified up to constants of proportionalities. Model-based *Sampler* constructors are available for use with the `mcmc()` engine as well as stand-alone functions that can be used independently.

2.4.1 Approximate Bayesian Computation (ABC)

Approximate Bayesian Computation in the framework of MCMC (also known as Likelihood-Free MCMC) as proposed by *[59]* for simulating autocorrelated draws from a posterior distribution without evaluating its likelihood. Also see *[85]* for a thorough review of Likelihood-Free MCMC.

Model-Based Constructor

ABC (*params::ElementOrVector{Symbol}*, *scale::ElementOrVector{T<:Real}*, *summary::Function*, *epsilon::Real; kernel::KernelDensityType=SymUniform*, *dist::Function=(Tsim*, *Tobs*) → sqrt(sum(abs2, Tsim - Tobs)), proposal::SymDistributionType=Normal, maxdraw::Integer=1, nsim::Integer=1, decay::Real=1.0, randeps::Bool=false, args...)

Construct a `Sampler` object for ABC sampling. Parameters are assumed to be continuous, but may be con-

strained or unconstrained.

Arguments

- `params` : stochastic node(s) to be updated with the sampler. Constrained parameters are mapped to unconstrained space according to transformations defined by the *Stochastic* `unlist()` function.
- `scale` : scaling value or vector of the same length as the combined elements of nodes `params` for the `proposal` distribution. Values are relative to the unconstrained parameter space, where candidate draws are generated.
- `summary` : function that takes a vector of observed or simulated data and returns a summary statistic or vector of statistics.
- `epsilon` : target tolerance for determining how similar observed and simulated data summary statistics need to be in order to accept a candidate draw.
- `kernel` : weighting kernel density of type `Biweight`, `Cosine`, `Epanechnikov`, `Normal`, `SymTriangularDist`, `SymUniform`, or `Triweight` to use in measuring similarity between observed and simulated data summary statistics. Specified `epsilon` determines the standard deviation of Normal kernels and widths of the others.
- `dist` : positive function for the kernel density to compute distance between vectors of observed (`Tobs`) and simulated (`Tsim`) data summary statistics (default: Euclidean distance).
- `proposal` : symmetric distribution of type `Biweight`, `Cosine`, `Epanechnikov`, `Normal`, `SymTriangularDist`, `SymUniform`, or `Triweight` to be centered around current parameter values and used to generate proposal draws. Specified `scale` determines the standard deviations of Normal proposals and widths of the others.
- `maxdraw` : maximum number of unaccepted candidates to draw in each call of the sampler. Draws are generated until one is accepted or the maximum is reached. Larger values increase acceptance rates at the expense of longer runtimes.
- `nsim` : number of data sets to simulate in deciding whether to accept a candidate draw. Larger values lead to closer approximations of the target distribution at the expense of longer runtimes.
- `decay` : if `0 < decay <= 1`, the rate at which internal tolerances are monotonically decreased from the initial distance between observed and simulated summary statistics toward the maximum of each subsequent distance and `epsilon`; if `decay = 0`, internal tolerances are fixed at `epsilon`.
- `randeps` : whether to perturb internal tolerances by random exponential variates.
- `args...` : additional keyword arguments to be passed to the `dist` function.

Value

Returns a `Sampler{ABCTune}` type object.

Example

See the *ABC Line*, *GK*, and other *Examples*.

ABCTune Type

Declaration

```
type ABCTune
```

Fields

- `datakeys::Vector{Symbol}` : stochastic "data" nodes in the full conditional distribution for parameters to be updated and nodes at which summary statistics are computed separately in the sampling algorithm.
- `Tsim::Vector{Vector{Float64}}` : simulated data summary statistics for the `nsim` data sets.
- `epsilon::Vector{Float64}` : internal tolerances for the data sets.
- `epsilonprime::Vector{Float64}` : perturbed tolerances if `randeps=true` or `epsilon` otherwise.

2.4.2 Adaptive Mixture Metropolis (AMM)

Implementation of the Roberts and Rosenthal *[79]* adaptive (multivariate) mixture Metropolis *[45][46][63]* sampler for simulating autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Model-Based Constructor

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

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

Arguments

- `params` : stochastic node(s) to be updated with the sampler. Constrained parameters are mapped to unconstrained space according to transformations defined by the *Stochastic* `unlist()` 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).
- `args...` : additional keyword arguments to be passed to the `AMMVariate` constructor.

Value

Returns a `Sampler{AMMTune}` type object.

Example

See the *Seeds* and other *Examples*.

Stand-Alone Function

sample!(*v::AMMVariate; adapt::Bool=true*)

Draw one sample from a target distribution using the AMM 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 will contain the `tune` field returned by the previous call.
- `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 = Dict(
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
)

## Log-transformed Posterior(b0, b1, log(s2)) + Constant
logf = function(x::DenseVector)
   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], eye(3), logf)
for i in 1:n
  sample!(theta, adapt = (i <= burnin))
  sim[i, :, 1] = [theta[1:2]; exp(theta[3])]
end
describe(sim)
```

AMMVariate Type

Declaration

```
const AMMVariate = SamplerVariate{AMMTune}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::AMMTune` : tuning parameters for the sampling algorithm.

Constructor

`AMMVariate`(*x::AbstractVector{T<:Real}*, *Sigma::Matrix{U<:Real}*, *logf::Function;* *beta::Real=0.05*, *scale::Real=2.38*)

Construct an `AMMVariate` object that stores simulated values and tuning parameters for AMM sampling.

Arguments

- `x` : initial values.
- `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.
- `logf` : function that takes a single `DenseVector` argument of parameter values at which to compute the log-transformed density (up to a normalizing constant).
- `beta` : proportion of weight given to draws from the non-adaptive proposal with covariance factorization `SigmaL`, relative to draws from the adaptively tuned proposal with covariance factorization `SigmaLm`, during adaptive updating.
- `scale` : factor (`scale^2 / length(x)`) by which the adaptively updated covariance matrix is scaled—default value adopted from Gelman, Roberts, and Gilks *[32]*.

Value

Returns an `AMMVariate` type object with fields set to the supplied `x` and tuning parameter values.

AMMTune Type

Declaration

```
type AMMTune <:  SamplerTune
```

Fields

- `logf::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density, or null if not supplied.
- `adapt::Bool` : whether the proposal distribution is being adaptively tuned.
- `beta::Float64` : proportion of weight given to draws from the non-adaptive proposal with covariance factorization `SigmaL`, relative to draws from the adaptively tuned proposal with covariance factorization `SigmaLm`, during adaptive updating.
- `m::Int` : 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::Matrix{Float64}` : running mean of `v * v'` during adaptive updating. Used in the calculation of `SigmaLm`.

- `scale::Float64` : factor (`scale^2 / length(v)`) by which the adaptively updated covariance matrix is scaled.
- `SigmaL::LowerTriangular{Float64}` : Cholesky factorization of the non-adaptive covariance matrix.
- `SigmaLm::Matrix{Float64}` : pivoted factorization of the adaptively tuned covariance matrix.

2.4.3 Adaptive Metropolis within Gibbs (AMWG)

Implementation of a Metropolis-within-Gibbs sampler *[63][79][95]* for iteratively simulating autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Model-Based Constructor

AMWG(*params::ElementOrVector{Symbol}*, *sigma::ElementOrVector{T<:Real}; adapt::Symbol=:all*, *args...*)

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

Arguments

- `params` : stochastic node(s) to be updated with the sampler. Constrained parameters are mapped to unconstrained space according to transformations defined by the *Stochastic* `unlist()` function.
- `sigma` : scaling value or vector of the same length as the combined elements of nodes `params`, defining initial standard deviations for univariate normal proposal distributions. 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).
- `args...` : additional keyword arguments to be passed to the `AMWGVariate` constructor.

Value

Returns a `Sampler{AMWGTune}` type object.

Example

See the *Birats*, *Blocker*, and other *Examples*.

Stand-Alone Function

sample!(*v::AMWGVariate; adapt::Bool=true*)

Draw one sample from a target distribution using the AMWG 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 will contain the `tune` field returned by the previous call.
- `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*. Also, see the *Line: Block-Specific Sampling with AMWG and Slice* example.

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

using Mamba

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

## Log-transformed Posterior(b0, b1, log(s2)) + Constant
logf = function(x::DenseVector)
   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], 1.0, logf)
for i in 1:n
  sample!(theta, adapt = (i <= burnin))
  sim[i, :, 1] = [theta[1:2]; exp(theta[3])]
end
describe(sim)
```

AMWGVariate Type

Declaration

```
const AMWGVariate = SamplerVariate{AMWGTune}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::AMWGTune` : tuning parameters for the sampling algorithm.

Constructor

AMWGVariate(*x::AbstractVector{T<:Real}*, *sigma::ElementOrVector{U<:Real}*, *logf::Function;* *batchsize::Integer=50*, *target::Real=0.44*)

Construct an `AMWGVariate` object that stores simulated values and tuning parameters for AMWG sampling.

Arguments

- `x` : initial values.
- `sigma` : scaling value or vector of the same length as the combined elements of nodes `params`, defining initial standard deviations for univariate normal proposal distributions. Standard deviations are relative to the unconstrained parameter space, where candidate draws are generated.
- `logf` : function that takes a single `DenseVector` argument of parameter values at which to compute the log-transformed density (up to a normalizing constant).
- `batchsize` : number of samples that must be accumulated before applying an adaptive update to the proposal distributions.
- `target` : target acceptance rate for the algorithm.

Value

Returns an `AMWGVariate` type object with fields set to the supplied `x` and tuning parameter values.

AMWGTune Type

Declaration

```
type AMWGTune <:  SamplerTune
```

Fields

- `logf::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density, or null if not supplied.
- `adapt::Bool` : whether the proposal distribution is being adaptively tuned.
- `accept::Vector{Int}` : number of accepted candidate draws generated for each element of the parameter vector during adaptive updating.
- `batchsize::Int` : number of samples that must be accumulated before applying an adaptive update to the proposal distributions.
- `m::Int` : number of adaptive update iterations that have been performed.
- `sigma::Vector{Float64}` : updated values of the proposal standard deviations if `m > 0`, and user-supplied values otherwise.
- `target::Float64` : target acceptance rate for the adaptive algorithm.

2.4.4 Binary Hamiltonian Monte Carlo (BHMC)

Implementation of the binary-state Hamiltonian Monte Carlo sampler of Pakman *[68]*. The sampler simulates autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Model-Based Constructor

BHMC (*params::ElementOrVector{Symbol}*, *traveltime::Real*)

Construct a `Sampler` object for BHMC sampling. Parameters are assumed to have binary numerical values (0 or 1).

Arguments

- `params` : stochastic node(s) to be updated with the sampler.
- `traveltime` : length of time over which particle paths are simulated. It is recommended that supplied values be of the form $(n+\frac{1}{2})\pi$, where optimal choices of $n \in \mathbb{Z}^+$ are expected to grow with the parameter space dimensionality.

Value

Returns a `Sampler{BHMCTune}` type object.

Example

See the *Pollution* and other *Examples*.

Stand-Alone Function

sample! (*v::BHMCVariate*)

Draw one sample from a target distribution using the BHMC sampler. Parameters are assumed to have binary numerical values (0 or 1).

Arguments

- `v` : current state of parameters to be simulated.

Value

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

Example

```
#########################################################################
↪######
## Linear Regression
##   y ~ MvNormal(X * (beta0 .* gamma), 1)
##   gamma ~ DiscreteUniform(0, 1)
#########################################################################
↪######

using Mamba

## Data
n, p = 25, 10
X = randn(n, p)
beta0 = randn(p)
gamma0 = rand(0:1, p)
y = X * (beta0 .* gamma0) + randn(n)

## Log-transformed Posterior(gamma) + Constant
logf = function(gamma::DenseVector)
  logpdf(MvNormal(X * (beta0 .* gamma), 1.0), y)
end

## MCMC Simulation with Binary Hamiltonian Monte Carlo Sampling
```

```
t = 10000
sim = Chains(t, p, names = map(i -> "gamma[$i]", 1:p))
gamma = BHMCVariate(zeros(p), (2 * p + 0.5) * pi, logf)
for i in 1:t
  sample!(gamma)
  sim[i, :, 1] = gamma
end
describe(sim)
```

BHMCVariate Type

Declaration

```
const BHMCVariate = SamplerVariate{BHMCTune}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::BHMCTune` : tuning parameters for the sampling algorithm.

Constructor

`BHMCVariate`(*x::AbstractVector{T<:Real}*, *traveltime::Real*, *logf::Function*)

Construct a `BHMCVariate` object that stores simulated values and tuning parameters for BHMC sampling.

Arguments

- `x` : initial values.
- `traveltime` : length of time over which particle paths are simulated. It is recommended that supplied values be of the form $(n+\frac{1}{2})\pi$, where optimal choices of $n \in \mathbb{Z}^+$ are expected to grow with the parameter space dimensionality.
- `logf` : function that takes a single `DenseVector` argument of parameter values at which to compute the log-transformed density (up to a normalizing constant).

Value

Returns a `BHMCVariate` type object with fields set to the supplied `x` and tuning parameter values.

BHMCTune Type

Declaration

```
type BHMCTune <:  SamplerTune
```

Fields

- `logf::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density, or null if not supplied.
- `traveltime::Float64` : length of time over which particle paths are simulated.

- `position::Vector{Float64}` : initial particle positions.
- `velocity::Vector{Float64}` : initial particle velocites.
- `wallhits::Int` : number of times particles are reflected off the 0 threshold.
- `wallcrosses::Int` : number of times particles travel through the threshold.

2.4.5 Binary Individual Adaptation (BIA)

Implementation of the binary-state Individual Adaptation sampler of Griffin, et al. *[43]* which adjusts a general proposal to the data. The sampler simulates autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Model-Based Constructor

BIA(*params::ElementOrVector{Symbol}; args...*)

Construct a `Sampler` object for BIA sampling. Parameters are assumed to have binary numerical values (0 or 1).

Arguments

- `params` : stochastic node(s) to be updated with the sampler.
- `args...` : additional keyword arguments to be passed to the `BIAVariate` constructor.

Value

Returns a `Sampler{BIATune}` type object.

Example

See the *Pollution* and other *Examples*.

Stand-Alone Function

sample!(*v::BIAVariate*)

Draw one sample from a target distribution using the BIA sampler. Parameters are assumed to have binary numerical values (0 or 1).

Arguments

- `v` : current state of parameters to be simulated.

Value

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

Example

```
#########################################################################
↪######
## Linear Regression
##   y ~ MvNormal(X * (beta0 .* gamma), 1)
##   gamma ~ DiscreteUniform(0, 1)
#########################################################################
↪######

using Mamba
```

```
## Data
n, p = 25, 10
X = randn(n, p)
beta0 = randn(p)
gamma0 = rand(0:1, p)
y = X * (beta0 .* gamma0) + randn(n)

## Log-transformed Posterior(gamma) + Constant
logf = function(gamma::DenseVector)
  logpdf(MvNormal(X * (beta0 .* gamma), 1.0), y)
end

## MCMC Simulation with Binary Individual Adaptation Sampler
t = 10000
sim = Chains(t, p, names = map(i -> "gamma[$i]", 1:p))
gamma = BIAVariate(zeros(p), logf)
for i in 1:t
  sample!(gamma)
  sim[i, :, 1] = gamma
end
describe(sim)
```

BIAVariate Type

Declaration

```
const BIAVariate = SamplerVariate{BIATune}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::BIATune` : tuning parameters for the sampling algorithm.

Constructor

`BIAVariate`(*x::AbstractVector{T<:Real}, logf::Function; A::Vector{Float64} = ones(x) / length(x), D::Vector{Float64} = ones(x) / length(x), epsilon::Real = 0.01 / length(x), decay::Real = 0.55, target::Real = 0.45*)

Construct a `BIAVariate` object that stores simulated values and tuning parameters for BIA sampling.

Arguments

- `x` : initial values.
- `logf` : function that takes a single `DenseVector` argument of parameter values at which to compute the log-transformed density (up to a normalizing constant).
- `A` : vector of probabilities to switch the elements of `x` from 0 to 1 (i.e. added).
- `D` : vector of probabilities to switch elements from 1 to 0 (i.e. deleted).
- `epsilon` : range `(epsilon, 1 - epsilon)` for the elements of `A` and `D`, where `0 < epsilon < 0.5`.
- `decay` : rate of decay of the adaptation, where `0.5 < decay <= 1.0`.

- `target` : target mutation rate, where `0.0 < target < 1.0`.

Value

Returns a `BIAVariate` type object with fields set to the supplied `x` and tuning parameter values.

BIATune Type

Declaration

```
type BIATune <:  SamplerTune
```

Fields

- `logf::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density, or null if not supplied.
- `A::Vector{Float64}` : vector of probabilities to switch from 0 to 1.
- `D::Vector{Float64}` : vector of probabilities to switch from 1 to 0.
- `epsilon::Float64` : range `(epsilon, 1 - epsilon)` for the elements of `A` and `D`.
- `decay::Float64` : rate of decay of the adaptation.
- `target::Float64` : target mutation rate.
- `iter::Int` : iteration number for adaptive updating.

2.4.6 Binary MCMC Model Composition (BMC3)

Implementation of the binary-state MCMC Model Composition of Madigan and York *[58]* in which proposed updates are always state changes. Liu *[55]* shows this sampler is more efficient than Gibbs sampling for a binary vector. Schafer *[83][84]* proposes a method for block updates of binary vectors using this sampler. The sampler simulates autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Model-Based Constructor

BMC3 (*params::ElementOrVector{Symbol}; k::BMC3Form=1*)

Construct a `Sampler` object for BMC3 sampling. Parameters are assumed to have binary numerical values (0 or 1).

Arguments

- `params` : stochastic node(s) to be updated with the sampler.
- `k` : number of parameters or vector of parameter indices to select at random for simultaneous updating in each call of the sampler.

Value

Returns a `Sampler{BMC3Tune{typeof(k)}}` type object.

Example

See the *Pollution* and other *Examples*.

Stand-Alone Function

sample! (*v::SamplerVariate{BMC3Tune{F<:BMC3Form}}*)

Draw one sample from a target distribution using the BMC3 sampler. Parameters are assumed to have binary numerical values (0 or 1).

Arguments

- v : current state of parameters to be simulated.

Value

Returns v updated with simulated values and associated tuning parameters.

