JAGS Version 1.0.3 manual

Martyn Plummer

April 23, 2009

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 WinBUGS (`http://www.mrc-bsu.cam.ac.uk`). JAGS was written with three aims in mind: to have an engine for the BUGS language that runs on Unix; to be extensible, allowing users to write their own functions, distributions, and samplers; and to be a platform for experimentation with ideas in Bayesian modelling.

JAGS is designed to work closely with the R language and environment for statistical computation and graphics (`http://www.r-project.org`). In particular, you will need R to prepare the input data for JAGS, and you will find it useful to install the coda package for R to analyze the output.

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).

Chapter 2

Running a model in JAGS

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. Defining the model
2. Compiling
3. Initializing
4. Adapting and burning-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 analyzing MCMC output, and JAGS is designed for easy transition of data to and from R.

2.1 Definition

There are two parts to the definition of a model in JAGS: a description of the model and the definition of the data.

2.1.1 Model definition

The model is defined in a text file using a dialect of the BUGS language. 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 the standard 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)
}
```

Each relation defines a node in the model in terms of other nodes that appear on the right hand side. These are referred to as the parent nodes. Taken together, the nodes in the model (together with the parent/child relationships represented as directed edges) form a directed acyclic graph. 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 file (*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.

Nodes defined by a relation are embedded in named arrays. 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]`.

Deterministic nodes do not need to be embedded in node arrays. The node `Y[i]` could equivalently be defined as

```
Y[i] ~ dnorm(alpha + beta * (x[i] - x.bar), tau)
```

In this version of the model definition, the node previously defined as `mu[i]` still exists, but is not accessible to the user as it does not have a name. This ability to hide deterministic nodes by embedding them in other expressions underscores an important fact: only the stochastic nodes in a model are really important. Deterministic nodes are merely a syntactically convenient way of describing the relations between, or transformations of, the stochastic nodes.

2.1.2 Data

The data are defined in a separate file from the model definition, in the format created by the `dump()` function in R. 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 stripped when they are read into JAGS.

The data may contain missing values, but you cannot supply partially missing values for a multivariate node. In JAGS a node is either completely observed, or completely unobserved. The unobserved nodes are referred to as the *parameters* of the model. The data file therefore defines, by omission, the parameters of the model.

Here are the data for the `LINE` example:

```
‘x‘ <-
c(1, 2, 3, 4, 5)
#R-style comments, like this one, can be embedded in the data file
‘Y‘ <-
```

```
c(1, 3, 3, 3, 5)
‘N‘ <-
5
```

It is an error to supply a data value for a deterministic node. (See, however, section 7.0.4 on observable functions).

2.1.3 Node Array dimensions

Array declarations

JAGS allows the option of declaring the dimensions of node arrays in the model file. The declarations are modelled on classic BUGS, and 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;
```

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. Using this method, empty indices are not allowed on the left hand side of any 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.

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 7.0.5).

```
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.

2.2 Compilation

When a model is compiled, a graph representing the model is created in computer memory. Compilation can fail for a number of reasons:

1. The graph contains a directed cycle. These are forbidden in JAGS.
2. A top-level parameter is undefined. Any node that is used on the right hand side of a relation, but is not defined on the left hand side of any relation, is assumed to be a constant node. Its value must be supplied in the data file.
3. The model uses a function or distribution that has not been defined in any of the loaded modules.

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. By default, JAGS only runs a single chain.

2.3 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 initial value is set.
3. The Samplers are chosen for each parameter in the model.

2.3.1 Parameter values

The user may supply an initial value file containing values for the model parameters. The file may not contain values for logical or constant nodes. The format is the same as the data file (see section 2.1.2).

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 method of choosing the initial value depends on whether the node is classified as a fixed effect or a random effect.[1]

Fixed effects are parameters for which all parents are observed. In this case, 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.

Random effects are parameters for which one or more parents are unobserved. The initial values of these nodes are chosen by taking a random sample from the prior distribution

These heuristic rules are designed to create reasonable default starting values in a wide range of models.

If you rely on automatic initial value generation and are running multiple parallel chains, then the values for the top-level model parameters will be the same in all chains. You may not want this behaviour, especially if you are using the Gelman and Rubin convergence diagnostic, which assumes that the initial values are over-dispersed with respect to the posterior distribution. In this case, you are advised to set the starting values manually using the "parameters in" statement.

2.3.2 RNGs

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 in the initial values file.

The name of the RNG is set as follows.

```
.RNG.name <- "name"
```

[1]Jean Baptiste Denis points out that this distinction between fixed and random effects is a little artificial since it depends on the parameterization of the model. However, within the context of a single parameterization, defined by a particular BUGS-language representation of the model, the distinction is unambiguous.

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 an integer value to `.RNG.seed`, *e.g.*

```
".RNG.seed" <- 314159
```

The second is way to save the state of the RNG from one JAGS session (see the "PARAMETERS TO" statement, section 3.1.12) 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,

```
".RNG.state" <- as.integer(c(20899,10892,29018))
```

is a valid state for the Marsaglia-Multicarry generator. You cannot supply an arbitrary integer 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. If you want to work with more than four parallel chains, then you need to explicitly supply the RNG name and the random seed for each chain.

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.

2.3.3 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.

The Model holds an internal 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 Factory at the beginning of the list, where it will be queried before all of the other Sampler Factories.

A report on the samplers chosen by the model, and the stochastic nodes they act on, can be generated using the "SAMPLERS TO" command. See section 3.1.13.

2.4 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 is 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 convergence.

By default, adaptive mode is turned off half way through first update of a JAGS model, although the user may also control the length of the adaptive phase directly. 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.

2.5 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, and can dump the stored values to file in the format used by the CODA package for R and S-PLUS.

More complex monitors can be defined. For example, the `dic` module defines a *deviance monitor*, which records the deviance of observed stochastic nodes, and a *pD monitor*, which estimates the contribution of a stochastic node to the *effective number of parameters* (p_D) of a model.

All monitors can be dumped to a file in R `dump()` format in the form of a list.

Chapter 3

Running JAGS

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

