JAGS Version 4.3.0 user manual

Martyn Plummer

28 June 2017

Contents

Chapter 1

Introduction

JAGS is "Just Another Gibbs Sampler". It is a program for the analysis of Bayesian models using Markov Chain Monte Carlo (MCMC) which is not wholly unlike OpenBUGS (`http://www.openbugs.info`). JAGS is written in C++ and is portable to all major operating systems.

JAGS is designed to work closely with the R language and environment for statistical computation and graphics (`http://www.r-project.org`). You will find it useful to install the coda package for R to analyze the output. Several R packages are available to run a JAGS model. See chapter 4.

JAGS is licensed under the GNU General Public License version 2. You may freely modify and redistribute it under certain conditions (see the file `COPYING` for details).

1.1 Downloading JAGS

JAGS is hosted on Sourceforge at `https://sourceforge.net/projects/mcmc-jags/`. Binary distributions of JAGS for Windows and macOS can be downloaded directly from there. Binaries for various Linux distributions are also available, but not directly from Sourceforge. See `http://mcmc-jags.sourceforge.net` for details.

You can also download the source tarball from Sourceforge. Details on installing JAGS from source are given in a separate JAGS Installation manual.

1.2 Getting help

The best way to get help on JAGS is to use the discussion forums hosted on Sourceforge at `https://sourceforge.net/p/mcmc-jags/discussion/`. You will need to create a Sourceforge account in order to post.

Bug reports can be sent to `martyn_plummer@users.sourceforge.net`. Occasionally, JAGS will stop with an error instructing you to send a message to this address. I am particularly interested in the following issues:

- Crashes, including both segmentation faults and uncaught exceptions.
- Incomprehensible error messages
- Models that should compile, but don't

- Output that cannot be validated against other software such as OpenBUGS
- Documentation erors

If you submit a bug report, it must be reproducible. Please send the model file, the data file, the initial value file and a script file that will reproduce the problem. Describe what you think should happen, and what did happen.

1.3 Acknowledgements

Many thanks to the BUGS development team, without whom JAGS would not exist. Thanks also to Simon Frost for pioneering JAGS on Windows and Bill Northcott for getting JAGS on Mac OS X to work. Kostas Oikonomou found many bugs while getting JAGS to work on Solaris using Sun development tools and libraries. Bettina Gruen, Chris Jackson, Greg Ridgeway and Geoff Evans also provided useful feedback. Special thanks to Jean-Baptiste Denis who has been very diligent in providing feedback on JAGS. Matt Denwood is a co-developer of JAGS with full access to the Sourceforge repository.

Chapter 2

The BUGS language

A JAGS model is defined in a text file using a dialect of the BUGS language [Lunn et al., 2012]. The model definition consists of a series of *relations* inside a block delimited by curly brackets `{` and `}` and preceded by the keyword `model`. Here is a simple linear regression example:

```
model {
    for (i in 1:N) {
          Y[i]   ~ dnorm(mu[i], tau)
          mu[i] <- alpha + beta * (x[i] - x.bar)
    }
    x.bar   <- mean(x)
    alpha    ~ dnorm(0.0, 1.0E-4)
    beta     ~ dnorm(0.0, 1.0E-4)
    sigma   <- 1.0/sqrt(tau)
    tau      ~ dgamma(1.0E-3, 1.0E-3)
}
```

2.1 Relations

Each relation defines a node in the model. The node on the left of a relation is defined in terms of other nodes – referred to as parent nodes – that appear on the right hand side. Taken together, the nodes in the model form a directed acyclic graph (with the parent/child relationships represented as directed edges). The very top-level nodes in the graph, with no parents, are constant nodes, which are defined either in the model definition (*e.g.* `1.0E-3`), or in the data when the model is compiled (*e.g.* `x[1]`).

Relations can be of two types. A *stochastic relation* (`~`) defines a stochastic node, representing a random variable in the model. A *deterministic relation* (`<-`) defines a deterministic node, the value of which is determined exactly by the values of its parents. The equals sign (`=`) can be used for a deterministic relation in place of the left arrow (`<-`).

2.2 Arrays and subsetting

Nodes defined by a relation are embedded in named arrays. Arrays can be vectors (one-dimensional), or matrices (two-dimensional), or they may have more than two dimensions.

Array names may contain letters, numbers, decimal points and underscores, but they must start with a letter. In the code example above, the node array `mu` is a vector of length N containing N nodes (`mu[1]`, ..., `mu[N]`). The node array `alpha` is a scalar. JAGS follows the S language convention that scalars are considered as vectors of length 1. Hence the array `alpha` contains a single node `alpha[1]`.

2.2.1 Subsetting vectors

Subsets of node arrays may be obtained with expressions of the form `alpha[e]` where `e` is any expression that evaluates to an integer vector. Typically expressions of the form `L:U` are used, where L and U are integer arguments to the operator ":". This creates an integer vector of length $U - L + 1$ with elements $L, L + 1, \ldots U - 1, U$.[1] Both the arguments L and U must be fixed because JAGS does not allow the length of a node to change from one iteration to the next.

2.2.2 Subsetting matrices

For a two-dimensional array `B`, the element in row r and column c is accessed as `B[r,c]`. Complex subsets may be obtained with expressions of the form `B[rows, cols]` where `rows` is an integer vector of row indices and `cols` is an integer vector of column indices. An index expression in any dimension may be omitted, and this implies taking the whole range of possible indices. So if `B` is an $M \times N$ matrix then `B[r,]` is the whole of row r, and is equivalent to `B[r,1:N]`. Likewise `B[,c]` is the whole of column c and is equivalent to `B[1:M,c]`. Finally, `B[,]` is the whole matrix, equivalent to `B[1:M,1:N]`. In this case, one may also simply write `B` without any square brackets. Note that writing `B[]` for a 2-dimensional array will result in a compilation error: the number of indices separated by commas within the square brackets must match the dimensions of the array.

2.2.3 Restrictions on index expressions

There are some restrictions on index expressions that are allowed on the left hand side of a relation. Firstly, they must be fixed, meaning that they must be expressed in terms of data, or constants defined in the model, or the index of an enclosing for loop (see below). As a counter-example, the following code snippet is illegal:

```
  i ~ dcat(p)
x[i] ~ dnorm(0, 1)
```

unless i is observed and is supplied with the data. This restriction ensures that the node `x[i]` can be unambiguously referred to elsewhere in the model – e.g. as a function argument or the parameter for a distribution – and does not change from one iteration to another. Secondly, for vector-valued subsets, the same index must not be repeated. For example, supposed μ is a 2-vector and T is a 2×2 matrix. Then `dmnorm(mu, T)` defines a bivariate normal random variable. However, this code snippet is illegal:

```
indices <- c(1,1)
x[indices] ~ dmnorm(mu, T)
```

[1] If $L > U$ then the operator : returns a vector of length zero, which may not be used to take array subsets

as it implies writing two different values to the same element `x[1]`.

2.3 Constructing vectors

Vectors can be constructed using the combine function `c` from the bugs module, *e.g.*

```
y <- c(x1, x2, x3)
```

creates a new vector y combining the elements of $x1$, $x2$, and $x3$.

The combine function can be used with a single array argument

```
v <- c(a)
```

In this case the return value v is a vector with the values of the input array a in column-major order (i.e. with the left hand index moving fastest). For example if a is a 2×2 matrix then this is equivalent to

```
v <- c(a[1,1], a[2,1], a[1,2], a[2,2])
```

2.4 For loops

For loops are used in the the BUGS language to simplify writing repeated expressions. A for loop takes the form:

```
for (i in exp) {
}
```

where `exp` is any expression that evaluates to an integer vector. The contents of the for loop inside the curly braces will then be expanded with the index `i` taking each value in the vector `exp`. For example

```
for (i in 1:3) {
   Y[i] ~ dnorm(mu, tau)
}
```

is equivalent to

```
Y[1] ~ dnorm(mu, tau)
Y[2] ~ dnorm(mu, tau)
Y[3] ~ dnorm(mu, tau)
```

As in the above example, for loops generally use the sequence operator `:` to generate an increasing sequence of integers. This operator has high precedence, so is important to use brackets to ensure that the expression is interpreted correctly. For example, to get a for loop from 1 to $m + 1$ the correct expression is

```
for (i in 1:(m+1)) {
}
```

Omitting the brackets as in the following expression:

```
for (i in 1:m+1) {
}
```

gives a for loop from 2 to $m + 1$.

The sequence operator `:` can only produce increasing sequences. If $n < m$ then `m:n` produces a vector of length zero and when this is used in a for loop index expression the contents of loop inside the curly brackets are skipped. Note that this behaviour is different from the sequence operator in R, where `m:n` will produce a *decreasing* sequence if $n < m$.

2.5 Data transformations

Sometimes it is useful to transform the data before analysing them. For example, a transformation of outcomes may be necessary to reduce skewness and allow the outcome to be represented by a normal or other symmetric distribution.

You can transform the data supplied to JAGS in a separate block of relations preceded by the keyword `data`. Here for example, the input data `y` is transformed via a square-root transformation to `z`, which is then used as an outcome variable in the model block:

```
data {
   for (i in 1:N) {
      z[i] <- sqrt(y[i])
   }
}
model {
   for (i in 1:N) {
      z[i] ~ dnorm(mu, tau)
   }
   ...
}
```

In effect, the data block defines a distinct model, which describes how the data are generated. Each node in this model is forward-sampled once, and then the node values are read back into the data table.

2.5.1 Modelling simulated data

The data block is not limited to logical relations, but may also include stochastic relations. You may therefore use it in simulations, generating data from a stochastic model that is different from the one used to analyse the data in the `model` statement.

This example shows a simple location-scale problem in which the "true" values of the parameters `mu` and `tau` are generated from a given prior in the `data` block, and the generated data is analyzed in the `model` block.

```
data {
   for (i in 1:N) {
      y[i] ~ dnorm(mu.true, tau.true)
   }
   mu.true ~ dnorm(0,1);
```

```
   tau.true ~ dgamma(1,3);
}
model {
   for (i in 1:N) {
      y[i] ~ dnorm(mu, tau)
   }
   mu ~ dnorm(0, 1.0E-3)
   tau ~ dgamma(1.0E-3, 1.0E-3)
}
```

Beware, however, that every node in the `data` statement will be considered as data in the subsequent `model` statement. This example, although superficially similar, has a quite different interpretation.

```
data {
   for (i in 1:N) {
      y[i] ~ dnorm(mu, tau)
   }
   mu ~ dnorm(0,1);
   tau ~ dgamma(1,3);
}
model {
   for (i in 1:N) {
      y[i] ~ dnorm(mu, tau)
   }
   mu ~ dnorm(0, 1.0E-3)
   tau ~ dgamma(1.0E-3, 1.0E-3)
}
```

Since the names `mu` and `tau` are used in both `data` and `model` blocks, these nodes will be considered as *observed* in the model and their values will be fixed at those values generated in the `data` block.

2.6 Node Array dimensions

2.6.1 Array declarations

JAGS allows the option of declaring the dimensions of node arrays in the model file. The declarations consist of the keyword `var` (for variable) followed by a comma-separated list of array names, with their dimensions in square brackets. The dimensions may be given in terms of any expression of the data that returns a single integer value.

In the linear regression example, the model block could be preceded by

```
var x[N], Y[N], mu[N], alpha, beta, tau, sigma, x.bar;
```

2.6.2 Undeclared nodes

If a node array is not declared then JAGS has three methods of determining its size.

1. **Using the data.** The dimension of an undeclared node array may be inferred if it is supplied in the data file.

2. **Using the left hand side of the relations.** The maximal index values on the left hand side of a relation are taken to be the dimensions of the node array. For example, in this case:

```
for (i in 1:N) {
   for (j in 1:M) {
      Y[i,j] ~ dnorm(mu[i,j], tau)
   }
}
```

   Y would be inferred to be an $N \times M$ matrix. This method cannot be used when there are empty indices (*e.g.* `Y[i,]`) on the left hand side of the relation.

3. **Using the dimensions of the parents** If a whole node array appears on the left hand side of a relation, then its dimensions can be inferred from the dimensions of the nodes on the right hand side. For example, if `A` is known to be an $N \times N$ matrix and

```
B <- inverse(A)
```

   Then `B` is also an $N \times N$ matrix.

2.6.3 Querying array dimensions

The JAGS compiler has two built-in functions for querying array sizes. The `length()` function returns the number of elements in a node array, and the `dim()` function returns a vector containing the dimensions of an array. These two functions may be used to simplify the data preparation. For example, if `Y` represents a vector of observed values, then using the `length()` function in a for loop:

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

avoids the need to put a separate data value `N` in the file representing the length of `Y`.

For multi-dimensional arrays, the `dim` function serves a similar purpose. The `dim` function returns a vector, which must be stored in an array before its elements can be accessed. For this reason, calls to the `dim` function must always be in a data block (see section 2.5).

```
data {
   D <- dim(Z)
}
model {
   for (i in 1:D[1]) {
      for (j in 1:D[2]) {
         Z[i,j] <- dnorm(alpha[i] + beta[j], tau)
      }
```

```
    }
    ...
}
```

Clearly, the `length()` and `dim()` functions can only work if the size of the node array can be inferred, using one of the three methods outlined above.

Note: the `length()` and `dim()` functions are different from all other functions in JAGS: they do not act on nodes, but only on node *arrays*. As a consequence, an expression such as `dim(a %*% b)` is syntactically incorrect.

Chapter 3

Steps in running a model

JAGS is designed for inference on Bayesian models using Markov Chain Monte Carlo (MCMC) simulation. Running a model refers to generating samples from the posterior distribution of the model parameters. This takes place in five steps:

1. Definition of the model
2. Compilation
3. Initialization
4. Adaptation and burn-in
5. Monitoring

The next stages of analysis are done outside of JAGS: convergence diagnostics, model criticism, and summarizing the samples must be done using other packages more suited to this task. There are several R packages designed for running a JAGS model and analyzing the output. See chapter 4 for details.

3.1 Modules

The JAGS library is distributed along with dynamically loadable modules that extend its functionality. A module can define new objects of the following classes:

1. **Functions** and **distributions**, the basic building blocks of the BUGS language.
2. **Samplers**, the objects which update the parameters of the model at each iteration, and **sampler factories**, the objects that create new samplers for specific model structures.
3. **Monitors**, the objects that record sampled values for later analysis, and **monitor factories** that create them.
4. **Random number generators**, the objects that drive the MCMC algorithm and **RNG factories** that create them.

The `base` module and the `bugs` module are loaded automatically at start time. Others may be loaded by the user. It is important to load the modules you need before running the model.