Example

```
#########################################################################
↪######
## Linear Regression
##   y ~ MvNormal(X * (beta0 .* gamma), 1)
##   gamma ~ DiscreteUniform(0, 1)
#########################################################################
↪######

using Mamba

## Data
n, p = 25, 10
X = randn(n, p)
beta0 = randn(p)
gamma0 = rand(0:1, p)
y = X * (beta0 .* gamma0) + randn(n)

## Log-transformed Posterior(gamma) + Constant
logf = function(gamma::DenseVector)
  logpdf(MvNormal(X * (beta0 .* gamma), 1.0), y)
end

## MCMC Simulation with Binary MCMC Model Composition
t = 10000
sim1 = Chains(t, p, names = map(i -> "gamma[$i]", 1:p))
sim2 = Chains(t, p, names = map(i -> "gamma[$i]", 1:p))
gamma1 = BMC3Variate(zeros(p), logf)
gamma2 = BMC3Variate(zeros(p), logf, k=Vector{Int}[[i] for i in 1:p])
for i in 1:t
  sample!(gamma1)
  sample!(gamma2)
  sim1[i, :, 1] = gamma1
  sim2[i, :, 1] = gamma2
end
describe(sim1)
describe(sim2)

p = plot(sim1, [:trace, :mixeddensity])
draw(p, filename = "bmc3plot")
```

BMC3 Variate Type

Declaration

```
SamplerVariate{BMC3Tune{F<:BMC3Form}}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::BMC3Tune{F}` : tuning parameters for the sampling algorithm.

Constructor

`BMC3Variate`(*x::AbstractVector{T<:Real}*, *logf::Function; k::BMC3Form=1*)

Construct a `SamplerVariate` object that stores simulated values and tuning parameters for BMC3 sampling.

Arguments

- `x` : initial values.
- `logf` : function that takes a single `DenseVector` argument of parameter values at which to compute the log-transformed density (up to a normalizing constant).
- `k` : number of parameters or vector of parameter indices to select at random for simultaneous updating in each call of the sampler.

Value

Returns a `SamplerVariate{BMC3Tune{typeof(k)}}` type object with fields set to the supplied `x` and tuning parameter values.

BMC3Tune Type

Declaration

```
const BMC3Form = Union{Int, Vector{Vector{Int}}}
type BMC3Tune{F<:BMC3Form} <: SamplerTune
```

Fields

- `logf::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density, or null if not supplied.
- `k::F` : number of parameters or vector of parameter indices to select at random for simultaneous updating in each call of the sampler.

2.4.7 Binary Metropolised Gibbs (BMG)

Implementation of the binary-state Metropolised Gibbs sampler described by Schafer *[83][84]* in which components are drawn sequentially from full conditional marginal distributions and accepted together in a single Metropolis-Hastings step. The sampler simulates autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Model-Based Constructor

BMG (*params::ElementOrVector{Symbol}; k::BMGForm=1*)

Construct a `Sampler` object for BMG sampling. Parameters are assumed to have binary numerical values (0 or 1).

Arguments

- `params` : stochastic node(s) to be updated with the sampler.
- `k` : number of parameters or vector of parameter indices to select at random for simultaneous updating in each call of the sampler.

Value

Returns a `Sampler{BMGTune{typeof(k)}}` type object.

Example

See the *Pollution* and other *Examples*.

Stand-Alone Function

sample! (*v::SamplerVariate{BMGTune{F<:BMGForm}}*)

Draw one sample from a target distribution using the BMG sampler. Parameters are assumed to have binary numerical values (0 or 1).

Arguments

- `v` : current state of parameters to be simulated.

Value

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

Example

```
#########################################################################
↪######
## Linear Regression
##   y ~ MvNormal(X * (beta0 .* gamma), 1)
##   gamma ~ DiscreteUniform(0, 1)
#########################################################################
↪######

using Mamba

## Data
n, p = 25, 10
X = randn(n, p)
beta0 = randn(p)
gamma0 = rand(0:1, p)
y = X * (beta0 .* gamma0) + randn(n)
```

```
## Log-transformed Posterior(gamma) + Constant
logf = function(gamma::DenseVector)
  logpdf(MvNormal(X * (beta0 .* gamma), 1.0), y)
end

## MCMC Simulation with Binary Metropolised Gibbs
t = 10000
sim1 = Chains(t, p, names = map(i -> "gamma[$i]", 1:p))
sim2 = Chains(t, p, names = map(i -> "gamma[$i]", 1:p))
gamma1 = BMGVariate(zeros(p), logf)
gamma2 = BMGVariate(zeros(p), logf, k=Vector{Int}[[i] for i in 1:p])
for i in 1:t
  sample!(gamma1)
  sample!(gamma2)
  sim1[i, :, 1] = gamma1
  sim2[i, :, 1] = gamma2
end
describe(sim1)
describe(sim2)
```

BMG Variate Type

Declaration

`SamplerVariate{BMGTune{F<:BMGForm}}`

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::BMGTune{F}` : tuning parameters for the sampling algorithm.

Constructor

`BMGVariate`(*x::AbstractVector{T<:Real}*, *logf::Function; k::BMGForm=1*)

Construct a `SamplerVariate` object that stores simulated values and tuning parameters for BMG sampling.

Arguments

- `x` : initial values.
- `logf` : function that takes a single `DenseVector` argument of parameter values at which to compute the log-transformed density (up to a normalizing constant).
- `k` : number of parameters or vector of parameter indices to select at random for simultaneous updating in each call of the sampler.

Value

Returns a `SamplerVariate{BMGTune{typeof(k)}}` type object with fields set to the supplied `x` and tuning parameter values.

BMGTune Type

Declaration

```
const BMGForm = Union{Int, Vector{Vector{Int}}}
type BMGTune{F<:BMGForm} <: SamplerTune
```

Fields

- `logf::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density, or null if not supplied.
- `k::F` : number of parameters or vector of parameter indices to select at random for simultaneous updating in each call of the sampler.

2.4.8 Discrete Gibbs Sampler (DGS)

Implementation of a sampler for the simulation of discrete or discretized model parameters with finite support. Draws are simulated directly from a probability mass function that can be specified up to a constant of proportionality. Note that versions of this sampler evaluate the probability function over all points in the parameter space; and, as a result, may be very computationally intensive for large spaces.

Model-Based Constructor

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

Construct a `Sampler` object for which DGS sampling is to be applied separately to each of the supplied parameters. Parameters are assumed to have discrete univariate distributions with finite supports.

Arguments

- `params` : stochastic node(s) to be updated with the sampler.

Value

Returns a `Sampler{DSTune{Function}}` type object.

Example

See the *Eyes*, *Pollution*, and other *Examples*.

Stand-Alone Functions

sample! (*v::DGSVariate*)
sample! (*v::DiscreteVariate*)

Draw one sample directly from a target probability mass function. Parameters are assumed to have discrete and finite support.

Arguments

- `v` : current state of parameters to be simulated.

Value

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

Discrete Variate Types

Declaration

```
const DGSVariate = SamplerVariate{DSTune{Function}}
const DiscreteVariate = SamplerVariate{DSTune{Vector{Float64}}}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::DSTune{F<:DSForm}` : tuning parameters for the sampling algorithm.

Constructors

`DGSVariate`(*x::AbstractVector{T<:Real}*, *support::Matrix{U<:Real}*, *mass::Function*)
`DiscreteVariate`(*x::AbstractVector{T<:Real}*, *support::Matrix{U<:Real}*, *mass::Vector{Float64}*)

Construct an object that stores simulated values and tuning parameters for discrete sampling.

Arguments

- `x` : initial values.
- `support` : matrix whose columns contain the vector coordinates in the parameter space from which to simulate values.
- `mass` : function that takes a single `DenseVector` argument of parameter values at which to compute the density (up to a normalizing constant), or a vector of sampling probabilities for the parameter space.

Value

Returns a `DGSVariate` or `DiscreteVariate` type object with fields set to the supplied `x` and tuning parameter values.

DSTune Type

Declaration

```
const DSForm = Union{Function, Vector{Float64}}
type DSTune{F<:DSForm} <: SamplerTune
```

Fields

- `mass::Nullable{F}` : density mass function or vector supplied to the constructor, or null if not supplied.
- `support::Matrix{Real}` : matrix whose columns contain the vector coordinates in the parameter space from which to simulate values.

2.4.9 Hamiltonian Monte Carlo (HMC)

Implementation of the Hybrid Monte Carlo (also known as Hamiltonian Monte Carlo) of Duane *[23]*. The sampler simulates autocorrelated draws from a distribution that can be specified up to a constant of proportionality. Code is derived from Neal's implementation *[65]*.

Model-Based Constructors

HMC(*params::ElementOrVector{Symbol}*, *epsilon::Real*, *L::Integer; dtype::Symbol=:forward*)
HMC(*params::ElementOrVector{Symbol}*, *epsilon::Real*, *L::Integer*, *Sigma::Matrix{T<:Real}; dtype::Symbol=:forward*)
Construct a `Sampler` object for HMC sampling. Parameters are assumed to be continuous, but may be constrained or unconstrained.

Arguments

- `params` : stochastic node(s) to be updated with the sampler. Constrained parameters are mapped to unconstrained space according to transformations defined by the *Stochastic* `unlist()` function.
- `epsilon` : step size.
- `L` : number of steps to take in the Leapfrog algorithm.
- `Sigma` : covariance matrix for the multivariate normal proposal distribution. The covariance matrix is relative to the unconstrained parameter space, where candidate draws are generated. If omitted, the identity matrix is assumed.
- **`dtype`** [type of differentiation for gradient calculations. Options are]
  - `:central` : central differencing.
  - `:forward` : forward differencing.

Value

Returns a `Sampler{HMCTune}` type object.

Example

See the *Dyes* and other *Examples*.

Stand-Alone Function

`sample!`(*v::HMCVariate*)
Draw one sample from a target distribution using the HMC sampler. Parameters are assumed to be continuous and unconstrained.

Arguments

- `v` : current state of parameters to be simulated.

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 = Dict(
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
)

## Log-transformed Posterior(b0, b1, log(s2)) + Constant and Gradient␣
↪Vector
logfgrad = function(x::DenseVector)
  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 Hamiltonian Monte Carlo
## Without (1) and with (2) a user-specified proposal covariance matrix
n = 5000
sim1 = Chains(n, 3, names = ["b0", "b1", "s2"])
sim2 = Chains(n, 3, names = ["b0", "b1", "s2"])
epsilon = 0.1
L = 50
Sigma = eye(3)
theta1 = HMCVariate([0.0, 0.0, 0.0], epsilon, L, logfgrad)
theta2 = HMCVariate([0.0, 0.0, 0.0], epsilon, L, Sigma, logfgrad)
for i in 1:n
  sample!(theta1)
  sample!(theta2)
  sim1[i, :, 1] = [theta1[1:2]; exp(theta1[3])]
  sim2[i, :, 1] = [theta2[1:2]; exp(theta2[3])]
end
describe(sim1)
describe(sim2)
```

HMCVariate Type

Declaration

```
const HMCVariate = SamplerVariate{HMCTune}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::HMCTune` : tuning parameters for the sampling algorithm.

Constructors

`HMCVariate`(*x::AbstractVector{T<:Real}*, *epsilon::Real*, *L::Integer*, *logfgrad::Function*)
`HMCVariate`(*x::AbstractVector{T<:Real}*, *epsilon::Real*, *L::Integer*, *Sigma::Matrix{U<:Real}*, *logfgrad::Function*)

Construct an `HMCVariate` object that stores simulated values and tuning parameters for HMC sampling.

Arguments

- `x` : initial values.
- `epsilon` : step size.
- `L` : number of steps to take in the Leapfrog algorithm.
- `Sigma` : covariance matrix for the multivariate normal proposal distribution. The covariance matrix is relative to the unconstrained parameter space, where candidate draws are generated. If omitted, the identity matrix is assumed.
- `logfgrad` : function that takes a single `DenseVector` argument at which to compute the log-transformed density (up to a normalizing constant) and gradient vector, and returns the respective results as a tuple.

Value

Returns an `HMCVariate` type object with fields set to the supplied `x` and tuning parameter values.

HMCTune Type

Declaration

```
type HMCTune <:  SamplerTune
```

Fields

- `logfgrad::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density and gradient vector, or null if not supplied.
- `epsilon::Float64` : step size.
- `L::Int` : number of steps to take in the Leapfrog algorithm.
- `SigmaL::Union{UniformScaling{Int}, LowerTriangular{Float64}}` : Cholesky factorization of the covariance matrix for the multivariate normal proposal distribution.

2.4.10 Metropolis-Adjusted Langevin Algorithm (MALA)

Implementation of the Metropolis-Adjusted Langevin Algorithm of Roberts and Tweedie *[81]* and Roberts and Stramer *[80]*. The sampler simulates autocorrelated draws from a distribution that can be specified up to a constant of proportionality. MALA is related to Hamiltonian Monte Carlo as described thoroughly by Girolami and Calderhead *[40]*.

Model-Based Constructors

MALA(*params::ElementOrVector{Symbol}*, *epsilon::Real; dtype::Symbol=:forward*)
MALA(*params::ElementOrVector{Symbol}*, *epsilon::Real*, *Sigma::Matrix{T<:Real}; dtype::Symbol=:forward*)

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

Arguments

- `params` : stochastic node(s) to be updated with the sampler. Constrained parameters are mapped to unconstrained space according to transformations defined by the *Stochastic* `unlist()` function.
- `epsilon` : factor by which the drift and covariance matrix of the proposal distribution are scaled.
- `Sigma` : covariance matrix for the multivariate normal proposal distribution. The covariance matrix is relative to the unconstrained parameter space, where candidate draws are generated. If omitted, the identity matrix is assumed.
- **`dtype`** [type of differentiation for gradient calculations. Options are]
  - `:central` : central differencing.
  - `:forward` : forward differencing.

Value

Returns a `Sampler{MALATune}` type object.

Example

See the *Dyes* and other *Examples*.

Stand-Alone Function

sample!(*v::MALAVariate*)

Draw one sample from a target distribution using the MALA sampler. Parameters are assumed to be continuous and unconstrained.

Arguments

- `v` : current state of parameters to be simulated.

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 = Dict(
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
)

## Log-transformed Posterior(b0, b1, log(s2)) + Constant and Gradient␣
↪Vector
logfgrad = function(x::DenseVector)
  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 Metropolis-Adjusted Langevin Algorithm
## Without (1) and with (2) a user-specified proposal covariance matrix
n = 5000
sim1 = Chains(n, 3, names = ["b0", "b1", "s2"])
sim2 = Chains(n, 3, names = ["b0", "b1", "s2"])
epsilon = 0.1
Sigma = eye(3)
theta1 = MALAVariate([0.0, 0.0, 0.0], epsilon, logfgrad)
theta2 = MALAVariate([0.0, 0.0, 0.0], epsilon, Sigma, logfgrad)
for i in 1:n
  sample!(theta1)
  sample!(theta2)
  sim1[i, :, 1] = [theta1[1:2]; exp(theta1[3])]
  sim2[i, :, 1] = [theta2[1:2]; exp(theta2[3])]
end
describe(sim1)
describe(sim2)
```

MALAVariate Type

Declaration

```
const MALAVariate = SamplerVariate{MALATune}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::MALATune` : tuning parameters for the sampling algorithm.

Constructors

`MALAVariate`(*x::AbstractVector{T<:Real}*, *epsilon::Real*, *logfgrad::Function*)
`MALAVariate`(*x::AbstractVector{T<:Real}*, *epsilon::Real*, *Sigma::Matrix{U<:Real}*, *logfgrad::Function*)

Construct a `MALAVariate` object that stores simulated values and tuning parameters for MALA sampling.

Arguments

- `x` : initial values.
- `epsilon` : factor by which the drift and covariance matrix of the proposal distribution are scaled.
- `Sigma` : covariance matrix for the multivariate normal proposal distribution. The covariance matrix is relative to the unconstrained parameter space, where candidate draws are generated. If omitted, the identity matrix is assumed.
- `logfgrad` : function that takes a single `DenseVector` argument of parameter values at which to compute the log-transformed density (up to a normalizing constant) and gradient vector, and returns the respective results as a tuple.

Value

Returns a `MALAVariate` type object with fields set to the supplied `x` and tuning parameter values.

MALATune Type

Declaration

```
type MALATune <:  SamplerTune
```

Fields

- `logfgrad::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density and gradient vector, or null if not supplied.
- `epsilon::Float64` : factor by which the drift and covariance matrix of the proposal distribution are scaled.
- `SigmaL::Union{UniformScaling{Int}, LowerTriangular{Float64}}` : Cholesky factorization of the covariance matrix for the multivariate normal proposal distribution.

2.4.11 Missing Values Sampler (MISS)

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

Model-Based Constructor

MISS (*params::ElementOrVector{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 node(s) that contain missing values (`NaN`) to be updated with the sampler.

Value

Returns a `Sampler{Dict{Symbol, MISSTune}}` type object.

Example

See the *Bones* and other *Examples*.

MISSTune Type

Declaration

```
type MISSTune
```

Fields

- `dims::Tuple` : dimensions of a stochastic node to be updated.
- `valueinds::Vector{Int}` : indices to missing values in the node.
- `distrinds::Vector{Int}` : indices to node distributions from which to sample the missing values.

2.4.12 No-U-Turn Sampler (NUTS)

Implementation of the No-U-Turn Sampler extension (algorithm 6) *[48]* to Hamiltonian Monte Carlo *[65]* for simulating autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Model-Based Constructor

NUTS (*params::ElementOrVector{Symbol}; dtype::Symbol=:forward, args...*)

Construct a `Sampler` object for NUTS 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 node(s) to be updated with the sampler. Constrained parameters are mapped to unconstrained space according to transformations defined by the *Stochastic* `unlist()` function.

- **dtype** [type of differentiation for gradient calculations. Options are]
  - :central : central differencing.
  - :forward : forward differencing.
- args... : additional keyword arguments to be passed to the NUTSVariate constructor.

Value

Returns a Sampler{NUTSTune} type object.

Example

See the *Dyes*, *Equiv*, and other *Examples*.

Stand-Alone Function

sample!(*v::NUTSVariate; adapt::Bool=false*)

Draw one sample from a target distribution using the NUTS 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 will contain the tune field returned by the previous call.
- adapt : whether to adaptively update the epsilon step size parameter.

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 = Dict(
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
)

## Log-transformed Posterior(b0, b1, log(s2)) + Constant and Gradient␣
↪Vector
logfgrad = function(x::DenseVector)
  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], logfgrad)
for i in 1:n
  sample!(theta, adapt = (i <= burnin))
  if i > burnin
    sim[i, :, 1] = [theta[1:2]; exp(theta[3])]
  end
end
describe(sim)
```

NUTSVariate Type

Declaration

```
const NUTSVariate = SamplerVariate{NUTSTune}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::NUTSTune` : tuning parameters for the sampling algorithm.

Constructors

`NUTSVariate`(*x::AbstractVector{T<:Real}*, *epsilon::Real*, *logfgrad::Function; target::Real=0.6*)
`NUTSVariate`(*x::AbstractVector{T<:Real}*, *logfgrad::Function; target::Real=0.6*)

Construct a `NUTSVariate` object that stores simulated values and tuning parameters for NUTS sampling.

Arguments

- `x` : initial values.
- `epsilon` : step size parameter.
- `logfgrad` : function that takes a single `DenseVector` argument of parameter values at which to compute the log-transformed density (up to a normalizing constant) and gradient vector, and returns the respective results as a tuple. If `epsilon` is not specified, the function is used by the constructor to generate an initial step size value.
- `target` : target acceptance rate for the algorithm.

Value

Returns a `NUTSVariate` type object with fields set to the supplied `x` and tuning parameter values.

NUTSTune Type

Declaration

```
type NUTSTune <:  SamplerTune
```

Fields

- `logfgrad::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density and gradient vector, or null if not supplied.
- `adapt::Bool` : whether the proposal distribution is being 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 `m > 0`, and the user-supplied 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::Int` : number of adaptive update iterations m that have been performed.
- `mu::Float64` : dual averaging parameter, defined as $\mu = \log(10\epsilon_0)$.
- `nalpha::Int` : 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.

2.4.13 Random Walk Metropolis (RWM)

Random walk Metropolis-Hastings algorithm *[46][63]* in which parameters are sampled from symmetric distributions centered around the current values. The sampler simulates autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Model-Based Constructor

RWM(*params::ElementOrVector{Symbol}*, *scale::ElementOrVector{T<:Real}; args...*)

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

Arguments

- `params` : stochastic node(s) to be updated with the sampler. Constrained parameters are mapped to unconstrained space according to transformations defined by the *Stochastic* `unlist()` function.

- `scale` : scaling value or vector of the same length as the combined elements of nodes `params` for the `proposal` distribution. Values are relative to the unconstrained parameter space, where candidate draws are generated.
- `args...` : additional keyword arguments to be passed to the `RWMVariate` constructor.

Value

Returns a `Sampler{RWMTune}` type object.

Example

See the *Dyes* and other *Examples*.

Stand-Alone Function

`sample!`(*v::RWMVariate*)

Draw one sample from a target distribution using the RWM sampler. Parameters are assumed to be continuous and unconstrained.

Arguments

- `v` : current state of parameters to be simulated.

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 = Dict(
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
)

## Log-transformed Posterior(b0, b1, log(s2)) + Constant
logf = function(x::DenseVector)
   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 Random Walk Metropolis
n = 5000
burnin = 1000
sim = Chains(n, 3, names = ["b0", "b1", "s2"])
theta = RWMVariate([0.0, 0.0, 0.0], [0.5, 0.25, 1.0], logf,
                   proposal = SymUniform)
for i in 1:n
  sample!(theta)
  sim[i, :, 1] = [theta[1:2]; exp(theta[3])]
end
describe(sim)
```

RWMVariate Type

Declaration

```
const RWMVariate = SamplerVariate{RWMTune}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::RWMTune` : tuning parameters for the sampling algorithm.