```
jags <script file>
```

Output from JAGS is printed to the standard output, even when a script file is being used. The JAGS interface is designed to be forgiving. It will print a warning message if you make a mistake, but otherwise try to keep going. This may create a cascade of error messages, of which only the first is informative.

3.1 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.

3.1.1 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 (3.1.15)

3.1.2 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.

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 (3.1.11).

3.1.3 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.

3.1.4 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 may be used before a model has been initialized. You may only supply the values of unobserved stochastic nodes in the parameters file. Logical nodes and constant nodes are forbidden.

See also: PARAMETERS TO (3.1.12)

3.1.5 INITIALIZE

```
. initialize
```

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

3.1.6 UPDATE

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

Updates the model by `n` iterations.

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. In this case the model must be run again with a longer adaptation phase, starting from the MODEL IN statement.

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.

3.1.7 ADAPT

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

Updates the model by `n` iterations and then turns of adaptive mode.

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

3.1.8 MONITOR

In JAGS, a monitor is an object that records a value based on the current state of a node. The simplest monitor simply records the value of the node itself. This is called a "trace" monitor. More complex monitors can be defined that do additional calculations. For example, the `dic` module defines a "deviance" monitor that records the deviance of an observed stochastic node, and a "pD" monitor that estimates the contribution of a node to the effective number of parameters in the model.

In order to distinguish between different classes of Monitor, all monitor statements in the JAGS scripting language have a `type` option. If this is omitted, a trace monitor is assumed.

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

Sets a monitor for variable `<varname>`, The `thin` option sets the thinning interval of the monitor so that it will only record every nth value.

```
. monitor [, type(<name>)]
```

A monitor keyword on its own, with no variable name, will create a set of monitors for default nodes in the model. The nodes chosen as "default" nodes depend on the type of monitor. For a trace monitor, the default nodes are all fixed effects (*i.e.* all model parameters with observed parents).

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

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

```
. monitors to <filename> [, type(<name>)]
```

Dumps the values of all monitors of the given type to the given file in a format that can be read into R using the `source()` function. The R structure is a list with one entry for each sampler.

Note the plural ("monitors to" not "monitor to").

3.1.9 CODA

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

If the named node has a trace monitor, this dumps the monitored values of to files `CODAindex.txt`, `CODAindex1.out`, `CODAindex2.txt`, ... in a form that can be read by the `coda` package of R. The file name stem may be changed from `CODA` to another value by using the `stem()` option. The wild-card character "*" may be used to dump all monitored nodes

3.1.10 EXIT

```
. exit
```

Exits JAGS. JAGS will also exit when it reads an end-of-file character. Note that although JAGS behaves in many ways like classic BUGS, it will not automatically dump the contents of the monitors on exit. You must use the CODA statement before exiting.

3.1.11 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 (3.1.2)

3.1.12 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 (3.1.4)

3.1.13 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.

3.1.14 LOAD

```
. load <module>
```

Loads a module into JAGS (see chapter 4). Once loaded, a module cannot be unloaded in the same JAGS session.

3.1.15 MODEL CLEAR

```
. model clear
```

Clears the current model. The data table (see section 3.1.2) remains intact

3.1.16 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"`.

3.1.17 Change Directory (CD)

```
. cd <dirname>
```

Changes the working directory to `<dirname>`

3.1.18 Directory list (DIR)

```
. dir
```

Lists the files in the current working directory.

3.1.19 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.

3.2 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 "Feedback" below) 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 `yacc`. They generally take the form "syntax error, unexpected FOO, expecting BAR" and are not always abundantly clear.

If a model compiles and initializes correctly, but an error occurs during updating, then the current state of the model will be dumped to a file named `jags.dumpN.R` where N is the chain number. You should then load the dumped data into R to inspect the state of each chain when the error occurred.

Chapter 4

Modules

The JAGS library is distributed along with certain 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. If the module defines a new distribution, then it will typically also define a new sampler for that distribution.
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.

4.1 The base module

The base module supply the base functionality for the JAGS library to function correctly. It is loaded first by default.

4.1.1 Base Samplers

The `base` module defines samplers that use highly generic update methods. These sampling methods only require basic information about the stochastic nodes they sample. Conversely, they may not be fully efficient.

Three samplers are currently defined:

1. The Finite sampler can sample a discrete-valued node with fixed support of less than 20 possible values. The node must not be bounded using the `T(,)` construct
2. The Real Slice Sampler can sample any scalar real-valued stochastic node.
3. The Discrete Slice Sampler can sample any scalar discrete-valued stochastic node.

4.1.2 Base RNGs

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.

The base RNG factory can only generate four independent RNGs from a single seed. Hence JAGS is limited to four parallel chains, unless the user specifies the RNG name and seed in each initial value file.

4.1.3 Base Monitors

The `base` module defines the TraceMonitor class (type "trace"). This is the monitor class that simply records the current value of the node at each iteration.

When default nodes for monitoring are requested, the factory object for TraceMonitors will select all fixed effects (Unobserved stochastic nodes with observed parents) in the model. In a typical model, these will be the highest-level model parameters.

4.2 The bugs module

The `bugs` module defines some of the functions and distributions from WinBUGS. These are described in more detail in sections 5 and 6. The `bugs` module also defines conjugate samplers for efficient Gibbs sampling.

4.3 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. This sampler is still experimental and has not been tested on a sufficiently wide range of problems.

4.4 The dic module

The `dic` module defines new monitor classes for Bayesian model criticism using deviance-based measures. The `deviance` monitor records the deviance of an observed stochastic node. The `pD` monitor can be used to estimate the contribution of a stochastic node to the *effective number of parameters* in the model. Finally, the `popt` monitor can be used to estimate the optimism of the expected deviance.

The `rjags` package provides classes and functions for manipulating penalized deviances. See the rjags manual for details. The use of these monitors through the command-line interface is described below.

4.4.1 The deviance monitor

The JAGS library already has the facility to create a deviance node even without the `dic` module being loaded. The deviance node gives the *total* deviance of the model. The deviance monitor defined in the `dic` module allows more a more detailed evaluation of the model deviance by recording the deviance of *individual* nodes. The command