The two most useful optional modules are the `glm` module (chapter 10), which provides efficient samplers for generalized linear mixed modules, and the `dic` module (chapter 11), which provides novel monitors for assessing model fit using deviance statistics.

3.2 Compilation

When a JAGS model is compiled, the model description is combined with the data to create a *virtual graphical model*, which is a representation of the model as a directed acyclic graph (DAG) in computer memory.

One of the interesting features of the BUGS language is that it does not clearly distinguish between data and parameters. A stochastic node – *i.e.* a node defined on the left hand side of a stochastic relation – may be either observed (data) or unobserved (parameter). The distinction is made only when the model is compiled and depends entirely on whether a data value was supplied or not.

Data may also be supplied for *constant nodes*. These are nodes that are used on the right hand side of a relation (or in the index expression of a for loop) but are never defined on the left hand side. In fact you must supply data for these nodes or the JAGS compiler will stop with an error message.

Data may not be supplied for deterministic nodes – *i.e.* those defined on the left hand side of a deterministic relation. These nodes always calculate their value according to the current value of the parents.

Recall the simple linear regression model from chapter 2.

```
model {
    for (i in 1:N) {
          Y[i]   ~ dnorm(mu[i], tau)
          mu[i] <- alpha + beta * (x[i] - x.bar)
    }
    x.bar   <- mean(x)
    alpha    ~ dnorm(0.0, 1.0E-4)
    beta     ~ dnorm(0.0, 1.0E-4)
    sigma   <- 1.0/sqrt(tau)
    tau      ~ dgamma(1.0E-3, 1.0E-3)
}
```

This model may be compiled by combining with data values for `x`, `Y` and `N`. Exactly how the data are supplied depends on which interface is being used. With any of the R interfaces to JAGS, the user can create a list of values to be supplied as data, e.g.

```
line_data <- list("x" = c(1, 2, 3, 4, 5),
                  "Y" = c(1, 3, 3, 3, 5),
                  "N" = 5)
```

and then supply this list as the `data` argument to any of the modelling functions (NB the above is R code, not BUGS code). This is explored in more detail in chapter 4. With the

command line interface, the data are supplied in a separate file with values written in the `dump()` format of the R language, e.g.

```
#contents of file line-data.R
"x" <- c(1, 2, 3, 4, 5)
"Y" <- c(1, 3, 3, 3, 5)
"N" <- 5
```

See section 5.2 for more details.

In our simple linear regression model, data values must always be supplied for `N`, since it is used to index the for loop. Data must also be supplied for `x` since it is used on the right hand side of the relation defining `mu` but never defined on the left hand side of a relation.

In this example, data values are also supplied for the outcome variables `Y[1]` to `Y[5]`. The parameters of the model are the three stochastic nodes for which no data are supplied: `alpha`, `beta` and `tau`. JAGS will generate samples from the posterior distribution of these parameters given `Y`.

As can be seen from the example above, data values are supplied for whole node arrays If a node array contains both observed and unobserved nodes, then the data should contain missing values (`NA`) for the unobserved elements. Suppose that if `Y[2]` and `Y[5]` were unobserved, then the data supplied for the node array `Y` would be.

```
Y = c(1, NA, 3, 3, NA)
```

Then `Y[2]` and `Y[5]` are parameters and JAGS will generate samples from their posterior predictive distributions, given the observed `Y` values.

Multivariate nodes cannot be partially observed, so if a node takes up two or more elements of a node array, then the corresponding data values must be all present or all missing.

3.2.1 Compilation failures

Compilation may fail for a number of reasons. The three most common are:

1. The model uses a function or distribution that has not been defined in any of the loaded modules.
2. The graph contains a directed cycle. These are forbidden in JAGS.
3. A node is used in an expression but never defined by a relation or supplied with the data.

3.2.2 Parallel chains

The number of parallel chains to be run by JAGS is also defined at compilation time. Each parallel chain should produce an independent sequence of samples from the posterior distribution.

3.3 Initialization

Before a model can be run, it must be initialized. There are three steps in the initialization of a model:

1. The initial values of the model parameters are set.
2. A Random Number Generator (RNG) is chosen for each parallel chain, and its seed is set.
3. The Samplers are chosen for each parameter in the model.

3.3.1 Initial values

The user may supply initial value files – one for each chain – containing initial values for the model parameters. Initial values may not be supplied for logical or constant nodes.

The format for initial values is the same as the one used for data (i.e. a list using the R interface or a separate file when using the command line interface). As with the data, initial values are supplied for whole node arrays and may include missing values for elements of the array that are not to be initialized. This need typically arises with contrast parameters. Suppose X is a categorical covariate, taking values from 1 to 4, which is used as a predictor variable in a linear model:

```
for (i in 1:N) {
   Y[i] ~ alpha + beta[x[i]]
}
# Prior distribution
alpha ~ dnorm(0, 1.0E-3)
beta[1] <- 0
for(i in 2:4) {
   beta[i] ~ dnorm(0, 1.0E-3)
}
```

There is one parameter for each level of x ($\beta_1 \ldots \beta_4$), but the first parameter β_1 is set to zero for identifiability. The remaining parameters $\beta_2 \ldots \beta_4$ represent contrasts with respect to the first level of X.

A suitable set of initial values (for a single chain) would be

```
list(alpha = 0, beta = c(NA, 1.03, -2.03, 0.52))
```

This allows parameter values to be supplied for the stochastic elements of `beta` but not the constant first element.

If initial values are not supplied by the user, then each parameter chooses its own initial value based on the values of its parents. The initial value is chosen to be a "typical value" from the prior distribution. The exact meaning of "typical value" depends on the distribution of the stochastic node, but is usually the mean, median, or mode.

If you rely on automatic initial value generation and are running multiple parallel chains, then the initial values will be the same in all chains.[1] You are advised to set the starting values manually.

[1] This is undesirable behaviour and it will be changed in a future release of JAGS.

3.3.2 Random Number Generators

Each chain in JAGS has its own random number generator (RNG). RNGs are more correctly referred to as *pseudo*-random number generators. They generate a sequence of numbers that merely looks random but is, in fact, entirely determined by the initial state. You may optionally set the name of the RNG and its initial state with the initial values for the parameters.

There are four RNGs supplied by the `base` module in JAGS with the following names:

```
"base::Wichmann-Hill"
"base::Marsaglia-Multicarry"
"base::Super-Duper"
"base::Mersenne-Twister"
```

There are two ways to set the starting state of the RNG. The simplest is to supply the name of the RNG and its seed (a single integer value) with the initial values, *e.g.* using the R interface to JAGS:

```
list(".RNG.name" = "base::Wichmann-Hill",
     ".RNG.seed" = 314159,
     ...)
```

Here we use `...` to denote that the same list includes initial values for the parameters (not shown).

It is also possible to save the state of the RNG from one JAGS session and use this as the initial state of a new chain. The state of any RNG in JAGS can be saved and loaded as an integer vector with the name `.RNG.state`. For example,

```
list(".RNG.name" = "base::Marsaglia-Multicarry",
     ".RNG.state" = as.integer(c(20899,10892,29018)),
     ...)
```

is a valid state for the Marsaglia-Multicarry generator. You cannot supply an arbitrary integer vector to `.RNG.state`. Both the length of the vector and the permitted values of its elements are determined by the class of the RNG. The only safe way to use `.RNG.state` is to re-use a previously saved state.

If no RNG names are supplied, then RNGs will be chosen automatically so that each chain has its own independent random number stream. The exact behaviour depends on which modules are loaded. The `base` module uses the four generators listed above for the first four chains, then recycles them with different seeds for the next four chains, and so on.

By default, JAGS bases the initial state on the time stamp. This means that, when a model is re-run, it generates an independent set of samples. If you want your model run to be reproducible, you must explicitly set the `.RNG.seed` for each chain.

3.3.3 Samplers

A Sampler is an object that acts on a set of parameters and updates them from one iteration to the next. During initialization of the model, samplers are chosen automatically for all parameters.

JAGS holds a list of *sampler factory* objects, which inspect the graph, recognize sets of parameters that can be updated with specific methods, and generate sampler objects for

them. The list of sampler factories is traversed in order, starting with sampling methods that are efficient, but limited to certain specific model structures and ending with the most generic, possibly inefficient, methods. If no suitable sampler can be generated for one of the model parameters, an error message is generated.

The user has no direct control over the process of choosing samplers. However, you may indirectly control the process by loading a module that defines a new sampler factory. The module will insert the new sampler sactory at the beginning of the list, where it will be queried before all of the other sampler factories. You can also optionally turn on and off sampler factories.

3.4 Adaptation and burn-in

In theory, output from an MCMC sampler converges to the target distribution (*i.e.* the posterior distribution of the model parameters) in the limit as the number of iterations tends to infinity. In practice, all MCMC runs are finite. By convention, the MCMC output is divided into two parts: an initial "burn-in" period, which is discarded, and the remainder of the run, in which the output is considered to have converged (sufficiently close) to the target distribution. Samples from the second part are used to create approximate summary statistics for the target distribution.

By default, JAGS keeps only the current value of each node in the model, unless a monitor has been defined for that node. The burn-in period of a JAGS run is therefore the interval between model initialization and the creation of the first monitor.

When a model is initialized, it may be in *adaptive mode*, meaning that the Samplers used by the model may modify their behaviour for increased efficiency. Since this adaptation may depend on the entire sample history, the sequence generated by an adapting sampler is no longer a Markov chain, and is not guaranteed to converge to the target distribution. Therefore, adaptive mode must be turned off at some point during burn-in, and a sufficient number of iterations must take place *after* the adaptive phase to ensure successful burnin.

All samplers have a built in test to determine whether they have converged to their optimal sampling behaviour. If any sampler fails this validation test, a warning will be printed. To ensure optimal sampling behaviour, the model should be run again from scratch using a longer adaptation period.

3.5 Monitoring

A *monitor* in JAGS is an object that records sampled values. The simplest monitor is a *trace monitor*, which stores the sampled value of a node at each iteration. Other types of monitor are available: monitors of type "mean" and "variance" defined by the base module store the running mean and variance, respectively, of a given node (See section 8.2.4). The dic module defines various monitors based on deviance statistics for model criticism (See chapter 11).

3.6 Errors

There are two kinds of errors in JAGS: runtime errors, which are due to mistakes in the model specification, and logic errors which are internal errors in the JAGS program.

Logic errors are generally created in the lower-level parts of the JAGS library, where it is not possible to give an informative error message. The upper layers of the JAGS program are supposed to catch such errors before they occur, and return a useful error message that will help you diagnose the problem. Inevitably, some errors slip through. Hence, if you get a logic error, there is probably an error in your input to JAGS, although it may not be obvious what it is. Please send a bug report (see "Getting help" on page 4) whenever you get a logic error.

Error messages may also be generated when parsing files (model files, data files, command files). The error messages generated in this case are created automatically by the program `bison`. They generally take the form "syntax error, unexpected FOO, expecting BAR" and are not always abundantly clear, but this is out of my control.

Chapter 4

Running JAGS from R

The native interface from R to JAGS is the `rjags` package [Plummer, 2016]. This provides an object-based functional interface to JAGS, in which most of the steps of running a model are separated into different function calls. Several alternative packages have been developed that offer a unified interface to JAGS. The main ones are detailed below, but there are many other R packages that interface to JAGS. See the list of reverse depends, imports, suggests and enhances at `https://cran.r-project.org/package=rjags`

To illustrate each package, we consider the linear regression model from chapter 2 with a trivial data set of 5 observations. The data and initial values for this model are given below and are used by all R interfaces:

```
line_data <- list("x" = c(1, 2, 3, 4, 5), "Y" = c(1, 3, 3, 3, 5), "N"=5)
line_inits <- list(list("alpha" = 3, "beta"= 0, "tau" = 1.5),
                   list("alpha" = 0, "beta" = 1, "tau" = 0.375))
```

4.1 rjags

To run a model with the `rjags` package, we first create a model object with the `jags.model()` function.

```
library(rjags)
model <- jags.model("line.bug", data=line_data, inits=line_inits, n.chains=2)
```

This compiles and initializes the model, and if necessary runs an adaptive phase for 1000 iterations (see chapter 3 for an explanation of these steps). If adaptation is required then a progress bar made of `'+'` signs will be printed.

The object created by `jags.model` is of class "jags". It is an R interface to the virtual graphical model created by JAGS and does not contain any samples.

To get samples from the posterior distribution of the parameters, we use the `coda.samples` function after first using the `update` function to run the Markov Chain for a burn-in period of 1000 iterations.

```
update(model, n.iter=1000)
samples <- coda.samples(model, variable.names=c("alpha", "beta", "sigma"),
                        n.iter=1000)
```

The `samples` object is of class `mcmc.list` defined by the `coda` package [Plummer et al., 2006]. It contains monitored samples of the variables `alpha`, `beta`, and `sigma` for the two parallel chains.

To get an informative summary of the samples, we use the `summary` function:

```
> summary(samples)

Iterations = 1001:2000
Thinning interval = 1
Number of chains = 2
Sample size per chain = 1000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

        Mean     SD Naive SE Time-series SE
alpha 3.0001 0.5344 0.011949       0.012392
beta  0.7997 0.3921 0.008769       0.009133
sigma 1.0326 0.7120 0.015920       0.030337

2. Quantiles for each variable:

         2.5%    25%    50%    75% 97.5%
alpha 1.95654 2.7452 2.9916 3.2488 4.158
beta  0.01763 0.6178 0.8037 0.9877 1.559
sigma 0.41828 0.6376 0.8276 1.1838 2.804
```

The `mcmc.list` class also has a plot method. Hence

```
plot(samples)
```

will produce trace plots and density plots for the sampled values.

4.2 runjags

The `runjags` package [Denwood, 2016] includes many enhancements to JAGS, including a custom JAGS module that contains some additional distributions in the Pareto family. The basic function for running a model with the `runjags` package is `run.jags()`, which can be used as follows:

```
library(runjags)
out <- run.jags(model="line.bug", monitor=c("alpha","beta","sigma","dic"),
                data=line_data, n.chains=2, inits=line_inits)
```

The optional `method` argument to `run.jags` allows the user to choose whether the model is run through the `rjags` interface (section 4.1) or through the command line interface of JAGS (chapter 5). In addition, the `method` argument controls how parallel chains are run. Running parallel chains in different processes can offer considerable speed improvements on a multi-processor computer.