Constructor

`RWMVariate`(*x::AbstractVector{T<:Real}, scale::ElementOrVector{U<:Real}, logf::Function; proposal::SymDistributionType=Normal*)

Construct a `RWMVariate` object that stores simulated values and tuning parameters for RWM sampling.

Arguments

- `x` : initial values.
- `scale` : scalar or vector of the same length as `x` for the `proposal` distribution.
- `logf` : function that takes a single `DenseVector` argument of parameter values at which to compute the log-transformed density (up to a normalizing constant).
- `proposal` : symmetric distribution of type `Biweight`, `Cosine`, `Epanechnikov`, `Normal`, `SymTriangularDist`, `SymUniform`, or `Triweight` to be centered around current parameter values and used to generate proposal draws. Specified `scale` determines the standard deviations of Normal proposals and widths of the others.

Value

Returns a `RWMVariate` type object with fields set to the supplied `x` and tuning parameter values.

RWMTune Type

Declaration

```
type RWMTune <:  SamplerTune
```

Fields

- `logf::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density, or null if not supplied.
- `scale::Union{Float64, Vector{Float64}}` : scaling for the proposal distribution.
- `proposal::SymDistributionType` : proposal distribution.

2.4.14 Shrinkage Slice (Slice)

Implementation of the shrinkage slice sampler of Neal *[64]* for simulating autocorrelated draws from a distribution that can be specified up to a constant of proportionality.

Model-Based Constructor

`Slice`(*params::ElementOrVector{Symbol}*, *width::ElementOrVector{T<:Real}*, *::Type{F<:SliceForm}=Multivariate; transform::Bool=false*)

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

Arguments

- `params` : stochastic node(s) to be updated with the sampler.
- `width` : scaling value or vector of the same length as the combined elements of nodes `params`, defining initial widths of a hyperrectangle from which to simulate values.
- **`F`** [sampler type. Options are]
  - `Univariate` : sequential univariate sampling of parameters .
  - `Multivariate` : joint multivariate 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* `unlist()` 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{SliceTune{Univariate}}` or `Sampler{SliceTune{Multivariate}}` type object if sampling univariately or multivariately, respectively.

Example

See the *Birats*, *Rats*, and other *Examples*.

Stand-Alone Functions

`sample!`(*v::SliceUnivariate*)
`sample!`(*v::SliceMultivariate*)

Draw one sample from a target distribution using the Slice univariate or multivariate sampler. Parameters are assumed to be continuous, but may be constrained or unconstrained.

Arguments

- `v` : current state of parameters to be simulated.

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*. Also, see the *Line: Block-Specific Sampling with AMWG and Slice* example.

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

using Mamba

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

## Log-transformed Posterior(b0, b1, log(s2)) + Constant
logf = function(x::DenseVector)
   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 Slice Sampling
## With multivariate (1) and univariate (2) updating
n = 5000
sim1 = Chains(n, 3, names = ["b0", "b1", "s2"])
sim2 = Chains(n, 3, names = ["b0", "b1", "s2"])
width = [1.0, 1.0, 2.0]
theta1 = SliceUnivariate([0.0, 0.0, 0.0], width, logf)
theta2 = SliceMultivariate([0.0, 0.0, 0.0], width, logf)
for i in 1:n
  sample!(theta1)
  sample!(theta2)
  sim1[i, :, 1] = [theta1[1:2]; exp(theta1[3])]
  sim2[i, :, 1] = [theta2[1:2]; exp(theta2[3])]
end
describe(sim1)
describe(sim2)
```

Slice Variate Types

Declaration

```
const SliceUnivariate = SamplerVariate{SliceTune{Univariate}}
const SliceMultivariate = SamplerVariate{SliceTune{Multivariate}}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::SliceTune{F<:SliceForm}` : tuning parameters for the sampling algorithm.

Constructors

`SliceUnivariate`(*x::AbstractVector{T<:Real}*, *width::ElementOrVector{U<:Real}*, *logf::Function*)
`SliceMultivariate`(*x::AbstractVector{T<:Real}*, *width::ElementOrVector{U<:Real}*, *logf::Function*)

Construct an object that stores simulated values and tuning parameters for Slice sampling.

Arguments

- `x` : initial values.
- `width` : scaling value or vector of the same length as the combined elements of nodes `params`, defining initial widths of a hyperrectangle from which to simulate values.
- `logf` : function that takes a single `DenseVector` argument of parameter values at which to compute the log-transformed density (up to a normalizing constant).

Value

Returns an object of the same type as the constructor name for univariate or multivariate sampling, respectively, with fields set to the supplied `x` and tuning parameter values.

SliceTune Type

Declaration

```
const SliceForm = Union{Univariate, Multivariate}
type SliceTune{F<:SliceForm} <: SamplerTune
```

Fields

- `logf::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density, or null if not supplied.
- `width::Union{Float64, Vector{Float64}}` : initial widths of hyperrectangles from which to simulate values.

2.4.15 Slice Simplex (SliceSimplex)

Implementation of the slice simplex sampler as described by Cowles et al. *[19]* for simulating autocorrelated draws of parameters on the simplex $\{\theta_1, \ldots, \theta_d : \theta_i \geq 0, \sum_{i=1}^{d} \theta_i = 1\}$ and from a distribution that can be specified up to a constant of proportionality.

Model-Based Constructor

`SliceSimplex`(*params::ElementOrVector{Symbol}; args...*)

Construct a `Sampler` object for which SliceSimplex sampling is to be applied separately to each of the supplied parameters. Parameters are assumed to be continuous and constrained to a simplex.

Arguments

- `params` : stochastic node(s) to be updated with the sampler.
- `args...` : additional keyword arguments to be passed to the `SliceSimplexVariate` constructor.

Value

Returns a `Sampler{SliceSimplexTune}` type object.

Example

See the *Asthma*, *Eyes*, and other *Examples*.

Stand-Alone Function

`sample!`(*v::SliceSimplexVariate*)

Draw one sample from a target distribution using the SliceSimplex sampler. Parameters are assumed to be continuous and constrained to a simplex.

Arguments

- `v` : current state of parameters to be simulated.

Value

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

Example

```
#########################################################################
↪######
## Multinomial Model
##   y ~ Multinomial(n, rho)
##   rho ~ Dirichlet(1, ..., 1)
#########################################################################
↪######

using Mamba

## Data
n, k = 100, 5
rho0 = rand(Dirichlet(ones(k)))
y = rand(Multinomial(n, rho0))

## Log-transformed Posterior(rho) + Constant
logf = function(rho::DenseVector)
  logpdf(Multinomial(n, rho), y)
```

```
end

## MCMC Simulation with Slice Simplex Sampling
t = 10000
sim = Chains(t, k, names = map(i -> "rho[$i]", 1:k))
rho = SliceSimplexVariate(fill(1 / k, k), logf)
for i in 1:t
  sample!(rho)
  sim[i, :, 1] = rho
end
describe(sim)

p = plot(sim)
draw(p, filename = "slicesimplexplot")
```

SliceSimplexVariate Type

Declaration

```
const SliceSimplexVariate = SamplerVariate{SliceSimplexTune}
```

Fields

- `value::Vector{Float64}` : simulated values.
- `tune::SliceSimplexTune` : tuning parameters for the sampling algorithm.

Constructor

`SliceSimplexVariate`(*x::AbstractVector{T<:Real}*, *logf::Function; scale::Real=1.0*)

Construct a `SliceSimplexVariate` object that stores simulated values and tuning parameters for slice simplex sampling.

Arguments

- `x` : initial values.
- `scale` : value `0 < scale <= 1` by which to scale the standard simplex to define an initial space from which to simulate values.
- `logf` : function that takes a single `DenseVector` argument of parameter values at which to compute the log-transformed density (up to a normalizing constant).

Value

Returns a `SliceSimplexVariate` type object with fields set to the supplied `x` and tuning parameter values.

SliceSimplexTune Type

Declaration

```
type SliceSimplexTune <:  SamplerTune
```

Fields

- `logf::Nullable{Function}` : function supplied to the constructor to compute the log-transformed density, or null if not supplied.
- `scale::Float64` : value `0 < scale <= 1` by which to scale the standard simplex to define an initial space from which to simulate values.

The following table summarizes the (d-dimensional) sample spaces over which each method simulates draws, whether draws are generated univariately or multivariately, and whether transformations are applied to map parameters to the sample spaces.

Table 2.3: Summary of sampling methods and their characteristics.

		Model-Based Constructors			Stand-Alone Functions	
Method	Sample Space	Univariate	Multivariate	Transformations	Univariate	Multivariate
ABC	$\mathbb{R}^d$	No	Yes	Yes	No	No
AMM	$\mathbb{R}^d$	No	Yes	Yes	No	Yes
AMWG	$\mathbb{R}^d$	Yes	No	Yes	Yes	No
BHMC	$\{0,1\}^d$	No	Yes	No	No	Yes
BIA	$\{0,1\}^d$	No	Yes	No	No	Yes
BMC3	$\{0,1\}^d$	Yes	Yes	No	Yes	Yes
BMG	$\{0,1\}^d$	Yes	Yes	No	Yes	Yes
DGS	Finite $S \subset \mathbb{Z}^d$	Yes	No	No	No	Yes
HMC	$\mathbb{R}^d$	No	Yes	Yes	No	Yes
MALA	$\mathbb{R}^d$	No	Yes	Yes	No	Yes
MISS	Parameter-defined	Yes	Yes	No	No	No
NUTS	$\mathbb{R}^d$	No	Yes	Yes	No	Yes
RWM	$\mathbb{R}^d$	No	Yes	Yes	No	Yes
Slice	$S \subseteq \mathbb{R}^d$	Yes	Yes	Optional	Yes	Yes
SliceSimplex	d-simplex	No	Yes	No	No	Yes

2.5 Examples

2.5.1 OpenBUGS

The following examples are taken from OpenBUGS *[44]*, 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.

Volume I

Rats: A Normal Hierarchical Model

An example from OpenBUGS *[44]* and section 6 of Gelfand *et al.* *[29]* 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 = Dict{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] = Int[div(i - 1, 5) + 1 for i in 1:150]
rats[:week] = Int[(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,
    (alpha, beta, rat, Xm, s2_c) ->
      begin
        mu = alpha[rat] + beta[rat] .* Xm
        MvNormal(mu, sqrt(s2_c))
      end,
    false
  ),

  alpha = Stochastic(1,
    (mu_alpha, s2_alpha) -> Normal(mu_alpha, sqrt(s2_alpha)),
    false
  ),

  alpha0 = Logical(
    (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,
    (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 = [
  Dict(: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),
  Dict(: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, 100.0),
          Slice([:mu_alpha, :s2_alpha], [100.0, 10.0], Univariate),
          AMWG(:beta, 1.0),
          Slice([:mu_beta, :s2_beta], 1.0, Univariate)]
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
   s2_c  37.2543133 6.026634572 0.0695895819 0.2337982327  664.4576
mu_beta   6.1830663 0.108042927 0.0012475723 0.0017921615 3634.4474
 alpha0 106.6259925 3.459210115 0.0399435178 0.0526804390 3750.0000

Quantiles:
           2.5%      25.0%       50.0%       75.0%       97.5%
   s2_c 27.778388  33.0906026  36.4630047  40.5538472  51.5713716
mu_beta  5.969850   6.1110307   6.1836454   6.2538831   6.3964953
 alpha0 99.815707 104.3369878 106.6105679 108.9124224 113.5045347
```

Pumps: Gamma-Poisson Hierarchical Model

An example from OpenBUGS *[44]* and George *et al.* *[36]* 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 = Dict{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,
    (theta, t, N) ->
      UnivariateDistribution[
        begin
          lambda = theta[i] * t[i]
          Poisson(lambda)
        end
        for i in 1:N
      ],
    false
  ),

  theta = Stochastic(1,
    (alpha, beta) -> Gamma(alpha, 1 / beta),
    true
  ),

  alpha = Stochastic(
    () -> Exponential(1.0)
  ),

  beta = Stochastic(
    () -> Gamma(0.1, 1.0)
  )

)


## Initial Values
inits = [
```

```
  Dict(:y => pumps[:y], :alpha => 1.0, :beta => 1.0,
       :theta => rand(Gamma(1.0, 1.0), pumps[:N])),
  Dict(: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, Univariate),
          Slice(:theta, 1.0, Univariate)]
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
     beta 0.93036099 0.517274134 0.00597296721 0.01824153419  804.11618
    alpha 0.69679849 0.264420102 0.00305326034 0.00722593007 1339.06428
 theta[1] 0.05991674 0.025050913 0.00028926303 0.00032725274 3750.00000
 theta[2] 0.10125873 0.078887354 0.00091091270 0.00129985769 3683.18182
 theta[3] 0.08910382 0.037307731 0.00043079257 0.00045660986 3750.00000
 theta[4] 0.11529009 0.030274265 0.00034957710 0.00032583333 3750.00000
 theta[5] 0.59971611 0.316811730 0.00365822675 0.00585119652 2931.65686
 theta[6] 0.60967188 0.134761371 0.00155609028 0.00174949908 3750.00000
 theta[7] 0.86767451 0.669943846 0.00773584520 0.02858200254  549.40360
 theta[8] 0.85445727 0.668132036 0.00771492422 0.02485547216  722.57105
 theta[9] 1.55721556 0.753564449 0.00870141275 0.03109274798  587.38464
theta[10] 1.98475207 0.405438843 0.00468160451 0.00912748779 1973.09587

Quantiles:
              2.5%       25.0%      50.0%      75.0%      97.5%
     beta 0.189558591 0.55034346 0.84013720 1.22683690 2.134324768
    alpha 0.286434561 0.50214938 0.66295652 0.85427430 1.319308645
 theta[1] 0.021291808 0.04164372 0.05696335 0.07435302 0.118041776
 theta[2] 0.008789962 0.04279772 0.08116810 0.13771833 0.305675244
 theta[3] 0.032503106 0.06202730 0.08293498 0.11024076 0.176883533
 theta[4] 0.063587574 0.09389172 0.11222268 0.13443866 0.182555751
 theta[5] 0.153474125 0.36947945 0.54009121 0.77234190 1.364481338
 theta[6] 0.371096082 0.51395467 0.60089332 0.69531957 0.898866079
```

```
 theta[7] 0.077416146 0.37391993 0.71230582 1.17659703 2.616100803
 theta[8] 0.072479432 0.35973235 0.69319639 1.16347299 2.655857786
 theta[9] 0.463821785 1.01169918 1.43745818 1.96871605 3.351798915
theta[10] 1.269842527 1.70167020 1.95748757 2.24000075 2.861197982

## 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]  5.5784000 3.3457654 0.038633571 0.042295418 3750.0000
 y[2]  1.6009333 1.7532320 0.020244579 0.022399464 3750.0000
 y[3]  5.5854667 3.2976980 0.038078537 0.040503717 3750.0000
 y[4] 14.5666667 5.4547523 0.062986054 0.061718435 3750.0000
 y[5]  3.1118667 2.4165260 0.027903638 0.035965796 3750.0000
 y[6] 19.0780000 6.0320229 0.069651801 0.071700830 3750.0000
 y[7]  0.9008000 1.1756177 0.013574864 0.031805581 1366.2355
 y[8]  0.8922667 1.1550634 0.013337523 0.025610086 2034.1808
 y[9]  3.2585333 2.4039269 0.027758156 0.069046712 1212.1504
y[10] 20.7933333 6.2068800 0.071670876 0.113975236 2965.6896

Quantiles:
      2.5% 25.0% 50.0% 75.0% 97.5%
 y[1]    1     3     5     8    13
 y[2]    0     0     1     2     6
 y[3]    1     3     5     7    13
 y[4]    5    11    14    18    27
 y[5]    0     1     3     4     9
 y[6]    9    15    19    23    32
 y[7]    0     0     1     1     4
 y[8]    0     0     1     1     4
 y[9]    0     1     3     5     9
y[10]   10    16    20    25    34
```

Dogs: Loglinear Model for Binary Data

An example from OpenBUGS *[44]*, Lindley and Smith *[54]*, and Kalbfleisch *[52]* 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 = Dict{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
     0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 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
     0 1 1 0 0 1 1 1 1 0 1 0 1 0 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
     0 0 0 0 0 0 1 1 1 1 0 0 1 0 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
     0 0 0 0 0 0 0 1 1 0 0 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
     0 0 0 0 0 0 0 0 0 0 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
     0 0 0 1 1 0 1 0 0 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
     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 0 0 0 0 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
     0 0 0 0 1 0 1 0 1 1 1 1 1 0 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
     0 0 0 0 1 1 0 1 0 1 1 0 1 0 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
     0 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0 0 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
     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 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
     0 0 0 0 1 0 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
     0 0 0 0 1 1 0 0 1 1 1 0 1 0 1 0 1 0 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]
)
dogs[:Dogs] = size(dogs[:Y], 1)
dogs[:Trials] = size(dogs[:Y], 2)

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


## Model Specification
model = Model(

  y = Stochastic(2,
    (Dogs, Trials, alpha, xa, beta, xs) ->
      UnivariateDistribution[
        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(
    alpha -> exp(alpha)
  ),

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

  B = Logical(
    beta -> exp(beta)
  )

)


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


## Sampling Scheme
scheme = [Slice([:alpha, :beta], 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
 beta -0.0788601 0.011812880 0.00013640339 0.00018435162 3750.0000
    B  0.9242336 0.010903201 0.00012589932 0.00017012187 3750.0000
alpha -0.2441654 0.024064384 0.00027787158 0.00045249868 2828.2310
    A  0.7835846 0.018821132 0.00021732772 0.00035466850 2816.0882

Quantiles:
          2.5%        25.0%       50.0%       75.0%       97.5%
 beta -0.10326776 -0.086633844 -0.07848501 -0.07046663 -0.05703080
    B  0.90188545  0.917012804  0.92451592  0.93195884  0.94456498
```

```
alpha -0.29269315 -0.260167645 -0.24381478 -0.22784322 -0.19872975
    A  0.74625110  0.770922334  0.78363276  0.79624909  0.81977141
```

Seeds: Random Effect Logistic Regression

An example from OpenBUGS *[44]*, Crowder *[20]*, and Breslow and Clayton *[10]* 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 = Dict{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,
    (alpha0, alpha1, x1, alpha2, x2, alpha12, b, n, N) ->
      UnivariateDistribution[
        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,
    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 = [
  Dict(:r => seeds[:r], :alpha0 => 0, :alpha1 => 0, :alpha2 => 0,
       :alpha12 => 0, :s2 => 0.01, :b => zeros(seeds[:N])),
  Dict(: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, 0.01),
          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  1.310728093 0.26053104 0.0026053104 0.0153996801 286.21707
 alpha1  0.088700176 0.26872879 0.0026872879 0.0128300598 438.70341
 alpha0 -0.556154341 0.17595432 0.0017595432 0.0101730837 299.15388
alpha12 -0.746440855 0.43006756 0.0043006756 0.0251658152 292.04607
     s2  0.085705306 0.09738014 0.0009738014 0.0080848189 145.07755

Quantiles:
             2.5%         25.0%        50.0%       75.0%       97.5%
 alpha2  0.8040593795  1.148881827  1.309947687  1.48076318  1.82815608
 alpha1 -0.4250164771 -0.093637900  0.094390643  0.26007581  0.62353933
 alpha0 -0.9149197759 -0.666632319 -0.551292851 -0.44262420 -0.22244775
alpha12 -1.5457041398 -1.027576522 -0.757250262 -0.49149187  0.17029702
     s2  0.0011739822  0.021117624  0.059376315  0.11140082  0.35234645
```

Surgical: Institutional Ranking

An example from OpenBUGS *[44]* 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 = Dict{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,
    (n, p, N) ->
      UnivariateDistribution[Binomial(n[i], p[i]) for i in 1:N],
    false
  ),
```

```
  p = Logical(1,
    b -> invlogit.(b)
  ),

  b = Stochastic(1,
    (mu, s2) -> Normal(mu, sqrt(s2)),
    false
  ),

  mu = Stochastic(
    () -> Normal(0, 1000)
  ),

  pop_mean = Logical(
    mu -> invlogit(mu)
  ),

  s2 = Stochastic(
    () -> InverseGamma(0.001, 0.001)
  )

)