```
monitor, type(deviance)
```

will create a deviance monitor for all observed stochastic nodes. These monitors cannot be dumped to file in CODA format. Instead they are dumped using the command

```
monitors to "myfile.R", type(deviance)
```

The file "myfile.R" can then be read into R with the `source()` function. This creates a list named "deviance" which contains the sampled values.

4.4.2 The pD monitor

The `pD` monitor estimates the contribution to the effective number of parameters (p_D) [3] from an observed stochastic node by comparing the deviance deviance across multiple chains [1]. It is created by using the option `type(pD)`. If the model has only one chain then a `pD` monitor cannot be defined.

Like the deviance monitor, the `pD` monitor factory will create a monitor for every observed stochastic node using the command.

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

The sum of the monitor values gives the effective number of parameters of the model, in the same way that the sum of the deviance monitor values gives the total deviance of the model.

Again, the values of the `pD` monitors are dumped to file using

```
monitors to "myfile.R", type(pD)
```

When read into R using the `source()` function, this creates a list object named "pD" containing the sampled values.

4.4.3 The `popt` monitor

The `popt` monitor works exactly like the `pD` monitor, but is created using the option `type(popt)`. It gives an estimate of the optimism of the expected deviance (p_{opt}), which can be added to the mean deviance to give the penalized expected deviance [2].

Under asymptotically favourable conditions in which $p_D \ll n$, where n is the sample size

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

For generalized linear models, a better approximation is

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

where $p_D = \sum_i p_{D_i}$.

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 check the output of the `popt` monitor with MCMC diagnostics.

4.5 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 5

Functions

Functions allow deterministic nodes to be defined using the `<-` ("gets") operator. Most of the functions in JAGS are scalar functions taking scalar arguments. However, JAGS also allows arbitrary vector- and array-valued functions, such as the matrix multiplication operator `%*%` and the transpose function `t()` defined in the `bugs` module, and the matrix exponential function `mexp()` defined in the `msm` module. JAGS also uses an enriched dialect of the BUGS language with a number of operators that are used in the S language.

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

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

5.1 Base functions

The functions defined by the `base` module all appear as infix or prefix operators. The syntax of these operators is built into the JAGS parser. They are therefore considered part of the modelling language. Table 5.1 lists them in reverse order of precedence.

Logical operators convert numerical arguments to logical values: zero arguments are converted to FALSE and non-zero arguments to TRUE. Logical and comparison operators return the value 1 if the result is TRUE and 0 if the result is FALSE. Comparison operators are non-associative: the expression `x < y < z`, for example, is syntactically incorrect.

The `%special%` function is an exception in table 5.1. 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 "%" Such functions are automatically recognized as infix operators by the JAGS model parser, with precedence defined by table 5.1.

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

Table 5.1: Base functions listed in reverse order of precedence

5.2 Functions in the bugs module

5.2.1 Scalar functions

Table 5.2 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.

Table 5.3 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.

5.3 Scalar-valued functions with vector arguments

Table 5.4 lists the scalar-valued functions in the `bugs` module that take general arguments. Unless otherwise stated in table 5.4, the arguments to these functions may be scalar, vector, or higher-dimensional arrays.

The `max()` and `min()` functions work like the corresponding R functions. They take a variable number of arguments and return the maximum/minimum element over all supplied arguments. This usage is compatible with WinBUGS, although more general.

Usage	Description	Value	Restrictions on arguments
`abs(x)`	Absolute value	Real	
`cos(x)`	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	
`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	
`sqrt(x)`	Square-root	Real	$x >= 0$
`step(x)`	Test for $x \geq 0$	Logical	
`trunc(x)`	Round to integer towards zero	Integer	

Table 5.2: Scalar functions in the `bugs` module

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 5.3: Link functions in the `bugs` module

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

Table 5.4: Scalar-valued functions with general arguments in the `bugs` module

Usage	Description	Restrictions
`inverse(a)`	Matrix inverse	a is a square matrix
`mexp(a)`	Matrix exponential	a is a square matrix
`rank(v)`	Ranks of elements 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 5.5: Vector- or matrix-valued functions in the `bugs` module

5.4 Vector- and array-valued functions

Table 5.5 lists vector- or matrix-valued functions in the `bugs` module.

The `sort` and `rank` functions behaves like their R namesakes: `sort` accepts a vector and returns the same values sorted in ascending order; `rank` returns a vector of ranks. This is distinct from WinBUGS, which has two scalar-valued functions `rank` and `ranked`.