The output of the `run.jags()` function is an object of class "runjags". The print method for "runjags" objects prints a summary of the posterior distribution of the parameters.

```
> print(out)

JAGS model summary statistics from 20000 samples
(chains = 2; adapt+burnin = 5000):

       Lower95  Median Upper95    Mean      SD    Mode     MCerr MC%ofSD SSeff
alpha   1.9171  3.0004  4.0096  2.9971 0.54331  2.9936 0.0037325     0.7 21189
beta  0.046318 0.79919  1.5288 0.79989 0.39328 0.79952 0.0028322     0.7 19283
sigma  0.31105 0.82308   2.155  1.0107 0.73651 0.68375 0.0094755     1.3  6042

            AC.10   psrf
alpha  -0.0029037      1
beta   -0.0044582 1.0004
sigma -0.00090713      1

Model fit assessment:
DIC = 21.16815
[PED not available from the stored object]
Estimated effective number of parameters:  pD = 8.24059

Total time taken: 0.9 seconds
```

The print method also shows some convergence diagnostics for each parameter: `SSeff` is the effective sample size, controlling for autocorrelation of the Markov chain; `AC.10` is the autocorrelation at lag 10; and `psrf` is the Gelman-Rubin convergence diagnostic [Gelman and Rubin, 1992], or the *potential scale reduction factor*.

If the chain is showing signs of poor mixing then the `extend.jags` function can be used to continue running the model. The `autorun.jags` is an alternative to `run.jags` that takes control of the run-length of the MCMC process and runs the chain until convergence.

The default plot method for "runjags" objects shows a traceplot, the cumulative distribution function, the density function and the autocorrelation function for each variable. A cross-correlation plot for all variables is also generated. There are a number of other methods available for "runjags" objects, including `as.mcmc.list` for extraction of the MCMC objects, and `as.jags` for conversion to an object that can be used with the rjags package. The reverse operation is also possible by using `as.runjags` on an rjags model.

One of the motivations for the runjags package is to allow comments embedded directly within the the JAGS model code to specify variables to be monitored, and to indicate any variables for which data or initial values should be extracted automatically from the R session. The `template.jags` function uses these features to create complete model, data and initial value definitions based on a data frame and formula-based representation of a generalised linear mixed model as provided by the user.

4.3 R2jags

The R2jags package [Su and Yajima, 2015] was modelled on the pre-existing R2WinBUGS package [Sturtz et al., 2005] which provided the first interface from WinBUGS to R. With R2jags and R2WinBUGS the user may switch between the two BUGS engines while keeping a consistent R interface. The R2OpenBUGS package is an updated version of R2WinBUGS that works with OpenBUGS.

Like the `runjags` package, the `R2jags` package offers a single interface to JAGS that carries out all the steps of running the model, with reasonable default values. The interface function is `jags()`.

```
library(R2jags)
out <- jags(data=line_data, inits=line_inits,
            parameters.to.save=c("alpha","beta","sigma"),
            model.file="line.bug", n.chains=2)
```

The output from the `jags` function is an object of class "rjags". The print method for "rjags" objects summarizes the output (There is also a `plot` method).

```
> out
Inference for Bugs model at "line.bug", fit using jags,
 2 chains, each with 2000 iterations (first 1000 discarded)
 n.sims = 2000 iterations saved
         mu.vect sd.vect  2.5%    25%    50%    75%  97.5%  Rhat n.eff
alpha      2.992   0.552 1.888  2.757  3.002  3.258  4.037 1.001  2000
beta       0.797   0.381 0.041  0.618  0.795  0.988  1.549 1.005  1000
sigma      1.014   0.664 0.417  0.636  0.840  1.165  2.581 1.002  2000
deviance  12.981   3.595 8.832 10.358 12.104 14.660 22.215 1.002  1100

For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 6.5 and DIC = 19.4
DIC is an estimate of expected predictive error (lower deviance is better).
```

In addition to summaries of the posterior distribution, the print method for "rjags" objects includes `Rhat`, the Gelman-Rubin convergence diagnostic [Gelman and Rubin, 1992] and `n.eff` the effective sample size.

If an "rjags" object shows signs of lack of convergence, then it can be passed to the `autojags` function which will update it until convergence.

4.4 jagsUI

The `jagsUI` package offers similar facilities to the `runjags` and `R2jags` package. In fact `jagsUI` offers two interface functions – `jags()` and `autojags()` – with the same names as the ones in the `R2jags` package. It is therefore not a good idea to load both packages in the same R session.

Compared with the other packages, the output from `jagsUI` tends to be more verbose and didactic. It may therefore be suitable for users who are not familiar with MCMC.

The `jags` function is illustrated below.

```
library(jagsUI)
out <- jags(data=line_data, inits=line_inits,
            parameters.to.save=c("alpha","beta","sigma"),
            model.file="line.bug", n.chains=2,
            n.iter=2000, n.burnin=1000)
```

Note that you must supply arguments `n.iter` and `n.burnin`. However, noted above, there is also an `autojags()` function that controls the run length for you.

The output of the `jags` function is an object of class "jagsUI". The print method for "jagsUI" objects gives summary statistics and diagnostics similar to the `R2jags` package, but with more explanation.

```
JAGS output for model 'line.bug', generated by jagsUI.
Estimates based on 2 chains of 1000 iterations,
burn-in = 0 iterations and thin rate = 1,
yielding 2000 total samples from the joint posterior.
MCMC ran for 0.001 minutes at time 2017-06-27 17:57:07.

           mean    sd  2.5%    50%  97.5% overlap0     f  Rhat n.eff
alpha     2.995 0.615 1.883  2.984  4.053    FALSE 0.997 1.002  2000
beta      0.798 0.402 0.062  0.813  1.486    FALSE 0.977 1.011   332
sigma     1.041 0.768 0.420  0.828  3.218    FALSE 1.000 1.023  2000
deviance 12.984 3.880 8.849 11.921 23.821    FALSE 1.000 1.002  2000

Successful convergence based on Rhat values (all < 1.1).
Rhat is the potential scale reduction factor (at convergence, Rhat=1).
For each parameter, n.eff is a crude measure of effective sample size.

overlap0 checks if 0 falls in the parameter's 95% credible interval.
f is the proportion of the posterior with the same sign as the mean;
i.e., our confidence that the parameter is positive or negative.

DIC info: (pD = var(deviance)/2)
pD = 7.5 and DIC = 20.51
DIC is an estimate of expected predictive error (lower is better).
```

Chapter 5

Running JAGS on the command line

JAGS has a command line interface. To invoke jags interactively, simply type `jags` at the shell prompt on Unix, a Terminal window on MacOS, or the Windows command prompt on Windows. To invoke JAGS with a script file, type

```
jags <script file>
```

The linear regression model from chapter 2 can be run with the following commands

```
model in "line.bug"      # Read model file
data in "line-data.R"    # Read data in from file
compile, nchains(2)      # Compile a model with two parallel chains
parameters in "line-inits1.R", chain(1) # Read initial values from file for chain 1
parameters in "line-inits2.R", chain(2) # Read initial values from file for chain 2
initialize               # Initialize the model
update 1000              # Adaptation (if necessary) and burnin for 1000 iterations
monitor alpha            # Set trace monitor for node alpha ...
monitor beta             # ... and beta
monitor sigma            # ... and sigma
update 10000             # Update model for 10000 iterations
coda *                   # All monitored values are written out to file
```

More examples can be found in the file `classic-bugs.tar.gz` which is available from Sourceforge (`http://sourceforge.net/projects/mcmc-jags/files`).

Output from JAGS is printed to the standard output, even when a script file is being used.

5.1 Scripting commands

JAGS has a simple set of scripting commands with a syntax loosely based on Stata. Commands are shown below preceded by a dot (.). This is the JAGS prompt. Do not type the dot in when you are entering the commands.

C-style block comments taking the form /* ... */ can be embedded anywhere in the script file. Additionally, you may use R-style single-line comments starting with #.

If a scripting command takes a file name, then the name may be optionally enclosed in quotes. Quotes are required when the file name contains space, or any character which is not alphanumeric, or one of the following: `_`, `-`, `.`, `/`, `\`.

In the descriptions below, angular brackets `<>`, and the text inside them, represents a parameter that should be replaced with the correct value by you. Anything inside square brackets `[]` is optional. Do not type the square brackets if you wish to use an option.

MODEL IN

```
. model in <file>
```

Checks the syntactic correctness of the model description in `file` and reads it into memory. The next compilation statement will compile this model.

See also: MODEL CLEAR (page 30)

DATA IN

```
. data in <file>
```

JAGS keeps an internal data table containing the values of observed nodes inside each node array. The DATA IN statement reads data from a file into this data table. See section 5.2 for details on the file format.

Several data statements may be used to read in data from more than one file. If two data files contain data for the same node array, the second set of values will overwrite the first, and a warning will be printed.

See also: DATA TO (page 29).

COMPILE

```
. compile [, nchains(<n>)]
```

Compiles the model using the information provided in the preceding model and data statements. By default, a single Markov chain is created for the model, but if the `nchains` option is given, then `n` chains are created

Following the compilation of the model, further DATA IN statements are legal, but have no effect. A new model statement, on the other hand, will replace the current model.

PARAMETERS IN

```
. parameters in <file> [, chain(<n>)]
```

Reads the values in `file` and writes them to the corresponding parameters in chain `n`. The file has the same format as the one in the DATA IN statement. The `chain` option may be omitted, in which case the parameter values in all chains are set to the same value.

The PARAMETERS IN statement must be used after the COMPILE statement and before the INITIALIZE statement. You may only supply the values of unobserved stochastic nodes in the parameters file, not logical or constant nodes.

See also: PARAMETERS TO (page 29)

INITIALIZE

```
. initialize
```

Initializes the model using the previously supplied data and parameter values supplied for each chain.

UPDATE

```
. update <n> [,by(<m>)]
```

Updates the model by `n` iterations.

If JAGS is being run interactively, a progress bar is printed on the standard output consisting of 50 asterisks. If the `by` option is supplied, a new asterisk is printed every `m` iterations. If this entails more than 50 asterisks, the progress bar will be wrapped over several lines. If `m` is zero, the printing of the progress bar is suppressed.

If JAGS is being run in batch mode, then the progress bar is suppressed by default, but you may activate it by supplying the `by` option with a non-zero value of `m`.

If the model has an adaptive sampling phase, the first UPDATE statement turns off adaptive mode for all samplers in the model after `n/2` iterations. A warning is printed if adaptation is incomplete. Incomplete adaptation means that the mixing of the Markov chain is not optimal. It is still possible to continue with a model that has not completely adapted, but it may be preferable to run the model again with a longer adaptation phase, starting from the MODEL IN statement. Alternatively, you may use an ADAPT statement (see below) immediately after initialization.

ADAPT

```
. adapt <n> [,by(<m>)]
```

Updates the model by `n` iterations keeping the model in adaptive mode and prints a message to indicate whether adaptation is successful. Successive calls to ADAPT may be made until the adaptation is successful. The next call to UPDATE then turns off adaptive mode immediately.

Use this instead of the first UPDATE statement if you want explicit control over the length of the adaptive sampling phase.

Like the UPDATE statement, the ADAPT statement prints a progress bar, but with plus signs instead of asterisks.

MONITOR

In JAGS, a monitor is an object that calculates summary statistics from a model. The most commonly used monitor simply records the value of a single node at each iteration. This is called a "trace" monitor.

```
. monitor <varname> [, thin(n)] [type(<montype>)]
```

The `thin` option sets the thinning interval of the monitor so that it will only record every nth value. The `thin` option selects the type of monitor to create. The default type is `trace`.

More complex monitors can be defined that do additional calculations. For example, the `dic` module defines a "deviance" monitor that records the deviance of the model at

each iteration, and a "pD" monitor that calculates an estimate of the effective number of parameters on the model [Spiegelhalter et al., 2002].

```
. monitor clear <varname> [type(<montype>)]
```

Clears the monitor of the given type associated with variable `<varname>`.

CODA

```
. coda <varname> [, stem(<filename>)]
```

This dumps monitored values for a given node to file in a form that can be read by the coda package of R. The wild-card character "*" may be used to dump all monitored nodes

Monitor types can be classified according to whether they pool values over iterations and whether they pool values over parallel chains (The standard trace monitor does neither). When monitor values are written out to file using the CODA command, the output files created depend on the pooling of the monitor, as shown in table 5.1. By default, all of these files have the prefix CODA, but this may be changed to any other name using the "stem" option to the CODA command.

Pool iterations	Pool chains	Output files
no	no	CODAindex.txt, CODAchain1.txt, ... CODAchainN.txt
no	yes	CODAindex0.txt, CODAchain0.txt
yes	no	CODAtable1.txt, ... CODAtableN.txt
yes	yes	CODAtable0.txt

Table 5.1: Output files created by the CODA command depending on whether a monitor pools its values over chains or over iterations

The standard CODA format for monitors that do not pool values over iterations is to create an index file and one or more output files. The index file is has three columns with, one each line,

1. A string giving the name of the (scalar) value being recorded
2. The first line in the output file(s)
3. The last line in the output file(s)

The output file(s) contain two columns:

1. The iteration number
2. The value at that iteration

Some monitors pool values over iterations. For example a mean monitor may record only the sample mean of a node, without keeping the individual values from each iteration. Such monitors are written out to a table file with two columns:

1. A string giving the name of the (scalar) value being recorded
2. The value (pooled over all iterations)

EXIT

```
. exit
```

Exits JAGS. JAGS will also exit when it reads an end-of-file character.

DATA TO

```
. data to <filename>
```

Writes the data (*i.e.* the values of the observed nodes) to a file in the R `dump` format. The same file can be used in a DATA IN statement for a subsequent model.

See also: DATA IN (page 26)

PARAMETERS TO

```
. parameters to <file> [, chain(<n>)]
```

Writes the current parameter values (*i.e.* the values of the unobserved stochastic nodes) in chain `<n>` to a file in R dump format. The name and current state of the RNG for chain `<n>` is also dumped to the file. The same file can be used as input in a PARAMETERS IN statement in a subsequent run.

See also: PARAMETERS IN (page 26)

SAMPLERS TO

```
. samplers to <file>
```

Writes out a summary of the samplers to the given file. The output appears in three tab-separated columns, with one row for each sampled node

- The index number of the sampler (starting with 1). The index number gives the order in which Samplers are updated at each iteration.
- The name of the sampler, matching the index number
- The name of the sampled node.

If a Sampler updates multiple nodes then it is represented by multiple rows with the same index number.

Note that the list includes only those nodes that are updated by a Sampler. Stochastic nodes that are updated by forward sampling from the prior are not listed.

LOAD

```
. load <module>
```

Loads a module into JAGS (see section 3.1). Loading a module does not affect any previously initialized models, but will affect the future behaviour of the compiler and model initialization.

