JAGS Version 0.90 manual

Martyn Plummer
International Agency for Research on Cancer

September 16, 2005

1 Introduction

JAGS is Just Another Gibbs Sampler. It is a program for analysis of Bayesian hierarchical models using Gibbs sampling that aims for the same functionality as classic BUGS (`http://www.mrc-bsu.cam.ac.uk`). If you want to understand what JAGS does, you need to be familiar with BUGS. For most questions, you should therefore consult the WinBUGS manual. This document is a short description of the differences between WinBUGS and JAGS to allow you to get started.

JAGS was written with three aims in mind: to have a BUGS engine that runs on Unix; to be extensible, allowing users to write their own functions and distributions; and to be a platform for experimentation with ideas in Bayesian modelling. To this last end, JAGS is licensed under the GNU General Public License. You may freely modify and redistribute it under certain conditions (see the file `COPYING` for details).

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.

2 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 output is currently quite verbose. It will be cut down in a future release when JAGS is more stable. 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 is is useful when you make a syntax error after a long run.

3 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 #.

3.1 SEED statement

```
. seed <n>
```

Sets the random seed of the random number generator (RNG) provided by the Rmath library. The RNG is Marsaglia-Multicarry, which requires two seeds. For user convenience, these will be generated from the single seed supplied by the user. The seed generation algorithm is copied from R.

This statement is optional. Without it, the random seed is determined by the time at startup. The starting seed is printed out so that the run can be reproduced, if necessary.

3.2 MODEL IN statement

```
. 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. The file name may be optionally enclosed in quotes, and this is necessary if the name contains spaces, or any character other than alphanumeric characters, and '_', '-', '.', '/', or '\'.

The model file should contain a BUGS language description of the model in the form used by WinBUGS.

You may, if you wish, include variable declarations as a comma-separated list in the style of classic BUGS, but these are optional. Here is the standard linear regression example:

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

The semi-colons are optional.

3.3 DATA IN statement

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

JAGS keeps an internal data table containing the values of observed nodes. The DATA IN statement reads data from a file into this data table.

The data must be in the form produced by the `dump()` command in the R language, which is the same format used by the `source()` command to read data into R. It consists of a series of R assignment statements. For example, 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
```

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

You may not supply the values of logical nodes in the data file. Only stochastic nodes and constant nodes are allowed.

See also: DATA TO (3.12).

3.4 COMPILE statement

```
. 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 (**NB JAGS only currently supports one chain**)

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

3.5 PARAMETERS IN statement

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

Reads the data values in `<file>` and writes them to the corresponding **unobserved** variables in chain `<n>`. The file has the same format as the data file. The `chain` option may be omitted, in which case $n = 1$ is assumed. (**NB JAGS currently only supports one chain**)

The PARAMETERS IN statement may be used at any time to overwrite the current parameter values.

You may only supply the values of stochastic nodes in the parameters file. Logical nodes and constant nodes are forbidden.

See also: PARAMETERS TO (3.13)

3.6 INITS statement

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

This is a synonym for PARAMETERS IN. It is kept for compatibility with existing script files, but is deprecated. Please use PARAMETERS IN instead.

3.7 INITIALIZE statement

```
. initialize
```

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

3.8 UPDATE statement

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

Updates the model by `<n>` iterations. A progress bar is printed on the standard output consisting of 40 asterisks. If the `by` option is supplied, a new asterisk is printed every `<m>` iterations. If this entails more than 40 asterisks, the progress bar will be wrapped over several lines. At the end of each line, the percentage of the total run completed is printed. If `<m>` is zero, the printing of the progress bar is suppressed.

3.9 MONITOR statement

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

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

```
. monitor clear <varname>
```

Clears the monitor associated with variable `<varname>`

3.10 CODA statement

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

Dumps the monitored values of variable `<varname>` to files `jags.ind` and `jags.out` in a form that can be read by the CODA package of R. The file name stem may be changed from `jags` to another value by using the `stem()` option. The wild-card character "*" may be used to dump all monitored nodes

3.11 EXIT statement

```
. 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.