Chapter 6

Distributions

Distributions are used to define stochastic nodes using the ~ operator. The distributions defined in the bugs module are listed in table 6.1 (real-valued distributions), 6.2 (discrete-valued distributions), and 6.3 (multivariate distributions).

Some distributions have restrictions on the valid parameter values, and these are indicated in the tables. If a Distribution object is given invalid parameter values when evaluating the log-likelihood, it returns $-\infty$. When a model is initialized, all stochastic nodes are checked to ensure that the initial parameter values are valid for their distribution.

Name	Usage	Density	Lower	Upper
Beta	`dbeta(a,b)` $a > 0$, $b > 0$	$\frac{x^{a-1}(1-x)^{b-1}}{\beta(a,b)}$	0	1
Chi-square	`dchisqr(k)` $k > 0$	$\frac{x^{\frac{k}{2}-1}\exp(-x/2)}{2^{\frac{k}{2}}\Gamma(\frac{k}{2})}$	0	
Double exponential	`ddexp(mu,tau)` $\tau > 0$	$\tau\exp(-\tau\lvert x-\mu\rvert)/2$		
Exponential	`dexp(lambda)` $\lambda > 0$	$\lambda\exp(-\lambda x)$	0	
Gamma	`dgamma(r, mu)` $\mu > 0$, $r > 0$	$\frac{\mu^r x^{r-1}\exp(-\mu x)}{\Gamma(r)}$	0	
Generalized gamma	`dgen.gamma(r,mu,beta)` $\mu > 0$, $\beta > 0$, $r > 0$	$\beta\mu^{\beta r}x^{\beta r-1}\exp\{-(\mu x)^\beta\}$	0	
Log-normal	`dlnorm(mu,tau)` $\tau > 0$	$\tau^{\frac{1}{2}}x^{-1}\exp\left\{-\tau(\log(x)-\mu)^2/2\right\}$	0	
Normal	`dnorm(mu,tau)` $\tau > 0$	$\left(\frac{\tau}{2\pi}\right)^{\frac{1}{2}}\exp\{-(x-\mu)^2\tau\}$		
Pareto	`dpar(alpha, c)` $\alpha > 0$, $c > 0$	$\alpha c^\alpha x^{-(\alpha+1)}$	c	
Student t	`dt(mu,tau,k)` $\tau > 0$, $k > 0$	$\frac{\Gamma(\frac{k+1}{2})}{\Gamma(\frac{k}{2})}\left(\frac{\tau}{k\pi}\right)^{\frac{1}{2}}\left\{1+\frac{\tau(x-\mu)^2}{k}\right\}^{-\frac{(k+1)}{2}}$		
Uniform	`dunif(a,b)` $a < b$	$\frac{1}{b-a}$	a	b
Weibull	`dweib(v, lambda)` $v > 0$, $\lambda > 0$	$v\lambda x^{v-1}\exp(-\lambda x^v)$	0	

Table 6.1: Univariate real-valued distributions in the `bugs` module

Name	Usage	Density	Lower	Upper
Bernoulli	`dbern(p)` $0 < p < 1$	$p^x(1-p)^{1-x}$	0	1
Binomial	`dbin(p,n)` $0 < p < 1$, $n \in \mathbb{N}^*$	$\binom{n}{x}p^x(1-p)^{n-x}$	0	n
Categorical	`dcat(p)` $p \in (\mathbb{R}^+)^N$	$\frac{p_x}{\sum_i p_i}$	1	N
Hypergeometric	`dhyper(n1,n2,m1,psi)` $0 \leq n_i$, $0 < m_1 \leq n_+$	$\binom{n_1}{x}\binom{n_2}{m_1-x}\psi^x$	$\max(0, n_+ - m_1)$	$\min(n_1, m_1)$
Negative binomial	`dnegbin(p, r)` $0 < p < 1$, $r \in \mathbb{N}^+$	$\binom{x+r-1}{x}p^r(1-p)^x$	0	
Poisson	`dpois(lambda)` $\lambda > 0$	$\frac{\exp(-\lambda)\lambda^x}{x!}$	0	

Table 6.2: Discrete univariate distributions in the `bugs` module

Name	Usage	Density
Dirichlet	`p ~ ddirch(alpha)` $\alpha_j \geq 0$	$\Gamma(\sum_i \alpha_i) \prod_j \frac{p_j^{\alpha_j - 1}}{\Gamma(\alpha_j)}$
Multivariate normal	`x ~ dmnorm(mu,Omega)` Ω positive definite	$\left(\frac{\lvert\Omega\rvert}{2\pi}\right)^{\frac{1}{2}} exp\{-(x-\mu)^T \Omega (x-\mu)/2\}$
Wishart	`Omega ~ dwish(R,k)` R pos. def.	$\frac{\lvert\Omega\rvert^{(k-p-1)/2} \lvert R\rvert^{k/2} \exp\{-\mathrm{Tr}(R\Omega/2)\}}{2^{pk/2}\Gamma_p(k/2)}$
Multivariate Student t	`x ~ dmt(mu,Omega,k)` Ω pos. def.	$\frac{\Gamma\{(k+p)/2\}}{\Gamma(k/2)(n\pi)^{p/2}} \lvert\Omega\rvert^{1/2} \left\{1 + \frac{1}{k}(x-\mu)^T \Omega (x-\mu)\right\}^{-\frac{(k+p)}{2}}$
Multinomial	`x ~ dmulti(p, n)` $\sum_i x_i = n$	$n! \prod_j \frac{p_j^{x_j}}{x_j!}$

Table 6.3: Multivariate distributions in the `bugs` module

Chapter 7

Differences between JAGS and WinBUGS

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

7.0.1 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 WinBUGS and you wish to convert it for JAGS, then use the command `bugs2jags`, which is supplied with the `coda` package.

7.0.2 Samplers

JAGS has a more limited set of samplers than WinBUGS, which leads to relatively poor performance.

7.0.3 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

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

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

JAGS contains two novel distributions to handle these situations.

1. The `dinterval` distribution represents interval-censored data. It has two parameters: t the original continuous variable, and $c[]$, a vector of cut points of length M, say. If `X` $\sim$ `dinterval(t, c)` then

   $X = 0$ if $t < c[1]$
   $X = m$ if $c[m] \leq t < c[m+1]$ for $1 \leq m < M$
   $X = M$ if $c[M] \leq t$.

2. The `dsum` distribution represents the sum of two variables. It has two parameters, $x1$ and $x2$. If `Y` $\sim$ `dsum(x1,x2)` then $Y = x1 + x2$.

These distributions exist to give a likelihood to data that is, in fact, a deterministic function of the parameters. The relation

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

is logically equivalent to

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

But the latter form does not create a contribution to the likelihood, and does not allow you to define Y as data. The likelihood function is trivial: it is 1 if the parameters are consistent with the data and 0 otherwise. The `dsum` distribution also requires a special sampler, which can currently only handle the case where both $x1$ and $x2$ are discrete-valued.

7.0.5 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)
   }
   ...
}
```

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

7.0.6 Directed cycles

Directed cycles are forbidden in JAGS. There are two important instances where directed cycles are used in BUGS.