UNLOAD

```
. unload <module>
```

Unloads a module. Currently initialized models are unaffected, but the functions, distribution, and factory objects in the model will not be accessible to future models.

LIST MODULES

```
. list modules
```

Prints a list of the currently loaded modules.

LIST FACTORIES

```
. list factories, type(<factype>)
```

List the currently loaded factory objects and whether or not they are active. The `type` option must be given, and the three possible values of `<factype>` are `sampler`, `monitor`, and `rng`.

SET FACTORY

```
. set factory "<facname>" <status>, type(<factype>)
```

Sets the status of a factor object. The possible values of `<status>` are `on` and `off`. Possible factory names are given from the LIST MODULES command.

MODEL CLEAR

```
. model clear
```

Clears the current model. The data table (see page 26) remains intact

Print Working Directory (PWD)

```
. pwd
```

Prints the name of the current working directory. This is where JAGS will look for files when the file name is given without a full path, *e.g.* `"mymodel.bug"`.

Change Directory (CD)

```
. cd <dirname>
```

Changes the working directory to `<dirname>`

Directory list (DIR)

```
. dir
```

Lists the files in the current working directory.

RUN

```
. run <cmdfile>
```

Opens the file `<cmdfile>` and reads further scripting commands until the end of the file. Note that if the file contains an EXIT statement, then the JAGS session will terminate.

5.2 Data format

The file format used by JAGS for representing data and initial values is the `dump()` format used by R. This format is valid R code, so a JAGS data file can be read into R using the `source()` function. The JAGS parser ignores all white space. Long expressions can therefore be split over several lines.

Scalar values are represented by a simple assignment statement

```
theta <- 0.1243
```

All numeric values are read in to JAGS as doubles, even if they are represented as integers (i.e. '12' and '12.0' are equivalent). Engineering notation may be used to represent large or small values (e.g 1.5e-3 is equivalent to 0.0015).

Vectors are denoted by the R collection function "c", which takes a number of arguments equal to the length of the vector. The following code denotes a vector of length 4:

```
x <- c(2, 3.5, 1.3e-2, 88)
```

There is no distinction between row and column-vectors.

Matrices and higher-dimensional arrays in R are created by adding a dimension attribute (`.Dim`) to a vector. In the R dump format this is done using the "structure" function. The first argument to the structure function is a vector of numeric values of the matrix, given in column major order (i.e. filling the left index first). The second argument, which must be given the tag `.Dim`, is the number of dimensions of the array, represented by a vector of integer values. For example, if the matrix "A" takes values

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

it is represented as

```
`A` <- structure(c(1, 2, 3, 4, 5, 6), .Dim=c(3,2))
```

The simplest way to prepare your data is to read them into R and then dump them. Only numeric vectors, matrices and arrays are allowed. More complex data structures such as factors, lists and data frames cannot be parsed by JAGS nor can non-numeric vectors. Any R attributes of the data (such as names and dimnames) are ignored when they are read into JAGS.

Chapter 6

Functions

Functions allow deterministic nodes to be defined using a left arrow `<-` or an equals sign `=` .

6.1 Link Functions

Table 6.1 lists the link functions in the `bugs` module. These are smooth scalar-valued functions that may be specified using an S-style replacement function notation. So, for example, the log link

```
log(y) <- x
```

is equivalent to the more direct use of its inverse, the exponential function:

```
y <- exp(x)
```

This usage comes from the use of link functions in generalized linear models.

6.2 Vectorization

Scalar functions taking scalar arguments are automatically vectorized. They can also be called when the arguments are arrays with conforming dimensions, or scalars. So, for example, the scalar c can be added to the matrix A using

```
B <- A + c
```

instead of the more verbose form

Link function	Description	Range	Inverse
`cloglog(y) <- x`	Complementary log log	$0 < y < 1$	`y <- icloglog(x)`
`log(y) <- x`	Log	$0 < y$	`y <- exp(x)`
`logit(y) <- x`	Logit	$0 < y < 1$	`y <- ilogit(x)`
`probit(y) <- x`	Probit	$0 < y < 1$	`y <- phi(x)`

Table 6.1: Link functions in the `bugs` module

```
D <- dim(A)
for (i in 1:D[1])
   for (j in 1:D[2]) {
      B[i,j] <- A[i,j] + c
   }
}
```

Inverse link functions are an exception to this rule (*i.e.* exp, icloglog, ilogit, phi) and cannot be vectorized.

6.3 Functions associated with distributions

All distributions in JAGS have a log density function associated with them. For example, if variable `x` has a normal distribution with mean `mu` and precision `tau`:

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

then the log density of `x` is given by

```
ldx <- logdensity.norm(x, mu, tau)
```

In general, if "`dfoo`" is the name of a distribution then the associated log density function is "`logdensity.foo`". The first argument of the log-density function is the sample value at which the log-density is evaluated and the remaining arguments are the parameters of the distribution.

For some scalar-valued distributions, additional functions are available that give the probability density, probability function and the quantile function. These distributions and associated functions are given in table 6.2. These distributions are all in the `bugs` module. See section 9.2 for details on the parameterization of these distributions.

The density, probability, and quantile functions associated with the normal distribution are illustrated below:

```
density.x  <- dnorm(x, mu, tau)     # Density of normal distribution at x
prob.x     <- pnorm(x, mu, tau)     # P(X <= x)
quantile90.x <- qnorm(0.9, mu, tau) # 90th percentile
```

6.4 Function aliases

A function may optionally have an alias, which can be used in the model definition in place of the canonical name. Aliases are used to avoid confusion with other software in which functions may have slightly different names. Table 6.3 shows the functions in the `bugs` module with an alias.

Distribution	Distribution	Quantile	
Bernoulli	dbern	pbern	qbern
Beta	dbeta	pbeta	qbeta
Binomial	dbin	pbin	qbin
Chi-squared	dchisqr	pchisqr	qchisqr
Double exponential	ddexp	pdexp	qdexp
Exponential	dexp	pexp	qexp
F	df	pf	qf
Gamma	dgamma	pgamma	qgamma
Generalized gamma	dgen.gamma	pgen.gamma	qgen.gamma
Noncentral hypergeometric	dhyper	phyper	qhyper
Logistic	dlogis	plogis	qlogis
Log-normal	dlnorm	plnorm	qlnorm
Negative binomial	dnegbin	pnegbin	qnegbin
Noncentral Chi-squared	dnchisqr	pnchisqr	qnchisqr
Normal	dnorm	pnorm	qnorm
Pareto	dpar	ppar	qpar
Poisson	dpois	ppois	qpois
Student t	dt	pt	qt
Weibull	dweib	pweib	qweib

Table 6.2: Functions to calculate the probability density, probability function, and quantiles of some of the distributions provided by the `bugs` module.

Function	Canonical name	Alias	Compatible with
Arc-cosine	arccos	acos	R
Hyperbolic arc-cosine	arccosh	acosh	R
Arc-sine	arcsin	asin	R
Hyperbolic arc-sine	arcsinh	asinh	R
Arc-tangent	arctan	atan	R

Table 6.3: Functions with aliases in `bugs` module

Chapter 7

Distributions

7.1 Truncating distributions

Some scalar-valued distributions in JAGS can be truncated. Truncation is represented in JAGS using the `T(,)` construct on the right hand side of a stochastic relation. If

```
X ~ dfoo(theta) T(L,U)
```

then *a priori* X is known to lie between L and U. This generates a likelihood

$$\frac{p(x \mid \theta)}{P(L \leq X \leq U \mid \theta)}$$

if $L \leq X \leq U$ and zero otherwise, where $p(x \mid \theta)$ is the density of X given θ according to the distribution `foo`. Note that calculation of the denominator may be computationally expensive.

The `T(,)` construct may also be used with one argument dropped. For example `T(L,)` means that the distribution is truncated below at value L and `T(,U)` means it is truncated above at value U.

A common use of the `T(,)` construct is to restrict a distribution to positive values, e.g.

```
theta ~ dnorm(0,1.0E-3) T(0,)
```

gives a diffuse-half normal prior to `theta` on positive values only.

Important note: The `T(,)` construct must not be used for censoring. Censoring and truncation are two closely related issues that frequently cause confusion in the BUGS language. If you have a variable X in your model such that $X > C$ but you are not sure whether X is censored or truncated then ask yourself: What should happen if you generate a new data set by sampling from the prior distribution?

- If it is always true that $X > c$ in any replicate data set, then X is *truncated* and you must use the $T(,)$ construct.
- If you might have $X \leq c$ in a replicate data set, but you happen to observe $X > c$ in this data set, then X is *censored*. Censored observations in JAGS are represented by the `dinterval` distribution (section 9.2.4).

7.2 Observable Functions

Logical nodes in the BUGS language are a convenient way of describing the relationships between observables (constant and stochastic nodes), but are not themselves observable. You cannot supply data values for a logical node. This restriction can occasionally be inconvenient, as there are important cases where the data are a deterministic function of unobserved variables. Three important examples are

1. Censored data, which commonly occurs in survival analysis. In the most general case, we know that unobserved failure time T lies in the interval $(L, U]$.

2. Rounded data, when there is a continuous underlying distribution but the measurements are rounded to a fixed number of decimal places.

3. Aggregate data when we observe the sum of two or more unobserved variables.

The `bugs` module contains novel distributions to handle these situations. These are documented in section 9.2.4. These distributions are referred to as *observable functions*. They exist to give a likelihood to data that is, in fact, a deterministic function of the parameters. For example, the relation

```
Y ~ sum(x1, x2)
```

allows you to model an observed random variable Y as the sum of two unobserved variables $x1$ and $x2$. If Y is not observed, then the above relation creates Y as a deterministic node and is logically equivalent to

```
Y <- x1 + x2
```

Important note: Observable functions can cause problems when you want to use deviance-based statistics to assess model fit (See the `dic` module in chapter 11). The likelihood generated by an observable function is trivial: it is 1 if the parameters are consistent with the data and 0 otherwise. This likelihood is used by JAGS when sampling to enforce the constraints on the parameters given by the observable function. However, it is not useful for evaluating the model fit because it has the wrong *focus* (See Spiegelhalter et al. [2002] for a discussion of focus in a hierarchical model).

7.3 Ordered distributions

A multivariate distribution in which the elements are ordered can be created from a scalar-valued distribution using the `sort` function. For example:

```
for (i in 1:3) {
   alpha0[i] ~ dnorm(0, 1.0E-3)
}
alpha[1:3] <- sort(alpha0)
```

The elements of `alpha` (α) satisfy ($\alpha_1 < \alpha_2 < \alpha_3$). It is very important to supply initial values for the unsorted parameters (`alpha0` in this example) as by default JAGS will initialize them to the same value and the sorted parameters (`alpha`) will be identical instead of ordered.

7.4 Distribution aliases

A distribution may optionally have an alias, which can be used in the model definition in place of the canonical name. Aliases are used to avoid confusion with other statistical software in which distributions may have slightly different names. Note however that the alias does not change the parameterization. For example, the Weibull distribution is parameterized differently in BUGS and R, and using the alias `dweibull` instead of the BUGS name `dweib` does not change this.

Chapter 8

The base module

The base module supplies the base functionality for the JAGS library to function correctly. It is loaded first and should never be unloaded.

In order to avoid a name clash with the `base` package of R, the file name of the base module is `basemod.so` on Unix/Linux or `basemod.dll` on Windows.

8.1 Functions in the base module

The functions defined by the `base` module are all infix or prefix operators. The syntax of these operators is built into the JAGS parser and they are considered part of the modelling language. Table 8.1 lists them in reverse order of precedence.

Type	Usage	Description
Logical operators	`x || y`	Or
	`x && y`	And
	`!x`	Not
Comparison operators	`x > y`	Greater than
	`x >= y`	Greater than or equal to
	`x < y`	Less than
	`x <= y`	Less than or equal to
	`x == y`	Equal
	`x != y`	Not equal
Arithmetic operators	`x + y`	Addition
	`x - y`	Subtraction
	`x * y`	Multiplication
	`x / y`	Division
	`x %special% y`	User-defined operators
	`x : y`	Sequence
	`-x`	Unary minus
Power function	`x^y`	

Table 8.1: Base functions listed in reverse order of precedence

There is no logical data type in the BUGS language. Logical operators, like all other

functions in JAGS, take numeric arguments and return numerical values. Zero arguments are considered to be FALSE and non-zero arguments to be TRUE. If the result of a logical operator is TRUE then the value 1 is returned, or if it is FALSE the value 0 is returned.

Comparison operators use the same convention as logical operators and return 1 or 0 for TRUE and FALSE respectively. Comparison operators are non-associative: the expression `x < y < z`, for example, is syntactically incorrect. Likewise `x != y == z` is an invalid expression.

The `%special%` function is an exception in table 8.1. It is not a function defined by the `base` module, but is a place-holder for any function with a name starting and ending with the character "`%`" For example, the `bugs` module defines a matrix multiplication operator `%*%`. Such functions are automatically recognized as infix operators by the JAGS model parser, with precedence defined by table 8.1.

The sequence operator in the expression `m:n` takes two integer arguments m and n (which may themselves be arithmetic expressions) and returns a vector of length $n-m+1$ containing the increasing integer sequence $m, m+1, \ldots, n-1, n$. Since the length of the resulting vector may not change from one iteration to another, the arguments m and n must be fixed. If $n < m$ then `m:n` returns a vector of length zero. See section 2.4 for the consequences of this in for loops.

For compatibility with OpenBUGS, the expression `x^y` can also be represented as `pow(x,y)`.

8.2 Samplers in the base module

The `base` module defines samplers that use highly generic update methods. These sampling methods only require basic information about the stochastic nodes they sample.

8.2.1 Inversion

The `base::Finite` factory produces the `base::Finite` sampler for discrete-valued, bounded stochastic nodes (*i.e.* those that can take only a finite number of values). The `base::Finite` sampler draws samples by inverting the cumulative distribution function. When the number of possible values is large, inversion can be slow and the discrete slice sampler (see below) is typically more efficient. For this reason, the `base::Finite` factory will not produce a sampler when there are more than 100 possible values. An exception to this 100-value limit is made for nodes with a `dcat` prior, which may be arbitrarily large.

8.2.2 Slice sampling

Slice sampling [Neal, 2003] is the "work horse" sampling method in JAGS. The `base:Slice` factory produces three samplers based on slice sampling:

- `base::RealSlicer` samples continuous scalar distributions
- `base::DiscreteSlicer` samples discrete scalar distribution
- `base::MSlicer` samples multivariate continuous distributions.