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


## Sampling Scheme
scheme = [NUTS(:b),
          Slice([:mu, :s2], 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 -2.550263247 0.151813688 0.001752993476 0.00352027397 1859.8115
pop_mean  0.073062651 0.010097974 0.000116601357 0.00022880854 1947.7088
      s2  0.183080212 0.161182244 0.001861172237 0.00629499754  655.6065
    p[1]  0.053571675 0.019444542 0.000224526231 0.00059140521 1080.9986
    p[2]  0.103203725 0.022015796 0.000254216516 0.00049928289 1944.3544
    p[3]  0.071050270 0.017662050 0.000203943787 0.00022426787 3750.0000
```

```
    p[4]  0.059863573 0.008208359 0.000094781965 0.00033190971  611.6074
    p[5]  0.052400628 0.013632842 0.000157418497 0.00052344316  678.3186
    p[6]  0.069799258 0.014697820 0.000169715805 0.00026854081 2995.6099
    p[7]  0.066927057 0.015703466 0.000181328008 0.00023888938 3750.0000
    p[8]  0.122296440 0.023319424 0.000269269516 0.00086456417  727.5137
    p[9]  0.070386216 0.014318572 0.000165336624 0.00019674962 3750.0000
   p[10]  0.077978945 0.019775182 0.000228344129 0.00031544646 3750.0000
   p[11]  0.101809685 0.017568938 0.000202868617 0.00037809859 2159.1405
   p[12]  0.068534503 0.011579540 0.000133709009 0.00015162331 3750.0000

Quantiles:
             2.5%         25.0%         50.0%         75.0%         97.5%
      mu -2.878686485 -2.637983754 -2.540103608 -2.455828143 -2.269200544
pop_mean  0.053217279  0.066733498  0.073094153  0.079013392  0.093706084
      s2  0.011890649  0.082427721  0.141031919  0.233947847  0.601137558
    p[1]  0.017683165  0.039573948  0.052942713  0.067219732  0.092219913
    p[2]  0.067482073  0.086541186  0.100283541  0.116521584  0.153268132
    p[3]  0.040088134  0.059099562  0.070320337  0.080842394  0.110595078
    p[4]  0.044965780  0.054145641  0.059405801  0.065149564  0.076754218
    p[5]  0.028224773  0.042599948  0.051613569  0.061247876  0.079162530
    p[6]  0.043031618  0.059518315  0.069130011  0.079935035  0.100122661
    p[7]  0.038586653  0.055992241  0.066465865  0.076006492  0.101351899
    p[8]  0.082604334  0.105852706  0.121313052  0.137724088  0.171202438
    p[9]  0.044453383  0.060622475  0.070120723  0.079078759  0.101046045
   p[10]  0.044439834  0.064694148  0.075946776  0.089236114  0.122899027
   p[11]  0.071966143  0.088360531  0.100133691  0.112632331  0.139968412
   p[12]  0.047466661  0.060611827  0.068267225  0.075638053  0.093309250
```

Magnesium: Meta-Analysis Prior Sensitivity

An example from OpenBUGS *[44]*.

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 = Dict{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,
    (nc, pc) ->
      UnivariateDistribution[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,
    (nt, pc, theta) ->
      UnivariateDistribution[
        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,
    (mu, tau) ->
      UnivariateDistribution[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,
```

```
    mu -> exp.(mu)
  ),

  tau = Logical(1,
    (priors, s2_0) ->
      Float64[
        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,
    s2_0 ->
      UnivariateDistribution[
        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 = [
  Dict(: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]),
  Dict(: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, 0.1),
          AMWG(:mu, 0.1),
          Slice(:pc, 0.25, Univariate),
          Slice(:priors, [1.0, 5.0, 5.0, 0.25, 0.25, 5.0], Univariate)]
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
tau[1] 0.55098858 0.35814901 0.0035814901 0.0221132365  262.31486
tau[2] 1.11557619 0.58886505 0.0058886505 0.0237788755  613.26606
tau[3] 0.83211110 0.49113676 0.0049113676 0.0222839957  485.75664
tau[4] 0.47864203 0.26258828 0.0026258828 0.0135868530  373.51920
tau[5] 0.48624861 0.35359386 0.0035359386 0.0215005369  270.46485
tau[6] 0.56841884 0.18877962 0.0018877962 0.0058505056 1041.17429
 OR[1] 0.47784058 0.15389133 0.0015389133 0.0066922017  528.79852
 OR[2] 0.42895913 0.32240192 0.0032240192 0.0081170895 1577.59150
 OR[3] 0.43118350 0.18264467 0.0018264467 0.0064385836  804.69879
 OR[4] 0.47587697 0.13947735 0.0013947735 0.0064893426  461.96170
 OR[5] 0.48545299 0.14603013 0.0014603013 0.0083912319  302.85415
 OR[6] 0.44554385 0.14121352 0.0014121352 0.0053818401  688.47941

Quantiles:
          2.5%       25.0%      50.0%      75.0%      97.5%
tau[1] 0.050143630 0.28821905 0.49325476 0.73991793 1.42033542
tau[2] 0.326282292 0.70071249 0.98873505 1.39092470 2.65606546
tau[3] 0.136936046 0.49195419 0.74461372 1.06769203 2.03061696
tau[4] 0.091858771 0.28921566 0.44085251 0.61784984 1.10639558
tau[5] 0.028866916 0.23628679 0.42220429 0.65955432 1.37318750
tau[6] 0.214834142 0.43417486 0.56753402 0.70014948 0.94171060
 OR[1] 0.206501871 0.37431326 0.47128600 0.57044051 0.80062151
 OR[2] 0.107428346 0.27074745 0.38362516 0.52062237 0.99299623
 OR[3] 0.145435475 0.30454141 0.41470065 0.53630024 0.83015778
 OR[4] 0.231777387 0.38069803 0.47049713 0.56112444 0.76292805
 OR[5] 0.207697044 0.38509504 0.48308284 0.59166588 0.75526778
 OR[6] 0.208377218 0.34750042 0.43313882 0.52797553 0.76192141
```

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

An example from OpenBUGS *[44]* and Breslow *[9]* 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 = Dict{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,
    (alpha, beta, gamma, x, lambda) ->
      UnivariateDistribution[
        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,
    s2 -> Normal(0, sqrt(s2)),
    false
  ),

  s2 = Stochastic(
    () -> InverseGamma(0.001, 0.001)
  )

)


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

```
  Dict(: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], 0.1)]
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
   s2  0.0690769709 0.04304237136 0.000497010494 0.001985478741 469.961085
gamma -0.0011250515 0.00034536546 0.000003987937 0.000025419899 184.590851
 beta  0.3543443166 0.07160779229 0.000826855563 0.007122805926 101.069088
alpha  2.0100584321 0.26156942610 0.003020343571 0.027100522461  93.157673

Quantiles:
           2.5%          25.0%         50.0%          75.0%          97.5%
   s2  0.0135820292  0.039788136  0.0598307554  0.08740879470  0.17747300708
gamma -0.0017930379 -0.001347435 -0.0011325733 -0.00090960998 -0.00039271486
 beta  0.2073237562  0.312747679  0.3582979574  0.40006921583  0.48789037325
alpha  1.5060295212  1.847111545  2.0062727893  2.16556036713  2.50547846044
```

Equiv: Bioequivalence in a Cross-Over Trial

An example from OpenBUGS *[44]* and Gelfand *et al.* *[29]* 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 = Dict{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,
    (delta, mu, phi, pi, s2_1, T) ->
      begin
        sigma = sqrt(s2_1)
        UnivariateDistribution[
          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,
    s2_2 -> Normal(0, sqrt(s2_2)),
    false
  ),

  mu = Stochastic(
    () -> Normal(0, 1000)
  ),

  phi = Stochastic(
    () -> Normal(0, 1000)
  ),

  theta = Logical(
```

```
    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(
    theta -> Int(0.8 < theta < 1.2)
  )

)


## Initial Values
inits = [
  Dict(:y => equiv[:y], :delta => zeros(10, 2), :mu => 0, :phi => 0,
       :pi => 0, :s2_1 => 1, :s2_2 => 1),
  Dict(: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], 1.0),
          Slice([:s2_1, :s2_2], 1.0, Univariate)]
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
 s2_2  0.0173121833 0.014549568 0.00014549568 0.0007329722  394.02626
 s2_1  0.0184397014 0.013837972 0.00013837972 0.0005689492  591.55873
   pi -0.1874240524 0.086420302 0.00086420302 0.0032257037  717.76558
  phi -0.0035569545 0.087590520 0.00087590520 0.0035141650  621.25503
theta  1.0002921934 0.088250458 0.00088250458 0.0036227671  593.40761
```

```
equiv  0.9751000000 0.155828169 0.00155828169 0.0036666529 1806.14385
   mu  1.4387396416 0.042269208 0.00042269208 0.0013735876  946.96847

Quantiles:
           2.5%          25.0%          50.0%         75.0%         97.5%
 s2_2  0.0016375061  0.0056159514  0.013968228  0.024613730  0.053154674
 s2_1  0.0017289114  0.0074958338  0.015849718  0.025963832  0.051967161
   pi -0.3579631753 -0.2432161807 -0.187946319 -0.130454165 -0.014910965
  phi -0.1723722017 -0.0623573600 -0.005681830  0.053144647  0.172913654
theta  0.8416658455  0.9395470702  0.994334281  1.054582177  1.188763456
equiv  1.0000000000  1.0000000000  1.000000000  1.000000000  1.000000000
   mu  1.3552569594  1.4110400018  1.438593809  1.466525521  1.519643109
```

Dyes: Variance Components Model

An example from OpenBUGS *[44]*, Davies *[21]*, and Box and Tiao *[8]* 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 = Dict{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(collect(1:dyes[:samples]), dyes[:batches])...)


## Model Specification
```

```
model = Model(

  y = Stochastic(1,
    (mu, batch, s2_within) -> MvNormal(mu[batch], sqrt(s2_within)),
    false
  ),

  mu = Stochastic(1,
    (theta, batches, s2_between) -> Normal(theta, sqrt(s2_between))
  ),

  theta = Stochastic(
    () -> Normal(0, 1000)
  ),

  s2_within = Stochastic(
    () -> InverseGamma(0.001, 0.001)
  ),

  s2_between = Stochastic(
    () -> InverseGamma(0.001, 0.001)
  )

)


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


## Sampling Schemes
scheme = [NUTS([:mu, :theta]),
          Slice([:s2_within, :s2_between], 1000.0)]

scheme2 = [MALA(:theta, 50.0),
           MALA(:mu, 50.0, eye(dyes[:batches])),
           Slice([:s2_within, :s2_between], 1000.0)]

scheme3 = [HMC(:theta, 10.0, 5),
           HMC(:mu, 10.0, 5, eye(dyes[:batches])),
           Slice([:s2_within, :s2_between], 1000.0)]

scheme4 = [RWM(:theta, 50.0, proposal=Cosine),
           RWM(:mu, 50.0),
           Slice([:s2_within, :s2_between], 1000.0)]


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

setsamplers!(model, scheme2)
```

```
sim2 = mcmc(model, dyes, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim2)

setsamplers!(model, scheme3)
sim3 = mcmc(model, dyes, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim3)

setsamplers!(model, scheme4)
sim4 = mcmc(model, dyes, inits, 10000, burnin=2500, thin=2, chains=2)
describe(sim4)
```

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 3192.2614 4456.129102 51.45494673 487.44961486   83.57109
     theta 1526.7186   24.549671  0.28347518   0.37724897 3750.00000
 s2_within 2887.5853 1075.174260 12.41504296  76.89117959  195.52607
     mu[1] 1511.4798   20.819711  0.24040531   0.52158448 1593.30921
     mu[2] 1527.9087   20.344151  0.23491402   0.30199960 3750.00000
     mu[3] 1552.6742   21.293738  0.24587891   0.70276515  918.08605
     mu[4] 1506.6440   21.349176  0.24651905   0.61821290 1192.57616
     mu[5] 1578.6636   25.512471  0.29459264   1.29216105  389.82685
     mu[6] 1487.1934   24.693967  0.28514137   1.23710390  398.44592

Quantiles:
              2.5%      25.0%     50.0%     75.0%        97.5%
s2_between  111.92351  815.9012 1651.4605 3269.5663 1.90261752x104
     theta 1475.08243 1513.5687 1527.1763 1540.0550 1.57444106x103
 s2_within 1566.61796 2160.3251 2654.9358 3324.8621 5.65161862x103
     mu[1] 1469.68693 1498.0985 1511.3084 1525.7344 1.55281296x103
     mu[2] 1486.15990 1514.8487 1527.6843 1541.2155 1.56770562x103
     mu[3] 1512.72912 1537.7046 1552.1976 1566.7188 1.59498301x103
     mu[4] 1463.91701 1492.3206 1506.9856 1521.3449 1.54854165x103
     mu[5] 1528.52480 1562.1831 1579.2515 1596.0167 1.62731017x103
     mu[6] 1440.27721 1470.7983 1486.1844 1502.9911 1.54464614x103
```

Stacks: Robust Regression

An example from OpenBUGS *[44]*, Brownlee *[14]*, and Birkes and Dodge *[5]* 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 = Dict{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,
    (mu, s2, N) ->
      UnivariateDistribution[Laplace(mu[i], s2) for i in 1:N],
    false
  ),

  beta0 = Stochastic(
    () -> Normal(0, 1000),
    false
  ),

  beta = Stochastic(1,
    () -> Normal(0, 1000),
    false
  ),

  mu = Logical(1,
    (beta0, z, beta) -> beta0 + z * beta,
    false
  ),

  s2 = Stochastic(
    () -> InverseGamma(0.001, 0.001),
    false
  ),

  sigma = Logical(
    s2 -> sqrt(2.0) * s2
  ),

  b0 = Logical(
    (beta0, b, meanx) -> beta0 - dot(b, meanx)
  ),

  b = Logical(1,
    (beta, sdx) -> beta ./ sdx
  ),

  outlier = Logical(1,
    (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 = [
  Dict(:y => stacks[:y], :beta0 => 10, :beta => [0, 0, 0], :s2 => 10),
  Dict(: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]   0.836863707 0.13085145 0.0015109423 0.0027601754 2247.4171
       b[2]   0.744454449 0.33480007 0.0038659382 0.0065756939 2592.3158
       b[3]  -0.116648437 0.12214077 0.0014103601 0.0015143922 3750.0000
         b0 -38.776564595 8.81860433 0.1018284717 0.0979006137 3750.0000
      sigma   3.487643717 0.87610847 0.0101164292 0.0279025494  985.8889
 outlier[1]   0.042666667 0.20211796 0.0023338572 0.0029490162 3750.0000
 outlier[3]   0.054800000 0.22760463 0.0026281519 0.0034398827 3750.0000
 outlier[4]   0.298000000 0.45740999 0.0052817156 0.0089200654 2629.5123
outlier[21]   0.606400000 0.48858046 0.0056416412 0.0113877443 1840.7583

Quantiles:
                2.5%         25.0%         50.0%         75.0%         97.5%
       b[1]   0.57218621   0.75741345   0.834874964   0.918345319   1.101502854
       b[2]   0.16177144   0.52291878   0.714951465   0.933171533   1.476258382
       b[3]  -0.36401372  -0.19028697  -0.113463801  -0.036994963   0.118538277
         b0 -56.70056875 -44.11785905 -38.698338454 -33.409149788 -21.453323631
      sigma   2.17947513   2.86899865   3.348631697   3.953033535   5.592773118
 outlier[1]   0.00000000   0.00000000   0.000000000   0.000000000   1.000000000
 outlier[3]   0.00000000   0.00000000   0.000000000   0.000000000   1.000000000
 outlier[4]   0.00000000   0.00000000   0.000000000   1.000000000   1.000000000
outlier[21]   0.00000000   0.00000000   1.000000000   1.000000000   1.000000000
```

Epilepsy: Repeated Measures on Poisson Counts

An example from OpenBUGS *[44]*, Thall and Vail *[92]* Breslow and Clayton *[10]* 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 = Dict{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,
    (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) ->
     UnivariateDistribution[
       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,
  s2_b1 -> Normal(0, sqrt(s2_b1)),
  false
),

b = Stochastic(2,
  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(
  (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 = [
  Dict(: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])),
  Dict(: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], 0.1),
          Slice(:b1, 0.5),
          Slice(:b, 0.5),
          Slice([:s2_b1, :s2_b], 1.0)]
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
      s2_b  0.13523750 0.031819272 0.00028460022 0.0025401820 156.91007
     s2_b1  0.24911885 0.073166731 0.00065442313 0.0044942066 265.04599
  alpha_V4 -0.09287934 0.083666872 0.00074833925 0.0051087325 268.21356
 alpha_Age  0.45830900 0.394536219 0.00352883922 0.0310012419 161.96291
  alpha_BT  0.24217000 0.190566444 0.00170447809 0.0163585849 135.70673
 alpha_Trt -0.75931393 0.397734236 0.00355744316 0.0337796459 138.63592
alpha_Base  0.91104974 0.135354470 0.00121064718 0.0111438503 147.52807
    alpha0 -1.35617079 1.313240197 0.01174597740 0.1021568814 165.25442

Quantiles:
               2.5%        25.0%       50.0%        75.0%        97.5%
      s2_b  0.07155807  0.112591385  0.13626498  0.158032604  0.193715882
     s2_b1  0.13817484  0.197135457  0.23761145  0.289655043  0.422805121
```

```
  alpha_V4 -0.25504531 -0.148157017 -0.09313603 -0.036681431  0.072099011
 alpha_Age -0.19666987  0.176356196  0.41608686  0.696647899  1.305075377
  alpha_BT -0.09025014  0.108102968  0.22656174  0.358354280  0.657804969
 alpha_Trt -1.63682212 -1.011390894 -0.75653998 -0.480870874 -0.016113397
alpha_Base  0.66318818  0.817700638  0.90268210  0.997417378  1.200619714
    alpha0 -4.16888778 -2.157932918 -1.26343143 -0.436226488  0.866195785
```

Blocker: Random Effects Meta-Analysis of Clinical Trials

An example from OpenBUGS *[44]* and Carlin *[15]* 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 = Dict{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,
    (mu, nc, N) ->
      begin
        pc = invlogit.(mu)
        UnivariateDistribution[Binomial(nc[i], pc[i]) for i in 1:N]
      end,
    false
  ),

  rt = Stochastic(1,
    (mu, delta, nt, N) ->
      begin
        pt = invlogit.(mu + delta)
        UnivariateDistribution[Binomial(nt[i], pt[i]) for i in 1:N]
      end,
    false
  ),

  mu = Stochastic(1,
    () -> Normal(0, 1000),
    false
  ),

  delta = Stochastic(1,
    (d, s2) -> Normal(d, sqrt(s2)),
    false
  ),

  delta_new = Stochastic(
    (d, s2) -> Normal(d, sqrt(s2))
  ),

  d = Stochastic(
    () -> Normal(0, 1000)
  ),

  s2 = Stochastic(
    () -> InverseGamma(0.001, 0.001)
  )

)


## Initial Values
inits = [
  Dict(:rc => blocker[:rc], :rt => blocker[:rt], :d => 0, :delta_new => 0,
       :s2 => 1, :mu => zeros(blocker[:N]), :delta => zeros(blocker[:N])),
  Dict(: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, 0.1),
          AMWG([:delta, :delta_new], 0.1),
          Slice([:d, :s2], 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
       s2  0.01822186 0.021121265 0.00024388736 0.0014150714 222.78358
        d -0.25563567 0.061841945 0.00071408927 0.0040205781 236.58613
delta_new -0.25005767 0.150325282 0.00173580684 0.0050219145 896.03592

Quantiles:
              2.5%         25.0%         50.0%         75.0%       97.5%
       s2  0.0006855452  0.0041648765  0.0107615659  0.024442084  0.07735715
        d -0.3734122953 -0.2959169814 -0.2581848849 -0.218341380 -0.12842580
delta_new -0.5385405488 -0.3279958446 -0.2557849252 -0.177588413  0.07986060
```

Oxford: Smooth Fit to Log-Odds Ratios

An example from OpenBUGS *[44]* and Breslow and Clayton *[10]* 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 = Dict{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, -1, -1, -1, -1, -1, -1, -1, -1, -1, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 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,
    (mu, n0, K) ->
      begin
        p = invlogit.(mu)
        UnivariateDistribution[Binomial(n0[i], p[i]) for i in 1:K]
      end,
    false
  ),
```