3.12 DATA TO statement

```
. 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 (3.3)

3.13 PARAMETERS TO statement

```
. 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 same file can be used as input in a PARAMETERS IN statement.

See also: PARAMETERS IN (3.5)

4 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 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.

5 Differences between JAGS and WinBUGS

Although JAGS aims for the same functionality as BUGS, there are a number of important differences. JAGS has a number of deficiencies compared to WinBUGS, such as the inability to simultaneously run two or more chains, while at the same time some new language features have been introduced that are not found in BUGS.

5.1 Data format

BUGS can read data in either "rectangular" or "S-plus" format. The S-plus format is, in fact, the format produced by the `dput()` function. JAGS uses an alternative data format produced by the `dump()` function in R. JAGS cannot read rectangular data files.

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 CODA version 0.6 or above.

5.2 Variable declarations

JAGS allows the dimensions of the variables to be defined in the model file. The declarations are modelled on classic BUGS, and consist of the keyword `var` followed by a comma-separated list of variable names, with their dimensions in square brackets. The dimensions may be given in terms of any constant expression, *i.e.* a scalar expression using the operators '$*$', '$/$', '$+$', '$-$' and any scalar data values.

As of JAGS version 0.75, **variable declarations are optional**. If a variable is not declared then JAGS has two methods of determining its size.

1. **Using the data table.** If data values are supplied in the data table (*via* DATA IN statements before compilation) then the dimension of an undeclared variable is inferred from this. Note that you can define the dimensions of *unobserved* variables this way by supplying a vector, matrix or array consisting entirely of missing values.
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 nodes. As with WinBUGS, empty indices are not allowed using this method since, for example, it is impossible to calculate the dimension of a node `X[1,]`. If indices are omitted entirely, the variable is taken to be a scalar.

5.3 Samplers

JAGS has a more limited set of samplers than WinBUGS. In particular, Gamermans's sampler for the fixed effects in a GLMM is not yet implemented, which leads to relatively poor performance.

5.4 Functions

The `max` and `min` functions are more general, and behave exactly like the corresponding functions in R. They take a variable number of arguments which may be scalar or vector-valued. The return value is the maximum/minimum value over all supplied arguments.

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.

5.5 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.

5.6 Observable Functions

Logical nodes in the BUGS langauage 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 real examples 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 $1 \leq m < M$; and $X = M$ 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.

5.7 Data transformations

As of version 0.75, 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.

5.8 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.

5.9 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. Under some circumstances, you can also represent truncated nodes with the `I(,)` construct. This works when L, U are constant nodes (You will note that this is syntactically the same as censoring in the BUGS language, even though the underlying concept is quite different).

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` described in section 5.6.

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

Should be replaced by this

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

6 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 `kdoc` to generate an on-line reference manual.

The JAGS source is divided into two main directories: `lib` 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.

functions which defines the standard JAGS functions and the `FuncTab` class that allows you to reference them by name.

distributions which defines the standard JAGS distribution and the `DistTab` class that allows you to reference them by name.

matrix which provides a C interface to various LAPACK functions that are used by multivariate functions and distributions in JAGS.

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 various samplers that 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`.

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 `terminal` directory contains the source code for a reference front end for the JAGS library, which uses the Stata-like syntax discussed above.

Currently, the JAGS library is not installed separately from the client. But in a future version, it will be. This will allow any program to link to the JAGS library, and in particular, a package for R is planned. This development will wait until any outstanding bugs in the library are resolved, and the library is optimized.

7 Feedback

Please send feedback to `<jags@iarc.fr>`. Since JAGS is currently in beta, 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.

8 Acknowledgments

Many thanks to the BUGS development team for creating a piece of software so good I had to copy its functionality. Thanks also to Simon Frost for pioneering JAGS on Windows and Bill Northcott for getting JAGS on Mac OS X to work. Bettina Gruen, Chris Jackson, Greg Ridgeway, Jean-Baptiste Denis and Geoff Evans also provided useful feedback.