Occasionally a slice sampler may fail with the message "Slicer stuck at value with infinite density". This occurs when sampling the gamma distribution (page 45) with shape parameter

less than 1, or the beta distribution (page 43) when either parameter is less than 1. In such cases, the density function is not bounded but goes to infinity at the boundary of the support (0 for the gamma distribution; 0 or 1 for the beta distribution). If this occurs, one solution is to use the `T(,)` construct to keep the distribution away from the boundary (See section 7.1)

8.2.3 RNGs in the base module

The `base` module defines four RNGs, taken directly from R, with the following names:

1. `"base::Wichmann-Hill"`
2. `"base::Marsaglia-Multicarry"`
3. `"base::Super-Duper"`
4. `"base::Mersenne-Twister"`

A single RNG factory object is also defined by the `base` module which will supply these RNGs for chains 1 to 4 respectively, if "RNG.name" is not specified in the initial values file. All chains generated by the base RNG factory are initialized using the current time stamp.

If you have more than four parallel chains, then the base module will recycle the same four RNGs, but using different seeds. If you want many parallel chains then you may wish to load the `lecuyer` module.

8.2.4 Monitors in the base module

Trace monitor

A trace monitor records the current value of a given node at each iteration. It is the default monitor type in JAGS, but it can be explicitly selected by choosing monitor type "trace".

Mean monitor

A mean monitor records a running mean of a given node. It is selected by choosing monitor type "mean". If you are only interested in the posterior mean of a node then a mean monitor is much more memory-efficient than a trace monitor. This is especially true if you want to estimate the posterior mean of thousands of nodes.

If you use a mean monitor then it is important to run several parallel chains in order to assess the sample variance.

Variance monitor

A variance monitor records the running variance of a given node. It is selected by choosing type "variance". See also mean monitor, above.

For a vector-valued node, a variance monitor records the variance of each element, but not the covariances between elements.

Chapter 9

The bugs module

The `bugs` module was originally intended to include the functions and distributions from OpenBUGS. It has expanded somewhat beyond its original purpose and now contains a number of novel functions and distributions.

9.1 Functions in the bugs module

9.1.1 Scalar functions

Table 9.1 lists the scalar-valued functions in the `bugs` module that also have scalar arguments. These functions are automatically vectorized when they are given vector, matrix, or array arguments with conforming dimensions.

9.1.2 Scalar-valued functions with array arguments

Table 9.2 lists the scalar-valued functions in the `bugs` module that take general arguments. Unless otherwise stated in table 9.2, the arguments to these functions may be scalar, vector, or higher-dimensional arrays.

9.1.3 Vector- and array-valued functions

Table 9.3 lists vector- or matrix-valued functions in the `bugs` module.

The `rep` function replicates elements of a vector. If

```
y <- rep(v[1:M], 2)
```

then y is of length $2M$ taking values `v[1]`, `v[1]`, `v[2]`, `v[2]`, ... `v[M]`, `v[M]`. The second argument can also be a vector of the same length as v, e.g.

```
y <- rep(v[1:M], t[1:M])
```

In this case element `v[i]` is replicated `t[i]` times.

The `sort` function accepts a vector and returns the same values sorted in ascending order; `rank` returns a vector of ranks of the elements; `order` returns a permutation that sorts the elements of the vector in ascending order. Ties are broken by giving the smallest rank/order in the output vector to the element with the smallest index in the input vector.

The `order` and `rank` functions are complementary in the sense that if

Usage	Description	Value	Restrictions on arguments
`abs(x)`	Absolute value	Real	
`arccos(x)`	Arc-cosine	Real	$-1 < x < 1$
`arccosh(x)`	Hyperbolic arc-cosine	Real	$1 < x$
`arcsin(x)`	Arc-sine	Real	$-1 < x < 1$
`arcsinh(x)`	Hyperbolic arc-sine	Real	
`arctan(x)`	Arc-tangent	Real	
`arctanh(x)`	Hyperbolic arc-tangent	Real	$-1 < x < 1$
`cos(x)`	Cosine	Real	
`cosh(x)`	Hyperbolic Cosine	Real	
`cloglog(x)`	Complementary log log	Real	$0 < x < 1$
`equals(x,y)`	Test for equality	Logical	
`exp(x)`	Exponential	Real	
`icloglog(x)`	Inverse complementary log log function	Real	
`ifelse(x,a,b)`	If x then a else b	Real	
`ilogit(x)`	Inverse logit	Real	
`log(x)`	Log function	Real	$x > 0$
`logfact(x)`	Log factorial	Real	$x > -1$
`loggam(x)`	Log gamma	Real	$x > 0$
`logit(x)`	Logit	Real	$0 < x < 1$
`phi(x)`	Standard normal cdf	Real	
`pow(x,z)`	Power function	Real	If $x < 0$ then z is integer
`probit(x)`	Probit	Real	$0 < x < 1$
`round(x)`	Round to integer away from zero	Integer	
`sin(x)`	Sine	Real	
`sinh(x)`	Hyperbolic Sine	Real	
`sqrt(x)`	Square-root	Real	$x >= 0$
`step(x)`	Test for $x \geq 0$	Logical	
`tan(x)`	Tangent	Real	
`tanh(x)`	Hyperbolic Tangent	Real	
`trunc(x)`	Round to integer towards zero	Integer	

Table 9.1: Scalar functions in the `bugs` module

Function	Description	Restrictions
`c(x1, x2, ...)`	Combine all arguments into a vector	
`inprod(x1,x2)`	Inner product	Dimensions of $x1$, $x2$ conform
`interp.lin(e,v1,v2)`	Linear Interpolation	e scalar, $v1, v2$ conforming vectors
`logdet(m)`	Log determinant	m symmetric positive definite matrix
`max(x1,x2,...)`	Maximum element among all arguments	
`mean(x)`	Mean of elements of x	
`min(x1,x2,...)`	Minimum element among all arguments	
`prod(x)`	Product of elements of x	
`sd(x)`	Standard deviation of elements of x	
`sum(x1,x2,...)`	Sum of elements of all arguments	

Table 9.2: Scalar-valued functions with general arguments in the `bugs` module , Ellipses (...) mean that the function takes a variable number of arguments

```
p <- rank(z)
q <- order(z)
```

Then

```
p[q] == 1:M
q[p] == 1:M
```

This complementarity allows you to shift indexing operations from the left-hand side to the right-hand side of an expression. For example, the following code using rank on the left:

```
p <- rank(z)
for (i in 1:N)) {
   y[p[i]] <- x[i]
}
```

is equivalent to the following code using order on the right:

```
q <- order(z)
for (i in 1:N) {
   y[i] <- x[q[i]]
}
```

When z is a stochastic vector, the first code is invalid – JAGS only allows fixed indices on the left hand side of a relation – but the second code still works.

9.2 Distributions in the bugs module

9.2.1 Continuous univariate distributions

Beta

The beta distribution with shape parameters $a > 0, b > 0$ is defined by

```
y ~ dbeta(a,b)
```

Usage	Description	Restrictions
`inverse(a)`	Matrix inverse	a is a symmetric positive definite matrix
`rank(v)`	Ranks of elements of v	v is a vector
`rep(v, t)`	Replicate elements of v	v, t are vectors, t is integer
`order(v)`	Ordering permutation of v	v is a vector
`sort(v)`	Elements of v in order	v is a vector
`t(a)`	Transpose	a is a matrix
`a %*% b`	Matrix multiplication	a, b conforming vector or matrices

Table 9.3: Vector- or matrix-valued functions in the `bugs` module

The density of the beta distribution is

$$\frac{y^{a-1}(1-y)^{b-1}}{\beta(a,b)}$$

for $0 \leq y \leq 1$.

The Dirichlet distribution (page 52) is a multivariate generalization of the beta distribution.

Chi-squared

The chi-squared distribution on $k > 0$ degrees of freedom is defined by

```
y ~ dchisqr(k)
```

The alias `dchisq` may also be used for compatibility with R.

The density of the chi-squared distribution is

$$\frac{y^{\frac{k}{2}-1}\exp(-y/2)}{2^{\frac{k}{2}}\Gamma(\frac{k}{2})}$$

for $y \geq 0$. The chi-squared distribution has mean k and variance $2k$.

The chi-squared distribution is a special case of the gamma distribution and is equivalent to

```
y ~ dgamma(1/2, k/2)
```

The degrees of freedom parameter k does not need to be an integer.

Double exponential

The double exponential (or Laplace) distribution with mean μ (mu) and precision τ (tau) is defined by

```
y ~ ddexp(mu, tau)
```

for $\tau > 0$. It has probability density function

$$\tau\exp(-\tau|y-\mu|)/2$$

The variance of the double exponential distribution is $1/\tau$.

Exponential

The exponential distribution with rate λ (lambda) is defined by

```
y ~ dexp(lambda)
```

for $\lambda > 0$. It has probability density function

$$\lambda \exp(-\lambda y)$$

for $y \geq 0$. The exponential distribution has mean $1/\lambda$ and variance $1/\lambda$.

In survival analysis, `T ~ dexp(lambda)` is the survival distribution for an event T that has constant hazard λ.

F

The F distribution on n and m degrees of freedom is defined by

```
y ~ df(n, m)
```

for $n > 0$ and $m > 0$. It has probability density function

$$\frac{\Gamma(\frac{n+m}{2})}{\Gamma(\frac{n}{2})\Gamma(\frac{m}{2})} \left(\frac{n}{m}\right)^{\frac{n}{2}} y^{\frac{n}{2}-1} \left\{1 + \frac{ny}{m}\right\}^{-\frac{(n+m)}{2}}$$

for $y \geq 0$.

The F distribution can be constructed as the ratio of two chi-squared distributions (suitably scaled).

```
y <- (x1/sqrt(n))/(x2/sqrt(m))
x1 ~ dchisqr(n)
x2 ~ dchisqr(m)
```

The F distribution has mean $m/(m-2)$ for $m > 2$, whereas for $0 < m \leq 2$ the mean is undefined. The variance is finite for $m > 4$.

Gamma

The gamma distribution with shape r and rate λ (lambda) is defined by

```
y ~ dgamma(r, lambda)
```

for $\lambda > 0$ and $r > 0$. It has probability density function

$$\frac{\lambda^r y^{r-1} \exp(-\lambda y)}{\Gamma(r)}$$

for $y \geq 0$.

When $r = 1$ the gamma distribution reduces to the exponential distribution with rate λ. The chi-squared distribution is also a special case of the gamma distribution. See page 44.

Generalized gamma

The generalized gamma distribution is a flexible 3-parameter family that includes both the gamma distribution and the Weibull distribution as special cases.

The generalized gamma distribution with parameters r, b, and λ (lambda) is defined by

```
y ~ dgen.gamma(r, lambda, b)
```

for $r > 0, b > 0, \lambda > 0$. For compatibility with OpenBUGS, the alias `dggamma` may also be used.

The generalized gamma distribution has probability density

$$\frac{b\lambda^{br} y^{br-1} \exp\{-(\lambda y)^b\}}{\Gamma(r)}$$

for $y \geq 0$.

When $b = 1$ the generalized gamma reduces to the gamma distribution with shape r and rate λ. When $r = 1$ it reduces to the Weibull distribution. However, it should be noted that the parameterization of the `dweib` distribution is not the same as the one used by the `dgen.gamma` distribution. See page 9.2.1 for more details.

Logistic

The logistic distribution with mean μ (mu) and precision τ (tau) is defined by

```
y ~ dlogis(mu, tau)
```

for $\tau > 0$. It has probability density function

$$\frac{\tau \exp\{(y-\mu)\tau\}}{[1 + \exp\{(y-\mu)\tau\}]^2}$$

The variance of the logistic distribution is $\pi^2/(3 * \tau)$, *i.e.* approximately 3.3 times larger than the variance of a normal distribution with the same precision parameter.

The cumulative distribution function of the logistic distribution is

$$P(Y \leq y) = \frac{1}{1 + \exp\{-(y-\mu)\tau\}}$$

which, for $\tau = 1$, is the logistic link function.

Log-normal

The log-normal distribution with parameters μ (mu) and τ (tau) is defined by

```
y ~ dlnorm(mu,tau)
```

for $\tau > 0$. Its probability density is

$$\left(\frac{\tau}{2\pi}\right)^{\frac{1}{2}} y^{-1} \exp\left\{-\tau(\log(y) - \mu)^2/2\right\}$$

for $y > 0$.

As the name suggests, the log-normal distribution is the log transformation of a normal distribution with mean μ and precision τ. It is equivalent to

```
y <- log(x)
x ~ dnorm(mu, tau)
```

Non-central chi-squared

The non-central chi-squared distribution on k degrees of freedom with non-centrality parameter δ (delta) is defined by

```
y ~ dnchisqr(k, delta)
```

for $k > 0$ and $\delta > 0$.

The non-central chi-squared distribution is equivalent to a Poisson mixture of chi-squared distributions:

```
y ~ dchisqr(k + 2*r)
r ~ dpois(delta/2)
```

It reduces to the chi-squared distribution when $\delta = 0$. See page 44.

Non-central t

The non-central Student t-distribution is defined by

```
y ~ dnt(mu, tau, df)
```

for $\tau > 0$ and $df > 0$. The probability density function has no simple closed-form expression, but the distribution can be defined constructively in terms of underlying normal and gamma distributions

```
Y <- U/sqrt(V)
U ~ dnorm(mu, tau)
V ~ dgamma(df/2, df/2)
```

The non-central t distribution is usually only defined for $\tau = 1$, in which case μ is the non-centrality parameter. The 3-parameter form here is a scaled version of a standard non-central t with non-centrality parameter $\delta = \mu\,\tau^{1/2}$. Due to a built-in limitation in the R math library, the absolute value of δ must be less than 37.62.

Normal

The normal (or Gaussian) distribution with mean μ (mu) and precision τ (tau) is defined by

```
y ~ dnorm(mu,tau}
```

for $\tau > 0$. Its probability density function is

$$\left(\frac{\tau}{2\pi}\right)^{\frac{1}{2}} \exp\{-\tau(y-\mu)^2/2\}$$

The variance of the normal distribution is $1/\tau$.

Pareto

The Pareto distribution with shape parameter α (alpha) and scale parameter c is defined by

```
y ~ dpar(alpha, c)
```

for $\alpha > 0, c > 0$. It has probability density

$$\alpha c^{\alpha} y^{-(\alpha+1)}$$

for $y \geq c$. The shape parameter α is also known as the *tail index* because it determines the behaviour of the tail of the distribution:

$$P(Y \geq y) = \left(\frac{c}{y}\right)^{\alpha}$$

The Pareto distribution has mean $\alpha c/(\alpha - 1)$ if $\alpha > 1$, otherwise the mean is infinite because the tail decays to zero very slowly. Similarly, the variance is finite only for $\alpha > 2$.