```
  r1 = Stochastic(1,
    (mu, alpha, beta1, beta2, year, b, n1, K) ->
      UnivariateDistribution[
        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,
    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 = [
  Dict(:r0 => oxford[:r0], :r1 => oxford[:r1], :alpha => 0, :beta1 => 0,
       :beta2 => 0, :s2 => 1, :b => zeros(oxford[:K]),
       :mu => zeros(oxford[:K])),
  Dict(: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], 1.0),
          Slice(:s2, 1.0),
          Slice(:mu, 1.0),
```

```
          Slice(:b, 1.0)]
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
beta2  0.005477119 0.0035675748 0.00003567575 0.00033192987 115.519023
beta1 -0.043336269 0.0161754258 0.00016175426 0.00133361554 147.112695
alpha  0.565784774 0.0630050896 0.00063005090 0.00468384860 180.944576
   s2  0.026238992 0.0307989154 0.00030798915 0.00302056007 103.967091

Quantiles:
           2.5%         25.0%         50.0%         75.0%         97.5%
beta2 -0.0010499046  0.0028489198  0.0056500394  0.0077473623  0.013630865
beta1 -0.0745152363 -0.0543180318 -0.0434425931 -0.0321216097 -0.009920787
alpha  0.4438257884  0.5238801187  0.5675039159  0.6051427125  0.695968063
   s2  0.0007134423  0.0033352655  0.0146737037  0.0397132522  0.118202258
```

LSAT: Item Response

An example from OpenBUGS *[44]* and Boch and Lieberman *[6]* 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 = Dict{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,
    (beta, theta, alpha, N, T) ->
      UnivariateDistribution[
        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,
    alpha -> alpha - mean(alpha)
  ),

  beta = Stochastic(
    () -> Truncated(Flat(), 0, Inf)
  )

)


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


## Sampling Scheme
scheme = [AMWG(:alpha, 0.1),
          Slice(:beta, 1.0),
          Slice(:theta, 0.5)]
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
beta  0.80404469 0.072990202 0.00084281825 0.0067491110  116.95963
a[1] -1.26236241 0.104040037 0.00120135087 0.0025355922 1683.61261
a[2]  0.48004293 0.069256031 0.00079969977 0.0013701533 2554.91753
a[3]  1.24206491 0.068338749 0.00078910790 0.0018426724 1375.42777
a[4]  0.16982672 0.072942222 0.00084226422 0.0012659899 3319.68982
a[5] -0.62957215 0.086601562 0.00099998871 0.0018787409 2124.79816

Quantiles:
         2.5%         25.0%        50.0%        75.0%        97.5%
beta  0.678005795  0.75195190  0.79754709  0.85100547  0.96030934
a[1] -1.471693755 -1.33040793 -1.25998457 -1.19317801 -1.06168159
a[2]  0.347262397  0.43161040  0.48023957  0.52718291  0.61527668
a[3]  1.106413529  1.19669095  1.24105794  1.28858225  1.37451854
a[4]  0.023253916  0.11970853  0.17099598  0.21998896  0.30858397
a[5] -0.799988061 -0.68755932 -0.63052599 -0.57168504 -0.46028931
```

Bones: Latent Trait Model for Multiple Ordered Categorical Responses

An example from OpenBUGS *[44]*, Roche *et al.* *[82]*, and Thissen *[93]* 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 = Dict{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(
     [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,2,1,1,2,1,1,
      2,1,1,1,2,2,1,1,1,1,1,1,1,1,1,1,1,1,3,1,1,1,1,1,1,1,1,3,1,1,2,1,1,
      2,1,1,1,2,2,1,1,1,1,1,1,1,1,1,1,1,1,3,1,1,1,1,1,1,1,1,4,3,3,3,1,1,
      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,
      2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,3,3,1,NaN,2,1,3,2,5,5,5,5,5,5,
      2,2,2,2,2,2,2,2,2,2,NaN,2,2,2,2,2,2,2,3,3,3,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,
    (ncat, delta, theta, gamma, nChild, nInd) ->
      begin
        p = Array{Float64}(5)
        UnivariateDistribution[
          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 = [
  Dict(:grade => bones[:grade], :theta => [0.5,1,2,3,5,6,7,8,9,12,13,16,18]),
  Dict(:grade => bones[:grade], :theta => [1,2,3,4,5,6,7,8,9,10,11,12,13])
]


## Sampling Scheme
scheme = [MISS(:grade),
          AMWG(:theta, 0.1)]
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 0.20640874 0.0023834028 0.0039448110 2737.81276
 theta[2]  1.37861692 0.25824308 0.0029819342 0.0058663965 1937.82503
 theta[3]  2.35227822 0.27998526 0.0032329913 0.0067161153 1737.93707
 theta[4]  2.90165730 0.29713320 0.0034309987 0.0078730921 1424.33353
 theta[5]  5.54427283 0.50242324 0.0058014839 0.0169090038  882.88350
 theta[6]  6.70804782 0.57206890 0.0066056827 0.0221532973  666.83738
 theta[7]  6.49138381 0.60154625 0.0069460578 0.0219158412  753.39330
 theta[8]  8.93701249 0.73636136 0.0085027686 0.0336199950  479.71875
 theta[9]  9.03585289 0.65172497 0.0075254717 0.0233182299  781.15561
theta[10] 11.93125529 0.69360918 0.0080091090 0.0282955741  600.88678
theta[11] 11.53686992 0.92271657 0.0106546132 0.0493587234  349.46912
theta[12] 15.81482824 0.54261736 0.0062656056 0.0210976666  661.48275
theta[13] 16.93028146 0.72458739 0.0083668145 0.0323302069  502.30161

Quantiles:
               2.5%       25.0%       50.0%       75.0%       97.5%
 theta[1] -0.11215555  0.19557824  0.33881555  0.45840506  0.7174563
 theta[2]  0.91705346  1.19969433  1.36116575  1.53751273  1.9466119
 theta[3]  1.78287586  2.17136780  2.35623189  2.53035766  2.9211580
 theta[4]  2.32940825  2.69621746  2.89121336  3.10758151  3.4945343
 theta[5]  4.59142954  5.20543314  5.53392246  5.86525435  6.5867245
 theta[6]  5.56649066  6.30983797  6.70666338  7.09569168  7.8229872
```

```
 theta[7]  5.38663728  6.07628064  6.46533033  6.88636840  7.7051374
 theta[8]  7.47304526  8.43125608  8.96072241  9.45344704 10.2856733
 theta[9]  7.80477915  8.60559136  9.01498109  9.46962522 10.3024722
theta[10] 10.64129157 11.48379528 11.89611699 12.37737647 13.3873043
theta[11]  9.83558611 10.88717498 11.49029895 12.15757004 13.4263451
theta[12] 14.79250437 15.45889470 15.79840132 16.15824313 16.9593310
theta[13] 15.61843069 16.42289725 16.90719268 17.41900248 18.3895761
```

Inhalers: Ordered Categorical Data

An example from OpenBUGS *[44]* and Ezzet and Whitehead *[25]* 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 = Dict{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,
    (a1, a2, a3, mu, group, b, N, T) ->
      begin
        a = Float64[a1, a2, a3]
        UnivariateDistribution[
          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,
    (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,
    s2 -> Normal(0, sqrt(s2)),
    false
  ),

  a1 = Stochastic(
    a2 -> Truncated(Flat(), -1000, a2)
  ),

  a2 = Stochastic(
    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 = [
  Dict(:response => inhalers[:response], :beta => 0, :pi => 0, :kappa => 0,
       :a1 => 2, :a2 => 3, :a3 => 4, :s2 => 1, :b => zeros(inhalers[:N])),
  Dict(: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, 0.1),
          Slice([:a1, :a2, :a3], 2.0),
          Slice([:beta, :pi, :kappa, :s2], 1.0, Univariate)]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, inhalers, inits, 5000, burnin=1000, thin=2, chains=2)
describe(sim)
```

Results

Mice: Weibull Regression

An example from OpenBUGS *[44]*, Grieve *[42]*, and Dellaportas and Smith *[22]* 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 = Dict{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,
    (r, beta, tcensor, M, N) ->
      UnivariateDistribution[
        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,
    (beta, r) -> exp.(-beta / r) * log(2)^(1 / r)
  ),

  veh_control = Logical(
    beta -> beta[2] - beta[1]
  ),

  test_sub = Logical(
    beta -> beta[3] - beta[1]
  ),

  pos_control = Logical(
    beta -> beta[4] - beta[1]
  )

)


## Initial Values
inits = [
  Dict(:t => mice[:t], :beta => fill(-1, mice[:M]), :r => 1.0),
  Dict(:t => mice[:t], :beta => fill(-2, mice[:M]), :r => 1.0)
]


## Sampling Scheme
scheme = [MISS(:t),
          Slice(:beta, 1.0, Univariate),
          Slice(:r, 0.25)]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, mice, inits, 20000, burnin=2500, thin=2, chains=2)
describe(sim)
```

Results

Leuk: Cox Regression

An example from OpenBUGS *[44]* and Ezzet and Whitehead *[27]* 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 *[17]* 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 = Dict{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{Int}(N, T)
leuk[:dN] = Array{Int}(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,
    (Y, beta, Z, dL0, N, T) ->
      UnivariateDistribution[
        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,
    (c, r, t) -> c * r * (t[2:end] - t[1:(end - 1)]),
    false
  ),

  dL0 = Stochastic(1,
    (mu, c, T) -> UnivariateDistribution[Gamma(mu[j], 1 / c) for j in 1:T],
    false
  ),

  beta = Stochastic(
    () -> Normal(0, 1000)
  ),

  S0 = Logical(1,
    dL0 -> exp.(-cumsum(dL0)),
    false
  ),

  S_treat = Logical(1,
    (S0, beta) -> S0.^exp(-0.5 * beta)
  ),

  S_placebo = Logical(1,
    (S0, beta) -> S0.^exp(0.5 * beta)
  )

)


## Initial Values
inits = [
  Dict(:dN => leuk[:dN], :beta => 0, :dL0 => fill(1, leuk[:T])),
  Dict(:dN => leuk[:dN], :beta => 1, :dL0 => fill(2, leuk[:T]))
]


## Sampling Scheme
scheme = [AMWG(:dL0, 0.1),
          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
         beta 1.552064427 0.424977799 0.00490722093 0.0121343026 1226.59967
   S_treat[1] 0.983021837 0.014032858 0.00016203749 0.0006444873  474.09293
   S_treat[2] 0.966246426 0.020819923 0.00024040776 0.0009762155  454.84854
   S_treat[3] 0.956207131 0.024534430 0.00028329919 0.0011874261  426.91239
   S_treat[4] 0.936432950 0.031636973 0.00036531230 0.0015140463  436.62796
   S_treat[5] 0.913982241 0.037982886 0.00043858859 0.0017354084  479.04081
   S_treat[6] 0.879372323 0.047653044 0.00055024996 0.0021437962  494.09936
   S_treat[7] 0.867528817 0.051602796 0.00059585777 0.0024545959  441.96358
   S_treat[8] 0.821750962 0.064689997 0.00074697574 0.0030256573  457.12479
   S_treat[9] 0.805557685 0.068612901 0.00079227353 0.0032183437  454.51343
  S_treat[10] 0.771831536 0.076942474 0.00088845517 0.0035776788  462.51903
  S_treat[11] 0.735657019 0.085534730 0.00098766999 0.0040566359  444.58306
  S_treat[12] 0.712600209 0.089598298 0.00103459203 0.0040292944  494.47179
  S_treat[13] 0.691135357 0.094685927 0.00109333891 0.0044879053  445.12656
  S_treat[14] 0.664423491 0.098857036 0.00114150273 0.0046673584  448.61405
  S_treat[15] 0.636423003 0.102857125 0.00118769178 0.0047872478  461.63310
  S_treat[16] 0.565616003 0.112893079 0.00130357699 0.0049341458  523.49276
  S_treat[17] 0.471034334 0.120102602 0.00138682539 0.0050776645  559.47003
 S_placebo[1] 0.927789861 0.050170096 0.00057931437 0.0023120443  470.86627
 S_placebo[2] 0.859431833 0.067291385 0.00077701399 0.0030529117  485.83685
 S_placebo[3] 0.820804570 0.074342778 0.00085843646 0.0034349005  468.43490
 S_placebo[4] 0.748344200 0.085488930 0.00098714114 0.0038433955  494.75434
 S_placebo[5] 0.671088723 0.090818803 0.00104868521 0.0036921538  605.05098
 S_placebo[6] 0.564655613 0.097783399 0.00112910544 0.0038182815  655.83466
 S_placebo[7] 0.532173306 0.099628193 0.00115040728 0.0042364979  553.03234
 S_placebo[8] 0.418874416 0.097451300 0.00112527068 0.0038432054  642.96611
 S_placebo[9] 0.383210551 0.095687796 0.00110490749 0.0036248738  696.83078
S_placebo[10] 0.317122087 0.091201496 0.00105310416 0.0031779306  823.59766
S_placebo[11] 0.256739191 0.086400289 0.00099766461 0.0029358329  866.09938
S_placebo[12] 0.223014153 0.082260904 0.00094986710 0.0026813211  941.21603
S_placebo[13] 0.195547225 0.079430544 0.00091718492 0.0029106845  744.70591
S_placebo[14] 0.164855445 0.074276591 0.00085767220 0.0026970503  758.44802
S_placebo[15] 0.137032754 0.068763778 0.00079401571 0.0026184621  689.64703
S_placebo[16] 0.083796000 0.054748991 0.00063218689 0.0020660044  702.24679
S_placebo[17] 0.040920336 0.037737842 0.00043575906 0.0012818385  866.73735

Quantiles:
                  2.5%        25.0%       50.0%       75.0%       97.5%
         beta 0.7524244645 1.259092877 1.540715015 1.829373993 2.41862572
   S_treat[1] 0.9462141591 0.977211258 0.986955387 0.992864719 0.99836877
   S_treat[2] 0.9150951603 0.955732194 0.970764331 0.981528195 0.99312428
   S_treat[3] 0.8974993157 0.943194903 0.960671242 0.974341293 0.98958205
   S_treat[4] 0.8599068641 0.919023530 0.941699242 0.959578177 0.98196246
   S_treat[5] 0.8239755135 0.892499135 0.919841738 0.941506275 0.97166964
   S_treat[6] 0.7738755125 0.850772834 0.885177533 0.913944652 0.95521105
   S_treat[7] 0.7531822416 0.836159276 0.873267366 0.904856224 0.95037821
   S_treat[8] 0.6801713643 0.780967298 0.828861810 0.868789540 0.92883388
   S_treat[9] 0.6555328459 0.762836917 0.813277192 0.854664506 0.91930265
  S_treat[10] 0.6026848684 0.723608005 0.779448002 0.826762737 0.90169913
```

```
  S_treat[11] 0.5503082957 0.681881638 0.743447395 0.797449392 0.87990722
  S_treat[12] 0.5233565529 0.654970518 0.720927823 0.777608202 0.86466162
  S_treat[13] 0.4922868942 0.631085044 0.699537585 0.758920006 0.85262814
  S_treat[14] 0.4597297883 0.598392933 0.670665550 0.735415731 0.83825032
  S_treat[15] 0.4269750729 0.566537132 0.641228870 0.709931594 0.82208295
  S_treat[16] 0.3438735907 0.487880437 0.566936069 0.644918756 0.77749139
  S_treat[17] 0.2466734959 0.383395600 0.469857377 0.555064846 0.70280881
 S_placebo[1] 0.7997904171 0.903185030 0.939595572 0.964924422 0.99089079
 S_placebo[2] 0.7055958749 0.819313313 0.869352795 0.908385797 0.96318697
 S_placebo[3] 0.6563988800 0.773124226 0.828159645 0.876023887 0.94337131
 S_placebo[4] 0.5660793194 0.692444549 0.754818343 0.810412223 0.89462480
 S_placebo[5] 0.4851605247 0.609481976 0.675431460 0.736195314 0.83615527
 S_placebo[6] 0.3696872931 0.497149446 0.566089714 0.632288659 0.74932879
 S_placebo[7] 0.3402856402 0.462939149 0.532290556 0.601897323 0.72484128
 S_placebo[8] 0.2379614336 0.349405859 0.415075601 0.483568964 0.62050327
 S_placebo[9] 0.2091293870 0.314947858 0.378301304 0.447909630 0.58210955
S_placebo[10] 0.1562465305 0.250265102 0.311673305 0.378896633 0.51089391
S_placebo[11] 0.1092657207 0.193262012 0.249637651 0.312589668 0.44360059
S_placebo[12] 0.0845659713 0.162032458 0.216266931 0.276356661 0.40448425
S_placebo[13] 0.0650227582 0.137218626 0.187654643 0.246139543 0.37311156
S_placebo[14] 0.0479431220 0.110429658 0.156032281 0.209723502 0.33168432
S_placebo[15] 0.0342784539 0.086171750 0.127242826 0.176712122 0.29500763
S_placebo[16] 0.0125569906 0.042952635 0.072131767 0.112032446 0.22046612
S_placebo[17] 0.0021678381 0.013959709 0.029822826 0.055822282 0.14078639
```

Volume II

Birats: A Bivariate Normal Hierarchical Model

An example from OpenBUGS *[44]* and section 6 of Gelfand *et al.* *[29]* concerning 30 rats whose weights were measured at each of five consecutive weeks.