- Defining autoregressive priors
- Defining ordered priors

For the first case, the `GeoBUGS` extension to WinBUGS provides some convenient ways of defining autoregressive priors. These should be available in a future version of JAGS.

7.0.7 Censoring, truncation and prior ordering

These are three, closely related issues that are all handled using the `I(,)` construct in BUGS.

Censoring occurs when a variable X is not observed directly, but is observed only to lie in the range $(L, U]$. Censoring is an *a posteriori* restriction of the data, and is represented in WinBUGS by the `I(,)` construct, *e.g.*

```
X ~ dnorm(theta, tau) I(L,U)
```

where L and U are constant nodes.

Truncation occurs when a variable is known *a priori* to lie in a certain range. Although BUGS has no construct for representing truncated variables, it turns out that there is no difference between censoring and truncation for top-level parameters (*i.e.* variables with no unobserved parents). Hence, for example, this

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

is a perfectly valid way to describe a parameter θ with a half-normal prior distribution.

Prior ordering occurs when a vector of nodes is known *a priori* to be strictly increasing or decreasing. It can be represented in WinBUGS with symmetric $I(,)$ constructs, *e.g.*

```
X[1] ~ dnorm(0, 1.0E-3) I(,X[2])
X[2] ~ dnorm(0, 1.0E-3) I(X[1],)
```

ensures that $X[1] \leq X[2]$.

JAGS makes an attempt to separate these three concepts.

Censoring is handled in JAGS using the new distribution `dinterval` (section 7.0.4). This can be illustrated with a survival analysis example. A right-censored survival time t_i with a Weibull distribution is described in WinBUGS as follows:

```
t[i] ~ dweib(r, mu[i]) I(c[i], )
```

where t_i is unobserved if $t_i > c_i$. In JAGS this becomes

```
is.censored[i] ~ dinterval(t[i], c[i])
t[i] ~ dweib(r, mu[i])
```

where `is.censored[i]` is an indicator variable that takes the value 1 if t_i is censored and 0 otherwise. See the MICE and KIDNEY examples in the “classic bugs” set of examples.

Truncation is represented in JAGS using the `T(,)` construct, which has the same syntax as the `I(,)` construct in WinBUGS, but has a different interpretation. 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.

Prior ordering of top-level parameters in the model can be achieved using the `sort` function, which sorts a vector in ascending order.

Symmetric truncation relations like this

```
alpha[1] ~ dnorm(0, 1.0E-3) I(,alpha[2])
alpha[2] ~ dnorm(0, 1.0E-3) I(alpha[1],alpha[3])
alpha[3] ~ dnorm(0, 1.0E-3) I(alpha[2],)
```

Should be replaced by this

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

Chapter 8

Development

At some point there will be a separate manual for JAGS developers. At the moment, if you want to start hacking JAGS then you must rely on the source file documentation. This is written in JavaDoc style and can be processed using `doxygen` to generate a reference manual in the `doc` directory.

The JAGS source is divided into three main directories: `lib`, `modules`, and `terminal`. The `lib` directory contains the JAGS library, which contains all the facilities for defining a Bayesian graphical model in the BUGS language, running the Gibbs sampler and monitoring the sampled values. The JAGS library is divided into several convenience libraries

sarray which defines the basic SArray class, modelled on an S language array, and its associated classes.

function which defines the interface for functions and the `FuncTab` class that allows you to reference them by name.

distribution which defines the interface for distribution and the `DistTab` class that allows you to reference them by name.

graph which defines the various Node classes used by JAGS when constructing a Bayesian graphical model, as well as the `Graph` class which is a container for nodes.

sampler which defines the interface for Samplers, which update stochastic nodes in the graph.

model which defines all the classes needed to create a model, including monitor classes.

compiler which contains the Compiler class and a number of supporting classes designed for an efficient translation of a BUGS-language description the model into a `Graph`.

rng which defines the interface for random number generators (RNGs) and the factories that create them.

util which contains some utility functions used in the rest of the JAGS library.

The `Console` class provides a clean interface to the JAGS library. The member functions of the `Console` class conduct all of the operations one may wish to do on a Bayesian graphical model. They are designed to catch any exceptions thrown by the library and print an informative message to either an output stream or an error stream, depending on the result.

The `modules` directory contains the source code for JAGS modules, which contain concrete classes corresponding to the abstract classes defined in the JAGS library.

The `terminal` directory contains the source code for a reference front end for the JAGS library, which uses the Stata-like syntax described in section 3.1.

Chapter 9

Feedback

Please send feedback to `martyn_plummer@users.sourceforge.net`. I am particularly interested in the following problems:

- Crashes, including both segmentation faults and uncaught exceptions.
- Incomprehensible error messages
- Models that should compile, but don't
- Output that cannot be validated against WinBUGS
- Documentation erors

If you want to send a bug report, it must be reproducible. 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.

Chapter 10

Acknowledgments

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 and who kindly proof-read this manual.

Testing of JAGS on IRIX 6.5 was carried out on Helix Systems at the National Institutes of Health, Bethesda, MD (`http://helix.nih.gov`).

Bibliography

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

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

[3] DJ Spiegelhalter, NG Best, BP 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.

Appendix A

Installation

JAGS is currently distributed in source form only. If you want to use it, you have to compile it. An exception is made for Microsoft Windows, for which a binary distribution is available.

JAGS has been successfully built on GNU/Linux, FreeBSD, Windows XP, Mac OS X, IRIX and Solaris. If you manage to build JAGS on any other platform, then please let me know at `martyn_plummer@users.sourceforge.net`.

These instructions assume that you in a unix environment. If you are working on Windows, then you should skip straight to section A.3.5

A.1 Configuration

JAGS follows the usual GNU convention of

```
./configure
make
make install
```

which is described in more detail in the file `INSTALL` in the top-level source directory. On some UNIX platforms, you may be required to use GNU make (gmake) instead of the native make command.