Student t

The Student t-distribution with location μ (mu), precision τ (tau) and degrees of freedom k is defined by

```
y ~ dt(mu,tau,k)
```

for $\tau > 0, k > 0$. It has probability density function

$$\frac{\Gamma(\frac{k+1}{2})}{\Gamma(\frac{k}{2})} \left(\frac{\tau}{k\pi}\right)^{\frac{1}{2}} \left\{1 + \frac{\tau(y-\mu)^2}{k}\right\}^{-\frac{(k+1)}{2}}$$

The t-distribution is equivalent to a scale mixture of normals:

```
y <- x/sqrt(s)
x ~ dnorm(mu, tau)
s ~ dgamma(k/2, k/2)
```

As $k \to \infty$ the t-distribution tends to the normal distribution with mean μ and precision τ.

For $k \leq 1$ all moments of the t-distribution are undefined. If $k > 1$ then the expectation is μ. The variance is $k/(\tau * (k-2))$ for $k > 2$ but for $1 < k \leq 2$ it is infinite.

The t-distribution is often represented as a one-parameter family indexed by the degrees of freedom parameter k. This is due to its origins as the distribution of Student's t-statistic. In the BUGS language the t-distribution also has location and precision parameters so that it can be used as an alternative to the normal distribution when heavier tails are required. A typical choice is $k = 4$ degrees of freedom, given finite mean and variance and zero skewness, but infinite kurtosis.

Uniform

The uniform distribution with limits a, b is defined by

```
y ~ dunif(a,b)
```

for $a < b$. It has probability density function

$$\frac{1}{b-a}$$

for $a \leq y \leq b$.

Weibull

The Weibull distribution with shape parameter v and rate parameter λ (lambda) is defined by

```
y ~ dweib(v, lambda)
```

for $v > 0, \lambda > 0$. For compatibility with R, the alias `dweibull` may also be used. Note, however, that the parameterization of the Weibull distribution in JAGS is different from the one in R, which uses a similar parameterization to the generalized gamma. See below.

Its probability density function is

$$v\lambda y^{v-1} \exp(-\lambda y^v)$$

The Weibull distribution is equivalent to a power transformation of an exponential distribution with rate λ.

```
y <- z^v
z ~ dexp(lambda)
```

When the `dweib` distribution is used as a survival distribution for an event time T, the hazard function is

$$h(t) = \lambda v t^{v-1}$$

Depending on the value of the shape parameter v, the hazard can be increasing ($v > 1$), decreasing ($v < 1$) or constant ($v = 1$) with time t. The parameter λ can be interpreted as a hazard ratio parameter (relative to the baseline hazard $h_0(t) = vt^{v-1}$ with $\lambda = 1$).

The Weibull distribution has an alternate parameterization for the accelerated life model. This is implemented `dgen.gamma` distribution when the first shape parameter set to 1.

```
y ~ dgen.gamma(1, lambda, b)
```

The hazard function is then

$$h(t) = v\lambda^b t^{b-1}$$

With the `dgen.gamma` distribution λ is not a hazard ratio. Instead, it is the factor by which the failure time T is accelerated as can be seen from the survival function

$$P(T \geq t) = \exp\left\{-(\lambda t)^b\right\}$$

9.2.2 Discrete univariate distributions

Bernoulli

The Bernoulli distribution (named after Swiss mathematician Jacob Bernoulli 1655-1705) with probability parameter p is defined by

```
y ~ dbern(p)
```

for $0 < p < 1$. The `dbern` distribution has probability mass function

$$\begin{aligned} P(Y = 1) &= p \\ P(Y = 0) &= 1 - p \end{aligned}$$

The Bernoulli distribution is a special case of the binomial distribution with size parameter equal to 1.

Binomial

The binomial distribution with probability parameter p and integer size parameter n is defined by

```
y ~ dbin(p,n)
```

for $0 < p < 1$ and $n \geq 1$. For compatibility with R, the alias `dbinom` may also be used. The probability mass function is

$$P(Y = y) = \binom{n}{x} p^y (1-p)^{n-y}$$

for $y = 0, 1, \ldots n$.

Categorical

The categorical distribution with probability parameter π (pi) is defined by

```
y ~ dcat(pi)
```

where π is a vector of non-negative probability weights of length N. The `dcat` distribution has probability mass function

$$P(Y = y) = \frac{\pi_y}{\sum_{i=1}^{N} \pi_i}$$

for $y = 1, \ldots N$. Note that the probability weights π do not need to sum to 1 (*i.e.* they are unnormalized), but at least one of the weights must be non-zero so that $\sum_i \pi_i > 0$.

Hypergeometric

The hypergeometric distribution can be represented as

```
y1 ~ dhyper(n1,n2,m1,psi)
```

where n_1, n_2, m_1 are all integers and ψ (psi) is a continuous parameter. The restrictions on the possible parameter values are $n_1 > 0$, $n_2 > 0$, $m_1 > 0$, $m_1 \leq n_1 + n_2$ and $\psi > 0$.

The hypergeometric distribution is simplest to describe when the odds ratio parameter ψ is equal to 1. The distribution can then be explained in terms of an urn model represented in the 2×2 table below.

	white	black	total
drawn	y_1	y_2	m_1
remaining	$n_1 - y_1$	$n_2 - y_1$	$n_1 + n_2 - m_1$
	n_1	n_2	$n_1 + n_2$

Suppose that an urn contains n_1 white balls and n_2 black balls. A total of m_1 balls are drawn from the urn without replacement. Then y_1, the number of white balls drawn from the urn, has a hypergeometric distribution

```
y1 ~ dhyper(n1,n2,m1,1)
```

The non-central hypergeometric distribution with odds ratio parameter $\psi \neq 1$ arises when m_1 is the sum of two binomial distributions

```
m1 ~ sum(y1, y2)
y1 ~ dbin(p1, n1)
y2 ~ dbin(p2, n2)
```

If m_1 is observed, the conditional distribution of y_1 has a non-central hypergeometric distribution with odds ratio parameter

$$\psi = \frac{p_1(1-p_2)}{(1-p_1)p_2}$$

There are two important things to note. Firstly, the value of m_1 is not fixed *a priori* but is a random variable. Secondly, the hypergeometric distribution does not depend on the absolute values of the underlying binomial probability parameters p_1, p_2 but only their odds ratio ψ.

The hypergeometric distribution has probability mass function

$$P(Y = y_1) = \frac{\binom{n_1}{y_1}\binom{n_2}{m_1 - y_1}\psi^{y_1}}{\sum_i \binom{n_1}{i}\binom{n_2}{m_1 - i}\psi^i}$$

for $\max(0, n_1 + n_2 - m_1) \leq y_1 \leq \min(n_1, m_1)$. The rather complex expressions for the lower and upper limits of y_1 are derived from the restriction that none of the cells in the 2×2 table above can be negative.

Negative binomial

The negative binomial distribution with probability parameter p and size parameter r is defined by

```
y ~ dnegbin(p, r)
```

for $0 < p \leq 1$ and $r \geq 0$. For compatibility with R, the alias `dnbinom` may also be used. The probability mass function is

$$P(Y = y) = \binom{y + r - 1}{y} p^r (1-p)^y$$

For $y = 0, 1, 2, \ldots$.

The size parameter r does not need to be an integer. However when r is an integer, the negative binomial distribution can be interpreted in terms of a series of Bernoulli trials, *i.e.* independent experiments with a binary outcome where the probability of "success" is p. The number of failures that occur before r successes are achieved has a negative binomial distribution.

The negative binomial distribution has mean $r(1-p)/p$ and variance $(1-p)r/p^2$. It collapses to a single point if either $r = 0$ or $p = 1$. In this case $P(Y = 0) = 1$. Conversely, the value $p = 0$ is not permitted (even if $r = 0$) because "success" in the Bernoulli trials is then impossible.

Both the binomial and negative binomial distributions can be explained in terms of Bernoulli trials. For the binomial distribution (page 50) the number of trials is fixed and the number of success is random; for the negative binomial the number of successes is fixed and the number of trials is random.

Poisson

The Poisson distribution (named after the French mathematician Siméon Denis Poisson 1781–1840) with rate λ (lambda) is defined by

```
y ~ dpois(lambda)
```

for $\lambda > 0$. It has probability mass function

$$P(Y = y) = \frac{\exp(-\lambda)\lambda^y}{y!}$$

for $y = 0, 1, 2 \ldots$.

The Poisson distribution has expectation λ and variance λ.

9.2.3 Multivariate distributions

Dirichlet

The Dirichlet distribution (named after German mathematician Peter Gustave Lejeune Dirichlet[1] 1805–1859) is defined by

```
y[1:N] ~ ddirch(alpha[1:N])
```

where alpha (α) is a vector of non-negative prior weights. The outcome y satisfies $y_i \geq 0$ and $\sum_i y_i = 1$ (i.e. the Dirichlet distribution is defined on the N-dimensional simplex).

The name `ddirch` is due to a historical misspelling of the name Dirichlet in WinBUGS. The distribution was renamed `ddirich` in OpenBUGS and so this alias is also allowed in JAGS.

When $\alpha_j > 0$ for all elements j of the vector α, the probability density function of the Dirichlet distribution is

$$\Gamma(\sum_i \alpha_i) \prod_j \frac{y_j^{\alpha_j - 1}}{\Gamma(\alpha_j)}$$

JAGS also allows some of the prior weights to be zero representing structural zeros. The corresponding elements of the outcome are fixed to zero (*i.e.* $y_j = 0$ if $\alpha_j = 0$) At least one element of α must be non-zero so that $\sum_j \alpha_j > 0$.

The outcome y represents a probability distribution over the integers $1 \ldots N$. The Dirichlet distribution is often used as a prior distribution for the probability parameters of the `dcat` or `dmulti` distribution, e.g.

```
z ~ dcat(pi[1:N])
pi[1:N] ~ ddirch(alpha[1:N])
```

Note however, that these distributions also accept a vector of unnormalized probability weights. In this context, it is worth noting that the Dirichlet distribution can be constructed by generating independent gamma random and then normalizing them.

[1] A note on pronunciation. Although the name Lejeune Dirichlet is of French origin, the *ch* is pronounced with a hard "k" sound and not a soft "sh" as it would be in French

```
for (i in 1:n) {
   pi[i] <- w[i]/sum(w[1:N])
   w[i] ~ dgamma(alpha[i], 1)
}
```

Hence an equivalent prior on the `dcat` distribution is

```
z ~ dcat(w[1:N])
for (i in 1:N) {
   w[i] ~ dgamma(alpha[i], 1)
}
```

Using the `ddirch` distribution is faster, however, as JAGS recognizes it as the conjugate prior.

Multivariate Normal

There are two distributions in the bugs module representing the multivariate normal distribution. The main one is `dmnorm` defined by

```
y[1:N] ~ dmnorm(mu[1:N], Omega[1:N, 1:N])
```

where mu (μ) is a vector of mean parameters of length N and Omega (Ω) is an $N \times N$ symmetric positive-definite precision matrix.

The probability density function is

$$|\Omega|^{\frac{1}{2}} (2\pi)^{-\frac{N}{2}} \exp\{-(y-\mu)^T \Omega (y-\mu)/2\}$$

The mean of the multivariate normal distribution is μ and its variance-covariance matrix is Ω^{-1}.

Sometimes is more convenient to express the multivariate normal distribution in terms of its variance-covariance matrix Σ (Sigma), which is the inverse of the precision matrix ($\Sigma = \Omega^{-1}$). While the following construction should work in theory

```
y ~ dnorm(mu, Omega)
Omega <- inverse(Sigma)
```

in practice it may lead to runtime errors if Σ cannot be inverted. In such situations it is better to use `dmnorm.vcov`, which is parameterized in terms of the mean and the variance-covariance matrix.

```
y[1:N] ~ dmnorm.vcov(mu[1:N], Sigma[1:N, 1:N])
```

Multivariate t

The multivariate t distribution is a generalization of the Student t-distribution (page 48) to more than one dimension. It is defined by:

```
y[1:N] ~ dmt(mu[1:N], Omega[1:N, 1:N], k)
```

where μ (mu) is a vector of location parameters of length N, Ω (Omega) is an $N \times N$ symmetric positive-definite precision matrix and $k > 0$ is the number of degrees of freedom.

The probability density function of the multivariate t-distribution is

$$\frac{\Gamma\{(k+N)/2\}}{\Gamma(k/2)(n\pi)^{N/2}}|\Omega|^{1/2}\left\{1+\frac{1}{k}(y-\mu)^T\Omega(y-\mu)\right\}^{-\frac{(k+N)}{2}}$$

The multivariate t-distribution can be defined constructively in terms of underlying multivariate normal and gamma random variables. It is equivalent to:

```
for (i in 1:N) {
  y[i] <- z[i]/sqrt(S[i])
  S[i] ~ dgamma(k/2, k/2)
}
z[1:N] ~ dnorm(mu[1:N], Omega[1:N, 1:N])
```

When *Omega* is a diagonal matrix then the elements of y are independent and the distribution is equivalent to

```
for (i in 1:N) {
   y[i] ~ dt(mu[i], Omega[i,i], k)
}
```

Note that the comments about the moments of the t-distribution `dt` for small values of k also apply to the multivariate t-distribution (See page 48).

Multinomial

The multinomial distribution represents sampling with replacement.

```
y[1:N] ~ dmulti(pi[1:N], k)
```

for pi (π), a vector of unnormalized probability weights of length N, and integer $k \geq 1$ giving the number of samples to draw. The outcome y is a vector of non-negative integers, counting the number of times each element was sampled. The outcome always satisfies $\sum_i y_i = k$.

The probability mass function for the multinomial distribution is

$$k! \prod_{j=1}^{N} \frac{\pi_j^{x_j}}{y_j!}$$

For sampling without replacement, see the `dsample` distribution below.

Sampling with replacement

The `dsample` distribution represents sampling without replacement

```
y[1:N] ~ dsample(pi[1:N], k)
```

where `pi` (π) is a vector of unnormalized probability weights and k is an integer in the range $1 \ldots N$ giving the number of samples to draw. The outcome y, is a vector of binary indicators of length N, so that $y_i = 1$ if element i has been sampled and $y_i = 0$ otherwise.

For sampling with replacement, see the multinomial distribution.

Wishart

The Wishart distribution (named after Scottish mathematician John Wishart 1898 – 1956) is defined by

```
Omega ~ dwish(R,k)
```

for positive definite $m \times m$ matrix R and degrees of freedom $k \geq m$. The resulting matrix Ω (Omega) is an $m \times m$ positive-definite matrix.