Model

Weights are modeled as

$$
\begin{aligned}
Y_{i,j} &\sim \text{Normal}\left(\mu_{i,j}, \sigma_C\right) \qquad i = 1, \ldots, 30; j = 1, \ldots, 5 \\
\mu_{i,j} &= \beta_{1,i} + \beta_{2,i} x_j \\
\boldsymbol{\beta}_i &\sim \text{MvNormal}\left(\boldsymbol{\mu}_\beta, \boldsymbol{\Sigma}\right) \\
\boldsymbol{\mu}_\beta &\sim \text{MvNormal}\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1e6 & 0\\0 & 1e6\end{bmatrix}\right) \\
\boldsymbol{\Sigma} &\sim \text{InverseWishart}\left(2, \begin{bmatrix}200 & 0\\0 & 0.2\end{bmatrix}\right) \\
\sigma_C^2 &\sim \text{InverseGamma}(0.001, 0.001),
\end{aligned}
$$

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
birats = Dict{Symbol, Any}(
  :N => 30, :T => 5,
  :x => [8.0, 15.0, 22.0, 29.0, 36.0],
  :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],
  :mean => [0.0, 0.0],
  :var => [1.0e6 0.0
           0.0   1.0e6],
  :Omega => [200.0 0.0
               0.0 0.2]
)


## Model Specification
model = Model(

  Y = Stochastic(2,
    (beta, x, sigmaC, N, T) ->
      UnivariateDistribution[
        Normal(beta[i, 1] + beta[i, 2] * x[j], sigmaC)
        for i in 1:N, j in 1:T
      ],
    false
  ),

  beta = Stochastic(2,
    (mu_beta, Sigma, N) ->
```

```
      MultivariateDistribution[
        MvNormal(mu_beta, Sigma)
        for i in 1:N
      ],
    false
  ),

  mu_beta = Stochastic(1,
    (mean, var) -> MvNormal(mean, var)
  ),

  Sigma = Stochastic(2,
    Omega -> InverseWishart(2, Omega),
    false
  ),

  sigma2C = Stochastic(
    () -> InverseGamma(0.001, 0.001),
    false
  ),

  sigmaC = Logical(
    sigma2C -> sqrt(sigma2C)
  )

)


## Initial Values
inits = [
  Dict(:Y => birats[:Y], :beta => repmat([100 6], birats[:N], 1),
       :mu_beta => [0, 0], :Sigma => eye(2), :sigma2C => 1.0),
  Dict(:Y => birats[:Y], :beta => repmat([50 3], birats[:N], 1),
       :mu_beta => [10, 10], :Sigma => 0.3 * eye(2), :sigma2C => 10.0)
]


## Sampling Scheme
scheme = [AMWG([:beta, :mu_beta], repmat([10.0, 1.0], birats[:N] + 1)),
          AMWG(:Sigma, 1.0),
          Slice(:sigma2C, 10.0)]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, birats, 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[1] 106.7046188 2.258246468 0.0260759841 0.081338282 770.81949
mu_beta[2]   6.1804557 0.104047928 0.0012014420 0.004102793 643.14317
    sigmaC   6.1431758 0.460583341 0.0053183583 0.021005830 480.76933

Quantiles:
               2.5%       25.0%       50.0%       75.0%      97.5%
mu_beta[1] 102.3595659 105.2252185 106.6914834 108.164852 111.2001520
mu_beta[2]   5.9720904   6.1130035   6.1817455   6.248025   6.3848798
    sigmaC   5.3167022   5.8123935   6.1206859   6.445419   7.0971249
```

Jaws: Repeated Measures Analysis of Variance

An example from OpenBUGS *[44]* and Elston and Grizzle *[24]* 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, bone heights are arranged into a 1-dimensional vector on which a *Block-Diagonal Multivariate Normal Distribution* is specified. Also not that since $\boldsymbol{\Sigma}$ is a covariance matrix, it is symmetric with `M * (M + 1) / 2` unique (upper or lower triangular) parameters, where `M` is the matrix dimension.

Analysis Program

```
using Mamba

## Data
jaws = Dict{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,
    (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,
    M -> InverseWishart(4.0, eye(M))
  )

)


## Initial Values
inits = [
  Dict(:y => jaws[:y], :beta0 => 40, :beta1 => 1, :Sigma => eye(M)),
  Dict(:y => jaws[:y], :beta0 => 10, :beta1 => 10, :Sigma => eye(M))
]


## Sampling Scheme
scheme = [Slice([:beta0, :beta1], [10, 1]),
          AMWG(:Sigma, 0.1)]
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
Sigma[1,1]  6.7915801 2.0232463 0.0233624358 0.1421847433 202.48437
Sigma[1,2]  6.5982624 1.9670001 0.0227129612 0.1469366529 179.20433
Sigma[1,3]  6.1775526 1.9084389 0.0220367541 0.1532226770 155.13523
Sigma[1,4]  5.9477070 1.9358258 0.0223529913 0.1545185214 156.95367
Sigma[2,2]  6.9308723 2.0236387 0.0233669666 0.1531630007 174.56542
Sigma[2,3]  6.6005767 1.9885583 0.0229618936 0.1600864592 154.30055
Sigma[2,4]  6.3803028 2.0196017 0.0233203515 0.1612393116 156.88795
Sigma[3,3]  7.4564163 2.1925641 0.0253175499 0.1705734202 165.22733
Sigma[3,4]  7.4518620 2.2712194 0.0262257824 0.1733769737 171.60713
Sigma[4,4]  8.0594440 2.4746352 0.0285746264 0.1784057891 192.39975
     beta1  1.8742617 0.2272166 0.0026236712 0.0071954415 997.16079
     beta0 33.6379701 1.9912509 0.0229929845 0.0632742554 990.37090

Quantiles:
              2.5%       25.0%      50.0%      75.0%      97.5%
Sigma[1,1]  3.7202164  5.3419070  6.5046777  7.8684049 11.5279247
Sigma[1,2]  3.5674344  5.2009878  6.3564397  7.6419462 11.2720602
Sigma[1,3]  3.2043099  4.8527075  5.9476859  7.1929746 10.8427648
Sigma[1,4]  2.9143241  4.5808041  5.6958961  6.9962164 10.6253935
Sigma[2,2]  3.7936234  5.4940524  6.6730872  8.0151463 11.6796110
Sigma[2,3]  3.4721419  5.2183567  6.3620683  7.6617912 11.4419940
Sigma[2,4]  3.2133129  4.9659531  6.1310937  7.4443619 11.3714037
Sigma[3,3]  4.1213458  5.9139585  7.1780478  8.6551856 12.8617596
Sigma[3,4]  4.0756709  5.8561719  7.1240011  8.7006336 13.0597624
Sigma[4,4]  4.4482953  6.3090779  7.6484712  9.4043857 14.0451233
     beta1  1.4349627  1.7279142  1.8707215  2.0159440  2.3445976
     beta0 29.4960557 32.3922780 33.6327451 34.9577696 37.4067853
```

Eyes: Normal Mixture Model

An example from OpenBUGS *[44]*, Bowmaker *[7]*, and Robert *[77]* concerning 48 peak sensitivity wavelength measurements taken on a set of monkey's eyes.

Model

Measurements are modelled as the mixture distribution

$$
\begin{aligned}
y_i &\sim \text{Normal}(\lambda_{T_i}, \sigma) \qquad i = 1, \ldots, 48 \\
T_i &\sim \text{Categorical}(P) \\
\lambda_1 &\sim \text{Normal}(0, 1000) \\
\lambda_2 &= \lambda_1 + \theta \\
\theta &\sim \text{Uniform}(0, 1000) \\
\sigma^2 &\sim \text{InverseGamma}(0.001, 0.001) \\
P &= \text{Dirichlet}(1, 1)
\end{aligned}
$$

where y_i is the measurement on monkey i.

Analysis Program

```
using Mamba

## Data
eyes = Dict{Symbol, Any}(
  :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,
    (lambda, T, s2, N) ->
      begin
        sigma = sqrt(s2)
        UnivariateDistribution[
          begin
            mu = lambda[Int(T[i])]
            Normal(mu, sigma)
          end
          for i in 1:N
        ]
      end,
    false
  ),

  T = Stochastic(1,
    (P, N) -> UnivariateDistribution[Categorical(P) for i in 1:N],
    false
  ),

  P = Stochastic(1,
    alpha -> Dirichlet(alpha)
  ),

  lambda = Logical(1,
    (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 = [
  Dict(:y => eyes[:y], :T => fill(1, eyes[:N]), :P => [0.5, 0.5],
       :lambda0 => 535, :theta => 5, :s2 => 10),
  Dict(:y => eyes[:y], :T => fill(1, eyes[:N]), :P => [0.5, 0.5],
       :lambda0 => 550, :theta => 1, :s2 => 1)
]


## Sampling Scheme
scheme = [DGS(:T),
          Slice([:lambda0, :theta], [5.0, 1.0]),
          Slice(:s2, 2.0, transform=true),
          SliceSimplex(:P, scale=0.75)]
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
     P[1]   0.60357102 0.08379221 0.0009675491 0.0013077559 3750.00000
     P[2]   0.39642898 0.08379221 0.0009675491 0.0013077559 3750.00000
       s2  14.45234459 4.96689604 0.0573527753 0.1853970211  717.73584
lambda[1] 536.75250337 0.88484533 0.0102173137 0.0304232594  845.90821
lambda[2] 548.98693469 1.18938418 0.0137338256 0.0625489045  361.58042

Quantiles:
              2.5%         25.0%        50.0%        75.0%        97.5%
     P[1]   0.43576584   0.54842468   0.60395494   0.66066383   0.76552882
     P[2]   0.23447118   0.33933617   0.39604506   0.45157532   0.56423416
       s2   8.55745263  11.37700782  13.39622105  16.26405571  27.08863166
lambda[1] 535.08951002 536.16891895 536.73114152 537.30710397 538.61179506
lambda[2] 546.57859195 548.24188698 548.97732892 549.74377421 551.38780148
```

The BUGS Book

Asthma: State Transitions in a Clinical Trial

An example from the BUGS book *[56]* concerning transitions between five clinical states in a randomized trial of treatments (seretide and fluticasone) for asthma.

Model

A discrete-time Markov model (equivalent to independent multinomial models) is fit with probability vector $\boldsymbol{q}_i$ governing the state in the following week conditionally on the current state. Possible states are successfully treated, unsuccessfully treated, hospital-managed exacerbation, primary care-managed exacerbation, and treatment failure. The fifth state, treatment failure, is absorbing (patients cannot move out of it). The model is given by

$$\begin{aligned} y_{ij} &\sim \text{Multinomial}(M_i, (q_{i1}, \ldots, q_{i5})) \qquad i = 1, \ldots, 3 \\ \boldsymbol{q}_i &\sim \text{Dirichlet}(1, \ldots, 1) \end{aligned}$$

where y_{ij} is the number of transitions from state i to j

Analysis Program

```
using Mamba

## Data
asthma = Dict{Symbol, Any}(
  :y =>
    [210  60  0  1   1
     88 641  0  4  13
     1    0  0  0   1],
  :M =>
    [272, 746, 2]
)


## Model Specification
model = Model(

  y = Stochastic(2,
    (M, q) ->
      MultivariateDistribution[
        Multinomial(M[i], vec(q[i, :]))
        for i in 1:length(M)
      ],
    false
  ),

  q = Stochastic(2,
    M ->
      MultivariateDistribution[
        Dirichlet(ones(5))
        for i in 1:length(M)
      ],
    true
  )
```

```
)


## Initial Values
inits = [
  Dict{Symbol, Any}(
    :y => asthma[:y],
    :q => vcat([rand(Dirichlet(ones(5)))' for i in 1:3]...)
  )
  for i in 1:3
]


## Sampling Scheme
scheme = [SliceSimplex(:q)]
setsamplers!(model, scheme)


## MCMC Simulations
sim = mcmc(model, asthma, inits, 10000, burnin=2500, thin=2, chains=3)
describe(sim)
```

Results

```
Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2,3
Samples per chain = 3750

Empirical Posterior Estimates:
           Mean           SD         Naive SE          MCSE          ESS
q[1,1] 0.7615754849 0.0272201055 0.000256633616 0.001595328676  291.12484
q[1,2] 0.2204851131 0.0265594084 0.000250404504 0.001578283076  283.18288
q[1,3] 0.0034735444 0.0037556875 0.000035408962 0.000069146577 2950.10543
q[1,4] 0.0072778962 0.0053705520 0.000050634050 0.000107382077 2501.34876
q[1,5] 0.0071879614 0.0053672180 0.000050602617 0.000112487762 2276.60625
q[2,1] 0.1191655126 0.0121038180 0.000114115890 0.000530342898  520.87229
q[2,2] 0.8543825941 0.0130973639 0.000123483131 0.000564705907  537.92673
q[2,3] 0.0012103544 0.0013675802 0.000012893670 0.000023654051 3342.67820
q[2,4] 0.0066582050 0.0030978699 0.000029206998 0.000059253193 2733.39774
q[2,5] 0.0185833339 0.0051526335 0.000048579495 0.000120299727 1834.54864
q[3,1] 0.2936564126 0.1740764923 0.001641208908 0.005020530045 1202.21215
q[3,2] 0.1394405572 0.1262073820 0.001189894609 0.002598964624 2358.13563
q[3,3] 0.1424463856 0.1308387051 0.001233559141 0.002834635296 2130.48310
q[3,4] 0.1417886606 0.1328770997 0.001252777310 0.003532010313 1415.32558
q[3,5] 0.2826679840 0.1709210331 0.001611458954 0.004864906134 1234.36071

Quantiles:
            2.5%           25.0%          50.0%         75.0%         97.5%
q[1,1] 0.706775811059 0.74365894715 0.7617692483 0.7804747794 0.8125428018
q[1,2] 0.169898760444 0.20150542266 0.2199140987 0.2386805795 0.2733246918
q[1,3] 0.000059528767 0.00083424954 0.0022254753 0.0048346451 0.0138405267
q[1,4] 0.000716682148 0.00327126033 0.0060441127 0.0099507177 0.0205431907
q[1,5] 0.000740554613 0.00319616653 0.0058781978 0.0098161771 0.0209389022
q[2,1] 0.096435484007 0.11060974070 0.1187015139 0.1271183450 0.1443574095
```

```
q[2,2] 0.828193845237 0.84541382625 0.8550642143 0.8634806977 0.8792757738
q[2,3] 0.000018088918 0.00028620290 0.0007503763 0.0016511135 0.0049571810
q[2,4] 0.002033379513 0.00434074296 0.0061948558 0.0084704327 0.0138428435
q[2,5] 0.010047515668 0.01486876182 0.0180566543 0.0217580995 0.0303924987
q[3,1] 0.035105559792 0.15629556479 0.2700039867 0.4070428885 0.6826474093
q[3,2] 0.002527526439 0.03939967503 0.1024828775 0.2077906692 0.4616490169
q[3,3] 0.002612503802 0.03901018402 0.1041953555 0.2089759155 0.4719934399
q[3,4] 0.002859691566 0.03827679857 0.1016367373 0.2072463302 0.4865686565
q[3,5] 0.033209396793 0.14610521565 0.2577690743 0.3965462343 0.6524841408
```

2.5.2 Contributed

Additional examples are provided below to further illustrate features of the package.

GK: Approximate Bayesian Computation

Approximate Bayesian Computation (ABC) is often useful when the likelihood is costly to compute. The generalized GK distribution *[76]* is a distribution defined by the inverse of its cumulative distribution function. It is therefore easy to sample from, but difficult to obtain analytical likelihood functions. These properties make the GK distribution well suited for ABC. The following is a simulation study that mirrors *[1]*.

Model

$$\begin{aligned} x_i &\sim \text{GK}\left(A, B, g, k\right) \qquad i = 1, \ldots, 1000 \\ A, B, g, k &\sim \text{Uniform}(0, 10) \end{aligned}$$

Analysis Program

```
using Mamba

@everywhere extensions = quote
  using Distributions
  import Distributions: location, scale, skewness, kurtosis, minimum, maximum,
         quantile

  immutable GK <: ContinuousUnivariateDistribution
    A::Float64
    B::Float64
    g::Float64
    k::Float64
    c::Float64

    function GK(A::Real, B::Real, g::Real, k::Real, c::Real)
      ## check args
      0.0 <= c < 1.0 || throw(ArgumentError("c must be in [0, 1)"))

      ## create distribution
      new(A, B, g, k, c)
    end
```

```
    GK(A::Real, B::Real, g::Real, k::Real) = GK(A, B, g, k, 0.8)
  end

  ## Parameters
  location(d::GK) = d.A
  scale(d::GK) = d.B
  skewness(d::GK) = d.g
  kurtosis(d::GK) = d.k
  asymmetry(d::GK) = d.c

  minimum(d::GK) = -Inf
  maximum(d::GK) = Inf

  function quantile(d::GK, p::Float64)
    z = quantile(Normal(), p)
    z2gk(d, z)
  end

  function z2gk(d::GK, z::Float64)
    term1 = exp(-skewness(d) * z)
    term2 = (1.0 + asymmetry(d) * (1.0 - term1) / (1.0 + term1))
    term3 = (1.0 + z^2)^kurtosis(d)
    location(d) + scale(d) * z * term2 * term3
  end
end

@everywhere eval(extensions)

d = GK(3, 1, 2, 0.5)
x = rand(d, 1000)

allingham = Dict{Symbol, Any}(
  :x => x
)

model = Model(
  x = Stochastic(1, (A, B, g, k) -> GK(A, B, g, k), false),
  A = Stochastic(() -> Uniform(0, 10)),
  B = Stochastic(() -> Uniform(0, 10)),
  g = Stochastic(() -> Uniform(0, 10)),
  k = Stochastic(() -> Uniform(0, 10))
)

inits = [
  Dict{Symbol, Any}(:x => x, :A => 3.5, :B => 0.5,
                    :g => 2.0, :k => 0.5),
  Dict{Symbol, Any}(:x => x, :A => median(x),
                    :B => sqrt(var(x)),
                    :g => 1.0, :k => 1.0),
  Dict{Symbol, Any}(:x => x, :A => median(x),
                    :B => diff(quantile(x, [0.25, 0.75]))[1],
                    :g => mean((x - mean(x)).^3) / (var(x)^(3 / 2)),
                    :k => rand())
]

sigma1 = 0.05
sigma2 = 0.5
```

```
stats = x -> quantile(x, [0.1, 0.25, 0.5, 0.75, 0.9])
epsilon = 0.1

scheme = [ABC([:A, :B, :k],
              sigma1, stats, epsilon, maxdraw=50, decay=0.75, randeps=true),
          ABC(:g, sigma2, stats, epsilon, maxdraw=50, decay=0.75)]
setsamplers!(model, scheme)

sim = mcmc(model, allingham, inits, 10000, burnin=2500, chains=3)
describe(sim)
p = plot(sim)
draw(p, filename="gk")
```

Results

```
Iterations = 2501:10000
Thinning interval = 1
Chains = 1,2,3
Samples per chain = 7500

Empirical Posterior Estimates:
     Mean         SD         Naive SE       MCSE         ESS
k 0.3510874 0.132239097 0.00088159398 0.0082306353  258.13865
A 3.0037486 0.068834777 0.00045889851 0.0018699334 1355.07569
B 1.0575823 0.129916443 0.00086610962 0.0074502510  304.07900
g 2.0259195 0.311756472 0.00207837648 0.0114243024  744.68323

Quantiles:
     2.5%       25.0%      50.0%      75.0%     97.5%
k 0.12902213 0.25821839 0.34355889 0.42948590 0.6759640
A 2.86683504 2.95778075 3.00296541 3.04971216 3.1358843
B 0.80207071 0.96839859 1.05571418 1.14595782 1.3123378
g 1.56533594 1.79945626 1.97432281 2.19565833 2.7212103
```

Line: Approximate Bayesian Computation

A simple example to demonstrate the Approximate Bayesian Computation (ABC) sampler within the MCMC framework, based on the linear regression model defined in the *Tutorial* section. ABC sampling is applied separately to the `:beta` and `:s2` parameter blocks. Different summary statistics are specified to show a range of functions that could be used. More common practice is to use the same data summaries for all ABC-sampled parameters.

Analysis Program

```
using Mamba

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

line[:xmat] = [ones(5) line[:x]]
```

```
## Model Specification
model = Model(
  y = Stochastic(1,
    (xmat, beta, s2) -> MvNormal(xmat * beta, sqrt(s2)),
    false
  ),
  beta = Stochastic(1, () -> MvNormal(2, sqrt(100))),
  s2 = Stochastic(() -> InverseGamma(0.01, 0.01))
)


## 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
]


## Tuning Parameters
scale1 = [0.5, 0.25]
summary1 = identity
eps1 = 0.5

scale2 = 0.5
summary2 = x -> [mean(x); sqrt(var(x))]
eps2 = 0.1


## User-Defined Sampling Scheme
scheme = [
  ABC(:beta, scale1, summary1, eps1, kernel=Normal, maxdraw=100, nsim=3),
  ABC(:s2,   scale2, summary2, eps2, kernel=Epanechnikov, maxdraw=100, nsim=3)
]
setsamplers!(model, scheme)


## MCMC Simulation with Approximate Bayesian Computation
sim = mcmc(model, line, inits, 10000, burnin=1000, chains=3)
describe(sim)
```

Results

```
Iterations = 1001:10000
Thinning interval = 1
Chains = 1,2,3
Samples per chain = 9000

Empirical Posterior Estimates:
           Mean        SD       Naive SE       MCSE       ESS
     s2 1.30743333 1.99877929 0.0121641834 0.083739029 569.73624
```

```
beta[1] 0.72349922 1.03842764 0.0063196694 0.039413390 694.16848
beta[2] 0.77469344 0.31702542 0.0019293553 0.011392989 774.30630

Quantiles:
            2.5%        25.0%      50.0%      75.0%     97.5%
     s2  0.048095084 0.23351203 0.57947788 1.45858829 7.7639321
beta[1] -1.309713807 0.12616636 0.67263204 1.27579373 3.1735176
beta[2]  0.107216316 0.59367961 0.77867235 0.95156086 1.4043715
```

Line: Block-Specific Sampling with AMWG and Slice

An example based on the linear regression model defined in the *Tutorial* section. The program below illustrates use of the stand-alone `amwg!()` and `slice!()` functions to sample different parameter blocks within the same MCMC algorithm.

Analysis Program

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

using Mamba

## Data
data = Dict{Symbol, Any}(
  :x => [1, 2, 3, 4, 5],
  :y => [1, 3, 3, 3, 5]
)

## Log-transformed unnormalized joint posterior for b0, b1, and log(s2)
logf = function(x::DenseVector)
   b0 = x[1]
   b1 = x[2]
   logs2 = x[3]
   (-0.5 * length(data[:y]) - 0.001) * logs2 -
     (0.5 * sum(abs2, data[:y] - b0 - b1 * data[:x]) + 0.001) / exp(logs2) -
     0.5 * b0^2 / 1000 - 0.5 * b1^2 / 1000
end

## Log-transformed unnormalized full conditional densities for the model
## parameters beta and log(s2) defined below in the MCMC simulation
logf_beta(x) = logf([x; logs2])
logf_logs2(x) = logf([beta; x])

## MCMC simulation
n = 10000
burnin = 1000
sim = Chains(n, 3, names = ["b0", "b1", "s2"])
beta = AMWGVariate([0.0, 0.0], 1.0, logf_beta)
logs2 = SliceMultivariate([0.0], 5.0, logf_logs2)
for i in 1:n
```

```
  sample!(beta, adapt = (i <= burnin))
  sample!(logs2)
  sim[i, :, 1] = [beta; exp.(logs2)]
end
describe(sim)
```

Results

```
Iterations = 1:10000
Thinning interval = 1
Chains = 1
Samples per chain = 10000

Empirical Posterior Estimates:
      Mean        SD       Naive SE       MCSE       ESS
b0 0.64401798 0.99315634 0.0099315634 0.060725564  267.4805
b1 0.78985612 0.29790444 0.0029790444 0.017888106  277.3481
s2 1.20785292 2.96033511 0.0296033511 0.062566344 2238.7222

Quantiles:
       2.5%       25.0%      50.0%     75.0%     97.5%
b0 -1.33127385 0.075527035 0.6403226 1.1902679 2.7665517
b1  0.16055439 0.625740352 0.7923275 0.9527029 1.3903861
s2  0.16673189 0.381185645 0.6538295 1.2373814 5.6065938
```

Pollution: Bayesian Variable Selection

Data from McDonald and Schwing *[61]* that include 15 independent variables and a measure of mortality on 60 U.S. metropolitan areas in 1959-1961. Originally, the data were used to illustrate ridge regression (the full covariate matrix has a huge condition number). This dataset was included in a review of Bayesian variable selection techniques by O'Hara et al *[67]*.