Operating System	Library	Replaces
Mac OS X	Accelerate framework	BLAS, LAPACK
Solaris	Sun Performance library	BLAS, LAPACK
IRIX	Scientific Computing Software library	BLAS, LAPACK
GNU/Linux	Intel Math Kernel Library	BLAS, LAPACK
	Automatically tuned linear algebra system (ATLAS)	BLAS
	K. Goto's BLAS	BLAS

Table 1: Optimized replacements for BLAS and LAPACK

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, Mac OS X, IRIX and Solaris. If you manage to build JAGS on any other platform, then please let me know at `<jags@iarc.fr>`.

A.1 Prerequisites

JAGS depends on three external libraries, BLAS, LAPACK and Rmath. You will need to install these libraries before building JAGS.

A.1.1 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 of JAGS.

Reference implementations of these libraries are available from NetLib (`http://www.netlib.org/lapack`). Vendor-supplied linear algebra libraries that supply the same functions are generally faster. Table 1 gives a list of replacements for BLAS and LAPACK:

If the configure script fails to find the BLAS or LAPACK libraries, you may specify their locations using the flags –with-blas and –with-lapack, *e.g*

```
./configure --with-blas=/usr/lib/libblas.so.3.0 \
            --with-lapack=/usr/lib/liblapack.so.3.0
```

A.1.2 Rmath

The Rmath library is a part of the R distribution that can be compiled as a standalone library. It provides functions to calculate the density, distribution and quantile functions of common univariate distributions as well as providing a random number generator.

To build it, unpack the R source and follow the directions in the directory `src/nmath/standalone`. After building Rmath, typing `make install` will copy the header file `Rmath.h` to a place where the C preprocessor will find it (typically `/usr/local/include`) and the libraries `libRmath.a` and `libRmath.so` to a place where your linker will find them (typically `/usr/local/lib`). Don't forget to run `ldconfig` if you want to use the dynamic library.

If you don't want to install the Rmath header and library files, you will have to set the following environment variables before building JAGS(assuming a Bourne shell).

```
export CPPFLAGS="-I<RSRCDIR>/src/include/"
export LDFLAGS="-L<RSRCDIR>/src/nmath/standalone/"
```

Building JAGS then follows the standard procedure:

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

This installs the JAGS executable in `/usr/local/bin`, along with the JAGS library and header files in `/usr/local/lib` and `/usr/local/include/JAGS` respectively.

A.2 Platform-specific notes

A.2.1 GNU/Linux

The BLAS and LAPACK libraries should be provided as part of your Linux distribution. Binary packages of libRmath are available for Debian, and for most RPM-based distributions on CRAN (`http://cran.r-project.org`).

You must install the *development* package for libRmath in order to compile JAGSİf your distribution also includes development packages for BLAS and LAPACK, you must also install these.

A.2.2 Solaris

JAGS has been successfully built using gcc on Solaris 2.9. You cannot use the Sun Performance Library with gcc, so you must install the BLAS and LAPACK libraries from Netlib, and these must also be compiled with gcc. Use the Makeconf.LINUX file from the INSTALL directory.

A.2.3 IRIX

JAGS has been successfully built using the MipsPro compiler for IRIX. You do not need to install LAPACK and BLAS, as JAGS will detect and use the Scientific Computing Software library (scs). When building the R math library, you need a version later than R 2.1.1.

A.2.4 Mac OS X: instructions from Bill Northcott

If trying to build software on Mac OS X you really need to use Tiger (10.4.x) or Panther (10.3.x). The open source support has improved greatly in recent releases. You also need the latest version of Apple's Xcode development tools. These are 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 libRmath. The GNU g77 Fortran compiler is included in the R binary distribution available on CRAN. Install the R binary and select "GNU Fortran", and "Tcl/Tk" from the optional components in the customize step of the installer.

IMPORTANT for Tiger (10.4.x) users: The default C/C++ compiler for Tiger is gcc-4. This is not compatible with g77. Before trying to build anything using g77, the default compiler must be changed with the command `sudo gcc_select 3.3`. (It won't hurt to do this on Panther as well.) gcc-4 uses a new Fortran compiler known as gfortran. This should be usable in the near future, but for the moment some hacking would be required.