The probability density function is

$$\frac{|\Omega|^{(k-m-1)/2}|R|^{k/2}\exp\{-\mathrm{Tr}(R\Omega/2)\}}{2^{mk/2}\Gamma_p(k/2)}$$

where Γ_p is the multivariate gamma function.

The Wishart distribution is often used as a prior for the precision of a multivariate normal distribution. It is therefore worth spending some time on the interpretation of the parameters R and k in this context. There is an implicitly defined prior on the variance-covariance matrix $\Sigma = \Omega^{-1}$, which has an inverse-Wishart distribution. The expectation of Ω is kR^{-1}. Thus R/k is a prior guess for the variance-covariance matrix Σ. When R is diagonal and $k = m+1$ the correlation parameters

$$\rho_{ij} = \frac{\sigma_{ij}}{\sqrt{\sigma_{ii}\sigma_{jj}}}$$

for $i \neq j$ have a uniform distribution on the interval $[-1, 1]$. For $k > m + 1$ the prior on ρ_{ij} is concentrated around 0, so that larger values of k indicate stronger prior belief that the elements of the multivariate normal distribution are independent. For $k = m$ the Wishart prior puts most weight on the extreme values $\rho_{ij} = 1$ and $\rho_{ij} = -1$. This is not recommended unless, of course, you have prior information that the correlations are strong.

9.2.4 Observable functions in the bugs module

Observable functions (section 7.2) are used in JAGS to represent observations that are *deterministic* functions of unobserved parameters.

Interval censoring: `dinterval`

The observable function `dinterval` is used in JAGS to represent censored outcomes. Before explaining how to set up such a model it is first necessary to explain how `dinterval` works as a function and as a distribution.

When `dinterval` is used as a function it divides up a continuous variable into categories according to a given set of cut-points, similar to the `cut` function in R:

```
y <- dinterval(t, c[1:M])
```

where $c_1 \ldots c_M$ is an increasing vector of cut points. The value of y is given by the table below.

$y = 0$	if	$t \leq c_1$
$y = 1$	if	$c_1 < t \leq c_2$
$\ldots$		
$y = M - 1$	if	$c_{M-1} < t \leq c_M$
$y = M$	if	$c_M < t$.

When `dinterval` is used as a distribution:

```
y ~ dinterval(t, c[1:M])
```

it generates a likelihood $P(y \mid t, c)$ for the parameters c and t that takes the value 1 when t lies within an interval defined by the cutpoints $c_1 \ldots c_M$ and 0 otherwise. The lower and upper limits of the interval are defined by the table below:

Value of y	Lower limit	Upper limit
0	$-\infty$	c_1
1	c_1	c_2
...		
M-1	c_{M-1}	c_M
M	c_M	∞

More succinctly, if we define $c_0 = -\infty$ and $c_{M+1} = \infty$ then

$$P(y \mid t, c) = \begin{cases} 1 & \text{if } c_y < t \leq c_{y+1} \\ 0 & \text{otherwise} \end{cases}$$

When t is unobserved, the likelihood from the `dinterval` distribution imposes the *a posteriori* restriction that t must lie in the interval $c_y < t \leq c_{y+1}$.

Censored data occur when outcomes are not observed directly but are known only to be above a certain value (right-censoring), or below a certain value (left-censoring), or in between two values (interval-censoring). The `dinterval` distribution can be used to represent all three forms of censoring.

To set up a right-censored survival model the failure time t_j is augmented with a *potential censoring time* c_j and a binary censoring indicator I_j for each observation.

```
I[j] ~ dinterval(t[j], c[j])
```

The model is set up as follows:

- When the failure time is uncensored then t_j is *observed* and the censoring indicator I_j is observed with value zero.
- When the failure time is right-censored, *i.e.* we know only that $t_j > c_j$, then t_j is *unobserved* and the censoring indicator I_j is observed with value one.

All observations must have a potential censoring time c_j, even if the failure time t_j is not in fact censored. In survival analysis there is always a maximum value of the failure time t_{max} defined by the end of the observation period. So set $c_j = t_{max}$ for uncensored failure times.

The example below shows the data for a simple example with 5 observation. Observations 3 and 4 are censored at times `c[3] = 2.1` and `c[4] = 0.7` respectively. Observations 1,2, and 5 are uncensored.

```
t <- c(1.2, 4.7, NA, NA, 3.2)
c <- c(tmax, tmax, 2.1, 0.7, tmax)
I <- c(0, 0, 1, 1, 0)
```

A practical disadvantage of the `dinterval` distribution is that initial values must be set for the censored times. In the above example, suitable starting values would be.

```
t <- c(NA, NA, 2.2, 0.8, NA)
```

Here the censored times are set to a value slightly higher than the censoring time. Failure to set appropriate initial values for censored times when using the `dinterval` distribution will result in the error message "invalid parent values"for the censoring indicator.

Further examples of right-censored survival data can be found in the MICE and KIDNEY examples in the "classic bugs" set of examples.

Rounding: `dround`

The `dround` distribution represents rounded data. It has two parameters: t the original continuous variable and d, the number of decimal places to which the measurements are rounded. Thus if $t = 1.2345$ and $d = 2$ then the rounded value is 1.23. If $d = 0$ then `dround` rounds to the nearest whole number and of $d < 0$ it rounds to a power of 10, e.g. if $d = -2$ then the data are rounded to the nearest 100.

When used as a function

```
y <- dround(t, d)
```

the result y is equal to t rounded to d decimal places. When used as a distribution

```
y ~ dround(t, d)
```

`dround` generates a likelihood for parameters t, d that is equal to 1 if y is equal to t rounded to d decimal places, and 0 otherwise. A small amount of tolerance for "near equality" is allowed in this comparison to allow for the fact that, in general, decimal fractions cannot be represented exactly.

Summing: `sum`

The `sum` function (See table 9.2) is a scalar-valued function that returns the sum of all of its arguments.

```
y <- sum(x1, x2, x3)
```

The `sum` function takes a variable number of arguments. If an argument is a vector then all of its elements are summed together.

You can also use `sum` to represent the observed sum of two or more unobserved nodes

```
y ~ sum(x1, x2, x3)
```

where the arguments must be either all discrete or all continuous. An example can be seen on page 50, where the hypergeometric distribution is represented using the observed sum of two unobserved binomials.

The use of `sum` as an observable function is facilitated by an associated sampler `bugs::Sum`. If at least two of the arguments to `sum` are unobserved, then the sampler will update them together ensuring that the sum constraint is preserved. Furthermore, if one of the arguments is itself the sum of unobserved stochastic nodes, then the `bugs::Sum` sampler will update them together. For example, if

```
y ~ dsum(x1, z)
z <- x2 + x3
```

then the sampler will update `x1`, `x2`, and `x3` together. The sampler will also try to fix the initial values so that the constraint is satisfied from the first iteration. If it fails to do this then JAGS will stop with an error.

There are limits to what the `bugs::Sum` sampler can do. It cannot update a node that is subject to more than one `sum` constraint.

9.3 Samplers in the bugs module

`bugs::Censored`

The `bugs::Censored` factory creates samplers for nodes that are censored by the `dinterval` distribution (See page 55). If the node has a distribution that can be bounded then the `bugs::Censored` sampler draws samples from the truncated distribution.

`bugs::Sum`

The `bugs::Sum` factory creates samplers for nodes that are constrained by the `sum` observable function. See page 57 for more details.

`bugs::Conjugate`

The `bugs::Conjugate` factory creates samplers for nodes with conjugate distributions, *i.e.* where the full conditional distribution of the node belongs to the same family as the prior.

`bugs::Dirichlet`

The `bugs::Dirichlet` factory creates samplers for nodes with a Dirichlet prior distribution when the posterior distribution is *not* conjugate. It does this by modelling the Dirichlet distribution as a set of normalized gamma distributions. (See page 52)

`bugs::MNormal`

The `bugs::MNormal` factory creates samplers for nodes with a multivariate normal prior distribution when the posterior distribution is *not* conjugate.

The `bugs::MNormal` sampler uses an adaptive random walk Metropolis algorithm. It can be very inefficient. In particular, it may require an extremely long adaptation period.

Chapter 10

The glm module

10.1 Distributions in the glm module

10.1.1 Ordered categorical distributions

Ordered logistic

The ordered logistic distribution with location parameter μ (mu) and c is defined by defined by

```
y ~ dordered.logit(mu, c[1:M])
```

where c is an ordered vector of cutpoints of length M. The `dordered.logit` distribution is a discrete distribution taking values in the range $1, \ldots M+1$, with probabilities that are determined by the cumulative distribution function of the logistic distribution. Let

$$F(y \mid \mu) = \frac{1}{1 - e^{y-\mu}}$$

Then

$$\begin{array}{lcl} P(Y=1) & = & F(c_1 \mid \mu) \\ P(Y=y) & = & F(c_y \mid \mu) - F(c_{y-1} \mid \mu) \quad \text{for} y = 2 \ldots M \\ P(Y=M+1) & = & 1 - F(c_M \mid \mu) \end{array}$$

If the location parameter μ is a linear predictor then the `dordered.logit` distribution can be used to represent the ordered logistic regression model [McCullagh, 1980]. For example, suppose

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

where x is a binary variable. Then the coefficient β represents the log odds ratio

$$\beta = \log \left\{ \frac{P\left(Y \leq y \mid x = 1\right)}{P\left(Y > y \mid x = 1\right)} \frac{P\left(Y > y \mid x = 0\right)}{P\left(Y \leq y \mid x = 0\right)} \right\}$$

which is constant for $y = 1 \ldots M$

Ordered probit

The ordered probit distribution with location parameter μ (mu) and c is defined by defined by

```
y ~ dordered.probit(mu, c[1:M])
```

where c is an ordered vector of cutpoints of length M. The `dordered.probit` distribution is a discrete distribution taking values in the range $1, \ldots M+1$, and is therefore similar to the ordered logistic distribution (`dordered.logit`), except that the probabilities $P(Y = y)$ are determined by the cumulative distribution function of the normal distribution

$$F(y \mid \mu) = \int_{-\infty}^{y} \frac{1}{\sqrt{2\pi}} \exp\left\{-(x-\mu)^2/2\right\} dx$$

When μ is a linear predictor then the `dordered.probit` distribution represents the ordered probit regression model.

10.1.2 Distributions for precision parameters

One of the difficulties with the normal distribution in the BUGS language is that it is parameterized in terms of the precision, and not in terms of the variance or standard deviation. The same is true of other location-scale distributions (*e.g.* Student t, logistic, double exponential). The reasons for this are historical. When BUGS was first developed, it exploited conjugate distributions for maximum sampling efficiency. It turns out that conjugate priors are most easily expressed in terms of the precision. The conjugate prior for the precision of the normal distribution is the gamma distribution.

```
for (i in 1:N) {
   eps[i] ~ dnorm(0, tau)
}
tau ~ dgamma(shape, rate)
```

For the multivariate normal, the conjugate prior is the Wishart distribution on the precision matrix (inverse of the variance-covariance matrix)

```
for (i in 1:N) {
   eps[i, 1:M] ~ dmnorm(rep(0,M), tau)
}
tau ~ dwish(R, df)
```

Thus the BUGS language allows these conjugate priors to be expressed very simply. However, nobody has an intuitive understanding of what a prior distribution on the precision means. The choice of parameters `shape` and `rate` or `R` and `df` can be difficult.

The `glm` module provides two distributions for precision parameters that are parameterized in a way that is intuitively easier to understand. Associated samplers in the `glm` module ensure that the distributions are sampled efficiently, especially when they are used as priors for the precision of random effects.

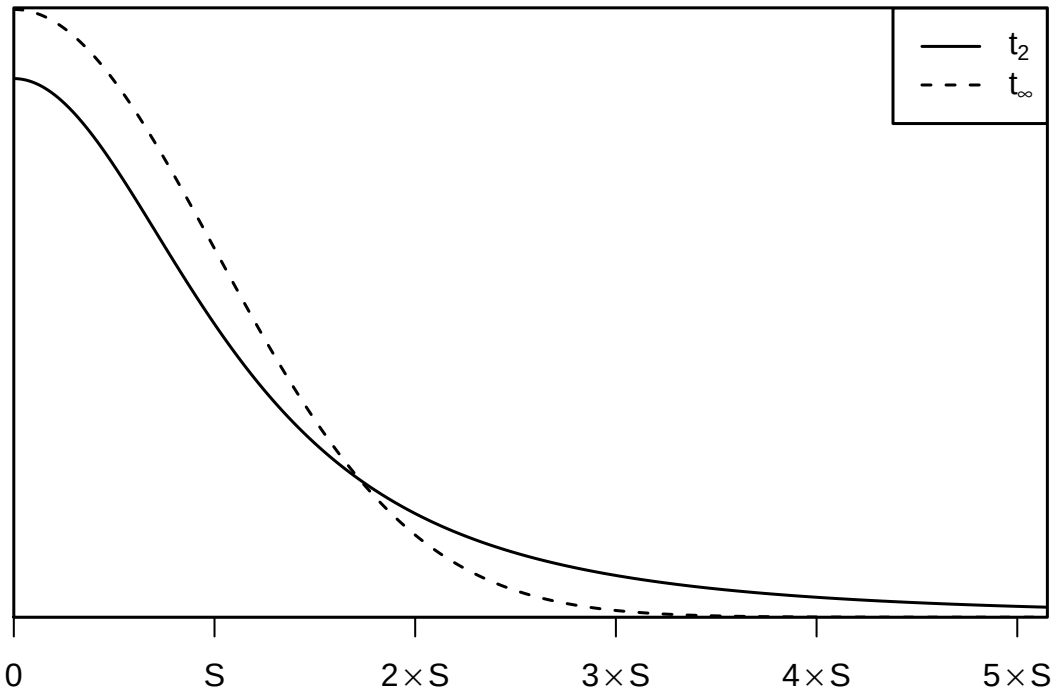

Figure 10.1: The half-t distribution with scale S and 2 degrees of freedom (solid line). The dotted line shows the limit as the degrees of freedom increases to infinity

Scaled gamma: `dscaled.gamma`

The scaled gamma distribution with scale s and degrees of freedom df is defined by

```
tau ~ dscaled.gamma(s, df)
```

where τ (tau) is a positive scalar-valued random variable.

If τ is the precision parameter of the normal distribution, then `dscaled.gamma` implicitly defines a prior on the standard deviation $\sigma = 1/\sqrt{\tau}$):

$$\sigma \sim s t^+_{df}$$

where t^+_{df} represents the t-distribution with df degrees of freedom restricted to positive values.