At configure time you also have the option of defining options such as:

- The names of the C, C++, and Fortran compilers. Although JAGS contains no Fortran code, you are required to define a Fortran compiler so that JAGS modules can be linked against libraries written in Fortran (such as BLAS and LAPACK)
- Optimization flags for the C and C++ compilers. JAGS is optimized by default if the GNU compiler (gcc) is used. Otherwise you must explicitly supply optimization flags.
- Installation directories. JAGS conforms to the GNU standards for where files are installed. You can control the installation directories in more detail using the flags that are listed when you type `./configure --help`.

A.1.1 Configuration for a private installation

If you do not have administrative privileges, you may wish to install JAGS in your home directory. This can be done with the following configuration options

```
export JAGS_HOME=$HOME/jags #or wherever you want it
./configure --bindir=$JAGS_HOME/bin --libdir=$JAGS_HOME/lib \
 --libexecdir=$JAGS_HOME/bin --includedir=$JAGS_HOME/include
```

You then need to modify your PATH environment variable to include `$JAGS_HOME/bin`. You may also need to set `LD_LIBRARY_PATH` to include `$JAGS_HOME/lib` (On Linux this is not necessary as the location of libjags and libjrmath is hard-coded into the JAGS binary).

A.2 BLAS and LAPACK

BLAS (Basic Linear Algebra System) and LAPACK (Linear Algebra Pack) are two libraries of routines for linear algebra. They are used by the multivariate functions and distributions in the `bugs` module. BLAS and LAPACK must be available as *shared* libraries. Although earlier versions of JAGS could be statically linked, this is no longer possible. Most unix-like operating system vendors supply shared libraries that provide the BLAS and LAPACK functions, although the libraries may not literally be called "blas" and "lapack". During configuration, a default list of these libraries will be checked. If `configure` cannot find a suitable library, it will stop with an error message.

You may use alternative BLAS and LAPACK libraries using the configure options `--with-blas` and `--with-lapack`

```
./configure --with-blas="-lmyblas" --with-lapack="-lmylapack"
```

If the BLAS and LAPACK libraries are in a directory that is not on the default linker path, you must set the `LDFLAGS` environment variable to point to this directory at configure time:

```
LDFLAGS="-L/path/to/my/libs" ./configure ...
```

If your BLAS and LAPACK libraries depend on other libraries that are not on the linker path, you must supply these dependency libraries as additional arguments to `--with-blas` and `--with-lapack` (See section A.3.1 for some examples of this).

At runtime, if you have linked JAGS against BLAS or LAPACK in a non-standard location, you must supply this location with the environment variable `LD_LIBRARY_PATH`, *e.g.*

```
LD_LIBRARY_PATH="/path/to/my/libs:${LD_LIBRARY_PATH}"
```

Alternatively, you may hard-code the paths to the blas and lapack libraries at compile time. This is compiler and platform-specific, but is typically achieved with

```
LDFLAGS="-L/path/to/my/libs -R/path/to/my/libs
```

Note: You should not link to BLAS and LAPACK libraries by giving the shared object files as arguments to `--with-blas` and `--with-lapack`:

```
#DON'T DO THIS!
./configure --with-blas=/path/to/my/libblas.so \
--with-lapack=/path/to/my/liblapack.so
```

Although JAGS will compile correctly, the resulting `bugs` module will not be linked to your BLAS or LAPACK libraries. Although this can be rectified using the `LD_PRELOAD` environment variable it is not recommended.

A.3 Platform-specific notes

A.3.1 GNU/Linux

Fortran compiler

The GNU FORTRAN compiler changed between gcc 3.x and gcc 4.x from `g77` to `gfortran`. Code produced by the two compilers is binary incompatible. If your BLAS and LAPACK libraries are linked against `libgfortran`, then they were built with `gfortran` and you must also use this to compile JAGS.

Most recent GNU/Linux distributions have moved completely to gcc 4.x. However, some older systems may have both compilers installed. Unfortunately, if `g77` is on your path then the configure script will find it first, and will attempt to use it to build JAGS. This results in a failure to recognize the installed BLAS and LAPACK libraries. In this event, set the `F77` variable at configure time.

```
F77=gfortran ./configure
```

BLAS and LAPACK

The **BLAS** and **LAPACK** libraries from Netlib (`http://www.netlib.org`) should be provided as part of your Linux distribution. If your Linux distribution splits packages into "user" and "developer" versions, then you must install the developer package (*e.g.* `blas-devel` and `lapack-devel`).

Suse Linux Enterprise Server (SLES) does not include BLAS and LAPACK in the main distribution. They are included in the SLES SDK, on a set of CD/DVD images which can be downloaded from the Novell web site. See `http://developer.novell.com/wiki/index.php/SLES_SDK` for more information.

It is quite common for the Netlib implementations of BLAS and LAPACK to break when they are compiled with the latest GNU compilers. Linux distributions that use "bleeding edge" development tools – such as **Fedora** – may ship with a broken version of BLAS and LAPACK. Normally, this problem is quickly identified and fixed. However, you need to take care to use the online updates of the BLAS and LAPACK packages from your Linux Distributor, and not rely on the version that came on the installation disk.

ATLAS

On Fedora Linux, pre-compiled atlas libraries are available via the `atlas` and `atlas-devel` RPMs. These RPMs install the atlas libraries in the non-standard directory `/usr/lib/atlas` (or `/usr/lib64/atlas` for 64-bit builds) to avoid conflicts with the standard `blas` and `lapack` RPMs. To use the atlas libraries, you must supply their location using the `LDFLAGS` variable (see section A.2)

```
./configure LDFLAGS="-L/usr/lib/atlas"
```

Runtime linking to the correct libraries is ensured by the automatic addition of `/usr/lib/atlas` to the linker path (see the file `/etc/ld.so.conf`), so you do not need to set the environment variable `LD_LIBRARY_PATH` at run time.