MacOS X 10.2 and onwards include optimised versions of the BLAS and LAPACK libraries. so nothing is needed for these. To build libRmath you will need need R source code from `http://www.r-project.org`. Only some versions of R are compatible with JAGS on MacOS X. Versions of the source 2.1 or later should work.

To build libRmath:

1. Get latest R source from `http://cran.r-project.org` and unpack it.

2. cd into the source code directory. Then type the following:

```
    ./configure --with-blas='-framework Accelerate' --with-lapack --with-aqua
    cd src/include
    make
    cd ../nmath/standalone
    make prefix=/usr/local
    sudo make prefix=/usr/local install
```

Notes: You need to be using an admin account for sudo to work. You can change the prefix to '∼/'. That will intall the Rmath library and header in your home directory and sudo will not be needed. This will install Rmath.h and both a static and dynamic libRmath.

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

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

Notes: You need to be using an admin account for sudo to work. Otherwise add 'prefix=∼/' to the configure line. You need to ensure /usr/local/bin is in your PATH in order for 'jags' to work from a shell prompt.

A.2.5 Windows

The following instructions describe how to build a statically linked version of JAGS for Windows. These instructions use MinGW, The Minimalist GNU system for Windows, which is available from `http://www.mingw.org`. You need some familiarity with Unix in order to follow the build instructions but, once built, the JAGS executable can be copied to any PC running Windows and can be run from the Windows command prompt.

NOTE: Although JAGS can be built using the CygWin POSIX emulation layer for Windows (`http://www.cygwin.com`), the performance of the resulting binary is **extremely poor**. Time tests carried out on the classic BUGS examples (volume 1) show that the CygWin version of JAGS runs on average, 8.8 times more slowly than the Linux version running on the same machine. By contrast, JAGS compiled with MinGW, as described here, runs on average 1.5 times slower than the Linux version. There is obviously room for improvement in the Windows binary. This may possibly be resolved using a commercial compiler and anyone who wishes to experiment is encouraged to contact me.

Preparing the build environment

You need to download the following

- The MinGW compiler
- MSYS

The MinGW compiler is distributed as a self extracting executable. Click on it and follow the on-screen instructions. You should choose a "full installation". On the next screen you have the option of downloading further updates from the Internet, and should do so. By default, MinGW will nstall into `c:\mingw`.

MSYS (the Minimal SYStem) is also a self-extracting executable. Once installed, 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, R and JAGS into this directory.

Building the required libraries

LAPACK

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

```
tar xfvz lapack.tgz
cd LAPACK
```

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
```

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

```
lib: blaslib lapacklib
```

It is not necessary to build `tmglib`. Type

```
make lib
```

Copy the resulting file `blas_MinGW.a` to `/mingw/lib/libblas.a` and `lapack_MinGW.a` to `/mingw/lib/liblapack.a`.

Rmath

Get the latest R source from `http://cran.r-project.org`. This is a gzipped tar file, like the others mentioned previously, but there is a difference: the R tarball contains symbolic links, which cannot be handled by the MSYS `tar` command. Unpacking the tarball will therefore give an error. At the time of writing, this error is not important for building the standalone math library, as it concerns only the R packages.

Once you have unpacked the R sources, use the following sequence of commands to build the Rmath library

```
cd R-<version>
./configure --with-readline=no --with-x=no
cd src/include
make
cd ../nmath/standalone
make static
```

This will create a file `libRmath.a` which you should copy to `/mingw/lib`. You should also copy the file `R-<version>/src/include/Rmath.h` to `/mingw/include`

Note that you cannot do a complete build of R in this way. These instructions are only for building the standalone Rmath library. There is extensive documentation on building R for Windows.

Compiling JAGS

To build JAGS, unpack the JAGS source and type the following commands from within MSYS

```
./configure
make
```

The `jags.exe` executable may then be found in the file `JAGS-0.90/src/terminal`. Copy it to somewhere on your Windows PATH. You can modify the PATH environment variable from the system dialogue in the control panel.

By default, the JAGS executable contains a lot of debugging information and is consequently quite large. If you are not going to do any debugging then you may wish to strip it:

```
strip jags.exe
```

This considerably reduces the file size.