Figure 10.1 illustrates the half-t distribution with 2 degrees of freedom. The density is flat for very small values ($\sigma \ll S$) and has a long flat tail for large values ($\sigma \gg S$). These features protect against mis-specification of the prior scale S. Choosing a large S is harmless as it simply makes the prior flatter, whereas if S is too small then the long flat tail ensures that sufficient data will dominate the prior. Gelman [2006] calls this a "weakly informative" prior. For comparison, figure 10.1 also shows the density on σ in the limit as the degrees of freedom df increases to infinity. The distribution then becomes a half-normal with standard deviation S. The effect of increasing the degrees of freedom is to diminish the weight of the tail.

Scaled Wishart: `dscaled.wishart`

The scaled Wishart distribution is defined by

```
Omega[1:M,1:M] ~ dscaled.wishart(s[1:M], df)
```

where s is a vector of prior scales of length M, *df* is the degrees of freedom, and Ω (Omega) is an $M \times M$ positive definite symmetric matrix (If you want all elements to have the same prior scale then use `rep(s, M)` as the first parameter).

The scaled Wishart is a multivariate generalization of the scaled gamma distribution. It was proposed by Huang and Wand [2013] as a flexible family of prior distributions for the precision parameter of the multivariate normal. When used in this way, the `dscaled.wishart` distribution implicitly defines a prior on the variance-covariance matrix $\Sigma = \Omega^{-1}$ with two important features:

1. Let $\sigma_i = (\Sigma_{ii})^{1/2}$ be the standard deviation of element i of the multivariate normal. Then σ_i has a half-t distribution with scale s_i and *df* degrees of freedom

2. Let $\rho_{ij} = \Sigma_{ij}/(\sigma_i \sigma_j)$ for $i \neq j$ be the correlation between elements i and j of the multivariate normal. For $df = 2$, all correlation parameters ρ_{ij} have a marginal uniform prior distribution. Larger values of *df* concentrate the prior around $\rho_{ij} = 0$.

10.2 Samplers in the glm module

The `glm` module implements samplers for efficient updating of generalized linear mixed models. The fundamental idea is to do block updating of the parameters in the linear predictor. The `glm` module is built on top of the Csparse and CHOLMOD sparse matrix libraries [Davis, 2006, Davis and Hager, 1999] which allows updating of both fixed and random effects in the same block. Currently, the methods only work on parameters that have a normal prior distribution.

Most of the samplers in the `glm` module are based in the idea of introducing latent variables that reduce the GLM to a linear model with normal outcomes. This idea was introduced by Albert and Chib [1993] for probit regression with a binary outcome, and was later refined and extended to logistic regression with binary outcomes by Holmes and Held [2006]. Another approach, auxiliary mixture sampling, was developed by Frühwirth-Schnatter et al. [2009] and is used for Poisson regression and logistic regression models with binomial outcomes. Polson et al. [2013] proposed an alternative data augmentation scheme for binary logistic regression models based on the Pólya-gamma distribution. Most of these data-augmentation schemes are compatible with each other. This means that you can have a heterogeneous glm with outcomes that are normal (linear), binary (logit or probit link), binomial (logit link), Poisson (log link), t (linear), or multivariate normal (linear).

The `glm` module also includes an alternative sampler for glms based on a proposal by Gamerman [1997]. It uses a stochastic version of the iteratively weighted least squares (IWLS) algorithm. However the IWLS sampler tends to break down when there are many random effects in the model. It uses Metropolis-Hastings updates, and the acceptance probability may be very small under these circumstances. Despite the poor performance of the IWLS sampler it remains in the `glm` module (with low priority) because in a future version of JAGS it can easily be converted to a Metropolis-adjusted Langevin algorithm (MALA).

Block updating in GLMMs frees the user from the need to centre predictor variables, like this:

```
y[i] ~ dnorm(mu[i], tau)
mu[i] <- alpha + beta * (x[i] - mean(x))
```

The second line can simply be written

```
mu[i] <- alpha + beta * x[i]
```

without affecting the mixing of the Markov chain.

Chapter 11

The dic module

11.1 Monitors in the dic module

The `dic` module defines new monitor classes for Bayesian model criticism using deviance-based measures.

11.1.1 The deviance monitor

The deviance monitor records the deviance of the model (*i.e.* the sum of the deviances of all the observed stochastic nodes) at each iteration. The command

```
monitor deviance
```

will create a deviance monitor *unless* you have defined a node called "deviance" in your model. In this case, you will get a trace monitor for your deviance node.

11.1.2 The `pD` monitor

The `pD` monitor is used to estimate the effective number of parameters (p_D) of the model [Spiegelhalter et al., 2002]. It requires at least two parallel chains in the model, but calculates a single estimate of p_D across all chains [Plummer, 2002]. A pD monitor can be created using the command:

```
monitor pD
```

Like the deviance monitor, however, if you have defined a node called "pD" in your model then this will take precedence, and you will get a trace monitor for your `pD` node.

Since the p_D monitor pools its value across all chains, its values will be written out to the index file "CODAindex0.txt" and output file "CODAoutput0.txt" when you use the CODA command.

The effective number of parameters is the sum of separate contributions from all observed stochastic nodes: $p_D = \sum_i p_{D_i}$. There is also a monitor that stores the sample mean of p_{D_i}. These statistics may be used as influence diagnostics [Spiegelhalter et al., 2002]. The mean monitor for p_{D_i} is created with:

```
monitor pD, type(mean)
```

Its values can be written out to a file "PDtable0.txt" with

```
coda pD, stem(PD)
```

11.1.3 The `popt` monitor

The `popt` monitor works exactly like the mean monitor for p_D, but records contributions to the optimism of the expected deviance (p_{opt_i}). The total optimism $p_{opt} = \sum_i p_{opt_i}$ can be added to the mean deviance to give the penalized expected deviance [Plummer, 2008].

The mean monitor for p_{opt_i} is created with

```
monitor popt, type(mean)
```

Its values can be written out to a file "POPTtable0.txt" with

```
coda popt, stem(POPT)
```

Under asymptotically favourable conditions in which $p_{D_i} \ll 1 \forall i$,

$$p_{opt} \approx 2p_D$$

For generalized linear models, a better approximation is

$$p_{opt} \approx \sum_{i=1}^{n} \frac{p_{D_i}}{1 - p_{D_i}}$$

The `popt` monitor uses importance weights to estimate p_{opt}. The resulting estimates may be numerically unstable when p_{D_i} is not small. This typically occurs in random-effects models, so it is recommended to use caution with the `popt` monitor until I can find a better way of estimating p_{opt_i}.

Chapter 12

The mix module

The `mix` module defines a novel distribution `dnormmix(mu,tau,pi)` representing a finite mixture of normal distributions. In the parameterization of the `dnormmix` distribution, μ, τ, and π are vectors of the same length, and the density of `y ~ dnormmix(mu, tau, pi)` is

$$f(y|\mu,\tau,\pi) = \sum_i \pi_i \tau_i^{\frac{1}{2}} \phi(\tau_i^{\frac{1}{2}}(y-\mu_i))$$

where $\phi()$ is the probability density function of a standard normal distribution.

The `mix` module also defines a sampler that is designed to act on finite normal mixtures. It uses tempered transitions to jump between distant modes of the multi-modal posterior distribution generated by such models [Neal, 1994, Celeux et al., 1999]. The tempered transition method is computationally very expensive. If you want to use the `dnormmix` distribution but do not care about label switching, then you can disable the tempered transition sampler with

```
set factory "mix::TemperedMix" off, type(sampler)
```

on the command line, or

```
set.factory("mix::TemperedMix", FALSE, type="sampler")
```

using the `rjags` interface, or

```
run.jags(..., modules="mix", factories="mix::TemperedMix sampler off")
```

using the `runjags` interface.

Chapter 13

The msm module

The `msm` module defines the matrix exponential function `mexp` and the multi-state distribution `dmstate` which describes the transitions between observed states in continuous-time multi-state Markov transition models.

Chapter 14

The lecuyer module

The `lecuyer` module provides the RNG factory `lecuyer::RngStream` that implements the random number generator of L'Ecuyer et al. [2002]. It is capable of generating a large number of independent RNG streams and is therefore useful when running many parallel chains.

Appendix A

Differences between JAGS and OpenBUGS

Although JAGS aims for the same functionality as OpenBUGS, there are a number of important differences.

A.1 Data format

There is no need to transpose matrices and arrays when transferring data between R and JAGS, since JAGS stores the values of an array in "column major" order, like R and FORTRAN (*i.e.* filling the left-hand index first).

If you have an S-style data file for OpenBUGS and you wish to convert it for JAGS, then use the command `bugs2jags`, which is supplied with the `coda` package.

A.2 Distributions

Structural zeros are allowed in the Dirichlet distribution. If

```
p ~ ddirch(alpha)
```

and some of the elements of alpha are zero, then the corresponding elements of p will be fixed to zero.

The Multinomial (`dmulti`) and Categorical (`dcat`) distributions, which take a vector of probabilities as a parameter, may use unnormalized probabilities. The probability vector is normalized internally so that

$$p_i \to \frac{p_i}{\sum_j p_j}$$

The non-central hypergeometric distribution (`dhyper`) uses the same parameterization as R, which is different from the parameterization used in OpenBUGS 3.2.2. OpenBUGS is parameterized as

```
X ~ dhyper(n, m, N, psi)      #OpenBUGS
```

where n, m, N are the following table margins:

x	-	n
-	-	-
m	-	N

This parameterization is symmetric in n, m. In JAGS, `dhyper` is parameterized as

```
X ~ dhyper(n1, n2, m1, psi) #JAGS
```

where $n1, n2, m1$ are

x	-	m1
-	-	-
n1	n2	-

A.3 Censoring and truncation

For compatibility with OpenBUGS, JAGS permits the use of `I(,)` for truncation when the the parameters of the truncated distribution are fixed. For example, this is permitted:

```
mu ~ dnorm(0, 1.0E-3) I(0, )
```

because the truncated distribution has fixed parameters (mean 0, precision 1.0E-3). In this case, there is no difference between censoring and truncation. Conversely, this is not permitted:

```
for (i in 1:N) {
   x[i] ~ dnorm(mu, tau) I(0, )
}
mu ~ dnorm(0, 1.0E-3)
tau ~ dgamma(1, 1)
}
```

because the mean and precision of $x_1 \ldots x_N$ are parameters to be estimated. JAGS does not know if the aim is to model truncation or censoring and so the compiler will reject the model. Use either `T(,)` or the `dinterval` distribution to resolve the ambiguity.

A.4 Data transformations

JAGS allows data transformations, but the syntax is different from BUGS. BUGS allows you to put a stochastic node twice on the left hand side of a relation, as in this example taken from the manual

```
for (i in 1:N) {
   z[i] <- sqrt(y[i])
   z[i] ~ dnorm(mu, tau)
}
```

This is forbidden in JAGS. You must put data transformations in a separate block of relations preceded by the keyword `data`:

```
data {
   for (i in 1:N) {
      z[i] <- sqrt(y[i])
   }
}
model {
   for (i in 1:N) {
      z[i] ~ dnorm(mu, tau)
   }
   ...
}
```

Bibliography

J. Albert and S. Chib. Bayesian analysis of binary and polychotomous response data. *Journal of the American Statistical Association*, 88:669–679, 1993.

G. Celeux, M. Hurn, and C. P. Robert. Computational and inferential difficulties with mixture posterior distributions. *Journal of the American Statistical Association*, 95:957–970, 1999.

T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

T. A. Davis and W. W. Hager. Modifying a sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 20:606–627, 1999.

M. J. Denwood. runjags: An R package providing interface utilities, model templates, parallel computing methods and additional distributions for MCMC models in JAGS. *Journal of Statistical Software*, 71(9):1–25, 2016. doi: 10.18637/jss.v071.i09.

S. Frühwirth-Schnatter, R. Frühwirth, L. Held, and H. Rue. Improved auxiliary mixture sampling for hierarchical models of non-gaussian data. *Statistics and Computing*, 19(4): 479–492, 2009.

D. Gamerman. Efficient sampling from the posterior distribution in generalized linear mixed models. *Statistics and Computing*, 7:57–68, 1997.

A. Gelman. Prior distributions for variance parameters in hierarchical models. *Bayesian Analysis*, 1:515–533, 2006.

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

C. Holmes and L. Held. Bayesian auxiliary variable models for binary and multinomial regression. *Bayesian Analysis*, 1(1):145–168, 2006.

A. Huang and M. Wand. Simple marginally noninformative prior distributions for covariance matrices. *Bayesian Analysis*, 8(2):439–452, 2013.

P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002.

D. Lunn, C. Jackson, N. Best, A. Thomas, and D. Spiegelhalter. *The BUGS Book A Practical Introduction to Bayesian Analysis*. CRC Press / Chapman and Hall, 2012.

P. McCullagh. Regression models for ordinal data (with discussion). *Journal of the Royal Statistical Society Series B*, 42:109–142, 1980.

R. Neal. Sampling from multimodal distributions using tempered transitions. *Statistics and Computing*, 6:353–366, 1994.

R. Neal. Slice sampling. *Annals of Statistics*, 31(3):705–767, 2003.

M. Plummer. Discussion of the paper by Spiegelhalter et al. *Journal of the Royal Statistical Society Series B*, 64:620, 2002.

M. Plummer. Penalized loss functions for Bayesian model comparison. *Biostatistics*, 9(3): 523–539, 2008.

M. Plummer. *rjags: Bayesian Graphical Models using MCMC*, 2016. URL `https://CRAN.R-project.org/package=rjags`. R package version 4-6.

M. Plummer, N. Best, K. Cowles, and K. Vines. Coda: Convergence diagnosis and output analysis for mcmc. *R News*, 6(1):7–11, 2006. URL `http://CRAN.R-project.org/doc/Rnews/`.

N. G. Polson, J. G. Scott, and J. Windle. Bayesian inference for logistic models using pólyagamma latent variables. *Journal of the American Statistical Association*, 108(504): 1339–1349, 2013. doi: 10.1080/01621459.2013.829001. URL `http://dx.doi.org/10.1080/01621459.2013.829001`.

D. Spiegelhalter, N. Best, B. Carlin, and A. van der Linde. Bayesian measures of model complexity and fit (with discussion). *Journal of the Royal Statistical Societey Series B*, 64: 583–639, 2002.

S. Sturtz, U. Ligges, and A. Gelman. R2winbugs: A package for running winbugs from r. *Journal of Statistical Software*, 12(3):1–16, 2005. URL `http://www.jstatsoft.org`.

Y.-S. Su and M. Yajima. *R2jags: Using R to Run 'JAGS'*, 2015. URL `https://CRAN.R-project.org/package=R2jags`. R package version 0.5-7.