AMD Core Math Library

The AMD Core Math Library (acml) provides optimized BLAS and LAPACK routines for AMD processors. To link JAGS with `acml`, you must supply the `acml` library, *and its dependencies*, as arguments to `--with-blas`. It is not necessary to set the `--with-lapack` argument as `acml` provides both sets of functions.

For example, to link to the 64-bit acml using gcc 4.0+:

```
LDFLAGS="-L/opt/acml4.0.0/gfortran64/lib" \
./configure --with-blas="-lacml -lacml_mv -lgfortran"
```

The last version that supports gcc 3.4 is `acml` 3.6.0. When using gcc 3.4, link against `libg2c`.

```
LDFLAGS="-L/opt/acml3.6.0/gnu64/lib" \
./configure --with-blas="-lacml -lacml_mv -lg2c"
```

See also section A.2 for run-time instructions.

Intel Math Kernel Library

The Intel Math Kernel library (MKL) provides optimized BLAS and LAPACK routines for Intel processors. The structure of the MKL changes with each major release, but a configuration that works with both MKL 9.x and MKL 10.0 is

```
MKLPATH=/opt/intel/mkl/<version>/lib/<arch> \
LDFLAGS="-L${MKLPATH}" \
./configure --with-blas="-lmkl -lguide -lpthread" --with-lapack="-lmkl_lapack"
```

MKL 10.0 allows an alternative "pure layered" linking model, in which individual libraries are selected from each layer (interface, threading, computational, and RTL) of the MKL architecture. Using the pure layered model, the configuration above can be re-expressed as

```
./configure \
--with-blas="-lmkl_intel -lmkl_intel_thread -lmkl_core -lguide -lpthread"
```

or, for 64-bit Linux,

```
./configure \
--with-blas="-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -lguide -lpthread"
```

See the MKL User's guide for details.

Using Intel Compilers

JAGS has been successfully built with the Intel C, C++ and Fortran compilers. Naturally, if you are using the Intel compilers, you should also link JAGS against MKL, as outlined above. The additional configure options required to use the Intel compilers are:

```
source /opt/intel/fc/10.0.023/bin/ifortvars.sh
source /opt/intel/cc/10.0.023/bin/iccvars.sh
CC=icc CXX=icpc F77=ifort ./configure
```

A.3.2 OpenSolaris

JAGS has been successfully built and tested on the Intel x86 platform under OpenSolaris 2008.05 using the Sun Studio Express 5/08 compilers.

```
./configure CC=cc CXX=CC F77=f95 \
CFLAGS="-xO3 -xarch=sse2" \
FFLAGS="-xO3 -xarch=sse2" \
CXXFLAGS="-xO3 -xarch=sse2"
```

The Sun Studio compiler is not optimized by default. Use the option `-xO3` for optimization (NB This is the letter "O" not the number "0") In order to use the optimization flag `-xO3` you must specify the architecture with the `-xarch` flag. The options above are for an Intel processor with SSE2 instructions. This must be adapted to your own platform.

To compile a 64-bit version of JAGS, add the option `-m64` to all the compiler flags.

Solaris provides two versions of the C++ standard library: `libCrun`, which is the default, and `libstlport4`, which conforms more closely to the C++ standard. JAGS may be linked to the stlport4 library by adding the options `-library=stlport4 -lCrun` to `CXXFLAGS`.

The configure script automatically detects the Sun Performance library, which implements the BLAS/LAPACK functions. Automatic detection may not work on older versions of Sun Studio, which used a different syntax for specifying this library. In this case, you may need to use the configure option

```
--with-blas="-xlic_lib=sunperf -lsunmath"
```

Using acml

AMD provides a version of their Core Math Library (acml) for Solaris. To use this library instead of the Sun Performance library add the following configure options (changing paths as appropriate):

```
--with-blas="-lacml -lacml_mv -lfsu" \
LDFLAGS="-L/opt/acml4.1.0/sun64/lib \
-R/opt/acml4.1.0/sun64/lib:/opt/SunStudioExpress/lib"
```

The acml library is only available in 64-bit mode, so the option `-m64` must also be added to all the compiler flags.

As with using acml on Linux (section A.3.1), the configure option `--with-blas` must include not only the acml library, but also its dependencies. The `LDFLAGS` option `-R` hard-codes the paths to these libraries into the JAGS modules that require them.

A.3.3 IRIX

JAGS has been successfully built using the MIPSpro 7.4 compiler on IRIX 6.5. The following configure options were used:

```
./configure CC=cc CXX=CC F77=f77 \
CFLAGS="-O2 -g2 -OPT:IEEE_NaN_inf=ON" \
CXXFLAGS="-O2 -g2 -OPT:IEEE_NaN_inf=ON"
```

and JAGS was built with `gmake` (GNU make).

BLAS and LAPACK functions on IRIX are provided by the Scientific Computing Software library (`scs`). The presence of this library is detected automatically by the configure script.

When using the MIPSpro compiler, optimization flags must be given explicitly at configure time. If this is not done, then JAGS will not be optimized at all and will run slowly.

A.3.4 Mac OS X: instructions from Bill Northcott

If trying to build software on Mac OS X you really need to use Leopard (10.5.x). Unless otherwise stated these instruction assume Leopard (10.5.x). The open source support has improved greatly in recent releases. You also need the latest version of Apple's Xcode development tools. The current version is Xcode 3.1. Earlier versions have serious bugs which affect R and JAGS. Xcode is available as a free download from `http://developer.apple.com`. You need to set up a free login to ADC. The Apple developer tools do not include a Fortran compiler. Without Fortran, you will not be able to build JAGS.

For instructions for building on Tiger or for older versions of R see previous versions of this manual.

The GNU gfortran Fortran compiler is included in the R binary distribution available on CRAN. Install the R binary and select all the optional components in the 'Customize' step of the installer. These instructions assume R-2.7.x.

The default C/C++ compiler for Leopard is gcc-4.x. Xcode 3.1 also includes gcc-4.2 and llvm-gcc4.2. The code has been successfully built with these optional compilers but will only run on Leopard. llvm is being actively developed by Apple and may produce better code.