Model

$$
\begin{aligned}
y_i &\sim \text{Normal}(\mu_i, \sigma) \qquad i = 1, \ldots, 60 \\
\mu_i &= \alpha + \sum_{j=1}^{15} \theta_j x_{ij} \\
\theta_j &= \beta_j \gamma_j \\
\alpha, \beta_j &\sim \text{Normal}(0, 1000) \\
\gamma_j &\sim \text{Bernoulli}(0.5) \\
\sigma^2 &\sim \text{InverseGamma}(0.0001, 0.0001)
\end{aligned}
$$

Analysis Program

```
using Mamba

## Data
data = [
 36 27 71  8.1 3.34 11.4 81.5 3243  8.8 42.6 11.7  21  15  59  59  921.87
 35 23 72 11.1 3.14 11.0 78.8 4281  3.5 50.7 14.4   8  10  39  57  997.88
 44 29 74 10.4 3.21  9.8 81.6 4260  0.8 39.4 12.4   6   6  33  54  962.35
 47 45 79  6.5 3.41 11.1 77.5 3125 27.1 50.2 20.6  18   8  24  56  982.29
 43 35 77  7.6 3.44  9.6 84.6 6441 24.4 43.7 14.3  43  38 206  55 1071.29
 53 45 80  7.7 3.45 10.2 66.8 3325 38.5 43.1 25.5  30  32  72  54 1030.38
 43 30 74 10.9 3.23 12.1 83.9 4679  3.5 49.2 11.3  21  32  62  56  934.70
 45 30 73  9.3 3.29 10.6 86.0 2140  5.3 40.4 10.5   6   4   4  56  899.53
 36 24 70  9.0 3.31 10.5 83.2 6582  8.1 42.5 12.6  18  12  37  61 1001.90
 36 27 72  9.5 3.36 10.7 79.3 4213  6.7 41.0 13.2  12   7  20  59  912.35
 52 42 79  7.7 3.39  9.6 69.2 2302 22.2 41.3 24.2  18   8  27  56 1017.61
 33 26 76  8.6 3.20 10.9 83.4 6122 16.3 44.9 10.7  88  63 278  58 1024.89
 40 34 77  9.2 3.21 10.2 77.0 4101 13.0 45.7 15.1  26  26 146  57  970.47
 35 28 71  8.8 3.29 11.1 86.3 3042 14.7 44.6 11.4  31  21  64  60  985.95
 37 31 75  8.0 3.26 11.9 78.4 4259 13.1 49.6 13.9  23   9  15  58  958.84
 35 46 85  7.1 3.22 11.8 79.9 1441 14.8 51.2 16.1   1   1   1  54  860.10
 36 30 75  7.5 3.35 11.4 81.9 4029 12.4 44.0 12.0   6   4  16  58  936.23
 15 30 73  8.2 3.15 12.2 84.2 4824  4.7 53.1 12.7  17   8  28  38  871.77
 31 27 74  7.2 3.44 10.8 87.0 4834 15.8 43.5 13.6  52  35 124  59  959.22
 30 24 72  6.5 3.53 10.8 79.5 3694 13.1 33.8 12.4  11   4  11  61  941.18
 31 45 85  7.3 3.22 11.4 80.7 1844 11.5 48.1 18.5   1   1   1  53  891.71
 31 24 72  9.0 3.37 10.9 82.8 3226  5.1 45.2 12.3   5   3  10  61  871.34
 42 40 77  6.1 3.45 10.4 71.8 2269 22.7 41.4 19.5   8   3   5  53  971.12
 43 27 72  9.0 3.25 11.5 87.1 2909  7.2 51.6  9.5   7   3  10  56  887.47
 46 55 84  5.6 3.35 11.4 79.7 2647 21.0 46.9 17.9   6   5   1  59  952.53
 39 29 75  8.7 3.23 11.4 78.6 4412 15.6 46.6 13.2  13   7  33  60  968.67
 35 31 81  9.2 3.10 12.0 78.3 3262 12.6 48.6 13.9   7   4   4  55  919.73
 43 32 74 10.1 3.38  9.5 79.2 3214  2.9 43.7 12.0  11   7  32  54  844.05
 11 53 68  9.2 2.99 12.1 90.6 4700  7.8 48.9 12.3 648 319 130  47  861.83
 30 35 71  8.3 3.37  9.9 77.4 4474 13.1 42.6 17.7  38  37 193  57  989.27
 50 42 82  7.3 3.49 10.4 72.5 3497 36.7 43.3 26.4  15  18  34  59 1006.49
 60 67 82 10.0 2.98 11.5 88.6 4657 13.5 47.3 22.4   3   1   1  60  861.44
 30 20 69  8.8 3.26 11.1 85.4 2934  5.8 44.0  9.4  33  23 125  64  929.15
 25 12 73  9.2 3.28 12.1 83.1 2095  2.0 51.9  9.8  20  11  26  58  857.62
 45 40 80  8.3 3.32 10.1 70.3 2682 21.0 46.1 24.1  17  14  78  56  961.01
 46 30 72 10.2 3.16 11.3 83.2 3327  8.8 45.3 12.2   4   3   8  58  923.23
 54 54 81  7.4 3.36  9.7 72.8 3172 31.4 45.5 24.2  20  17   1  62 1113.16
 42 33 77  9.7 3.03 10.7 83.5 7462 11.3 48.7 12.4  41  26 108  58  994.65
 42 32 76  9.1 3.32 10.5 87.5 6092 17.5 45.3 13.2  29  32 161  54 1015.02
 36 29 72  9.5 3.32 10.6 77.6 3437  8.1 45.5 13.8  45  59 263  56  991.29
 37 38 67 11.3 2.99 12.0 81.5 3387  3.6 50.3 13.5  56  21  44  73  893.99
 42 29 72 10.7 3.19 10.1 79.5 3508  2.2 38.8 15.7   6   4  18  56  938.50
 41 33 77 11.2 3.08  9.6 79.9 4843  2.7 38.6 14.1  11  11  89  54  946.19
 44 39 78  8.2 3.32 11.0 79.9 3768 28.6 49.5 17.5  12   9  48  53 1025.50
 32 25 72 10.9 3.21 11.1 82.5 4355  5.0 46.4 10.8   7   4  18  60  874.28
 34 32 79  9.3 3.23  9.7 76.8 5160 17.2 45.1 15.3  31  15  68  57  953.56
 10 55 70  7.3 3.11 12.1 88.9 3033  5.9 51.0 14.0 144  66  20  61  839.71
 18 48 63  9.2 2.92 12.2 87.7 4253 13.7 51.2 12.0 311 171  86  71  911.70
 13 49 68  7.0 3.36 12.2 90.7 2702  3.0 51.9  9.7 105  32   3  71  790.73
 35 40 64  9.6 3.02 12.2 82.5 3626  5.7 54.3 10.1  20   7  20  72  899.26
 45 28 74 10.6 3.21 11.1 82.6 1883  3.4 41.9 12.3   5   4  20  56  904.16
 38 24 72  9.8 3.34 11.4 78.0 4923  3.8 50.5 11.1   8   5  25  61  950.67
 31 26 73  9.3 3.22 10.7 81.3 3249  9.5 43.9 13.6  11   7  25  59  972.46
 40 23 71 11.3 3.28 10.3 73.8 1671  2.5 47.4 13.5   5   2  11  60  912.20
```

```
 41 37 78  6.2 3.25 12.3 89.5 5308 25.9 59.7 10.3  65  28 102  52  967.80
 28 32 81  7.0 3.27 12.1 81.0 3665  7.5 51.6 13.2   4   2   1  54  823.76
 45 33 76  7.7 3.39 11.3 82.2 3152 12.1 47.3 10.9  14  11  42  56 1003.50
 45 24 70 11.8 3.25 11.1 79.8 3678  1.0 44.8 14.0   7   3   8  56  895.70
 42 33 76  9.7 3.22  9.0 76.2 9699  4.8 42.2 14.5   8   8  49  54  911.82
 38 28 72  8.9 3.48 10.7 79.8 3451 11.7 37.5 13.0  14  13  39  58  954.44
]

pollution = Dict{Symbol, Any}(
  :y => data[:, end],
  :X => mapslices(x -> x / sqrt(var(x)), data[:, 1:(end - 1)], 1),
  :p => size(data, 2) - 1
)


## Model Specification
model = Model(

  y = Stochastic(1, (mu, sigma2) -> MvNormal(mu, sqrt(sigma2)), false),

  mu = Logical(1, (alpha, X, theta) -> alpha + X * theta, false),

  alpha = Stochastic(() -> Normal(0, 1000)),

  theta = Logical(1, (beta, gamma) -> beta .* gamma),

  beta = Stochastic(1,
    p -> UnivariateDistribution[Normal(0, 1000) for i in 1:p],
    false
  ),

  gamma = Stochastic(1, () -> Bernoulli(0.5)),

  sigma2 = Stochastic(() -> InverseGamma(0.0001, 0.0001))

)


## Gibbs Sampler for alpha and beta
Gibbs_alphabeta = Sampler([:alpha, :beta],
  (alpha, beta, sigma2, X, gamma, y) ->
    begin
      alphabeta_distr = [alpha.distr; beta.distr]
      alphabeta_mean = map(mean, alphabeta_distr)
      alphabeta_invcov = spdiagm(map(d -> 1 / var(d), alphabeta_distr))
      M = [ones(y)  X * spdiagm(gamma)]
      Sigma = inv(Symmetric(M' * M / sigma2 + alphabeta_invcov))
      mu = Sigma * (M' * y / sigma2 + alphabeta_invcov * alphabeta_mean)
      alphabeta_rand = rand(MvNormal(mu, Sigma))
      Dict(:alpha => alphabeta_rand[1], :beta => alphabeta_rand[2:end])
    end
)

Gibbs_sigma2 = Sampler([:sigma2],
  (mu, sigma2, y) ->
    begin
      a = length(y) / 2.0 + shape(sigma2.distr)
      b = sum(abs2, y - mu) / 2.0 + scale(sigma2.distr)
```

```
      rand(InverseGamma(a, b))
    end
)


## Initial Values
y = pollution[:y]
X = pollution[:X]
p = size(X, 2)

inits = [
  Dict(:y => y, :alpha => mean(y), :gamma => rand(0:1, p),
       :beta => inv(X' * X + eye(p)) * X' * y, :sigma2 => var(y)),
  Dict(:y => y, :alpha => 1, :gamma => rand(0:1, p),
       :beta => randn(p), :sigma2 => 1),
  Dict(:y => y, :alpha => 17, :gamma => rand(0:1, p),
       :beta => [15, -15, -10, 5, -10, -5, -10, 10, 40, -5, 0, 0, 0, 20, 5],
       :sigma2 => 1)
]


## Sampling Scheme (without gamma)
scheme0 = [Gibbs_alphabeta, Gibbs_sigma2]

## Binary Hamiltonian Monte Carlo
scheme1 = [BHMC(:gamma, (2 * p + 0.5) * pi); scheme0]
setsamplers!(model, scheme1)
sim1 = mcmc(model, pollution, inits, 10000, burnin=1000, thin=2, chains=3)
describe(sim1)
discretediag(sim1[:, :gamma, :]) |> showall

## Binary MCMC Model Composition
scheme2 = [BMC3(:gamma); scheme0]
setsamplers!(model, scheme2)
sim2 = mcmc(model, pollution, inits, 10000, burnin=1000, thin=2, chains=3)
describe(sim2)
discretediag(sim2[:, :gamma, :]) |> showall

## Binary Metropolised Gibbs Sampling
scheme3 = [BMG(:gamma); scheme0]
setsamplers!(model, scheme3)
sim3 = mcmc(model, pollution, inits, 10000, burnin=1000, thin=2, chains=3)
describe(sim3)
discretediag(sim3[:, :gamma, :]) |> showall

## Discrete Gibbs Sampling
scheme4 = [DGS(:gamma); scheme0]
setsamplers!(model, scheme4)
sim4 = mcmc(model, pollution, inits, 10000, burnin=1000, thin=2, chains=3)
describe(sim4)
discretediag(sim4[:, :gamma, :]) |> showall

## Individual Adaptation Sampling
scheme5 = [BIA(:gamma); scheme0]
setsamplers!(model, scheme5)
sim5 = mcmc(model, pollution, inits, 10000, burnin=1000, thin=2, chains=3)
describe(sim5)
discretediag(sim5[:, :gamma, :]) |> showall
```

Results

```
## Binary Hamiltonian Monte Carlo

Iterations = 1002:10000
Thinning interval = 2
Chains = 1,2,3,4
Samples per chain = 4500

Empirical Posterior Estimates:
               Mean            SD         Naive SE        MCSE         ESS
 gamma[1]    0.494666667   0.49998544 0.00372667146  0.0354668606  198.73265
 gamma[2]    0.147444444   0.35455827 0.00264272128  0.0244410034  210.44430
 gamma[3]    0.015611111   0.12396878 0.00092400872  0.0055061412  506.90895
 gamma[4]    0.082888889   0.27572186 0.00205510941  0.0154493001  318.51125
 gamma[5]    0.030722222   0.17256889 0.00128625256  0.0105253177  268.81567
 gamma[6]    0.318833333   0.46603724 0.00347363646  0.0336986753  191.25622
 gamma[7]    0.052777778   0.22359575 0.00166658436  0.0135283814  273.17152
 gamma[8]    0.039055556   0.19373256 0.00144399723  0.0068660579  796.14158
 gamma[9]    0.963111111   0.18849422 0.00140495300  0.0115061554  268.37102
gamma[10]    0.014388889   0.11909088 0.00088765098  0.0047463037  629.57277
gamma[11]    0.022888889   0.14955344 0.00111470550  0.0070783015  446.41078
gamma[12]    0.094222222   0.29214575 0.00217752582  0.0191631169  232.41643
gamma[13]    0.113000000   0.31660159 0.00235980895  0.0206601029  234.83414
gamma[14]    0.598888889   0.49013706 0.00365326592  0.0327788035  223.58820
gamma[15]    0.008333333   0.09090846 0.00067759165  0.0023285399 1524.19744
 theta[1]   10.963490356  11.99268806 0.08938821911  0.8074731491  220.58547
 theta[2]   -3.052823000   7.78859258 0.05805274150  0.5173033385  226.68742
 theta[3]   -0.180897422   1.69190595 0.01261072235  0.0669381806  638.85842
 theta[4]    1.445892275   5.77285839 0.04302834596  0.3387376553  290.43895
 theta[5]    0.220991524   1.67908086 0.01251512983  0.0755035072  494.54860
 theta[6]   -6.639050625  10.52547301 0.07845224381  0.7272107620  209.48969
 theta[7]   -0.569567370   2.97285383 0.02215834419  0.1667594094  317.80916
 theta[8]    0.469028055   2.61485474 0.01948997649  0.0924855110  799.36980
 theta[9]   32.862541601  12.24725603 0.09128565676  0.7176766659  291.21916
theta[10]   -0.120870758   1.29895656 0.00968185058  0.0429416368  915.02312
theta[11]   -0.111881745   3.05139958 0.02274378966  0.1397888103  476.48951
theta[12]  -11.124331528  40.10862414 0.29895203352  2.8217496939  202.04056
theta[13]    9.987999815  38.64436971 0.28803812538  2.7092141635  203.46316
theta[14]   14.017636096  12.48284270 0.09304161608  0.8185848401  232.54103
theta[15]    0.027299181   0.58218805 0.00433937350  0.0086425051 4500.00000
   sigma2 1675.673663690 421.46874551 3.14144255120 17.6104714349  572.78124
    alpha  937.545199598 187.23566554 1.39557225319 12.5267326120  223.40945

Quantiles:
              2.5%       25.0%       50.0%       75.0%        97.5%
 gamma[1]    0.000000    0.000000    0.000000    1.000000    1.0000000
 gamma[2]    0.000000    0.000000    0.000000    0.000000    1.0000000
 gamma[3]    0.000000    0.000000    0.000000    0.000000    0.0000000
 gamma[4]    0.000000    0.000000    0.000000    0.000000    1.0000000
 gamma[5]    0.000000    0.000000    0.000000    0.000000    1.0000000
 gamma[6]    0.000000    0.000000    0.000000    1.000000    1.0000000
 gamma[7]    0.000000    0.000000    0.000000    0.000000    1.0000000
 gamma[8]    0.000000    0.000000    0.000000    0.000000    1.0000000
 gamma[9]    0.000000    1.000000    1.000000    1.000000    1.0000000
gamma[10]    0.000000    0.000000    0.000000    0.000000    0.0000000
gamma[11]    0.000000    0.000000    0.000000    0.000000    0.0000000
gamma[12]    0.000000    0.000000    0.000000    0.000000    1.0000000
```

```
gamma[13]     0.000000     0.000000     0.000000     0.000000     1.0000000
gamma[14]     0.000000     0.000000     1.000000     1.000000     1.0000000
gamma[15]     0.000000     0.000000     0.000000     0.000000     0.0000000
 theta[1]     0.000000     0.000000     0.000000    22.137753    32.5989842
 theta[2]   -27.095671     0.000000     0.000000     0.000000     0.0000000
 theta[3]     0.000000     0.000000     0.000000     0.000000     0.0000000
 theta[4]     0.000000     0.000000     0.000000     0.000000    23.6631537
 theta[5]     0.000000     0.000000     0.000000     0.000000     2.2412885
 theta[6]   -30.773041   -15.262916     0.000000     0.000000     0.0000000
 theta[7]   -10.908665     0.000000     0.000000     0.000000     0.0000000
 theta[8]     0.000000     0.000000     0.000000     0.000000    10.3679336
 theta[9]     0.000000    25.808602    32.266120    39.686565    58.7865382
theta[10]     0.000000     0.000000     0.000000     0.000000     0.0000000
theta[11]     0.000000     0.000000     0.000000     0.000000     0.0000000
theta[12]  -163.621352     0.000000     0.000000     0.000000     0.0000000
theta[13]   -10.012373     0.000000     0.000000     0.000000   155.4851914
theta[14]     0.000000     0.000000    17.277061    24.764372    34.1579980
theta[15]     0.000000     0.000000     0.000000     0.000000     0.0000000
   sigma2  1074.425951  1385.703616  1605.131816  1878.631424  2759.1883162
    alpha   663.986294   798.280105   878.440305  1099.826940  1314.4802623
```

2.6 Discussion

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.

The package accommodates 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.

2.7 Supplement

2.7.1 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.