MacOS X 10.2 and onwards include optimised versions of the BLAS and LAPACK libraries. So nothing is needed for Leopard. Optimisation continues and Apple are working on using GPUs for this sort of math. Make sure your OS is up to date.

To ensure the JAGS configure script can find the Fortran compiler for a bash shell

```
export F77=/usr/local/bin/gfortran
```

or for tchsh

```
setenv F77 /usr/local/bin/gfortran
```

To build JAGS unpack the source code and cd into the source directory. Type the following:

```
./configure
make
```

(if you have multiple CPUs try `make -j 2` or `make -j 4`. It may need to be issued more than once)

```
sudo make install
```

You need to ensure `/usr/local/bin` is in your PATH in order for 'jags' to work from a shell prompt.

This will build the default architecture for you Mac: ppc on a G4 or G5 and i386 on an Intel Mac. If you want to build 64bit versions or multiple architecture fat binaries, you will need to ensure that libtool in the JAGS sources is version 1.5.24 or later. Then you can use configure commands like

```
CXXFLAGS="-O3 -arch i386 -arch x86_64" ./configure
```

Make will then build fat binaries. See the R Mac developers page `http://r.research.att.com/` for instructions to build fat R packages.

A.3.5 Windows

These instructions use MinGW, The Minimalist GNU system for Windows. You need some familiarity with Unix in order to follow the build instructions but, once built, JAGS can be installed on any PC running windows, where it can be run from the Windows command prompt.

Preparing the build environment

You need to install the following packages

- MinGW
- MSYS
- NSIS

MinGW (Minimalist GNU for Windows) is a build environment for Windows. There is an official release from `http://www.mingw.org`. However, we used the MinGW distribution that comes as part of the R tools for windows (`http://www.murdoch-sutherland.com/Rtools`), since the compilers in this distribution match the compilers used to build the binary distribution of R for windows, as well as the R packages distributed via CRAN (`http://cran.r-project.org`).

We used version 2.7 of `Rtools`. You only need to install the "MingGW components and tools", not the other components of `Rtools`. The installer will also ask if you wish to modify the Windows PATH. You do not need this.

MSYS (the Minimal SYStem) provides a bash shell for you to build Unix software. These instructions were tested with MSYS 1.0.10. It can be downloaded, as a self-extracting executable, from the MinGW web site `http://www.mingw.org`. At the end of the installation process, it will launch a post-install script that will allow you to use MSYS in conjunction with MinGW. Note that you do *not* need the MSYS developer toolkit (DTK) to compile JAGS.

MSYS creates a home directory for you in `c:\msys\<version>\home\<username>`, where `<version>` is the version of MSYS and `<username>` is your user name under Windows. You will need to copy and paste the source files for LAPACK and JAGS into this directory.

The Nullsoft Scriptable Install System (`http:\\nsis.sourceforge.net`) allows you to create a self-extracting executable that installs JAGS on the target PC. These instructions were tested with NSIS 2.33.

LAPACK

Download the LAPACK source file `lapack-lite-3.1.1.tgz` from `http://www.netlib.org/lapack` and unpack it in your home directory.

```
tar xfvz lapack-lite-3.1.1.tgz
cd lapack-lite-3.1.1
```

Replace the file make.inc with INSTALL/make.inc.LINUX. Then edit make.inc, replacing the line

```
PLAT = _LINUX
```

with something more sensible, like

```
PLAT = _MinGW
```

If using gcc 4.2 (the version that comes with Rtools 2.7), you need to set the FORTRAN and LOADER variables so that they use gfortran.

```
FORTRAN = gfortran
LOADER = gfortran
```

and uncomment the line

```
TIME = INT_ETIME
```

Edit the file Makefile so that it builds the BLAS library. The line that starts lib: should read

```
lib: blaslib lapacklib tmglib
```

Type

```
make
```

The compilation process is slow. Eventually, it will create two static libraries blas_MinGW.a and lapack_MingGW.a. These are insufficient for building JAGS: you need to create dynamic link library (DLL) for each one.

First create a definition file blas.def that exports all the symbols from the BLAS library

```
dlltool -z blas.def --export-all-symbols blas_MinGW.a
```

Then link this with the static library to create a DLL (blas.dll) and an import library (libblas.dll.a)

```
gcc -shared -o blas.dll -Wl,--out-implib=libblas.dll.a \
blas.def blas_MinGW.a -lgfortran
```

(If using gcc 3.4, the library should be linked with -lg2c instead of -lgfortran)

Repeat the same steps for the LAPACK library, creating an import library (liblapack.dll.a) and DLL (lapack.dll)

```
dlltool -z lapack.def --export-all-symbols lapack_MinGW.a
gcc -shared -o lapack.dll -Wl,--out-implib=liblapack.dll.a \
lapack.def lapack_MinGW.a  -L./ -lblas -lgfortran
```

Compiling JAGS

Unpack the JAGS source

```
tar xfvz JAGS-1.0.3.tar.gz
cd JAGS-1.0.3
```

and configure JAGS

```
./configure LDFLAGS="-L/path/to/import/libs/"
```

where `/path/to/import/libs` is a directory that contains the import libraries (`libblas.dll.a` and `liblapack.dll.a`). This must be an *absolute* path name, and not relative to the JAGS build directory.

Normally you will want to distribute the blas and lapack libraries with JAGS. In this case, put the DLLs and import libraries in the sub-directory `win32/lapack`. They will be detected and included with the distribution.

Make sure that the file `makensis.exe`, provided by NSIS, is in your PATH. For a typical installation of NSIS:

```
PATH=$PATH:/c/Program\ files/NSIS
```

Then type

```
make win32-installer
```

The self extracting archive will be in the subdirectory `win32`.

Note that you must go straight from the configure step to `make win32-installer` without the usual step of typing `make` on its own. The `win32-installer` make target resets the installation prefix, and this will cause an error if the source is already compiled.