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\beta}, \theta|\boldsymbol{y}) &= \log(p(\boldsymbol{y}|\boldsymbol{\beta}, \theta)p(\boldsymbol{\beta})p(\theta)) \\
&= (-n/2 - \alpha_\pi)\theta - \frac{1}{\exp\{\theta\}}\left(\frac{1}{2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \beta_\pi\right) \\
&\quad - \frac{1}{2}(\boldsymbol{\beta} - \boldsymbol{\mu}_\pi)^\top\boldsymbol{\Sigma}_\pi^{-1}(\boldsymbol{\beta} - \boldsymbol{\mu}_\pi) \\
\nabla\mathcal{L}(\boldsymbol{\beta}, \theta|\boldsymbol{y}) &= \begin{bmatrix} \frac{1}{\exp\{\theta\}}\boldsymbol{X}^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) - \boldsymbol{\Sigma}_\pi^{-1}(\boldsymbol{\beta} - \boldsymbol{\mu}_\pi) \\ -n/2 - \alpha_\pi + \frac{1}{\exp\{\theta\}}\left(\frac{1}{2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \beta_\pi\right) \end{bmatrix}
\end{aligned}$$

2.8 References

2.9 Indices

- genindex

Bibliography

[1] D Allingham, RA King, and KL Mengersen. Bayesian estimation of quantile distributions. *Statistics and Computing*, 19(2):189–201, 2009.

[2] 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.

[3] 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.

[4] J Bezanson, S Karpinski, V B Shah, and other contributors. The Julia Language. 2014. URL: http://julialang.org/.

[5] D Birkes and Y Dodge, editors. *Alternative Methods of Regression*. Wiley, New York, 1993.

[6] R D Boch and M Lieberman. Fitting a response model for n dichotomously scored items. *Psychometrika*, 35:179–197, 1970.

[7] 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.

[8] G E Box and G C Tiao, editors. *Bayesian Inference in Statistical Analysis*. Addison Wesley, Reading, MA, 1973.

[9] N E Breslow. Extra-Poisson variation in log-linear models. *Applied Statistics*, 33:38–44, 1984.

[10] N E Breslow and D G Clayton. Approximate inference in generalized linear mixed models. *Journal of the American Statistical Association*, 88:9–25, 1993.

[11] S Bromberger and other contributors. *LightGraphs*. 2016. julia software package. URL: https://github.com/JuliaGraphs/LightGraphs.jl.

[12] S Brooks and A Gelman. General methods for monitoring convergence of iterative simulations. *Journal of Computational and Graphical Statistics*, 7(4):434–455, 1998.

[13] 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.

[14] K A Brownlee, editor. *Statistical Theory and Methodology in Science and Engineering*. Wiley, New York, 1965.

[15] J B Carlin. Meta-analysis for 2 x 2 tables: a Bayesian approach. *Statistics in Medicine*, 11:141–159, 1992.

[16] 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.

[17] D Clayton. Bayesian analysis of frailty models. Technical Report, Medical Research Council Biostatistics Unit, Cambridge, 1994.

[18] 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.

[19] M K Cowles, J Yan, and B J Smith. Reparameterized and marginalized posterior and predictive sampling for complex Bayesian geostatistical models. *Journal of Computational and Graphical Statistics*, 2:262–282, 2009.

[20] M Crowder. Beta-Binomial ANOVA for proportions. *Applied Statistics*, 27:34–37, 1978.

[21] O L Davies. *Statistical Methods in Research and Production*. Olver & Boyd, Edinburgh and London, 1967.

[22] 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.

[23] S Duane, A D Kennedy, B J Pendleton, and D Roweth. Higher order hybrid Monte Carlo algorithms. *Physics Letters B*, 195:216–222, 1987.

[24] R C Elston and J E Grizzle. Estimation of time-response curves and their confidence bounds. *Biometrics*, 18:148–159, 1962.

[25] F Ezzet and J Whitehead. A random effects model for ordinal responses from a crossover trial. *Statistics in Medicine*, 10:901–907, 1993.

[26] Keno Fischer and other contributors. *GraphViz*. 2014. julia software package. URL: https://github.com/Keno/GraphViz.jl.

[27] 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.

[28] D Gamerman. *Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference*. Chapman & Hall/CRC, Boca Raton, FL, 1997.

[29] 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.

[30] 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.

[31] 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.

[32] A Gelman, G O Roberts, and W R Gilks. Efficient Metropolis jumping rules. *Bayesian Statistics*, 5:599–607, 1996.

[33] A Gelman and D B Rubin. Inference from iterative simulation using multiple sequences. *Statistical Science*, 7:457–511, 1992.

[34] 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.

[35] 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.

[36] E I George, U E Makov, and A F M Smith. Conjugate likelihood distributions. *Scandinavian Journal of Statistics*, 20:147–156, 1993.

[37] J Geweke. *Bayesian Statistics*, chapter Evaluating the Accuracy of Sampling-Based Approaches to Calculating Posterior Moments. Volume 4. Oxford University Press, New York, 1992.

[38] C J Geyer. Practical Markov chain Monte Carlo. *Statistical Science*, 7:473–511, 1992.

[39] W R Gilks, S Richardson, and D J Spiegelhalter, editors. *Monte Carlo in Practice*. Chapman & Hall/CRC, Boca Raton, FL, 1996.

[40] M Girolami and B Calderhead. Riemann manifold Langevin and Hamiltonian Monte Carlo methods. *Journal of the Royal Statistical Society: Series B*, 73(2):123–214, 2011.

[41] P W Glynn and W Whitt. Estimating the asymptotic variance with batch means. *Operations Research Letters*, 10:431–435, 1991.

[42] A P Grieve. Applications of Bayesian software: two examples. *Statistician*, 36:283–288, 1987.

[43] J E Griffin, K Łatuszyński, and M F J Steel. Individual adaptation: an adaptive mcmc scheme for variable selection problems. *arXiv:1412.6760v2 [stat.CO]*, 2014. URL: http://arxiv.org/abs/1412.6760v2.

[44] OpenBUGS Project Management Group. *OpenBUGS Examples*. 2014. version 3.2.3. URL: http://www.openbugs.net/w/Examples.

[45] H Haario, E Saksman, and J Tamminen. An adaptive Metropolis algorithm. *Bernoulli*, 7:223–242, 2001.

[46] W K Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[47] P Heidelberger and P Welch. Simulation run length control in the presence of an initial transient. *Operations Research*, 31:1109–1144, 1983.

[48] 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.

[49] SAS Institute Inc. *The MCMC Procedure*. SAS Institute Inc., Cary, NC, 2015.

[50] 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.

[51] D C Jones. *Gadfly*. 2014. julia software package. URL: https://github.com/dcjones/Gadfly.jl.

[52] J G Kalbfleisch. *Probability and Statistical Inference: Volume 2*. Springer-Verlag, New York, 1985.

[53] 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.

[54] 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.

[55] J S Liu. Peskun's Theorem and a modified discrete-state Gibbs sampler. *Biometrika*, 83(3):681–682, 1996.

[56] D Lunn, C Jackons, N Best, Thomas A,, and D Spiegelhalter. *The BUGS Book: A Practical Introduction to Bayesian Analysis*. Chapman & Hall/CRC, Boca Raton, FL, 2012.

[57] 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.

[58] D Madigan, J York, and D Allard. Bayesian graphical models for discrete data. *Revue Internationale de Statistique*, 63(2):215–232, 1995.

[59] P Marjoram, J Molitor, V Plagnol, and S Tavaré. Markov chain Monte Carlo without likelihoods. *Proceedings of the National Academy of Sciences of the United States of America*, 100(26):15324–15328, 2003.

[60] 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.

[61] G C McDonald and R C Schwing. Instabilities of regression estimates relating air pollution to mortality. *Technometrics*, 15(3):463–481, 1973.

[62] C McGilchrist and C Aisbett. Regression with frailty in survival analysis. *Biometrics*, 47:461–466, 1991.

[63] 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.

[64] R M Neal. Slice sampling (with discussion). *Annals of Statistics*, 31:705–767, 2003.

[65] R M Neal. *Handbook of Markov Chain Monte Carlo*, chapter MCMC Using Hamiltonian Dynamics, pages 113–162. CRC Press, 2011.

[66] 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.

[67] R B O'Hara and M J Sillanpää. A review of Bayesian variable selection methods: what, how and which. *Bayesian Analysis*, 4(1):85–117, 2009.

[68] A Pakman and L Paninski. Auxiliary-variable exact Hamiltonian Monte Carlo samplers for binary distributions. In C.j.c. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2409–2498. 2013. URL: http://media.nips.cc/nipsbooks/nipspapers/paper_files/nips26/1176.pdf.

[69] J H Park. *CRAN Task View: Bayesian Inference*. 2014. version 2014-05-16. URL: http://cran.r-project.org/web/views/Bayesian.html.

[70] A Patil, D Huard, and C J Fonnesbeck. PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software*, 35(4):1–81, 2010.

[71] 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.

[72] M Plummer, N Best, K Cowles, and K Vines. CODA: convergence diagnosis and output analysis for MCMC. *R News*, 6(1):7–11, 2006.

[73] 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.

[74] 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.

[75] A L Raftery and S Lewis. *Bayesian Statistics*, chapter How Many Iterations in the Gibbs Sampler? Volume 4. Oxford University Press, New York, 1992.

[76] GD Rayner and HL MacGillivray. Numerical maximum likelihood estimation for the g-and-k and generalized g-and-h distributions. *Statistics and Computing*, 12(1):57–75, 2002.

[77] C Robert. *Markov chain Monte Carlo in practice*, chapter Mixtures of distributions: inference and estimation. Chapman & Hall, 1994.

[78] C Robert and G Casella. *Monte Carlo Statistical Methods*. Springer, New York, 2nd edition, 2004.

[79] G O Roberts and J S Rosenthal. Examples of adaptive MCMC. *Journal of Computational and Graphical Statistics*, 18(2):349–367, 2009.

[80] G O Roberts and O Stramer. Langevin diffusions and Metropolis-Hastings algorithms. *Methodology and Computing in Applied Probability*, 4(4):337–357, 2002.

[81] G O Roberts and R L Tweedie. Exponential convergence of Langevin distributions and their discrete approximations. *Bernoulli*, 2(4):341–363, 1996.

[82] A F Roche, H Wainer, and D Thissen. *Skeletal maturity: The knee joint as a biological indicator*. Plenum, New York, 1975.

[83] C A Schäfer. *Monte Carlo Methods for Sampling High-Dimensional Binary Vectors*. PhD thesis, Université Paris-Dauphine, 2012.

[84] C A Schäfer and N Chopin. Sequential Monte Carlo on large binary sampling spaces. *Statistics and Computing*, 23(2):163–184, 2013.

[85] S A Sisson and Y Fan. *Handbook of Markov Chain Monte Carlo*, chapter Likelihood-Free MCMC, pages 313–335. CRC Press, 2011.

[86] B J Smith. boa: an R package for MCMC output convergence assessment and posterior inference. *Journal of Statistical Computing*, 21(11):1–37, 2007.

[87] B J Smith. *boa: Bayesian Output Analysis Program for MCMC*. 2008. R software package. URL: http://CRAN.R-project.org/package=boa.

[88] 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.

[89] 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.

[90] D Spiegelhalter, A Thomas, N Best, and D Lunn. *OpenBUGS User Manual*. March 2014. version 3.2.3. URL: http://www.openbugs.net/Manuals/Manual.html.

[91] 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.

[92] P F Thall and S C Vail. Some covariance models for longitudinal count data with overdispersion. *Biometrics*, 46:657–671, 1990.

[93] D Thissen. *MULITLOG Version 5: User's Guide*. Scientific Software, Mooresville, IN, 5th edition, 1986.

[94] A Thomas. *OpenBUGS Developer Manual*. March 2014. version 3.2.3. URL: http://www.openbugs.net/Manuals/Developer/Manual.html.

[95] L Tierney. Markov chains for exploring posterior distributions (with discussion). *Annals of Statistics*, 22:1701–1762, 1994.

[96] 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.

[97] J M White and other contributors. *Calculus*. 2014. julia software package. URL: https://github.com/johnmyleswhite/Calculus.jl.

[98] Stan Development Team. Stan: a C++ library for probability and sampling. 2014. URL: http://mc-stan.org/.

[99] Statisticat, LLC. *LaplacesDemon: Complete Environment for Bayesian Inference*. 2014. R software package. URL: http://www.bayesian-inference.com/software.

Index

Symbols

_logpdf{T<:Real}() (built-in function), 34

A

ABC() (built-in function), 56
AbstractChains, 44
AMM() (built-in function), 58
AMMVariate() (built-in function), 60
AMWG() (built-in function), 61
AMWGVariate() (built-in function), 63
autocor() (built-in function), 51

B

BHMC() (built-in function), 64
BHMCVariate() (built-in function), 65
BIA() (built-in function), 66
BIAVariate() (built-in function), 67
BMC3() (built-in function), 68
BMC3Variate() (built-in function), 70
BMG() (built-in function), 71
BMGVariate() (built-in function), 72

C

cat() (built-in function), 45
Chains, 44
Chains Types, 44
Chains() (built-in function), 44
ChainSummary, 49
changerate() (built-in function), 51
Convergence Diagnostics, 47
    Discrete, 48
    Gelman-Rubin-Brooks, 49
    Geweke, 49
    Heidelberger-Welch, 50
    Raftery-Lewis, 51
cor() (built-in function), 52

D

Dependent Types, 24
    AbstractDependent, 24
describe() (built-in function), 52
Deviance Information Criterion (DIC), 54
DGS() (built-in function), 73
DGSVariate() (built-in function), 74
dic() (built-in function), 54
discretediag() (built-in function), 48
DiscreteVariate() (built-in function), 74
Distributions, 30
    Block-Diagonal Normal, 33
    Flat, 30
    Matrix-Variate, 36
    Multivariate, 33
    Univariate, 30
    User-Defined Multivariate, 33
    User-Defined Univariate, 30
draw() (built-in function), 40, 55

E

Examples
    Asthma: State Transitions in a Clinical Trial, 150
    Birats: A Bivariate Normal Hierarchical Model, 142
    Blocker: Random Effects Meta-Analysis of Clinical Trials, 123
    Bones: Latent Trait Model for Multiple Ordered Categorical Responses, 131
    Dogs: Loglinear Model for Binary Data, 98
    Dyes: Variance Components Model, 113
    Epilepsy: Repeated Measures on Poisson Counts, 118
    Equiv: Bioequivalence in a Cross-Over Trial, 110
    Eyes: Normal Mixture Model, 147
    GK: Approximate Bayesian Computation, 152
    Inhalers: Ordered Categorical Data, 134
    Jaws: Repeated Measures Analysis of Variance, 145
    Leuk: Cox Regression, 138
    Line: Approximate Bayesian Computation, 154
    Line: Block-Specific Sampling with AMWG and Slice, 156
    Linear Regression, 7

LSAT: Item Response, 128
Magnesium: Meta-Analysis Prior Sensitivity, 105
Mice: Weibull Regression, 137
Oxford: Smooth Fit to Log-Odds Ratios, 125
Pollution: Bayesian Variable Selection, 157
Pumps: Gamma-Poisson Hierarchical Model, 95
Rats: A Normal Hierarchical Model, 92
Salm: Extra-Poisson Variation in a Dose-Response Study, 108
Seeds: Random Effect Logistic Regression, 101
Stacks: Robust Regression, 115
Surgical: Institutional Ranking, 103

F

first() (built-in function), 45

G

gelmandiag() (built-in function), 49
getindex() (built-in function), 39, 46
gettune() (built-in function), 42
gewekediag() (built-in function), 49
gradlogpdf
() (built-in function), 42
gradlogpdf() (built-in function), 42
graph() (built-in function), 40
graph2dot() (built-in function), 41

H

hcat() (built-in function), 45
heideldiag() (built-in function), 50
HMC() (built-in function), 75
HMCVariate() (built-in function), 77
hpd() (built-in function), 52

I

insupport{T<:Real}() (built-in function), 34

K

keys() (built-in function), 39

L

last() (built-in function), 45
length() (built-in function), 34
Logical Types, 25
AbstractLogical, 25
ArrayLogical, 25
ScalarLogical, 25
Logical() (built-in function), 26
logpdf
() (built-in function), 43
logpdf() (built-in function), 25, 29, 31, 43, 54

M

MALA() (built-in function), 78
MALAVariate() (built-in function), 80
maximum() (built-in function), 31
mcmc() (built-in function), 39
mcse() (built-in function), 53
minimum() (built-in function), 31
MISS() (built-in function), 81
Model, 38
Model() (built-in function), 38
ModelChains, 44
ModelChains() (built-in function), 44

N

Nodes
Input, 22
Logical, 22
Stochastic, 22
NUTS() (built-in function), 81
NUTSVariate() (built-in function), 83

P

Parallel Computing, 13
plot() (built-in function), 55
Posterior Predictive Distribution, 54
Posterior Summaries, 51
Autocorrelations, 51
Cross-Correlations, 52
Highest Posterior Density (HPD) Intervals, 52
Plotting, 55
Summary Statistics, 52
predict() (built-in function), 54

Q

quantile() (built-in function), 53

R

rafterydiag() (built-in function), 51
rand() (built-in function), 29
read() (built-in function), 47
readcoda() (built-in function), 47
relist
() (built-in function), 43
relist() (built-in function), 25, 29, 43
RWM() (built-in function), 84
RWMVariate() (built-in function), 86

S

sample
() (built-in function), 43, 58, 61, 64, 66, 69, 71, 73, 75, 78, 82, 85, 87, 90
Sampler Types, 36
ABCTune, 57
AMMTune, 60
AMMVariate, 59

AMWGTune, 63
AMWGVariate, 62
BHMCTune, 65
BHMCVariate, 65
BIATune, 68
BIAVariate, 67
BMC3Form, 70
BMC3Tune, 70
BMC3Variate, 69
BMGForm, 72
BMGTune, 72
BMGVariate, 72
DGSVariate, 73
DiscreteVariate, 73
DSForm, 74
DSTune, 74
HMCTune, 77
HMCVariate, 76
MALATune, 80
MALAVariate, 79
MISSTune, 81
NUTSTune, 84
NUTSVariate, 83
RWMTune, 86
RWMVariate, 86
Sampler, 36
SamplerTune, 37
SamplerVariate, 37
SliceForm, 89
SliceMultivariate, 88
SliceSimplexTune, 91
SliceSimplexVariate, 91
SliceTune, 89
SliceUnivariate, 88
Sampler() (built-in function), 36
SamplerVariate() (built-in function), 37
SamplerVariate{T<:SamplerTune}() (built-in function), 37
Sampling Functions
Adaptive Metropolis within Gibbs, 61
Adaptive Mixture Metropolis, 58
Approximate Bayesian Computation, 56
Binary Hamiltonian Monte Carlo, 63
Binary Individual Adaptation, 66
Binary MCMC Model Composition, 68
Binary Metropolised Gibbs, 70
Discrete Gibbs Sampler, 73
Hamiltonian Monte Carlo, 74
Metropolis-Adjusted Langevin Algorithm, 77
Missing Values Sampler, 80
No-U-Turn Sampler, 81
Random Walk Metropolis, 84
Shrinkage Slice, 87
Slice Simplex, 89
setindex
() (built-in function), 46
setinits
() (built-in function), 27, 28, 41
setinputs
() (built-in function), 41
setmonitor
() (built-in function), 25
setsamplers
() (built-in function), 42
show() (built-in function), 24, 37, 41
showall() (built-in function), 24, 37, 41
Slice() (built-in function), 87
SliceMultivariate() (built-in function), 89
SliceSimplex() (built-in function), 90
SliceSimplexVariate() (built-in function), 91
SliceUnivariate() (built-in function), 89
step() (built-in function), 45
Stochastic() (built-in function), 28
StochasticTypes, 27
AbstractStochastic, 27
ArrayStochastic, 27
ScalarStochastic, 27
summarystats() (built-in function), 53

U

unlist() (built-in function), 25, 29, 43
update
() (built-in function), 27, 30, 44

V

Variate Types, 22
AbstractVariate, 22
ArrayVariate, 22
MatrixVariate, 22
ScalarVariate, 22
VectorVariate, 22
vcat() (built-in function), 45

W

write() (built-in function), 47